CHAPTER 7

Three.js Framework

There are many WebGL frameworks that are available to abstract away the lower-level application programming interface (API) calls that we have covered in the first six chapters of the book. This abstraction helps to make WebGL development easier and more productive. We will discuss several WebGL frameworks in the next chapter. In this chapter we will concentrate on one of the most widely used frameworks – Three.js. We will cover the following:

- A background of the library
- How to start development with Three.js
- Falling back to a 2D canvas context for rendering if WebGL is not supported
- Three.js API calls to easily create cameras, objects, and use lighting models
- Show the equivalent Three.js code to some examples found in previous chapters, which used direct low-level WebGL API calls
- Introduce tQuery, a library that blends Three.js with jQuery selectors

Background

Three.js was created by Ricardo Cabello, aka Mr.Doob, and has been on gitHub since 2010. Since that time, it has received added help from many contributors and its user base has grown to a large size.

Three.js provides several different draw modes and can fall back to the 2D rendering context if WebGL is not supported. Three.js is a well-designed library and fairly intuitive to use. Default settings reduce the amount of initial work or "boilerplate" needed. Settings can be overridden as parameters passed in upon object construction or by calling the appropriate object methods afterwards.

Note There can be a mistaken notion among people starting out with WebGL that Three.js and WebGL development are one and the same. Just as the JavaScript framework, jQuery, is not the same as JavaScript, Three.js (or any other framework) is not the same as pure WebGL development.

If you are adept with an underlying language, you can usually understand framework code for it. The reverse is not true. Knowing a framework in no way guarantees that you know a language, so learning the low-level language is highly beneficial.

B. Danchilla, *Beginning WebGL for HTML5*

Features

Here are some of the many features of the Three.js framework:

- Gracefully falls back to 2D context when WebGL is not supported
- Built-in vector and matrix operators
- API wrapper implementation of cameras, lights, materials, shaders, objects, and common geometries
- Import and export utilities
- Good documentation and examples

Setup

We will now go over how to obtain the Three.js library code, its directory structure, and core objects.

Obtaining the Library

The Three.js project is hosted on github at `https://github.com/mrdoob/three.js`. The latest release can be downloaded from `https://github.com/mrdoob/three.js/downloads`. Or if you are familiar with git, you can clone the repository:

```
git clone https://github.com/mrdoob/three.js.git.
```

The library is under active development, and changes to the API are not uncommon. The latest complete API documentation can be found at the URL `mrdoob.github.com/three.js/docs/latest/`, which will redirect to the current version. There is a wiki page at `https://github.com/mrdoob/three.js/wiki/`, and there is no shortage of demos that use Three.js or articles about Three.js development on the Web. Some of the better articles are listed in Appendix D.

Directory Structure

Once you download or clone the repository, you can place the files within your active development folder. The directory structure shows the following folder layout:

```
/build     compressed versions of the source files
/docs      API documentation
/examples  examples
/gui       a drag-and-drop GUI builder that exports Three.js source
/src       source code, including the central Three.js file
/utils     utility scripts such as exporters
```

Within the `src` directory, components are split up nicely into the following subfolders:

```
/src
      /cameras      camera objects
      /core         core functionality such as color, vertex, face, vector, matrix, math
                    definitions, and so on
      /extra        utilities, helper methods, built-in effects, functionality, and plugins
      /lights       light objects
      /materials     mesh and particle material objects such as Lambert and Phong
      /objects       physical objects
```

```
/renderers      render mode objects
/scenes         scene graph object and fog functions
/textures       texture object
Three.js        central file
```

Basic Elements

There are several core object types in Three.js (see Table 7-1).

Table 7-1. *Core Objects in Three.js*

Base Object	Description
`THREE.Renderer`	The object that actually renders the scene. Implementations can be `CanvasRenderer`, `DOMRenderer,SVGRenderer`, or `WebGLRenderer`.
`THREE.Scene`	Scene graph that stores the objects and lights contained within a scene.
`THREE.Camera`	Virtual camera; can be `PerspectiveCamera or OrthographicCamera`.
`THREE.Object3D`	Many object types, including `Mesh, Line, Particle, Bone and Sprite`.
`THREE.Light`	Light model. Types can be `AmbientLight, DirectionalLight, PointLight`, or `SpotLight`

Two other notes about the object hierarchy: `THREE.Mesh` objects have an associated `THREE.Geometry` and `THREE.Material objects`, and in turn each `THREE.Geometry` contains `THREE.Vertex` and `THREE.Face` objects.

Basic Usage

Now that we have obtained the Three.js library, we are ready to start using it. We need to include the script, either from local sources, as follows:

```
<script src="./three.js/build/Three.js"></script>
```

Or remotely—from github, for example:

```
<script src="https://raw.github.com/mrdoob/three.js/master/build/Three.js"></script>
```

Hello World!

Using Three.js is very easy compared with the low-level coding that we have done so far. Having learned the base WebGL API calls already, though, we can fully appreciate the speedup of a framework while knowing (or at least presuming to know without actually checking the library code) what is going on underneath the surface Three.js API calls.

In our first example, shown in Figure 7-1, we will render an unlit rectangular cuboid in Three.js.

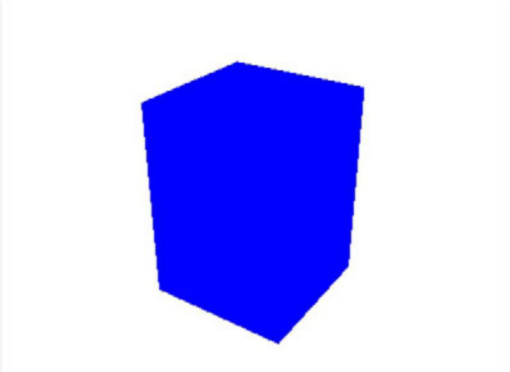

Figure 7-1. *A rectangular cuboid rendered with Three.js. No light makes the cuboid appear flat*

The full code of the example is fairly short compared with pure WebGL (see Listing 7-1). We will go into each section of the code in detail after the listing.

