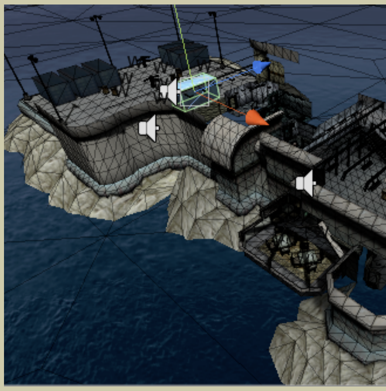


Creating a First Person Shooter (FPS)

Part 1

Author: Graham McAllister
Revised by: Jeff Aydelotte &
Amir Ebrahimi

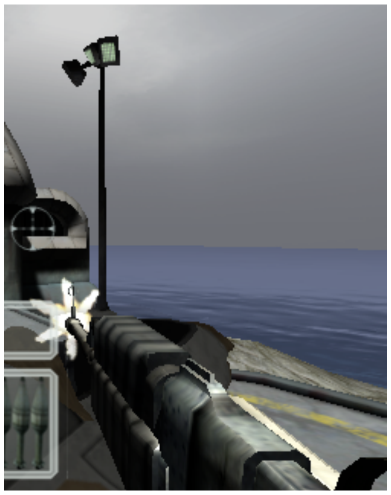
Time to complete: 3-4 hours

Last Revision: 10-July-2009

Contents

Part 1: Introduction

This tutorial will detail how to make a simple First Person Shooter (FPS). It will introduce fundamental 3D game programming concepts and give tips on how to think like a game programmer.

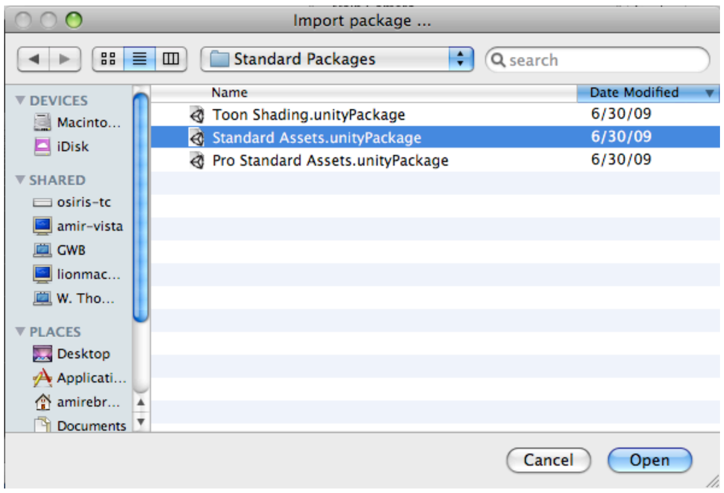


Prerequisites

This tutorial assumes that you are familiar with the Unity interface and basic scripting concepts.

Starting a new project

- Download FPS_Tutorial.zip, unzip, and open the project folder in Unity.
- Import the Standard Assets Unity Package from your Standard Packages folder where Unity is installed.



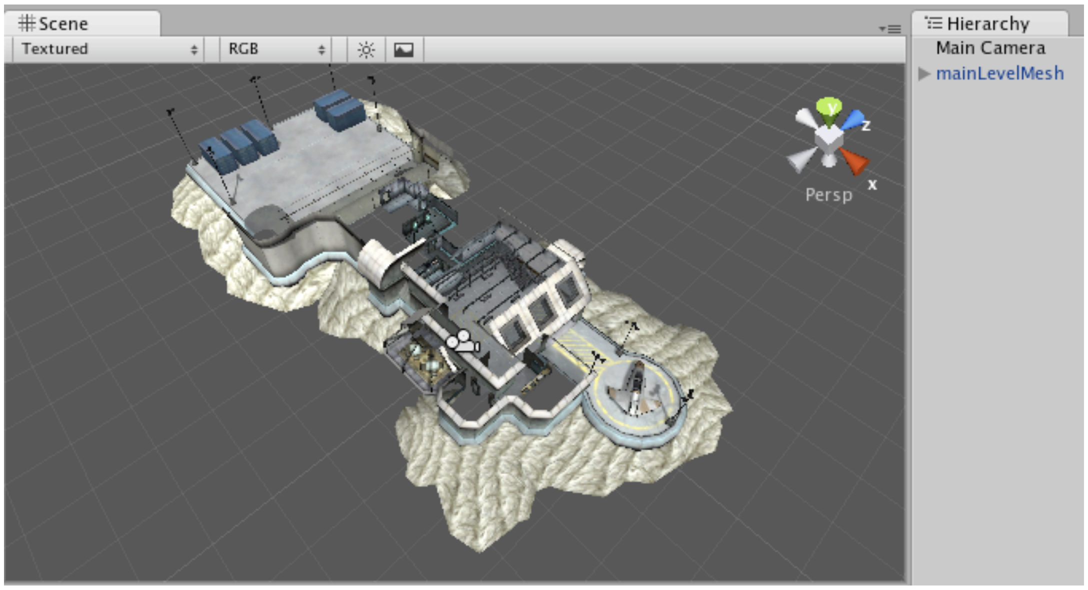
Name	Date Modified
Toon Shading.unityPackage	6/30/09
Standard Assets.unityPackage	6/30/09
Pro Standard Assets.unityPackage	6/30/09

