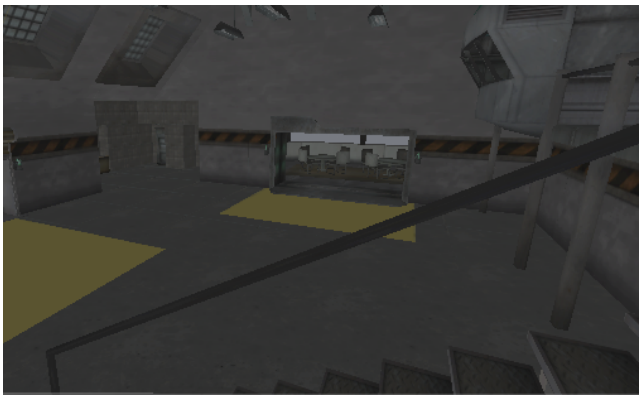
An Enhanced FPS in Unity



This intermediate-level tutorial extends upon the Basic FPS tutorial by introducing game elements such as multiple weapons, damage and enemies.

Time to complete: 3 - 4 hours.

Author: Graham McAllister

Contents

1. Aims of this tutorial
2. Prerequisites
3. Weapon Switching
4. Rocket launcher
5. Machine gun
6. Hit points
7. Sentry gun

Download the assets for this tutorial from:

www.otee.dk/tutorials/fps_assets.zip

1. Aims of this tutorial

This tutorial will detail how to add more games features to our FPS such as multiple weapons, damage (hit points) and basic enemies (sentry guns).

2. Prerequisites

You should already be familiar with the concepts discussed in the Basic FPS tutorial.

Note: any text that requires the user to take an **action** begins with a '-'.

Before we begin - level setup

- Create a new Unity project as described in the Basic FPS tutorial. Ensure that you have the FPS assets imported.

  Add the scene and the FPS controller.

  Note: In this tutorial no new scripts need to created, we'll be using the ones that were downloaded.

3. Weapon Switching

Before we discuss how to create each individual weapon, we need to write some code to manage how the weapons are initialized and switched from one to another. Let's look at the Javascript for **PlayerWeapons.js**:

```
function Awake () {
     // Select the first weapon
1    SelectWeapon(0);
}


function Update () {
     // Did the user press fire?
2    if (Input.GetButton ("Fire1"))
          BroadcastMessage("Fire");

     if (Input.GetKeyDown("1"))
     {
          SelectWeapon(0);
     }
     else if (Input.GetKeyDown("2"))
     {
          SelectWeapon(1);
     }
}

3 function SelectWeapon (index : int) {
     for (var i=0;i<transform.childCount;i++)
     {
          // Activate the selected weapon
          if (i == index)
               transform.GetChild(i).gameObject.SetActiveRecursively(true);
          // Deactivate all other weapons
          else
               transform.GetChild(i).gameObject.SetActiveRecursively(false);
     }
}
```

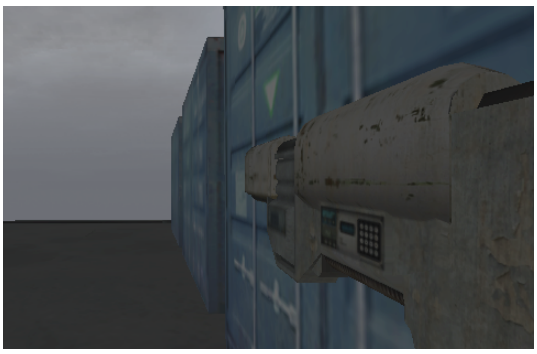
1. This function initializes weapon 0 as the default weapon.
2. This function detects keyboard input; the fire button, the "1" button for weapon 1 or the "2" button for weapon 2. The weapons will be children objects of the Main Camera.
3. This activates the corresponding weapon depending on keyboard input.

Let's use the above code.