Listing 7-1. Rendering an unlit rectangular cuboid

```
<!doctype html>
<html>
        <head>
                <title>Three.js Cube Test</title>
                <style>
                        body{ background-color: grey; }
                        canvas{ background-color: white; }
                </style>
                <script src=”./Three.js/build/Three.js”></script>
                <script>
                        var     CANVAS_WIDTH = 400,
                                CANVAS_HEIGHT= 300;

                        var     renderer = null,        //WebGL or 2D
                                scene = null,           //scene object
                                camera = null;          //camera object

                        function initWebGL()
                        {
                                setupRenderer();
                                setupScene();
                                setupCamera();

                                renderer.render(scene, camera);
                        }

                        function setupRenderer()
                        {
                                renderer = new THREE.WebGLRenderer();
                                renderer.setSize( CANVAS_WIDTH, CANVAS_HEIGHT );
```

```
                //where to add the canvas element
                document.body.appendChild( renderer.domElement );
        }

        function setupScene()
        {
                scene = new THREE.Scene();
                addMesh();
        }

        function setupCamera()
        {
                camera = new THREE.PerspectiveCamera(
                  35,                                   // Field of view
                  CANVAS_WIDTH / CANVAS_HEIGHT,         // Aspect ratio
                  .1,                                   // Near clip plane
                  10000                                 // Far clip plane
                );
                camera.position.set( -15, 10, 10 );
                camera.lookAt( scene.position );
                scene.add( camera );
        }

        function addMesh()
        {
                var cube = new THREE.Mesh(
                        new THREE.CubeGeometry( 5, 7, 5 ),
                        new THREE.MeshBasicMaterial( { color: 0x0000FF } )
                );
                scene.add(cube);
        }
    </script>
  </head>
  <body onload="initWebGL()"></body>

</html>
```

The code in Listing 7-1 is very straightforward. When scanning the listing, notice that we have not written vertex or fragment shaders or included a <canvas> tag. The shaders have been written for us by the library when the code is rendered and are based on our scene and camera setup. We will show later in the chapter how to specify shaders if needed.

Going through Listing 7-1, the first thing we do is add variables that will be used to set the size of our canvas and hold Three.js `WebGLRenderer`, Scene, and `PerspectiveCamera` objects:

```
var     CANVAS_WIDTH = 400,
        CANVAS_HEIGHT = 300;

var     renderer = null,     //WebGL or 2D
        scene = null,        //scene object
        camera = null;       //camera object
```

Then, as with low-level WebGL, we have an `onload` event. In Listing 7-1, the `onload` event calls the initWebGL function:

```
function initWebGL()
{
        setupRenderer();
        setupScene();
        setupCamera();

        renderer.render(scene, camera);
}
```

The names of the function give hints that we are going to set up a `WebGLRenderer` object, a `Scene` object, and a `Camera` object; and then run the renderer with our scene and camera objects. Each of the setup function calls are small and straightforward, starting with `setupRenderer`:

```
function setupRenderer()
{
        renderer = new THREE.WebGLRenderer();
        renderer.setSize( CANVAS_WIDTH, CANVAS_HEIGHT );

        //where to add the canvas element
        document.body.appendChild( renderer.domElement );
}
```

We choose the `WebGLRenderer` object as our renderer type and create a new instance of it. Then we set the renderer size to our canvas dimensions and attach the `domElement` of the renderer (a <`canvas`> element) to our document <`body`> tag.

Next we call `setupScene`:

```
function setupScene()
{
        scene = new THREE.Scene();
        addMesh();
}
```

We create a new Scene object that will store objects such as meshes and lighting. The `addMesh` function is this:

```
function addMesh()
{
        var cube = new THREE.Mesh(
                new THREE.CubeGeometry( 5, 7, 5 ),
                new THREE.MeshBasicMaterial( { color: 0x0000FF } )
        );
        scene.add(cube);
}
```

In this example, we create a cuboid mesh of dimensions `5x7x5`. We create a `MeshBasicMaterial` object with color property set to blue and do not add any lighting. Cuboid faces are not distinct in the rendering of Figure 7-1 because each face is the same color, and no lighting means that no normal vectors are used. Finally, in the `addMesh` function, we add this mesh to our scene object.

The `setupCamera` method creates and sets up a `PerspectiveCamera` object:

```
function setupCamera()
{
        camera = new THREE.PerspectiveCamera(
                45,                                 // Field of view
                CANVAS_WIDTH / CANVAS_HEIGHT,       // Aspect ratio
```

```
                .1,                                 // Near clip plane
                10000                               // Far clip plane
        );
        camera.position.set( 10, 10, 10 );
        camera.lookAt( scene.position );
        scene.add( camera );
}
```

We position our camera and tell it which direction to look. Then we add the camera object to the scene.

■ **Note** There is an equivalent orthogonal camera API call: THREE.OrthogonalCamera(float left, float right, float top, float bottom, float near, float far). Recall that an orthogonal camera is useful if we want objects with same-sized dimensions to appear the same size regardless of their distances within a scene.

Lastly we have the call:

```
renderer.render(scene, camera);
```

This call will render the scene using the `scene` graph object, which contains all the physical objects in the scene along and with the virtual `camera` object. The `renderer` object takes care of context handling and drawing to the underlying `canvas` element.

Let's examine all the details in Listing 7-1 that have been abstracted:

- No vertex points were specified; just the dimensions of the cuboid.
- The modelview or perspective matrices were not explicitly set. The `PerspectiveCamera` `position` and `lookAt` functions, along with the `scene.position` vector, were used to calculate them and pass along to the shaders for us.
- The shader pair in this example is completely computed for us.
- The `<canvas>` element is automatically added to our document.
- No vertex buffer objects or draw call is made by us. Which is used: `drawArrays` or `drawElements`? We cannot tell without looking at the source code of the library.

These are some nice abstractions for a basic scene to help an absolute beginner get started with three-dimensional animation. For more complex scenes, the amount of abstraction is even greater and can further increase productivity. Having a knowledge of the underlying workings of WebGL as we now do is also great because it allows us to understand the library code to help us troubleshoot when things do not work as expected.

Adding Some Details

We will now look at adjusting color, lighting, and mesh objects with Three.js.

Color

In Three.js, colors are initialized with hex values, which look similar to CSS but are numeric values prefixed with `0x` instead of a hash (#) tag. So pure red would be `0xFF0000`, and we would create a new red `Color` object with:

```
var myColor = new THREE.Color( 0xff0000 );
```

After initialization, color components are converted to RGB values between 0 and 1, and are available as the object properties `r`, `g`, and `b`. If you want to set the color component-wise yourself, you can use the function `setRGB`. To change the color to blue looks like this:

```
myColor.setRGB(0.0, 1.0, 0.0);
```

The Clear Color

To set the clear color in Three.js, we use the renderer method `setClearColor` or `setClearColorHex`:

```
var alpha = 1.0;
renderer.setClearColor(myColor, alpha);
```

Or equivalently:

```
renderer.setClearColorHex(0x00ff00, 1.0);
```

■ **Note** We can also specify the clear color in the WebGLRenderer constructor, along with other options. The default properties are shown here:

```
new THREE.WebGLRenderer({
    antialias: false,
    canvas: document.createElement( 'canvas' ),
    clearColor: 0x000000,
    clearAlpha: 0,
    maxLights: 4,
    stencil: true,
    preserveDrawingBuffer: false
});
```

When setting the `clearColor` in this manner, make sure to also set the `clearAlpha` to a nonzero value, such as this:

```
renderer = new THREE.WebGLRenderer( { clearColor: 0x007700, clearAlpha: 1 } );
```

Lighting

We will now add a light to our scene by adjusting `setupScene` and `addMesh` and adding a new method called `addLight`, which is shown in Listing 7-2. Changes are shown in bold.

