3D development with WebGL, Part 2: Code less, do more with WebGL libraries

Tame WebGL with higher-level libraries: Three.js and SceneJS

Sing Li (westmakaha@yahoo.com)
Consultant
Makawave

21 January 2014

The WebGL API gives JavaScript developers the ability to tap directly into the powerful built-in 3D graphics acceleration capabilities of today's PC and mobile-device hardware. Supported transparently in modern browsers, WebGL makes it possible to create high-performance 3D games, applications, and 3D-enhanced UIs for mainstream web users. This article is the second a three-part series for JavaScript developers who are new to WebGL. Series author Sing Li introduces you to two WebGL libraries that make 3D development far more efficient than coding directly to the WebGL API.

View more content in this series

WebGL, built into every modern browser, opens a system's hardware 3D rendering pipeline to direct programmatic access through JavaScript. In Part 1 of this three-part series, you worked through a WebGL example from scratch and learned some fundamental 3D development principles and techniques:

- Obtaining a 3D WebGL context from the HTML5 Document Object Model (DOM) canvas element
- Specifying the triangles that make up a 3D mesh
- Filling the vertices that control the color of each rendered pixel of a 3D object
- Compiling and linking the graphics processing unit (GPU) shader code written in OpenGL Shading Language (GLSL)
- Placing the camera and 3D objects in a 3D scene
- Interpolating vertex color values to create a color gradient
- Working with low-level binary formatted buffers in JavaScript
- Creating transformation matrices to rotate an object

The WebGL API is powerful but low-level. You did a lot of work in Part 1: It took more than 100 lines of raw WebGL code just to animate a single pyramid that rotates around the y-axis.

Significant design, coding, and maintenance effort are necessary for projects that code to raw WebGL.

Thankfully, the past decade of WebGL evolution has brought with it a fleet of easy-to-use higher-level API libraries. In this second part, I'll familiarize you with two popular WebGL libraries:

- Three.js, the de-facto standard Swiss-army-knife library for flexible WebGL development
- SceneJS, a library for construction of complex, intricate 3D scenes

Starting from the pyramid-rotation example, you'll work through successively more complex cases as you learn these libraries' fundamental features. I'll introduce additional 3D development concepts as you work through the examples.

Immediate tenfold productivity improvement

Download this article's sample code and load up pyramid3js.html in Chrome, Firefox, or Safari. Figure 1 shows a snapshot of the page.