After importing, you will see Unity’s built in "Standard Assets" in the Project Panel. When we create new assets, it’s best to put them in folders that group them according to their function, e.g. Rocket, Explosion, Audio , etc.

Setting up the game environment

Once the assets have been imported, you'll notice there are many folders in the Project panel.

- In the Projects panel, Select the mainLevelMesh from *Objects/mainLevelMesh/mainLevelMesh*.
- In the Inspector panel, inside of the FBXImporter options you'll find the "Generate Colliders" option. Toggle that option on. If we don't do this, the player will simply fall through the level (no collision).
- Drag the mainLevelMesh into the scene.



There is no need to add a light to the scene, the level is already fully lightmapped. The imported level uses lightmaps for all lighting which allows us to use pre-baked shadows. Lightmaps are very good for performance, especially if you want to create a complex lighting setup.

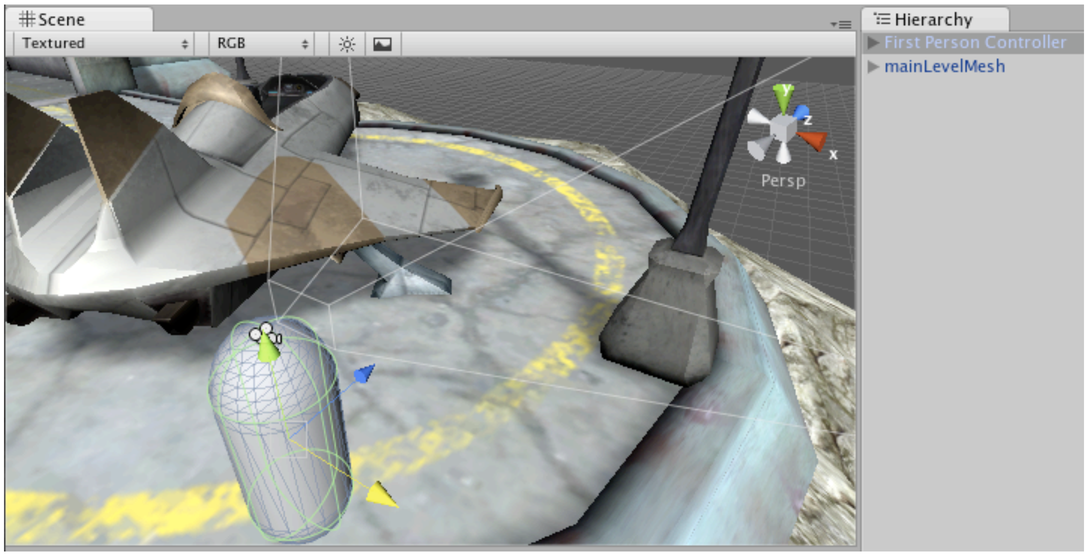
You're now ready to add a character into the environment.

Adding the Main Character

We're now going to add in a character for the player to control. Unity has a built in prefab specifically for a first-person controller. This can be found in the Project panel under Standard Assets->Prefabs.

- To add the First Person Controller, click on the arrow beside Standard Assets in the Project panel, as list of assets will appear. Find the folder called Prefabs and click on the arrow in the left hand side. You should now see the First person controller asset. Drag this into the Scene view.