Listing 7-2. Adding a light to the scene

```
function setupScene()
{
        scene = new THREE.Scene();
        addMesh();
        addLight();
}
```

```
function addMesh()
{
        var cube = new THREE.Mesh(
                        new THREE.CubeGeometry( 5, 7, 5 ),
                        new THREE.MeshLambertMaterial( { color: 0x0000FF } )
                 );
         scene.add(cube);
}

function addLight()
{
        var light = new THREE.PointLight( 0xFFFFFF );
        light.position.set( 20, 20, 20 );
        scene.add(light);
}
```

The result of our code modifications can be seen in Figure 7-2 and are in the `07/basic_lighting.html` file. In the `addLight` method of Listing 7-2 it takes only three lines of code to add a point light, specify the color and location of the light, and add it to our scene. It only takes changing the type of our `Mesh` material from `MeshBasicMaterial` to `MeshLambertMaterial` to use the Lambert shading model that was discussed in Chapter 4. We still have not needed to adjust the shader code.

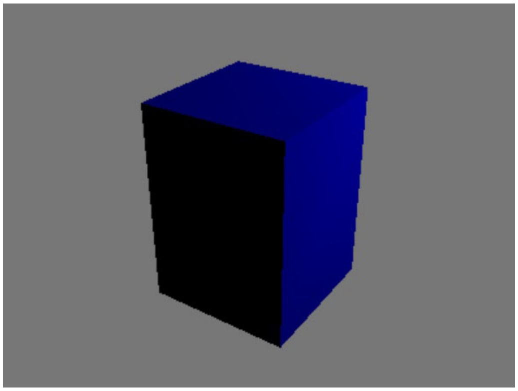

Figure 7-2. *Cuboid with clear color set to gray and a light that makes the 3D shape visible*

So far, we have used only the built-in `CubeGeometry` object. We will now cover the `Geometry` and `Mesh` objects in more detail.

Meshes

The basic `THREE.Mesh` object extends `THREE.Object3D` and stores a `Geometry` object and a `Material` object (among other things):

```
var myMesh=new THREE.Mesh(geometry, material);
```

As shown in Listing 7-1 and Listing 7-2, the material can be a Lambert model and created like this:

```
var material=new THREE.MeshLambertMaterial( { color: 0x0000FF } );
```

Preset geometry objects can be found in the /src/extras/geometries folder similar to the one we have used so far: CubeGeometry. If you look at the TorusGeometry.js source, the function signature is:

```
THREE.TorusGeometry=function ( radius, tube, segmentsR, segmentsT, arc ){ … }
```

To render a torus, we simply change the Geometry object in the addMesh code of Listing 7-2 to this:

```
function addMesh()
{
        var mesh = new THREE.Mesh(
                        new THREE.TorusGeometry( 4, 1.5, 20, 20 ),
                        new THREE.MeshLambertMaterial( { color: 0x0000FF } )
                );
        scene.add(mesh);
}
```

Figure 7-3 shows a torus geometry obtained by switching the Geometry object of a mesh.

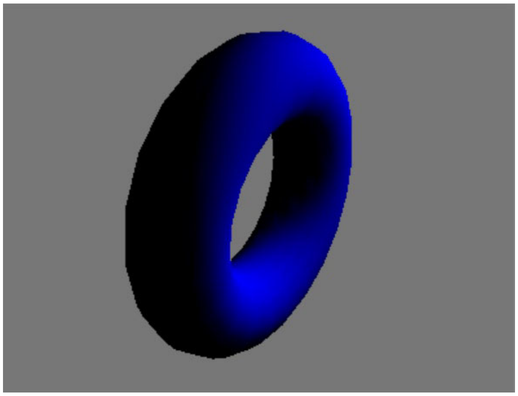

Figure 7-3. *A torus geometry rendered in Three.js*

Having existing geometries available is really nice. You do not need to understand or implement the math involved; someone has already done this for you! Each of these geometries extends the THREE.Geometry object found in /src/core/Geometry.js. In the base Geometry object are many properties such as vertices, colors, and faces along with built-in functionality such as computing normal vectors and bounding boxes, which are useful for collision detection.

Smooth shading is the default shading model, but we can also perform flat shading and show wireframe models very easily, as shown in Figure 7-4. To perform flat shading we adjust the material properties like so:

```
new THREE.MeshLambertMaterial( {
        color: 0x0000FF,
        shading: THREE.FlatShading
} )
```

Similarly to show the wireframe, we adjust the material properties to:

```
new THREE.MeshLambertMaterial( {
        color: 0x0000FF,
        wireframe: true
} )
```

On the left of Figure 7-4 is a flat shaded torus geometry; the wireframe of a torus is displayed on the right.

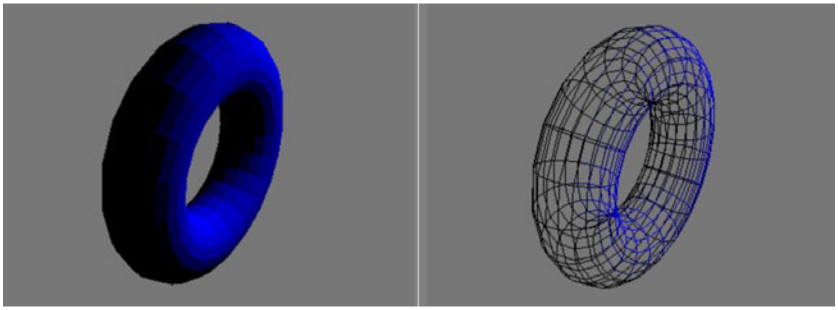

Figure 7-4. *Left: flat shading; right: wireframe*

Three.js caches values for performance improvements. If you change a `Geometry` object's properties, you need to inform Three.js to use the new values as we will now discuss.

Updating Objects

In Three.js, some values are updated automatically when adjusted, such as matrix transforms and cameras. However, for performance some values are not automatically updated. Instead you need to set a flag telling Three.js that the object needs updating.

For a `Geometry` object, update flag properties are `verticesNeedUpdate, elementsNeedUpdate, morphTargetsNeedUpdate, uvsNeedUpdate, normalsNeedUpdate, colorsNeedUpdate`, and `tangentsNeedUpdate`. For instance, you would tell Three.js that the normal vectors have been changed on an object named `geometry` by setting the `normalsNeedUpdate` flag with this:

```
geometry.normalsNeedUpdate;
```

Meshes also need their `dynamic` flag set:

```
geometry.dynamic=true;
```

Other objects such as textures may require flags as well. To update a texture you would set this:

```
texture.needsUpdate=true;
```

Complete details of how to update Three.js objects are available at https://github.com/mrdoob/Three.js/wiki/Updates.