- Create an empty game object called Weapons. Move this so that it is a child object to Main Camera (inside FPS controller). Our weapons will be added as children of this object.
- Assign the PlayerWeapons.js script to the Weapons game object under Main Camera.

We'll now create our first weapon.

4. Rocket Launcher



This section will describe how to make a rocket launcher style weapon.

Rocket Launcher

The rocket launcher is responsible for instantiating a rocket and giving it an initial velocity. The rocket will be launched directly at wherever the user is pointing and will be destroyed whenever it collides with another collider.

- Add an empty game object and name it RocketLauncher. Position the game object in the approximate position where the FPS Controller's hands would be.
- Add the RocketLauncher as a child to the Weapons game object inside the Main Camera in the Hierarchy View. This allows us to shoot wherever the camera is pointing and also makes sure that the RocketLauncher game object follows the FPS Controller as it moves around (as Main Camera is a child of FPS Controller).
- [TODO make the it's visible, it has to have no translation or rotation, this is done by choosing reset from the inspector]. Rotate and translate so that it looks good. Don't rotate the empty game object holder, just the machine gun model.
- Drag Objects/weapson/rocketLauncher model so that it is a child of the RocketLauncher empty game object.
- The code for the **RocketLauncher.js** script is as follows:

```
var projectile : Rigidbody;
var initialSpeed = 20.0;
var reloadTime = 0.5;
var ammoCount = 20;
private var lastShot = -10.0;

function Fire ()
{
     // Did the time exceed the reload time?
1 →  if (Time.time > reloadTime + lastShot && ammoCount > 0)
     {
          // create a new projectile, use the same position and rotation as the
          // Launcher.
          var instantiatedProjectile : Rigidbody = Instantiate (projectile,
               transform.position, transform.rotation);
          // Give it an initial forward velocity. The direction is along the z-axis of
          // the missile launcher's transform.
          instantiatedProjectile.velocity = transform.TransformDirection(Vector3 (0, 0,
               initialSpeed));

          // Ignore collisions between the missile and the character controller
          Physics.IgnoreCollision(instantiatedProjectile.collider,
               transform.root.collider);

1 →       lastShot = Time.time;
          ammoCount--;
     }
}
```

1. This code ensures that the weapon can't fire faster than **reloadTime**. It also checks that the user can fire only when they have sufficient ammo.

The behaviour for the RocketLauncher is similar to that in the previous FPS tutorial with the exception of the reload time and ammo count described above.

- Attach the RocketLauncher.js script to the RocketLauncher game object.

Rocket

We will now build the rocket in the scene and upload the final version to a prefab.

- Drag the Objects/weapons/rocket model into the Scene view.
- Attach the WeaponScripts/Rocket script to it.
- Add a box collider to the rocket. Make the box collider slightly larger than the actual rocket to prevent tunneling of collisions. Tunnelling of collisions is what happens when small, fast game objects avoid collision detection due to their size and speed. Making the box collider z-axis larger ensures collisions work correctly for these obejcts.
- [TODO insert timestep diagram here of 2 consecutive frames]
- Create a particle system, Game Object->Create Other->Particle System.
- Modify the elipsoid x,y,z sizes to 0.1.
- Modify the Rnd Velocity to 0.1 in each axis also.
- Change the particle emitter min size and max size to 0.5.
- Change the number of particles emitted to 100 (max and min).
- Drag the Particle Effects/smoke.mat onto the particle system.
- In the Particle Animator section, set each of the axis values to 0.5.
- In the RigidBody of the rocket, deselect 'useGravity'. This ensures that the rocket does not fall under gravity.
- Set the size grow variable to 3.
- Enable autodestruct on the particle system. This ensures that the particle system is removed from the game after the rocket has been destroyed.
- Drag the particle system in the Hierarchy View so that it is a child of the rocket. Reset the transform of the particle system so that it is centred on the rocket, then modify the position so that it is at the rear of the rocket.
- Select the rocket in the Hierarchy view and check that the smoke trail follows it around in the Scene view.