- You should see a cylinder object representing the player, 3 large arrows for altering the location in 3D space for the object (if you do not see 3 arrows then press the 'W' key), and a white mesh which shows the object's viewport (where it is currently looking). The FPS Controller is now the default camera, by moving this object, you change the current view in the Game View. You'll also notice that the FPS Controller has a camera icon on top of it. Move the character so that it is above ground level in the environment.
- As we no longer have any need for the Main Camera, you can delete it.
- Press Play, you should now be able to move around the level by using the mouse and keyboard (cursor keys to move or W,A,S,D).



You've now created a very simple FPS, let's give the player a weapon.

Adding a weapon

We're now going to give the player a grenade type object to throw in the environment. To do this, you'll need to create some Javascript to tell Unity about the behavior of the weapon.

So what do we want to do? We want to allow the player to shoot wherever the camera is pointing. However, let's first think about our game character and their weapons. Our game character is seen through the first person view, with the camera positioned at eye level. If the player fires a weapons, the weapon should be launched from wherever their hands are, not from eye level. This means we have to add in a game object to represent the grenade launcher, and place it where the player's hand would be when they hold this weapons. This ensures that the object fires from the correct location.

Create the weapon launcher

First, let's add in a game object to represent the grenade launcher. A game object is any item in the 3D world (player, level, sounds), components are the properties that the game object has. Therefore you apply components to game objects.

- From the main menu select *GameObject* -> *Create Empty*, and rename the object to "Launcher" in the Hierarchy panel. Note, this object is invisible as it is and empty object, however it is just a placeholder for our missile launcher.

Now let's get in close to our FPS controller so we can see where to position the weapon launcher.

- Select the FPS controller in the Hierarchy Panel and ensuring that your cursor is over the Scene View, press 'F' (frame selected). This focuses us on the player (the currently selected item).

- Now select the Launcher from the Hierarchy Panel and choose *Game Object* -> *Move to view* from the main menu. Notice how the Launcher object is now placed near the FPS Controller. We can now use the handles to place the launcher approximately where the hands should be.

NOTE *You can make the character left or right-handed by altering the location of this object, no need to change any code.*

- Make sure your Unity window layout is in 2 by 3 mode (*Window* -> *Layouts* -> *2 by 3*), and press the Play button. Make sure Launcher is selected in the Hierarchy View and whilst watching the Scene View, move the character around. You'll notice that our launcher object does not move with our character (you can press Play to stop the game running now).

- To solve this problem drag the Launcher object onto the Main Camera object that belongs to the FPS Controller in the Hierarchy panel. It is okay to break the Prefab connection. Now run the game again and watch the Scene View, the Launcher should now move with our character. We have associated the Launcher game object with an object that moves in all 3 axes (the camera).

Create the Missile object

Let's create the missile that will be launched when the user clicks the fire button.

- For now we'll use something simple, a sphere. Create a new prefab object by clicking on *Assets* -> *Create* -> *Prefab* from the Unity menu bar, and rename it to Missile.

- Now create a sphere (*GameObject* -> *Create Other* -> *Sphere*).

- Now drag the Sphere game object from the Hierarchy Panel onto the Missile prefab in the Project Panel. The icon for the prefab will now change. You can delete the Sphere object from the Hierarchy View.

TIP *Any game object which you know you'll need to instantiate at run-time should generally be a prefab.*

Write the Missile Launcher code

The FPS Controller is a prefab consisting of several game object and components. The FPS Controller itself is a cylindrical mesh that only rotates around the y-axis, so if we attach the launcher code (script) to this, then we won't be able to shoot up and down. Therefore, we're going to attach our launcher code to the Main Camera inside the FPS Controller as the camera can look in any direction.

Let's create our first Javascript code which will describe the behaviour of the Launcher.

- Select *Assets* -> *Create* -> *JavaScript*, which creates a blank Javascript document. A new asset appears in the Project Panel entitled NewBehaviourScript. Rename this script to MissileLauncher.

TIP *You can specify which external code editor you want Unity to use by clicking on Unity -> Preferences then selecting the External Script Editor box.*

- Create a new directory inside your Project View called WeaponScripts, we'll put all our weapons scripts in here. Move your MissileLauncher javascript into here, and move the Missile prefab also.