Falling Back to the 2D Canvas Context

One of the really nice things about Three.js is the ability to fall back to the 2D canvas context if WebGL is not supported. We can do this with the new code shown in bold text in Listing 7-3.

Listing 7-3. Testing for WebGL support and falling back to the 2D canvas context if needed

```
function setupRenderer()
{
        var test_canvas = document.createElement('canvas');
        var gl = null;
        try{
```

```
                gl = (  test_canvas.getContext("webgl") ||
                        test_canvas.getContext("experimental-webgl")
                     );
        }catch(e){
        }
        if(gl)
        {
                renderer = new THREE.WebGLRenderer();
                console.log('webgl!');
        }else{
                renderer = new THREE.CanvasRenderer();
                console.log('canvas');
        }
        test_canvas = undefined;
        renderer.setSize( CANVAS_WIDTH, CANVAS_HEIGHT );
        renderer.setClearColorHex(0x777777, 1.0);
        //where to add the canvas element
        document.body.appendChild( renderer.domElement );
}
```

This output of the code in Listing 7-3 run in two browsers, one with and one without WebGL support, is shown in Figure 7-5. The images are not identical, but compared to not rendering anything, this ability to fall back with no code alterations other than that of Listing 7-3 is fantastic! It provides graceful degradation for users who do not have a browser with WebGL capabilities.

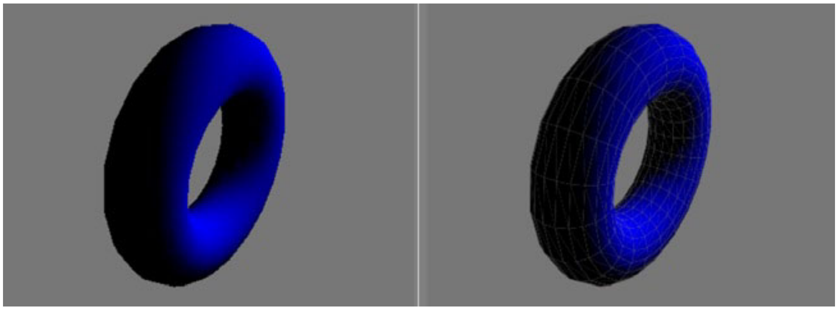

Figure 7-5. *Left: browser supporting WebGL; right: falling back to canvas context*

Shaders

To use shaders in Three.js, set the object material to be of type `ShaderMaterial`, where `vs_source` and `fs_source` are loaded sources from either embedded code or external files:

```
var material=new THREE.ShaderMaterial({
        vertexShader: vs_source,
        fragmentShader: fs_source
});
```

In addition, the constructor takes other optional parameters such as `attributes` and `uniforms`, which we will examine later on in the chapter.

Revisiting Earlier Book Code

We will now reproduce some of the earlier examples of the book using Three.js so that an adequate comparison can be made in terms of using a framework versus lower-level API usage. Along the way, we will uncover new Three.js API functions and configuration parameters, so porting our existing code is a great way to get our feet wet in a new API.

2D Rendering

Remember the “bowtie” two-triangle example of Chapter 1 (Figure 1-4)? Let’s reproduce it with Three.js. At this point, we have used only built-in meshes, but we do not know how to create a custom mesh, even a simple one, with Three.js.

Custom Mesh

To build a custom mesh, we first create a new `Geometry` object. Then we create `Vector3` objects for each vertice and add them to the `Geometry` object’s `vertices` property array. We then add vertice triplets to the `faces` array property of the `Geometry` object. Finally, we add our `Geometry` object to a new `Mesh` object. This is shown in Listing 7-4.

Listing 7-4. Creating a custom mesh with Three.js

```
function addMesh()
{
        var triangleVertices = [
                //left triangle
                -0.5, 0.5, 0.0,
                 0.0, 0.0, 0.0,
                -0.5, -0.5, 0.0,

                //right triangle
                0.5, 0.5, 0.0,
                0.0, 0.0, 0.0,
                0.5, -0.5, 0.0
        ];

        var geometry = new THREE.Geometry();
        for(var i=0; i<triangleVertices.length; i += 3)
        {
                var vertex = new THREE.Vector3(
                                triangleVertices[i],
                                triangleVertices[i + 1],
                                triangleVertices[i + 2]
                                );
                geometry.vertices.push(vertex);
        }
```

```
        geometry.faces.push( new THREE.Face3(0, 1, 2) );
        geometry.faces.push( new THREE.Face3(3, 4, 5) );

        var mesh = new THREE.Mesh(
                geometry,
                new THREE.MeshBasicMaterial( { color: 0xFFFFFF } )
        );

        scene.add(mesh);
}
```

Now when we run the code, we produce the image on the left of Figure 7-6. Only one triangle is rendered. This is because the winding order is opposite in our triangles. To fix this, we have two options. First, we can render both sides of the mesh:

```
mesh.doubleSided = true;
```

However, this is a performance hit and we do not want to get into the habit of doing this. The other option is to fix the winding order of the second face:

```
geometry.faces.push( new THREE.Face3(3, 5, 4) );
```

After this adjustment, we get the image on the right of Figure 7-6. The full code is in the file `07/bowtie.html`. Notice that even though we have specified the vertex data, we are not responsible to bind it to a VBO.

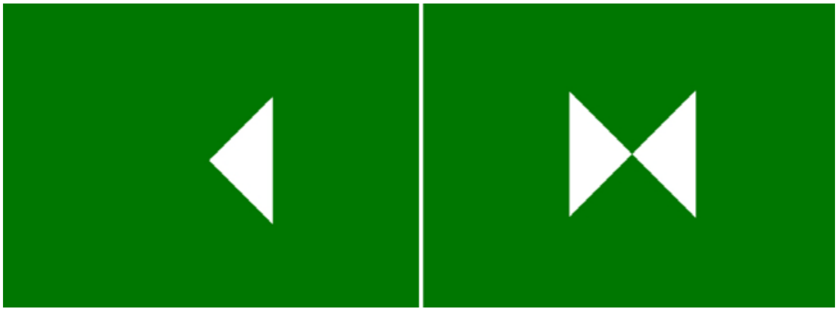

Figure 7-6. *Left: triangle faces with opposite winding order, only one is visible; right: triangle faces with the same winding order*

Separate Vertex Colors

To have separate colors per vertex, we need to assign them to the `geometry.faces[n].vertexColors` attributes and NOT the `geometry.colors` attribute. The `geometry.colors` attribute is used for other objects such as particles, but not for meshes. Instead of setting the color property of our mesh material, we now set the `vertexColors` property:

```
new THREE.MeshBasicMaterial(
{
        vertexColors: THREE.VertexColors
```

■ **Note** If we do not need per vertex coloring, we can also set the color of each face with `geometry.faces[n].color` and using `vertexColors: THREE.FaceColors` in our `Material` setup.

Changing the addMesh code to that in Listing 7-5 will produce the same colored output as in Figure 1-8. A working example can be found in the 07/bowtie_color.html file.