We've now made our rocket, complete with smoke trail. We're now ready to upload our changes to the prefab.

- Firstly, create an empty prefab to upload our changes to. Call the prefab 'rocket'
- Select the rocket in the Hierarchy view and drag it onto the new rocket prefab.
- Create a new directory in the Project view called 'WeaponPrefabs' to store our weapon prefabs in.

Here's the Javascript for **Rocket.js**:

```
// The reference to the explosion prefab
var explosion : GameObject;
var timeOut = 3.0;

// Kill the rocket after a while automatically
function Start () {
     Invoke("Kill", timeOut);
}


function OnCollisionEnter (collision : Collision) {
     // Instantiate explosion at the impact point and rotate the explosion
     // so that the y-axis faces along the surface normal
     var contact : ContactPoint = collision.contacts[0];
     var rotation = Quaternion.FromToRotation(Vector3.up, contact.normal);
    Instantiate (explosion, contact.point, rotation);

     // And kill ourselves
     Kill ();
}

function Kill ()
{
     // Stop emitting particles in any children
     var emitter : ParticleEmitter= GetComponentInChildren(ParticleEmitter);
     if (emitter)
          emitter.emit = false;

     // Detach children - We do this to detach the trail rendererer which should be set
     // up to auto destruct
     transform.DetachChildren();

     // Destroy the projectile
     Destroy(gameObject);
}


@script RequireComponent (Rigidbody)
```

1. The Kill function firstly finds the particle emitter in the child hierarchy and turns its emitter off. Next, it detaches any children (Smoketrail) from the object the script is attached to (the rocket) and destroys the rocket.
2. The most important line is the transform.DetachChildren() function. This is called prior to destroying the gameObject (the rocket) so that when the rocket is destroyed, the trail will now remain as it is no longer a child.
3. The '@script" command ensures that a Rigidbody is attached to the component that the script is attached to (as the script requires a Rigidbody).

Once a rocket instance collides with another collider, we want to destroy the rocket game object. However, if the trail is directly attached to the rocket, this will be destroyed also, and the trail will disappear suddenly. This is not what happens in real life, and this is why it is important to detach the smoketrail from the rocket before destroying the rocket.

Notice that the rocket can be killed in one of two ways, either it is destroyed if it has survived for more then 3 seconds (e.g. shot into the air), or it is destroyed if it collides with an object.

- Attach the Javascript to the Rocket game object.

- [TODO , hook up the rocket to the rocketlauncher variable].
- Play the game, when you fire a rocket, it should have a smoke trail following it.

Explosions

You probably noticed that when you fired a rocket there was no explosion when it collided, we'll add one in now.

- Drag the Small-Explosion prefab onto the exposed variable Explosion in the Rocket component of the Rocket prefab.

We still need to define the behaviour of our explosion, create a new Javascript and name it Explosion-Simple. Here's the code for the **Explosion-Simple.js** script:

```
var explosionRadius = 5.0;
var explosionPower = 10.0;
var explosionDamage = 100.0;

var explosionTime = 1.0;

function Start () {

     var explosionPosition = transform.position;
1    var colliders : Collider[] = Physics.OverlapSphere (explosionPosition,
     explosionRadius);


     for (var hit in colliders) {
          if (!hit)
               continue;

          if (hit.rigidbody)
          {
2              hit.rigidbody.AddExplosionForce(explosionPower, explosionPosition,
                    explosionRadius, 3.0);


               var closestPoint = hit.rigidbody.ClosestPointOnBounds(explosionPosition);
               var distance = Vector3.Distance(closestPoint, explosionPosition);

               // The hit points we apply decrease with distance from the hit point
3              var hitPoints = 1.0 - Mathf.Clamp01(distance / explosionRadius);
               hitPoints *= explosionDamage;

               // Tell the rigidbody or any other script attached to the hit object
               // how much damage is to be applied
4              hit.rigidbody.SendMessageUpwards("ApplyDamage", hitPoints,
                    SendMessageOptions.DontRequireReceiver);

          }
     }

    // stop emitting ?
    if (particleEmitter) {
        particleEmitter.emit = true;
           yield WaitForSeconds(0.5);
           particleEmitter.emit = false;
    }

    // destroy the explosion
     Destroy (gameObject, explosionTime);
}
```