Let's have a look at the complete Javascript for MissileLauncher. You can type in the code sections as you follow.

Thinking from a high level, what do we want to do? Well, we want to detect when the user has hit the fire button, then instantiate a new missile, and launch it with a certain velocity towards where the user was aiming. Let's examine the code in detail:

```
var projectile : Rigidbody;
var speed = 20;

function Update()
{
```

This is the beginning of the script, which defines to properties and declares a function called `Update`.

```
    if( Input.GetButtonDown( "Fire1" ) )
```

First we want to check when the user hits the fire button. "Fire1" is mapped to both the left mouse button and the current keyboard configuration (this can be edited from menu *Editor -> Project Settings -> Input*).

```
{
    var instantiatedProjectile : Rigidbody = Instantiate(
        projectile, transform.position, transform.rotation );
```

We declare a variable to hold a reference to the instantiated object. The type of this variable is Rigidbody, this is because the missile should have physics behavior.

To instantiate an object in Unity we use the `Instantiate` function with three parameters, the object to instantiate, the 3D position it should be created at, and the rotation the object should have. There is also another constructor, check the API guide, but we'll use this one for now.

The first parameter, projectile, is the object we want to create. So what is projectile anyway? The projectile is set by you inside the editor. To allow this to happen, we declare the variable **projectile** outside of any function, which makes it public and exposes it to the Unity editor. The projectile object could have been created in code, however if you want to modify (tweak) a variable, it's better to do this from within the Unity editor.

The second parameter, **transform.position**, creates the projectile at the same position in 3D space as the launcher. Why the launcher? Well, the script we're creating will be attached to the launcher so if we want to get our current 3D position, **transform.position** gives that to us (the **transform** keyword accesses the Transform of the Game Object where the script is attached).

The third parameter, **transform.rotation**, is similar to the second parameter, except it creates the missile with the same rotation properties as the launcher.

```
    instantiatedProjectile.velocity =
        transform.TransformDirection( Vector3( 0, 0, speed ) );
```

The next part of the code makes our missile move. To do this, we give the missile a velocity, but in which direction (x, y, or z)? In the Scene View, click on the FPS Controller, the move arrows appear (press 'W' if the arrows do not appear), one is red, one is green, and one is blue. Red denotes the x-axis, green denotes the y-axis, and blue denotes the z-axis. As blue is pointing in the direction that the player is facing, we want to give our missile a velocity in the z-axis.

Velocity is a property of **instantiatedProjectile**. How did we know this? Well, **instantiatedProjectile** is of type Rigidbody, and if we look at the API we see that velocity is a property. Take a look at some of the other properties that Rigidbody has, too. To set the velocity we need to specify the speed in each axis. However, there is one slight issue. Objects in 3D are specified using two coordinate models; local and world. In local space, coordinates are relative to the object you are working with. In world space, coordinates are absolute: up, for example, is always the same direction for all objects. **Rigidbody.velocity** must be specified in world space. So, when assigning the velocity you need to convert from the z-axis in local space (the forward direction) to its respective world space direction. You do this using the `transform.TransformDirection` function which takes a Vector3 as an argument. The variable **speed** is also exposed, so we can tweak it in the editor.

```
        Physics.IgnoreCollision( instantiatedProjectile. collider,
            transform.root.collider );
    }
}
```

Finally, we turn off collisions between the missile and the player. If we didn't do this the missile would collide with the player when it was instantiated. Look this up in the API under `IgnoreCollision`.

Here is the full code to *MissileLauncher.js*:

```
var projectile : Rigidbody;
var speed = 20;

function Update()
{
    if( Input.GetButtonDown( "Fire1" ) )
    {
        var instantiatedProjectile : Rigidbody = Instantiate(
            projectile, transform.position, transform.rotation );

        instantiatedProjectile.velocity =
            transform.TransformDirection( Vector3( 0, 0, speed ) );

        Physics.IgnoreCollision( instantiatedProjectile. collider,
            transform.root.collider );
    }
}
```

Insert the described code into MissileLauncher Javascript and save it.