Listing 7-5. Per vertex color values

```
function addMesh()
{
        var triangleVertices = [
                //left triangle
                -0.5, 0.5, 0.0,
                 0.0, 0.0, 0.0,
                -0.5, -0.5, 0.0,

                //right triangle
                0.5, 0.5, 0.0,
                0.0, 0.0, 0.0,
                0.5, -0.5, 0.0
        ];

        var triangleVerticeColors = [
                //left triangle
                 1.0, 0.0, 0.0,
                 1.0, 1.0, 1.0,
                 1.0, 0.0, 0.0,

                //right triangle
                0.0, 0.0, 1.0,
                0.0, 0.0, 1.0,          //these two colors are switched
                1.0, 1.0, 1.0,          //from the chapter 1 example as the
                                        //vertice order is changed here
        ];

        var geometry = new THREE.Geometry();
        var colors = [];

        for(var i=0; i<triangleVertices.length; i += 3)
        {
                var vertex = new THREE.Vector3();
                vertex.set(
                                triangleVertices[i],
                                triangleVertices[i + 1],
                                triangleVertices[i + 2]
                        );
                geometry.vertices.push(vertex);

                var color = new THREE.Color();
                color.setRGB(
                        triangleVerticeColors[i],
                        triangleVerticeColors[i + 1],
                        triangleVerticeColors[i + 2]            );
                colors.push(color);
        }
```

```
        geometry.faces.push( new THREE.Face3(0, 1, 2) );
        geometry.faces.push( new THREE.Face3(3, 5, 4) );

        var f = 0;
        for(var i=0; i < colors.length; i+=3)
        {
                geometry.faces[f].vertexColors.push(colors[i]);
                geometry.faces[f].vertexColors.push(colors[i+1]);
                geometry.faces[f].vertexColors.push(colors[i+2]);
                ++f;
        }

        var mesh = new THREE.Mesh(
                geometry,
                new THREE.MeshBasicMaterial(
                                {
                                 vertexColors: THREE.VertexColors
                                }
                        )
        );
        scene.add(mesh);
}
```

The next component of Chapter 1's bowtie example was adding movement, which we will now cover with Three.js.

Movement

We will now move our two triangles, as we did in the first chapter. We do this a little differently from how we did in Listing 1-9. First, we will make the `geometry`, `mesh`, and `triangleVertices` that were local variables in Listing 7-5 globally available:

```
var      mesh = null,
         geometry = null,
         triangleVertices = [],
         angle = 0;
```

We also have added a variable to keep track of an angle. To animate the scene, we can use the same animation loop using the `renderAnimationFrame` polyfill that we discussed in Chapter 1 and have been using since. However, Three.js includes the polyfill, so we do not need to include an extra file just for it:

```
function initWebGL()
{
        setupRenderer();
        setupScene();
        setupCamera();

        (function animLoop(){
                updateGeometry();
                renderer.render(scene, camera);
                requestAnimationFrame( animLoop );
        })();
}
```

In our `addMesh` method of Listing 7-5, we need to add this line:

```
geometry.dynamic = true;
```

This informs Three.js that properties of the geometry will change. Lastly, we define the `updateGeometry` function, which controls how the vertices change:

```
function updateGeometry()
{
        var x_translation = Math.sin(angle)/2.0;

        for (var i = 0; i < geometry.vertices.length; i++) {
                geometry.vertices[i].x = triangleVertices[i*3] + x_translation;
        }

        angle += 0.01;
        geometry.verticesNeedUpdate = true;
}
```

The preceding code loops through each vertex and adjusts the `x` component to its original value from the `triangleVertices` array plus a translation amount. We will look at a simpler way to move an entire mesh in the next example. To see movement, it is essential that we tell Three.js that the vertices need to be updated with this line:

```
geometry.verticesNeedUpdate = true;
```

The Triangular Prism

Our next code revisits producing the triangular prism shown in Figure 1-16 and found in the file `01/triangular_prism_depth_test.html`. Our array data is the same as in Listing 1-11, and we will not relist it here. The rest of the `addMesh` method for a triangular prism is shown in Listing 7-6.

Listing 7-6. Add mesh function for triangular prism

```
function addMesh()
{

        var triangleVertices,           //same as in Listing 1-11
          triangleVerticeColors,        //same as in Listing 1-11
          triangleVertexIndices;        //same as in Listing 1-11
        …

        var colors = [];

        for(var i=0; i<triangleVertexIndices.length; i += 3)
        {
                var vertex = new THREE.Vector3();
                var color = new THREE.Color();
                vertex.set(
                        triangleVertices[i],
                        triangleVertices[i + 1],
                        triangleVertices[i + 2]
                );
                geometry.vertices.push(vertex);

                color.setRGB(
                        triangleVerticeColors[i],
```

```
                triangleVerticeColors[i + 1],
                triangleVerticeColors[i + 2]
        );
        colors.push(color);
    }
    for(var i=0; i<triangleVertexIndices.length; i += 3)
    {
        geometry.faces.push( new THREE.Face3(
                triangleVertexIndices[i],
                triangleVertexIndices[i + 1],
                triangleVertexIndices[i + 2]
        ) );
    }

    var f = 0;
    for(var i=0; i<triangleVertexIndices.length; i +=3 )
    {
        geometry.faces[f].vertexColors.push(colors[triangleVertexIndices[i]]);
        geometry.faces[f].vertexColors.push(colors[triangleVertexIndices[i + 1]]);
        geometry.faces[f].vertexColors.push(colors[triangleVertexIndices[i + 2]]);
        ++f;
    }

    geometry.dynamic = true;

    mesh = new THREE.Mesh(
        geometry,
        new THREE.MeshBasicMaterial(
                {
                 vertexColors: THREE.VertexColors
                }
        )
    );
    mesh.doubleSided = true;

    scene.add(mesh);
}
```

The code in Listing 7-6 generates our `vertices`, `faces`, and `vertexColors` properties of our geometry. We also set the mesh to `doubleSided` for this example instead of making the winding consistent. To rotate and translate the mesh, we will act directly on the `Mesh` object instead of each vertice property, as we did in the previous example:

```
function initWebGL()
{
    setupRenderer();
    setupScene();
    setupCamera();

    var original_mesh_x = mesh.position.x;

    (function animLoop(){
        //rotate mesh round y-axis
        mesh.position.x = original_mesh_x + 2.0*Math.cos(angle);
        mesh.rotation.y = angle;
        angle += 0.05;
```

```
                renderer.render(scene, camera);
                requestAnimationFrame( animLoop );
        })();
}
```

The key to this technique is storing the original x position. A working implementation can be found in the `07/triangular_prism.html` file.

Texturing

Our next example will texture the triangular prism as we did in Chapter 3, in the `03/multitexture.html` file. Some of the built-in geometries will automatically calculate default texture coordinates. This is not the case for our custom mesh, but we will now go over how to assign custom coordinates.