1. This returns an array of colliders within the volume of the sphere.
2. This adds an upward force to all rigidbodies within range of the explosion (the sphere). Basically this makes the explosion look good!
3. This calculates how much damage to apply to each rigidbody caught in the explosion. The degree of damage decreases the further a rigidbody is from the centre of the explosion.
4. This sends a message to apply the damage to the rigidbody.

- Add the Explosion-Simple script as a component to the Small-Explosion prefab.

This explosion script can be used as a **generic** explosion script for any game object that requires explosions. To tailor the script to suit each specific case, just modify variables such as:

1. explosionPower - the degree of force with which the explosion will cause nearby objects to move.

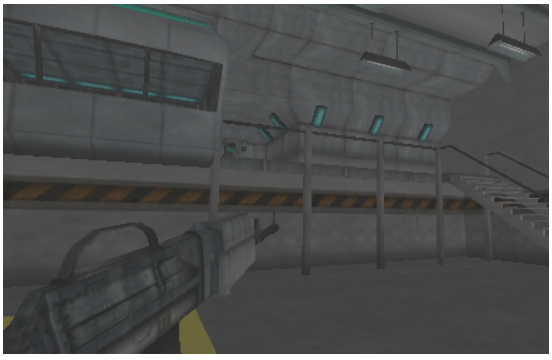
2. explosionDamage - how many hitpoints will the explosion cause.

3. explosionRadius - over what radius does the explosion have effect.

The Explosion script is very similar to that used in the previous FPS tutorial with the main difference being in the introduction of the **hitpoints**. The hitpoints variable scales the explosionDamage variable based on distance, with object at the outer edge of the radius having lesser damage than those at the centre of the explosion.

This means it's now possible for an explosion to inflict damage to objects near to where it collided. How to use hitpoints for each object will be discussed in more detail later.

- Play the game.



5. Machine Gun

The machine gun style weapon offers a faster rate of fire than the rocket launcher however each bullet impact does less damage.

- Crate an empty game object and name it MachineGun. Add this as a child object to Weapons in the Hierarchy View.

- Add Objects/weapons/machineGun to the empty MachineGun game object.

- Assign the machine gun script to the MachineGun game object.
- Assign muzzle_flash (it's a child of machineGun) to the muzzleFlash variable of machineGun.