- Attach the MissileLauncher script to the Launcher in the FPS Controller. You can confirm that the MissileLauncher script is now attached to the Launcher by selecting Launcher it in the Hierarchy View and checking that the MissileLauncher script is shown in the Inspector panel.

The missile prefab we previously created has not been associated with the **projectile** variable in our script, so we must do this inside the editor. The **projectile** variable is of type Rigidbody, so first we must ensure that our missile has a Rigidbody attached.

- Click on the Missile in the Project panel, then from the main menu bar select *Components* -> *Physics* -> *Rigidbody*. This adds a Rigidbody component to the missile we want to fire. We have to do this as the type of object we want to fire must match the type of the variable we exposed in the script.
- Associate the missile with the projectile variable in the script. First, click on Launcher in the Hierarchy panel, notice the variable Projectile in the Missile-Launcher script section in the Inspector panel. Now drag the Missile prefab from the Project panel and drop it onto the **Projectile** variable.
- Now if you run the game you will be able to shoot out a small sphere which will be affected by gravity.

Missile explosions

Now we're going to add an explosion when the missile collides with another object. To do this we need to attach behavior to the missile, i.e., create a new script and add it to the Missile game object.

- Create a new blank script by selecting *Assets* -> *Create* -> *Javascript* and rename this new script to Projectile. Drag the new script to the WeaponScripts folder in the Project panel.

So what do we want the Projectile to do? We want to detect when the missile has had a collision and create an explosion at that point. Let's look at the code:

```
var explosion : GameObject;

function OnCollisionEnter( collision : Collision )
{
```

Any code that we put inside the `OnCollisionEnter` function is executed whenever the object that the script is attached is in collision with another object.

```
    var contact : ContactPoint = collision.contacts[0];
```

The main task we want to do inside the `OnCollisionEnter` function is to instantiate a new explosion at the location in 3D space wherever the missile collided. So, where did the collision happen? The function `OnCollisionEnter` is passed a Collision class which contains this information. The point at which the collision occurred is stored in the **ContactPoint** variable.

```
    var rotation = Quaternion.FromToRotation( Vector3.up, contact.normal );
    var instantiatedExplosion : GameObject = Instantiate(
        explosion, contact.point, rotation );
```

Here we use the `Instantiate` function to create an explosion. We know that instantiate takes 3 parameters; (1) the object to instantiate, (2) the 3D position, and (3) the rotation of the object.

We will assign this to a game object with a particle system later. We'll also want to assign this using the editor, so we'll expose this variable by making it public (declaring it outside of any function).

The second parameter, the 3D point at which to `Instantiate` the explosion was determined from the collision class in point 2.

The third parameter, the rotation to set the explosion to, needs some explanation. We want the rotation of the explosion so that the y-axis of the explosion follows along the surface normal of the surface the missile collided with. This means that for walls the explosion should face outwards, for floors it will face upwards. So what we want to do is translate the y-axis of the explosion in local space (the y-axis is upwards), to the surface normal (in world space) of the object that was collided with. Essentially the rotation is saying 'make the y-axis of the object point in the same direction as the normal of the surface with which it collided'.

```
    Destroy( gameObject );
}
```

Finally, we want to make the missile disappear from the game as it has now collided, which is done by calling the `Destroy()` function with **gameObject** as a parameter (**gameObject** denotes the object the script is associated with).

Here is the full code to *Projectile.js*:

```
var explosion : GameObject;
```

```
function OnCollisionEnter( collision : Collision )
{
    var contact : ContactPoint = collision.contacts[0];

    var rotation = Quaternion.FromToRotation( Vector3.up, contact.normal );
    var instantiatedExplosion : GameObject = Instantiate(
        explosion, contact.point, rotation );

    Destroy( gameObject );
}
```

- Add the code into the Projectile Javascript and save it.
- Attach the Projectile Javascript to the Missile prefab by selecting *Component -> Scripts -> Projectile*.

Now we have to create the explosion that we want to occur whenever the missile collides.

- First, create a new Prefab (call it Explosion) that will store the explosion asset.
- The standard assets contain a nice explosion prefab with a particle system and a light around the explosion. Drag the explosion prefab under *Standard Assets/Particles/explosion* into the Hierarchy View.
- Adjust the settings on the explosion until you're happy with how it looks, then drag the Explosion from the Hierarchy view onto the Explosion Prefab in the Project panel.