First, we load our textures:

```
var     texture = [],
        textureImage = [],
        STONE_TEXTURE = 0,
        WEBGL_LOGO_TEXTURE = 1;
…
setupTexture();
…
function setupTexture()
{
        texture[STONE_TEXTURE] = THREE.ImageUtils.loadTexture(
                                        "textures/stone-128px.jpg");
        texture[WEBGL_LOGO_TEXTURE] = THREE.ImageUtils.loadTexture(
                                        "textures/webgl_logo-512px.png");

        for(var i=0; i<texture.length;++i)
        {
                texture[i].wrapT = texture[i].wrapS = THREE.RepeatWrapping;
                texture[i].needsUpdate = true;
        }
}
```

■ **Note** We need to ensure that `THREE.ImageUtils.loadTexture()` finishes before our scene is rendered. We show a couple approaches to guarantee this later in the chapter.

And now we will set our per vertex texture coordinates, which are stored as an array in the geometry's `faceVertexUvs` property:

```
function addMesh()
{
        …
        var uvs = [];
        for(var i=0; i<triangleVertexIndices.length; i += 3)
        {
                var vertex = new THREE.Vector3();
                var color = new THREE.Color();
                vertex.set(
```

```
                                triangleVertices[i],
                                triangleVertices[i + 1],
                                triangleVertices[i + 2]
                        );
                geometry.vertices.push(vertex);

                var tex = [];
                for(var j=0; j<3;++j)
                {
                        var a = triangleVertexIndices[i+j];
                        var s = null,
                            t = null;

                        if(i >= 24)
                        {
                                s = triangleVertices[a*3 + 1];
                                t = triangleVertices[a*3 + 2];
                        }else{
                                s = triangleVertices[a*3];
                                t = triangleVertices[a*3 + 1];
                        }
                        s = (s+2.0) * .25;
                        t = (t+2.0) * .25;
                        tex.push(new THREE.UV(s, t));
                }
                uvs.push(tex);

                color.setRGB(
                        triangleVerticeColors[i],
                        triangleVerticeColors[i + 1],
                        triangleVerticeColors[i + 2]
                );
                colors.push(color);
        }

        …

        geometry.faceVertexUvs = [];
        for(var z=0;z<uvs.length;z++){
                geometry.faceVertexUvs.push(uvs);
        }

        …

        mesh = new THREE.Mesh(
                geometry,
                new THREE.MeshBasicMaterial(
                                {
                                  map: texture[STONE_TEXTURE]
                                }
                        )
        );

        mesh.doubleSided = true;
        scene.add(mesh);
}
```

The preceding code applies the stone texture, and the example can be run from the `07/triangular_prism_textured.html` file.

How do we use two textures, one as a decal as we did in Chapter 3? To accomplish this, we will have to do our only shader coding with Three.js in this chapter, using the `ShaderMaterial` object.

ShaderMaterial

As mentioned earlier, the `ShaderMaterial` requires vertex and fragment shader sources. We also can provide `uniform` and `attribute` values. Three.js automatically sets many mesh properties such as vertex position and texture coordinates assigned as program attributes from our object properties. In addition, the model view and perspective uniforms are also assigned. This is nice, but may appear a little magical as well.

To decal a texture on top of another texture, as we did in the 03/`multitexture.html` file, we first assign variables for our uniforms and shader material:

```
var      uniforms = null,
         shaderMaterial = null;
```

Next we adjust our addMesh method:

```
function addMesh()
{
        …
        setupShaderMaterial();
        mesh = new THREE.Mesh(
                geometry,
                shaderMaterial
        );

        mesh.doubleSided = true;
        scene.add(mesh);
}
```

The `setupShaderMaterial` method is shown in Listing 7-7. In the method we set our textures as uniform variables. The `type` parameter represents the variable type: texture, int, float, and so on. Then we load our sources with Ajax (again, this could be embedded sources instead) and then create and store a new `ShaderMaterial` object.

Listing 7-7. Using a ShaderMaterial

```
function setupShaderMaterial()
{
        uniforms = {
                uSampler: { type: "t", value: 0, texture: texture[STONE_TEXTURE] },
                uSampler2: { type: "t", value: 1, texture: texture[WEBGL_LOGO_TEXTURE] }
        };

        var     vs_source = null,
                fs_source = null;

        //get shader sources with jQuery Ajax

        $.ajax({
            async: false,
            url: './multitexture.vs',
            success: function (data) {
```

```
                vs_source = data.firstChild.textContent;
            },
            dataType: 'xml'
        });

        $.ajax({
            async: false,
            url: './multitexture.fs',
            success: function (data) {
                fs_source = data.firstChild.textContent;
            },
            dataType: 'xml'
        });

        shaderMaterial = new THREE.ShaderMaterial( {
                uniforms: uniforms,
                vertexShader: vs_source,
                fragmentShader: fs_source
        } );
}
```

We define our shaders, which are different from the ones written in Chapter 3. The shader program pair is shorter now and uses some “magically set” attributes and uniforms in Listing 7-8.

Listing 7-8. A Three.js shader program for two textures

```
<script type="x-shader/x-vertex">
        varying highp vec2 vTextureCoord;

        void main(void) {
                gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
                vTextureCoord = uv;
        }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
        varying highp vec2 vTextureCoord;
        uniform sampler2D uSampler;
        uniform sampler2D uSampler2;

        void main(void) {
                highp vec4 stoneColor = texture2D(uSampler, vec2(vTextureCoord.st));
                highp vec4 webglLogoColor = texture2D(uSampler2, vec2(vTextureCoord.st));
                gl_FragColor = mix(stoneColor, webglLogoColor, webglLogoColor.a);
        }
</script>
```

In Listing 7-8, the `projectionMatrix` and `modelViewMatrix` variables are uniforms passed in from Three.js for our projection and model view transforms. The vertex positions values are passed in as the `position` variable attribute.

■ **Note** It is important to realize that Listing 7-8 is not a valid shader program on its own. These sources are not passed directly to the `shaderSource`, and `compileShader` WebGL methods. Instead, behind the scenes, Three.js checks for set values and inserts attributes and uniforms into the shader source before finalizing the source and compiling it. You can observe this by viewing the source of your browser and demonstrated in Figures 7-7 and 7-8. Then Three.js attaches and links the shader program and selects to use it as we manually do in other book chapters.