Figure 1. Rotating pyramid: Three.js version

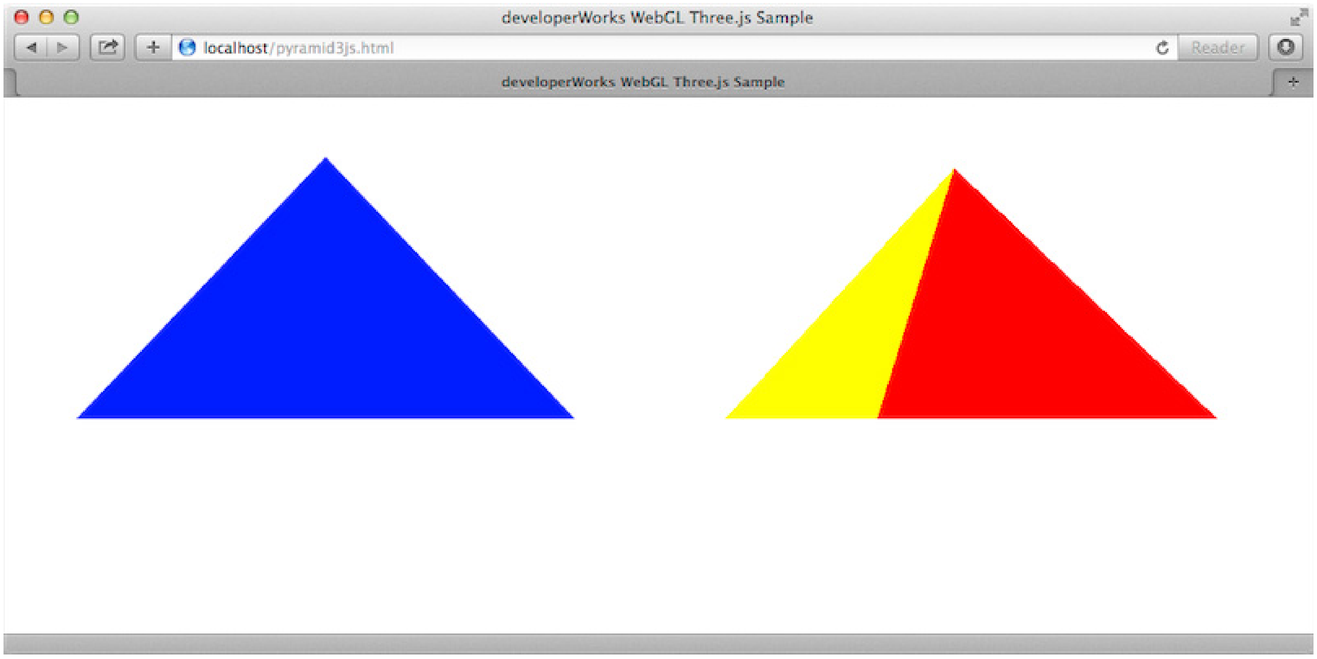

developerWorks WebGL Three.js Sample
localhost/pyramid3js.html
Reader
developerWorks WebGL Three.js Sample

Figure 1 shows the WebGL pyramid.html example that you're familiar with from Part 1, but this time it's coded with Three.js. You can't tell the difference between this version and the raw WebGL version just by looking at the page. But under the hood, the original 100+ lines of WebGL code in `draw3D()` became about 10 lines of Three.js code. To understand how Three.js significantly simplifies a WebGL developer's work, take a look at the source code in Listing 1.

Listing 1. Rotating pyramid with Three.js (pyramid3js.html)

```
<!doctype html>
<html>
<head>
```

```
  <title>developerWorks WebGL Three.js Example</title>
  <script src="Three.js" ></script>
  <script type="text/javascript">
  function draw2D()  {
      var canvas = document.getElementById("shapecanvas");
      var c2dCtx = null;
      var exmsg = "Cannot get 2D context from canvas";
      try {
        c2dCtx = canvas.getContext('2d');
      }
      catch (e)
      {
        exmsg = "Exception thrown: " + e.toString();
      }
      if (!c2dCtx) {
        alert(exmsg);
        throw new Error(exmsg);
      }
      c2dCtx.fillStyle = "#0000ff";
      c2dCtx.beginPath();
      c2dCtx.moveTo(250, 40);
      c2dCtx.lineTo(450, 250);         // Bottom Right
      c2dCtx.lineTo(50, 250);         // Bottom Left
      c2dCtx.closePath();
      c2dCtx.fill();

  }

  function draw3D()  {
    function animate() {
      requestAnimationFrame(animate);
      pyramid.rotateY(Math.PI / 180);
      renderer.render(scene, camera);
    }
    var geo = new THREE.CylinderGeometry(0,2,2,4,1, true);
    var faceColors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00];
    faceColors.forEach( function(color, idx) { geo.faces[2 * idx + 1].color.setHex(color);});
    var pyramid = new THREE.Mesh(geo, new THREE.MeshBasicMaterial({vertexColors: THREE.FaceColors}));
    var camera = new THREE.PerspectiveCamera(45, 1,0.1, 100);
    pyramid.position.y = 1;  camera.position.z = 6;
    var scene = new THREE.Scene();
    scene.add(pyramid);
    var renderer = new THREE.WebGLRenderer();
    renderer.setSize(500,500);
    var span = document.getElementById("shapecanvas2");
    span.appendChild( renderer.domElement );
    animate();
  }

</script>

</head>
<body onload="draw2D();draw3D();">

  <canvas id="shapecanvas" class="front" width="500" height="500"></canvas>
  <span id="shapecanvas2" style="border: none;" width="500" height="500"></span>

  <br/>
  </body>

</html>
```

Three.js renderers

Three.js was a popular JavaScript 3D rendering library long before WebGL's pervasive availability in common browsers. To support browsers that lack WebGL capability, a Three.js renderer (output module) that uses 2D drawing primitives (Canvas 2D Context API) runs on HTML5. This renderer is performance-constrained (because it can't access the 3D hardware) and lacks support for advanced Three.js features.