Now we can assign the explosion to the missile:

- Ensuring that Missile prefab is selected, fill in the Explosion variable by dragging the Explosion object in the Project panel onto the Missile's **Explosion** variable in the Inspector panel.

Defining Explosion Behavior

Now we have to create one more script which will define the behavior of the explosion itself.

- Create a new script, call it Explosion, and put it in the Weapons folder. Double click on the Explosion script to edit it.

Another common function type for scripts is called `Start()`. Code placed inside here is executed only once when the object it belongs to is instantiated. All we want to do for now is to delete the explosion from the game after a certain amount of time. To do this we use the `Destroy()` function with a second parameter which specifies how long to wait before deleting.

```
var explosionTime = 1.0;

function Start()
{
    Destroy( gameObject, explosionTime );
}
```

The **explosionTime** variable is exposed to the Unity editor so it can be easily tweaked.

- Insert the above code into the Explosion script, deleting the `Update()` function in the process.
- Attach the Explosion script to the Explosion prefab by first clicking on the Explosion prefab and then selecting *Components* -> *Scripts* -> *Explosion*.

Notice how we've tried to group the behavior as neatly as possibly, code that relates to the explosion is contained within the script that is attached to the Explosion prefab.

- Run the game and wherever your missile collides with the environment, you should see the explosion particle system for a period of time.

Sound effects

Our game world has been a little quiet so far, let's add a sound effect to our explosion effect.

First let's assign an audio clip to the Explosion prefab.

- To allow the Explosion object to be able to accept an audio clip, we add the Audio Source component to it from the main menu (*Component* -> *Audio*-> *Audio Source*). You'll notice one of the properties of this component is the **Audio Clip**.
- Assign "RocketLauncherImpact" to the Explosion prefab's **Audio Clip** exposed variable. Unity can play many different audio formats.
- Run the game again and our explosion sound effect should be heard when each missile impact.

Adding a GUI

Now let's add a graphical user interface (GUI), more commonly known as a heads up display (HUD) to the game. Our simple GUI is only going to contain a crosshair.

Adding a crosshair

- Create a directory called GUI in your project view.
- Create a new script, call it Crosshair and drag it to the GUI folder.

Enter the following code into *Crosshair.js*:

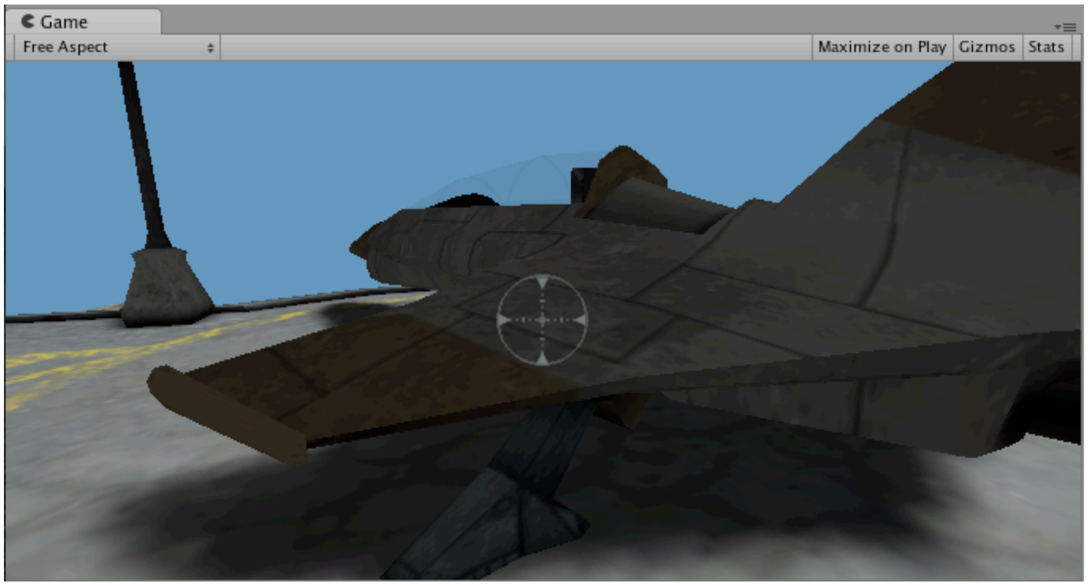
```
var crosshairTexture : Texture2D;
var position : Rect;

function Start()
{
    position = Rect( ( Screen.width - crosshairTexture.width ) / 2, ( Screen.height -
        crosshairTexture.height ) / 2, crosshairTexture.width, crosshairTexture.height );
}

function OnGUI()
{
    GUI.DrawTexture( position, crosshairTexture );
}
```

First we create two variables. The first variable is where we're going to put the image that we're going to use for our texture. The second variable is the rectangle which will store where to position the texture on the screen.