Part of the vertex shader produced is shown in Figure 7-7.

```
uniform mat4 viewMatrix;
uniform mat3 normalMatrix;
uniform vec3 cameraPosition;
attribute vec3 position;
attribute vec3 normal;
attribute vec2 uv;
attribute vec2 uv2;
#ifdef USE_COLOR
attribute vec3 color;
#endif
#ifdef USE_MORPHTARGETS
attribute vec3 morphTarget0;
attribute vec3 morphTarget1;
attribute vec3 morphTarget2;
attribute vec3 morphTarget3;
#ifdef USE_MORPHNORMALS
attribute vec3 morphNormal0;
attribute vec3 morphNormal1;
attribute vec3 morphNormal2;
attribute vec3 morphNormal3;
#else
attribute vec3 morphTarget4;
attribute vec3 morphTarget5;
attribute vec3 morphTarget6;
attribute vec3 morphTarget7;
#endif
#endif
#ifdef USE_SKINNING
attribute vec4 skinVertexA;
attribute vec4 skinVertexB;
attribute vec4 skinIndex;
attribute vec4 skinWeight;
#endif

varying highp vec2 vTextureCoord;

void main(void) {
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    vTextureCoord = uv;
}
```

Figure 7-7. *Part of the final vertex shader produced by Three.js from the initial vertex shader in Listing 7-8*

The full fragment shader generated code is shown in Figure 7-8. Compare the source code in these two figures with what we specify in Listing 7-8.

```
precision highp float;
#define MAX_DIR_LIGHTS 0
#define MAX_POINT_LIGHTS 0
#define MAX_SPOT_LIGHTS 0
#define MAX_SHADOWS 0

#define DOUBLE_SIDED

#define SHADOWMAP_SOFT


uniform mat4 viewMatrix;
uniform vec3 cameraPosition;

    varying highp vec2 vTextureCoord;
    uniform sampler2D uSampler;
    uniform sampler2D uSampler2;

    void main(void) {
        highp vec4 stoneColor = texture2D(uSampler, vec2(vTextureCoord.st));
        highp vec4 webglLogoColor = texture2D(uSampler2, vec2(vTextureCoord.st));
        gl_FragColor = mix(stoneColor, webglLogoColor, webglLogoColor.a);
    }
```

Figure 7-8. *Final fragment shader produced by Three.js from initial fragment shader in Listing 7-8*

Finally, we make `setupTexture` the document `onload` event now. In the `setupTexture` function, I have nested callbacks in the `loadTexture` function calls to ensure that the textures are loaded before initializing WebGL:

```
function setupTexture()
{
        texture[STONE_TEXTURE] = THREE.ImageUtils.loadTexture(
                "textures/stone-128px.jpg",
                {}, function() {
                texture[WEBGL_LOGO_TEXTURE] = THREE.ImageUtils.loadTexture(
                        "textures/webgl_logo-512px.png",
                        {}, function() {
                                for(var i=0; i<texture.length;++i)
                                {
                                        texture[i].wrapT = texture[i].wrapS =
                                                THREE.RepeatWrapping;
                                        texture[i].needsUpdate = true;
                                }
                                initWebGL();
                         }
                        );
                }
        );
}
```

Obviously, if we had more than a couple of textures, this approach would be very hard to read, and an alternate code structure would be preferable. We will show an alternate code structure later in the chapter. The full code of this example can be found in the `07/triangular_prism_textured_decal.html` file.

Lighting and Texturing

Our next example will be to re-create the three spheres and plane demonstrated in Chapter 4. We will use Phong lighting, blending, fog, and texturing. In this example, we do not use multiple textures per object and can accomplish everything without explicitly setting our shaders in Three.js. The final result is shown in Figure 7-9.

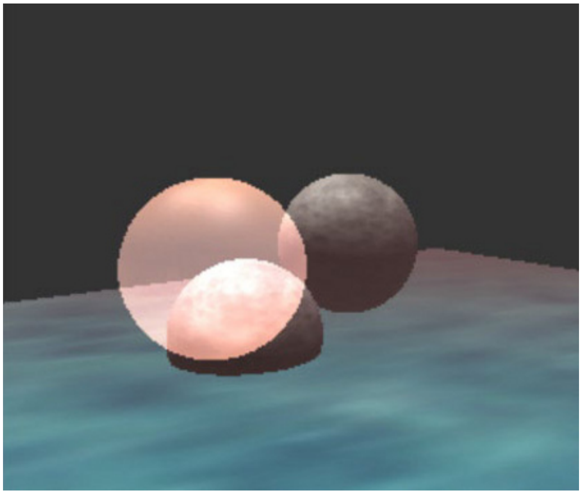

Figure 7-9. *Achieving a similar result to the examples in Chapter 4; this time with Three.js*

Here are the new variables that we will use in this example:

```
var     texture = [],
        STONE_TEXTURE = 0,
        GLASS_TEXTURE = 1,
        WATER_TEXTURE = 2,
        number_textures = 3,
        loaded_textures = 0,
        meshes = [],
        NUM_SPHERES = 3,
        PLANE_INDEX = 3;
```

To load our textures instead of nested callbacks, we now use the code in Listing 7-9. The advantage of it is that it is easier to read and adjust if we add more textures. Each time the callback is called, a global counter of loaded textures is incremented. When the expected number of textures loaded is reached, we call the `initWebGL` method.

Listing 7-9. Callback to adjust our loaded textures

```
function adjustLoadedTexture( tex )
{
        loaded_textures++;

        tex.wrapS = THREE.RepeatWrapping;
        tex.wrapT = THREE.RepeatWrapping;
        tex.needsUpdate = true;

        if( loaded_textures == number_textures )
```

```
        {
                initWebGL();
        }
}

function setupTexture()

{
        var texture_files = [
                "textures/stone-256px.jpg",
                "textures/glass-256px.jpg",
                "textures/water-256px.jpg"
        ];

        loaded_textures = 0;
        for(var i=0; i<texture_files.length;++i)
        {
                texture[i] = THREE.ImageUtils.loadTexture(
                        texture_files[i], {}, adjustLoadedTexture
                );
        }
}
```

■ **Note** The callback automatically passes the object returned from the `loadTexture` call as a parameter in the callback function, `adjustLoadedTexture`. Both of the following alternate function calls will not work:

```
texture[i] = THREE.ImageUtils.loadTexture(
        texture_files[i], {}, adjustLoadedTexture()
);
texture[i] = THREE.ImageUtils.loadTexture(
        texture_files[i], {}, adjustLoadedTexture( texture[i] )
);
```

To add fog to our scene, we do not need to implement this within a shader. We just assign a value to the `scene.fog` parameter by calling the method `THREE.FogExp2`:

```
scene.fog=new THREE.FogExp2( 0x775555, 0.11 );
```

The second parameter is the density of the fog. `FogExp2` is the exponent version of the fog equations that we discussed in Chapter 4. To perform the linear version, we would use `THREE.Fog(`*`color, near, far`*`)`.