Rendering 3D to the WebGL canvas context

In Listing 1, notice that the `shapecanvas2` element is now a `<span>` and not a `<canvas>`. The Three.js WebGL renderer takes care of creating the canvas element and rendering to the 3D context of the canvas (see the Three.js renderers sidebar).

In pyramid3js.html, the code in Listing 2 is responsible for associating the `<span>` with the Three.js-created canvas element.

Listing 2. Three.js-created canvas element

```
var renderer = new THREE.WebGLRenderer();
renderer.setSize(500,500);
var span = document.getElementById("shapecanvas2");
span.appendChild( renderer.domElement );
```

Creating a geometry and assigning face colors

As you know from Part 1, to create the pyramid itself, you must fill a low-level buffer with the vertices of the triangles that make up the pyramid. Three.js helps to eliminate a lot of code for that task by including library code that generates the vertices. By generating and filling the vertices buffer under the hood for you, Three.js lets you focus your coding on higher-level concerns.

To greatly simplify the construction and modeling of 3D scenes, Three.js includes library code for generating a large variety of primitive *geometries* typically used in 3D rendering. These geometries include cube, sphere, cylinder, torus, tetrahedron, icosahedron, octahedron, plane, tube, text, and many more. You can set up typical geometries — often generating a mesh with thousands of triangles — in a couple lines of code. In the case of the pyramid, the Three.js geometry to use is actually a cylinder:

```
var geo = new THREE.CylinderGeometry(0,2,2,4,1, true);
```

Materials in Three.js

`THREE.MeshBasicMaterial` doesn't require lighting to render.
`THREE.MeshLambertMaterial` interacts with nonambient lights (directional or spot light) and gives a smooth interpolated (Gouraud shaded) color effect on its surfaces.
`THREE.MeshPhongMaterial` also requires nonambient lights to render; it's used to represent shiny surfaces because it supports specular (extra-bright spot on shiny surface) reflection. The Phong and Lambert materials both support casting shadows.

A smooth cylinder is typically generated by using many triangles to approximate its curvature. This article's pyramid is simply a cylinder that is approximated by only four segments on its sides. (Each segment is typically two triangles.) The top radius is specified to be 0 — creating the pointed end of the pyramid and squeezing each two-triangle segment down to one triangle — and the bottom radius is 2. Height is specified to be 2, and the cylinder is specified to be open-ended (the last argument, `openEnded`, is `true`) to leave out the bottom square, which is never visible.

In pyramid3js.html, these two lines of code specify the colors of the four pyramid faces (red, green, blue, yellow) — replacing the 20 lines of raw binary buffer manipulation code used in Part 1 to specify vertex colors:

```
var faceColors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00];
    faceColors.forEach( function(color, idx) { geo.faces[2 * idx + 1].color.setHex(color);});
```

The geometry's faces (`geo.faces`)— eight triangular faces, two for each segment of the cylinder — are iterated through. Because the top radius is 0, one triangular face of each segment is invisible. The code sets only the odd-numbered faces, which represent the visible faces of each side of the pyramid.

Creating the mesh

Materials are used for creating Three.js meshes. Materials control how an object's surface is rendered and how light interacts with it (see the Materials in Three.js sidebar). You create a *mesh* with Three.js by associating a geometry with a material, which is similar to what you must do in raw WebGL. In pyramid3js.html, this line takes care of making the association for the pyramid:

```
var pyramid = new THREE.Mesh(geo, new THREE.MeshBasicMaterial({vertexColors: THREE.FaceColors}));
```

The versatile Object3D root class

Many objects in Three.js — including meshes, lights, and camera — derive from the Three.js `Object3D` root class. Subclasses of `Object3D` inherit many properties and methods that simplify 3D programming. For example, a `position` property tracks the object's position in 3D; a `scale` property tracks the scaling in each axis; the `rotateX()`, `rotateY()`, and `rotateZ()` methods make rotation around standard axes straightforward; and `translateX()`, `translateY()`, and `translateZ()` simplify translation. You'll work frequently with subclasses of `Object3D` when you program with Three.js, so it pays to become familiar with its properties and methods.