Here's the complete code for **MachineGun.js**:

```
var range = 100.0;
var fireRate = 0.05;
var force = 10.0;
var damage = 5.0;
var bulletsPerClip = 40;
var clips = 20;
```

```
var reloadTime = 0.5;
private var hitParticles : ParticleEmitter;
var muzzleFlash : Renderer;

private var bulletsLeft : int = 0;
private var nextFireTime = 0.0;
private var m_LastFrameShot = -1;

function Start ()
{
        hitParticles = GetComponentInChildren(ParticleEmitter);

        // We don't want to emit particles all the time, only when we hit something.
        if (hitParticles)
             hitParticles.emit = false;
        bulletsLeft = bulletsPerClip;
}

function LateUpdate()
{
        if (muzzleFlash)
        {
             // We shot this frame, enable the muzzle flash
             if (m_LastFrameShot == Time.frameCount)
             {
                  muzzleFlash.transform.localRotation = Quaternion.AxisAngle
                       (Vector3.forward, Random.value);
                  muzzleFlash.enabled = true;
             }
             // We didn't, disable the muzzle flash
             else
             {
                  muzzleFlash.enabled = false;
                  enabled = false;
             }
        }
}

function Fire ()
{
        if (bulletsLeft == 0)
             return;

        // If there is more than one bullet between the last and this frame
        // Reset the nextFireTime
        if (Time.time - fireRate > nextFireTime)
             nextFireTime = Time.time - Time.deltaTime;

        // Keep firing until we used up the fire time
        while( nextFireTime < Time.time && bulletsLeft != 0)
        {
             FireOneShot();
             nextFireTime += fireRate;
        }
}

function FireOneShot ()
{
        var direction = transform.TransformDirection(Vector3.forward);
        var hit : RaycastHit;

        // Did we hit anything?
```

```
        if (Physics.Raycast (transform.position, direction, hit, range))
        {
             // Apply a force to the rigidbody we hit
             if (hit.rigidbody)
                  hit.rigidbody.AddForceAtPosition(force * direction, hit.point);

             // Place the particle system for spawing out of place where we hit the
             // surface!

             // And spawn a couple of particles
             if (hitParticles)
             {
                  hitParticles.transform.position = hit.point;
                  hitParticles.transform.rotation =
                            Quaternion.FromToRotation (Vector3.up,
                            hit.normal);
                  hitParticles.Emit();
             }

             // Send a damage message to the hit object
             hit.collider.SendMessageUpwards("ApplyDamage", damage,
                  SendMessageOptions.DontRequireReceiver);

        }

        bulletsLeft--;

        // Register that we shot this frame,
        // so that the LateUpdate function enabled the muzzleflash renderer for one
        // frame
        m_LastFrameShot = Time.frameCount;
        enabled = true;

        // Reload gun in reload Time
        if (bulletsLeft == 0)
             StartCoroutine("Reload");
}



5 →  function Reload () {

        // Wait for reload time first - then add more bullets!
        yield WaitForSeconds(reloadTime);

        // We have a clip left reload
        if (clips > 0)
        {
             clips--;
             bulletsLeft = bulletsPerClip;
        }
}

function GetBulletsLeft () {
        return bulletsLeft;
}
```

1. The Start function is really just initializing the particle emitter (bullet spark) so that it is turned off.
2. The LateUpdate function is automatically called after an Update function is called. **Note** that the Update function is called in the PlayWeapons script which is attached to the Weapons game object (it's a parent of machineGun).

Generally the LateUpdate function will be used whenever you want to react to something that happened in Update. In this case, the player if firing in the Update function, and in LateUpdate we're applying the muzzle flash.

3. The Fire function calculates if the player should be able to fire based on the fire rate of the machine gun.
4. The FireOneShot function starts by casting out a ray in front of the FPS Controller to determine if the bullet has hit anything. We'll limit the range of the bullet to a certain distance.
   If the raycast did intersect with a rigidbody, then a force is applied in the forward direction to that rigidbody (a small force as it's only a machine gun).
   Next the bullet spark is instantiated at the location where the bullet (ray) struck. The particle emitter is oriented so that it is along the normal of the surface it struck.
   Next, damage is applied to the object by sending a damage message to the object that was hit.
5. The Reload function reloads a clip of ammo (if there are any left of course). The amount of time taken to reload can be tweaked in the inspector.

Configuring the particle emitter

The MachineGun needs a spark effect when bullets collide with rigidbodies, let's create one. There are two ways of creating a spark effect, an easy way, and a slightly harder way which is more configurable. Let's look at both.

Easy bullet sparks

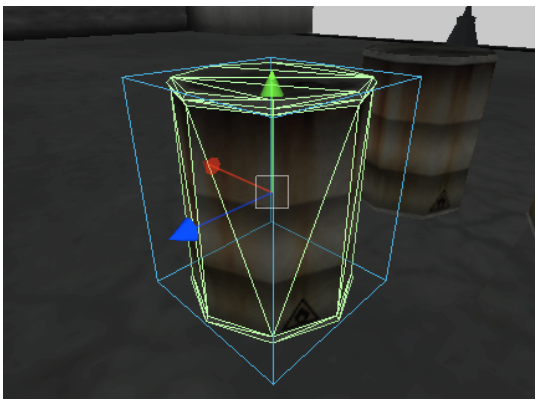
- Drag the Sparks prefab from the Standard Assets/Particles folder so that it's a child of **m**achineGun (not **M**achineGun) in the Hierarchy View.
- That's it! Play the game.