Inside of the Start() function we calculate where the texture will be drawn. Inside of the constructor we have 4 parameters which determine where the rectangle is located and how large it is. The first parameter tells us where the left side of the rectangle is. The second parameter sets the bottom of the rectangle. The third and fourth parameters determine the width and height of the rectangle.

Inside OnGUI() we use the functionality of the GUI class to draw items on screen. In this case we call DrawTexture() and pass it the position and crosshairTexture. This function will then draw our crosshair in the center of the screen.

- Save the script.
- Create a new empty game object and rename it "GUI".
- Drag your Crosshair script onto the GUI game object.
- Select the GUI game object and then drag the aim texture found at *Textures/aim* from the Project view to the **Crosshair Texture** variable in the Inspector
- Press play and there should be a crosshair in the center of the game screen



Physics

Now we want objects in our environment to behave as naturally as possible, this is achieved by adding physics. In this section we're going to add objects to the environment which will move when hit with our missile. First there are some new terms to explain.

Update

Previously, we have entered code inside an `Update` function, so that we can execute that code every frame. One example was to detect if the user had hit the fire button. However, frame rate is not a regular value as it is dependent on the complexity of the scene etc. This irregular timing between frames can lead to unstable physics. Therefore, whenever you want to update objects in your game which involve physics (rigidbodies), code should be placed inside a `FixedUpdate` function. The **deltaTime** value in Unity is used to determine the time taken between rendering two consecutive frames.

A general distinction between `Update` and `FixedUpdate` functions is as follows:

- `Update()` - code inside the function typically updates player behavior, game logic, etc. The value of **deltaTime** is not consistent when used within this function.

- `FixedUpdate()` - code inside this function typically updates rigidbodies over time (physics behavior). The **deltaTime** always returns the same value when called within a `FixedUpdate` function.

The frequency with which `FixedUpdate` is called is specified by the **Fixed Timestep** property in menu option *Edit* -> *Project Settings* -> *Time* and can also be altered. This property contains the time in seconds to get the frames per second value and take the reciprocal of this value.

TIP *When modifying the value for* **Fixed Timestep** *(effectively frames per second), be careful to strike a balance; a smaller timestep will give more stable physics (more accuracy), however the game performance will suffer. Ensure that the game runs quickly and the physics seem realistic.*

The final term to explain in this section is `yield`. This can be thought of as a statement to pause execution of the current function.

So let's get back to our game, what do we want to do:

- Allow the user to fire a missile (already done).
- If the missile collides with another rigidbody, determine if there are any other objects in the vicinity which have rigidbodies attached.
- For each rigidbody within the force of the explosion, give them a force in the upwards direction, causing them to react to the missile.

Let's look at the code for the updated Explosion Javascript:

```
var explosionTime = 1.0;
var explosionRadius = 5.0;
var explosionPower = 2000.0;

function Start()
{
    //Destroy the explosion in x seconds,
    //this will give the particle system and audio enough time to finish playing
    Destroy( gameObject, explosionTime );

    //Find all nearby colliders
    var colliders : Collider[] = Physics.OverlapSphere( transform.position,
        explosionRadius );
```

First we determine if there are any objects with colliders close to where the missile landed. The class function `Physics.OverlapSphere()` takes two parameters: a 3D position and radius and returns an array of colliders which are contained within the defined sphere.

```
    //Apply a force to all surrounding rigid bodies.
    for( var hit in colliders )
    {
```

Once the array has been obtained, each of the rigidbodies attached to the colliders can be given a force in a certain direction.

```
        if( hit.rigidbody )
        {
            hit.rigidbody.AddExplosionForce( explosionPower,
                transform.position, explosionRadius );
        }
    }
```

We then add a force (**explosionPower**) in the upwards direction (y-axis), at the location where the missile landed (**transform.position**). However, as the effect of explosions diminish over distance, the force should not be constant over the entire radius. Rigidbodies closer to the edge of the circumference should have a lesser effect applied than objects at the epicenter of the explosion. This effect is taken into consideration by this function. The exposed variables **explosionPower** and **explosionRadius** can be easily tweaked to create a suitable effect.