Other interesting adjustments that we have made for this example are to change the material used:

```
var material = new THREE.MeshPhongMaterial(
                {       ambient: 0xffffff,
                        color: colors[i],
                        specular: 0x555555,
                        shininess: 30,
                        map: tex
                }
        );
```

In this declaration, tex is a texture object. We specify blending on one of the spheres with this:

```
if(i == 2)
{
        material.blending = THREE.AdditiveBlending;
        material.blendSrc = THREE.SrcAlphaFactor;
        material.blendDst = THREE.OneFactor;
        material.transparent = true;
        material.depthTest = false;
}
```

Lastly, we have added a few more lights:

```
function addLight()
{
        var ambientLight = new THREE.AmbientLight( 0x111111 );
        scene.add(ambientLight);

        var pointLight = new THREE.PointLight( 0xFFFFFF );
        pointLight.position.set( 0, 10, 0 );
        scene.add(pointLight);

        var directionalLight = new THREE.DirectionalLight( 0xFFFFFF );
        directionalLight.position.set( 1, 2, 1 ).normalize();
        scene.add( directionalLight );
}
```

The point and directional light can have attenuation and intensity variations as with the lighting models that we implemented in Chapter 4.

Particle System

For our last example of the chapter, we will produce a particle system with Three.js similar to the one we created in Chapter 6. The result of the code is shown in Figure 7-10.

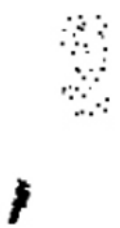

Figure 7-10. *Particle system produced with Three.js*

Creating our particle system is similar to the way we implemented it in Chapter 6, except now we place our particles inside of a Geometry as shown in Listing 7-10. Remember that particles are usually represented as single points, and we can also use a texture image mapped onto each point.

Listing 7-10. Initializing particles

```
function setupParticles()
{
        particleGeometry = new THREE.Geometry(),
        particleMaterial =
                new THREE.ParticleBasicMaterial({
                color: 0xFFFFFF,
                size: (Math.random() + 1.0) * .25
        });

        //fill empty data to capacity
        for( var i=0; i<MAX_NUMBER_OF_PARTICLES; ++i )
        {
                particleGeometry.vertices.push( initializeParticle() );
        }
}

function initializeParticle()
{
        var particle = new THREE.Vector3(
                        .5 * Math.random() - .25,
                        START_Y,
                        3.0);

        //add extra data
        particle.age = 0;
        particle.original = new THREE.Vector3(particle.x, particle.y, particle.z);
        particle.velocity = new THREE.Vector3(
                    5.0 * Math.random() - 10.0,
                    12.0 * Math.random() + 14.0,
                    0.5 + Math.random() * 4.0); //velocity [x,y,z]
        }
        return particle;
}
```

Next we set up a particle system that is basically a wrapper for a mesh and material:

```
//particle system
particleSystem = new THREE.ParticleSystem(
                    particleGeometry,
                    particleMaterial
                );
scene.add(particleSystem);
```

■ **Note** The object `THREE.Particle` also exists, but is used for `CanvasRenderer`, whereas `THREE.ParticleSystem` is used for the `WebGLRenderer`.

Finally, we adjust the particles during each iteration of the render loop, as shown in Listing 7-11.

Listing 7-11. Updating particles in the render loop

```
function adjustParticles(){
        var particles_old = particleGeometry.vertices.slice(); //copy
        particleGeometry.vertices = [];
        for( var i=0; i<particles_old.length; ++i )
        {
                //remove old particles
                //if past lifespan or below the start position, do not readd particle
                if(     ( particles_old[i].age < LIFESPAN ) &&
                        ( particles_old[i].y > (START_Y - 0.001) )
                  )
                {
                        particles_old[i].age += 1.0; //age
                        var pTime = particles_old[i].age/100.0;
                        particles_old[i].x = particles_old[i].original.x
                                        + particles_old[i].velocity.x * pTime;
                        particles_old[i].y = particles_old[i].original.y
                                        + particles_old[i].velocity.y * pTime
                                        - 4.9 * pTime * pTime;
                        particleGeometry.vertices =
                        particleGeometry.vertices.concat(particles_old[i]);
                }
        }

        currentNumberParticles = particleGeometry.vertices.length;

        //spawn new particles
        if( currentNumberParticles + MAX_SPAWN_PER_FRAME < MAX_NUMBER_OF_PARTICLES )
        {
                for( var n=0; n<MAX_SPAWN_PER_FRAME; ++n )
                {
                        var particle = initializeParticle();

                        particleGeometry.vertices.push(particle);
                        ++currentNumberParticles;
                }
        }
        particleGeometry.verticesNeedUpdate = true;
}
```

The working example can be found in the file `07/particle_system.html`.

Advanced Usage

There are many advanced built-in functions and algorithms in the Three.js library and currently more than 150 included examples that demonstrate usage. We cannot cover them in this book, but I encourage you to explore the API, examples, and source code of the library.

Import/Export

Files to import mesh files are available in the `/src/loaders` and `/src/extra/loaders` directories while files to export are in the `/utils/exporters` directory. We will show how to import a mesh which is converted to a JSON format specifically for Three.js in the next chapter.

tQuery

A promising looking project in development is called tQuery, which stands for: **T**hree.js + j**Query**. This library is written by Jerome Eteinne, who also writes the blog http://learningthreejs.com. tQuery is a thin wrapper on top of the Three.js library, which mimics jQuery chainability and can produce scenes with even less boilerplate code to get up and running than using Three.js alone. The project is available on gitHub at https://github.com/jeromeetienne/tquery.

The following code with tQuery produces the cylinder in Figure 7-11:

```
<!doctype html>
<html>
        <head>
                <title>tQuery Cylinder Example</title>
                <script
src="https://raw.github.com/jeromeetienne/tquery/master/build/tquery-all.js">
                </script>
        </head>
        <body>
                <script>
                        var world = tQuery.createWorld().boilerplate().start();
                        var object = tQuery.createCylinder().addTo(world);
                </script>
        </body>

</html>
```

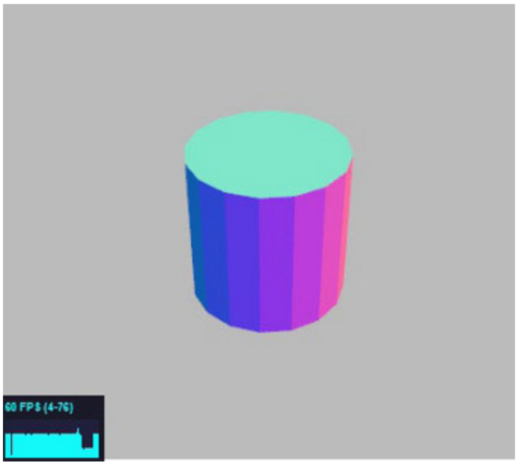

Figure 7-11. *Cylinder modelled with tQuery*

Of course, the previous example above is fairly stock, and the amount of flexibility that tQuery offers to customize meshes and scene details is very important.

Summary

This chapter showed the great power that a framework like Three.js combined with existing WebGL API knowledge can provide and how quickly we can develop code by using one.

In the next chapter, we will survey other WebGL frameworks and physics libraries. We will also show how to find and use existing mesh, shader and texture resources.