Configuring bullet sparks (more advanced)

- Create a new particle system (Game Object->Create Other->Particle system and name it BulletSpark. Add this as a child of **m**achineGun (not **M**achineGun - note case).

The default particle effect is not suitable, we'll look at how to edit the parameters of the particle system to create a bullet spark effect.

- Select BulletSpark, make sure Emit is selected in the Inspector Panel (this makes sure we can see the effects the parameters are having in real-time).
- In the Ellipsoid Particle Emitter section, change the Ellipsoid X,Y,Z values to X=0, Y=1, Z=0.
- In the Local Velocity section, change the X,Y,Z values to X=0, Y=1, Z=0.
- The lifespan of the particles is a too long, we can shorten them by changing the Max Energy parameter (1.0 or less should be fine).
- Play the game and check that the bullet sparks look good. Don't forget that 1 and 2 on the keyboard are used to switch weapons.



6. Hit Points

The Explosion and MachineGun Javascripts have already shown how to calculate the degree of damage caused by a projectile (rocket or bullet), and send this value to all nearby game objects. However, game objects do not yet know how to **respond** to this value.

Game objects can keep track of how healthy they are by using a **hitPoints** variable. Each object will be initialized to its own value (depending on how 'strong' it is). Each game object that should respond to damage should also have an **ApplyDamage()** function (note this is the function called from the Explosion and MachineGun scripts to apply the damage). This function will decrement hit points from the game object as necessary and call functions to handle what happens when the hit points reach 0 (typically a death or explosion state).

The next section will demonstrate how hitPoints and ApplyDamage() are used.

Exploding Barrels

The code we're about to look at is generic, so the Javascript can be added as a component to any object that can have damage applied to it. Here is the complete code for **DamageReceiver.js**:

```
var hitPoints = 100.0;
var detonationDelay = 0.0;
var explosion : Transform;
var deadReplacement : Rigidbody;

1 → function ApplyDamage (damage : float)
{
      // We already have less than 0 hitpoints, maybe we got killed already?
      if (hitPoints <= 0.0)
            return;

      hitPoints -= damage;
      if (hitPoints <= 0.0)
            Invoke("DelayedDetonate", detonationDelay);
}

function DelayedDetonate ()
{
2 →       BroadcastMessage ("Detonate");
}

function Detonate ()
{
      // Destroy ourselves
      Destroy(gameObject);

      // Create the explosion
3 →       if (explosion)
            Instantiate (explosion, transform.position, transform.rotation);
```

```
     // If we have a dead barrel then replace ourselves with it!
4    if (deadReplacement)
     {
          var dead : Rigidbody = Instantiate(deadReplacement, transform.position,
               transform.rotation);

          // For better effect we assign the same velocity to the exploded barrel
          dead.rigidbody.velocity = rigidbody.velocity;
          dead.angularVelocity = rigidbody.angularVelocity;
     }
}

// We require the barrel to be a rigidbody, so that it can do nice physics
@script RequireComponent (Rigidbody)
```

1. This function applies the damage to the game object which has been shot or caught in an explosion. If the object's hit points are already 0 or less, then nothing is done, else the decrement the hit point counter by the value passed in (damage). If the resulting hit points are now less than 0, call the DelayedDetonate function (delays can make the explosion look cool, no other reason).
2. This calls the Detonate method on the game object and its children.
3. If there's an explosion prefab attached to the barrel, then display that when the barrel's hit points reach zero.
4. If the game object has a dead equivalent, in this case a barrel which looks burnt out, then replace the normal game object with its dead equivalent. We make sure the object keeps travelling in the direction that it was going when its hit points reached zero.