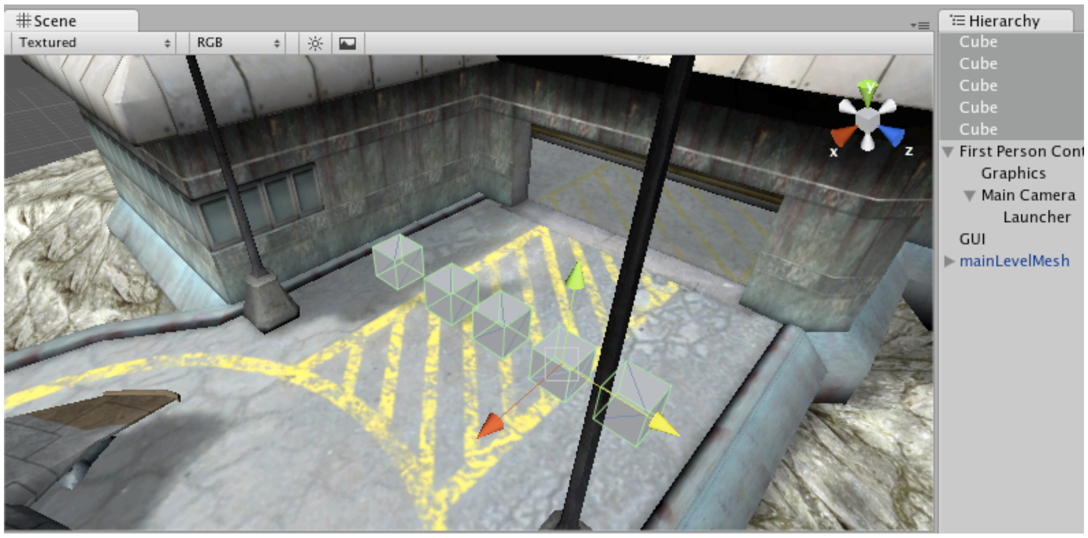
```
    //If we have a particle emitter attached, emit particles for .5 seconds
    if( particleEmitter )
    {
        particleEmitter.emit = true;
        yield WaitForSeconds( 0.5 );
        particleEmitter.emit = false;
    }
}
```

If a particle emitter is attached to our Explosion prefab then start emitting particles, wait for 0.5 seconds (`yield`), and then stop it emitting.

Here is the full code to *Explosion.js*:

```
var explosionTime = 1.0;
var explosionRadius = 5.0;
var explosionPower = 2000.0;

function Start()
{
    //Destroy the explosion in x seconds,
    //this will give the particle system and audio enough time to finish playing
    Destroy( gameObject, explosionTime );
    //Find all nearby colliders
    var colliders : Collider[] = Physics.OverlapSphere( transform.position,
        explosionRadius );

    //Apply a force to all surrounding rigid bodies.
    for( var hit in colliders )
    {
        if( hit.rigidbody )
        {
            hit.rigidbody.AddExplosionForce( explosionPower,
                transform.position, explosionRadius );
        }
    }

    //If we have a particle emitter attached, emit particles for .5 seconds
    if( particleEmitter )
    {
        particleEmitter.emit = true;
        yield WaitForSeconds( 0.5 );
        particleEmitter.emit = false;
    }
}
```

- Update the Explosion Javascript with the new code and save the script.
- Let's create objects to shoot. Create a cube in the scene and attach a rigidbody. Duplicate the cube several times and scatter them around.



- Try the game again and shoot objects which have rigidbodies attached. They should now react with an upwards force. Also, try to not shoot the objects directly, but close to them. This will clearly demonstrate how the upwards explosion force is applied over a radius and not only to each objet directly.

Save your scene.

Acknowledgments

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

The next tutorial in the series will show you how to build upon these fundamentals by introducing more advanced concepts such as AI and multiple weapons.