`Mesh` is an instance of the `Object3D` class, which is a pervasive root class in Three.js (see the The versatile Object3D root class sidebar). `{vertexColors:THREE.FaceColors}` tells Three.js to use the `color` property of the geometry object's `faces` array to render the color for each face. From the raw WebGL example in Part 1, you know that this means that the colors of all three vertices of a face are specified to be identical. You can also specify a different color for one or more vertices to produce a gradient (see `THREE.VertexColors` in the Three.js documentation for details). In any case, Three.js takes care of the details of generating the vertex-color array and loading the low-level buffers for you. What's more, even the GLSL shader coding — and the compiling and linking of the shaders — are hidden from you through the mere selection of a material.

Placing the camera and mesh

Placement of the camera and objects in a scene determines what is finally visible within the canvas viewport. With the raw WebGL example, creation of a 4x4 transformation matrix (`modelViewMatrix`) translates, rotates, and scales the objects into place. Three.js offers a significantly more intuitive programming interface via an object's `position`, `rotation`, and `scale` properties (which a mesh inherits from `Object3D`). The default value for position is `(0,0,0)`. To place the mesh into the scene at `(0,1,0)`, all that is necessary is:

```
pyramid.position.y = 1;
```

This code pulls the camera back from the origin by 6 units:

```
camera.position.z = 6;
```

Again, Three.js shields all the underlying matrix mathematics from you, providing a conceptually pure API for programming.

Setting the scene

To set up the scene for rendering in the raw WebGL example, you needed to specify camera-to-viewport projection as a 4x4 `projectionMatrix`. In Three.js, you use the following code to set up the perspective camera:

```
var camera = new THREE.PerspectiveCamera( 45, 1, 0.1, 100);
```

The pyramid mesh is placed into the scene by:

```
var scene = new THREE.Scene();
scene.add(pyramid);
```

And finally, this code renders a frame of the scene:

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

All of the tedium of compiling and linking GLSL shading code, and marshaling data into low-level GPU buffers prior to rendering a frame, is invisible to the Three.js developer.

Adding animated rotation

Finally, to animate the rotation of the pyramid around the y-axis, you needed to manipulate the `modelViewMatrix` in the raw WebGL example. With Three.js, just call the inherited `rotateY` method from `Object3D`, and supply an incremental radian (1 degree = PI / 180 radian):

```
pyramid.rotateY(Math.PI / 180);
```