Let's set up our game to use the DamageReceiver script on some barrels. Let's start by importing some assets.

- Import barrel from Objects/crateAndBarrel/barrel.
- Drag barrel from the Project View into the Scene View.
- Add a rigidbody component to the barrel.
- Add a box collider to the barrel (Component->Physics->Box Collider). You may want to tune the size of the box collider so that it is a better fit to the barrel. Do this by altering the size properties of Box Collider in the Inspector View.
- Attach the DamageReceiver script to Barrel in the Hierarchy View.
- Assign the explosion prefab to the Explosion property (in the Damage Receiver component of Barrel).
- Create a new prefab called **B**arrel (note capital B, the imported barrel has a small 'b').
- Drag the barrel we have configured from the Hierarchy View into the newly created prefab in the Project View.
- Add several Barrels to the Scene View (use duplicate as it's easier).
- Play the game.

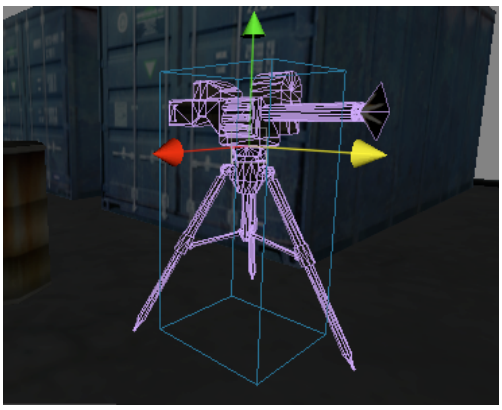
You'll notice when the barrels explode they just disappear, we'll now add a dead replacement barrel (a barrel that looks blown up).

- Create another prefab called Barrel-dead.
- Assign the original barrel to the prefab (the one we imported from Objects/ crateAndBarrel.

At the moment, the Barrel and Barrel-dead objects look the same, they both have a texture in common (barrel1).

We want Barrel-dead to have a texture that looks different from Barrel, so the user will know when it has exploded, something that looks burnt out would do. If we modify barrel1 texture to give this effect, then the Barrel object will also be modified, as Barrel and Barrel-dead both share this same texture. To solve this problem, we must create a new texture and assign that to Barrel-dead. We'll do this by firstly copying the barrel1 texture material and modifying it to resemble a burned out look.

- Select Barrel-dead in the Project View and click on barrel1 under the **Mesh Renderer** section of the Inspector View. A line appears showing where the barrel1 texture is in the Project View.
- Duplicate the barrel1 material (inside the **Materials** folder) (command+D) and name the copy barrelDead. Note the duplicate will likely be named barrel2, rename this barrelDead.
- We'll now modify the look of this texture so it looks burnt out. Make sure the barrelDead material is still selected and click on Main Color in the Inspector View. Drag each of the R,G,B sliders to the left-hand side (close to 0), this will give the texture a black (or burnt) appearance.
- Assign this new material to Barrel-dead by firstly selecting Barrel-dead in the Project View, then click on the drop down arrow beside Element 0 in the Mesh Renderer section of the Inspector View and assign the texture we just created, barrelDead.
- Verify that Barrel and Barrel-dead both look different by dragging them into the Scene View and comparing them. Delete BarrelDead from the Scene View again, as it should only appear once a barrel has been blown up.
- Next, add Box Collider and Rigidbody components to the Barrel-dead prefab (the Barrel prefab should already have them, check anyway).
- Assign the Barrel-dead prefab to the Dead Replacement property of Barrel.
- Play the game, the barrels should now explode and have the burned out effect.



7. Sentry Gun

Finally we'll add an enemy opponent to our game, a sentry gun. The sentry gun object will look for the player and shoot at them.

Let's start by importing the sentry gun weapon.

- Drag the sentryGun from Objects/weapons onto the Scene View.

-Add a box collider and a rigidbody to the sentryGun.

-Adjust the size and shape of the box collider so that it resembles a thin column which covers the turret of the gun. The tall, thin, column will make sure that the gun has a high centre of gravity and will fall over easily when shot.

- Attach the DamageReceiver script to the sentryGun. Assign the Explosion exposed variable.

We're now ready to examine the code for the sentry gun. Here's the full code for **SentryGun.js**:

```
var attackRange = 20.0;
var target : Transform;

1 function Start () {
      if (target == null && GameObject.FindWithTag("Player"))
            target = GameObject.FindWithTag("Player").transform;
}

2 function Update () {
      if (target == null)
            return;
      if (!CanSeeTarget ())
            return;
      // Rotate towards target
      var targetPoint = target.position;
      var targetRotation = Quaternion.LookRotation (targetPoint - transform.position,
            Vector3.up);
      transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation,
            Time.deltaTime * 2.0);


      // If we are almost rotated towards target - fire one clip of ammo
      var forward = transform.TransformDirection(Vector3.forward);
      var targetDir = target.position - transform.position;
3     if (Vector3.AngleBetween(forward, targetDir) * Mathf.Rad2Deg < 10.0)
            SendMessage("Fire");
}

4 function CanSeeTarget () : boolean
{
      if (Vector3.Distance(transform.position, target.position) > attackRange)
            return false;
      var hit : RaycastHit;
      if (Physics.Linecast (transform.position, target.position, hit))
            return hit.transform == target;

      return false;
}
```

1. The Start function checks to see if a target has been assigned for the gun (this can be done in the inspector), but it's much easier to assign the Player tag to the FPS controller using the inspector (we'll do this shortly).
2. If the player is within range, and the sentry gun can see the player, the gun turret will rotate from the its current rotation angle to the rotation angle of the player.
3. If the angle of rotation between the player and the current position of the sentry gun is less than 10 degrees, the sentry starts firing.
4. The CanSeeTarget function works out if the the sentry gun can see the target (in this case the player).

Let's finish setting up the sentry gun.

- Assign the target for the sentry gun. To do this, select FPS Controller in the Hierarchy View and then in the Tag drop down box, select Player.
- Attach the SentryGun script to the sentryGunRotateY child of sentryGun. This ensures that only the top of the sentry rotates and the tripod part remains stationary.
- Assign the explosion prefab to the Explosion property of the DamageReceiver component of sentryGun.
- Assign the sentryGun to the Dead Replacement property of the DamageReceiver component (or you can create an alternative dead replcement if you wish).
- Assign the MachineGun script to sentryGunRotateY.
- Assign the Muzzle Flash property of the Machine Gun component with the muzzleflash asset which is a child of sentryGunTop.
- Click on muzzleflash in the Hierarchy View, change its shader to Particles->Additive.
- Play the game. You should now be able to shoot the barrels and sentry gun.

Finally ...

Skybox

Let's add a sky effect to our scene.

- Import the skyBoxTest.mat from Asssets/Materials/

- Import B,D, F, L,R and U.tif from Assets/Skybox. The easiest way to do this is to copy the files then paste them into your project's Assets directory using Finder.

- Select skyBoxTest in the Project View. Assign the F texture to Front, B to Back etc.

- Select Edit->Render Settings. Drag the skyBoxTest onto the Skybox Material property.

You now have sky.

Summary

You now have an FPS game with the following:

- Multiple weapons
- A sentry gun enemy with simple AI.
- Barrels which react to physics and can be blown up.
- Objects which can have a different model when destroyed.
- Damage control (hit points).

Try adding other assets to the scene from the Assets directory, enjoy!

The next tutorial in the FPS series will demonstrate advanced concepts such as ragdoll character animation, advanced AI, and adding polish to our game.

Acknowledgments

Special thanks go to Joachim Ante (code) and Ethan Vosburgh (graphics) for their help in the making of this tutorial.