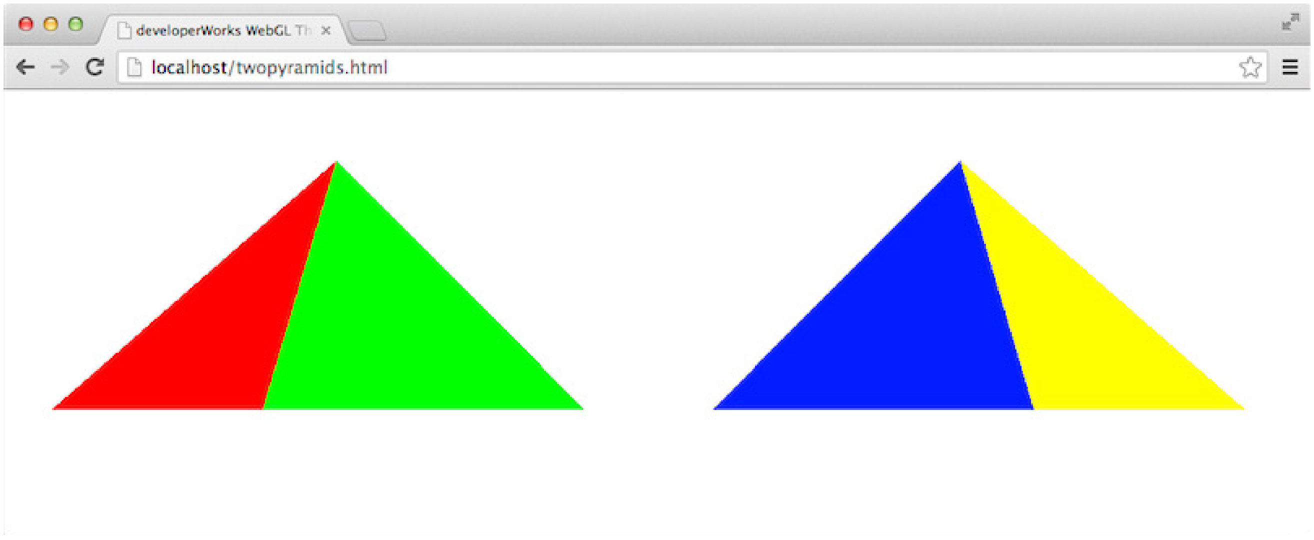
Upon every frame update, when rAF calls `animate()`, the pyramid rotates by 1 degree.

Leveraging objects in Three.js

Three.js elegantly abstracts away the complexity of low-level WebGL development, leaving you to focus on the higher-level details of 3D development (such as mesh creation, lighting, and animation). The next example takes the pyramid case a little further. You'll see how you can leverage the object orientation of the Three.js library to create a more complex scene quickly.

Load up twopyramids.html in your browser. Figure 2 shows the display.

Figure 2. Two pyramids using Three.js



In the scene captured in Figure 2, two pyramids rotate around the y-axis in opposite directions. Listing 3 shows the underlying code of twopyramids.html. The key differences between it and pyramid3js.html are highlighted in boldface.

Listing 3. Creating a clone of pyramid

```
function draw3D()  {
    function animate() {
      requestAnimationFrame(animate);
      pyramid1.rotateY(Math.PI/ 180);
      pyramid2.rotateY(- (Math.PI/ 180));
      renderer.render(scene, camera);
    }

    var geo = new THREE.CylinderGeometry(0,2,2,4,1, true);
    var faceColors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00];
    faceColors.forEach( function(color, idx)
       { geo.faces[2 * idx + 1].color.setHex(color);});
    var pyramid1 = new THREE.Mesh(geo,
       new THREE.MeshBasicMaterial({vertexColors: THREE.FaceColors}));
    pyramid1.position.set(-2.5, 1, 0);
    var pyramid2 = pyramid1.clone();
    pyramid2.position.set(2.5, 1, 0);

    var scene = new THREE.Scene();

    scene.add(pyramid1);
    scene.add(pyramid2);

    var camera = new THREE.PerspectiveCamera(  45, 1024/500,0.1, 100);
    camera.position.z = 6;

    var div = document.getElementById("shapecanvas2");
    var renderer = new THREE.WebGLRenderer();

    renderer.setSize(1024,500);
    div.appendChild( renderer.domElement );

    animate();
}
```

Scene graphs in 3D rendering

A *scene graph* is a data structure — almost always an acyclic tree — that holds the objects (meshes, lights, cameras, etc.) of a scene (often together with associated transformations). A scene graph is created when a 3D scene contains more than one object — which is always the case because the camera through which you view the scene is itself considered an object. The objects in a scene graph represent the displayed scene. It is the job of the 3D rendering library runtime to render the objects in the scene graph.

In twopyramids.html, the canvas size and camera positions differ from those in pyramid3js.html to accommodate the display of the two pyramids.

Listing 3 shows how simple it is to reuse 3D objects that you create in Three.js. The highlighted code simply calls the `clone()` method inherited from `Object3D` (the root class of `Mesh`) to create another instance of the pyramid. This task would not have been nearly as easy if you were coding procedurally with raw WebGL.

`pyramid2`, the cloned `Mesh`, is customized with a different position and rotation via this code:

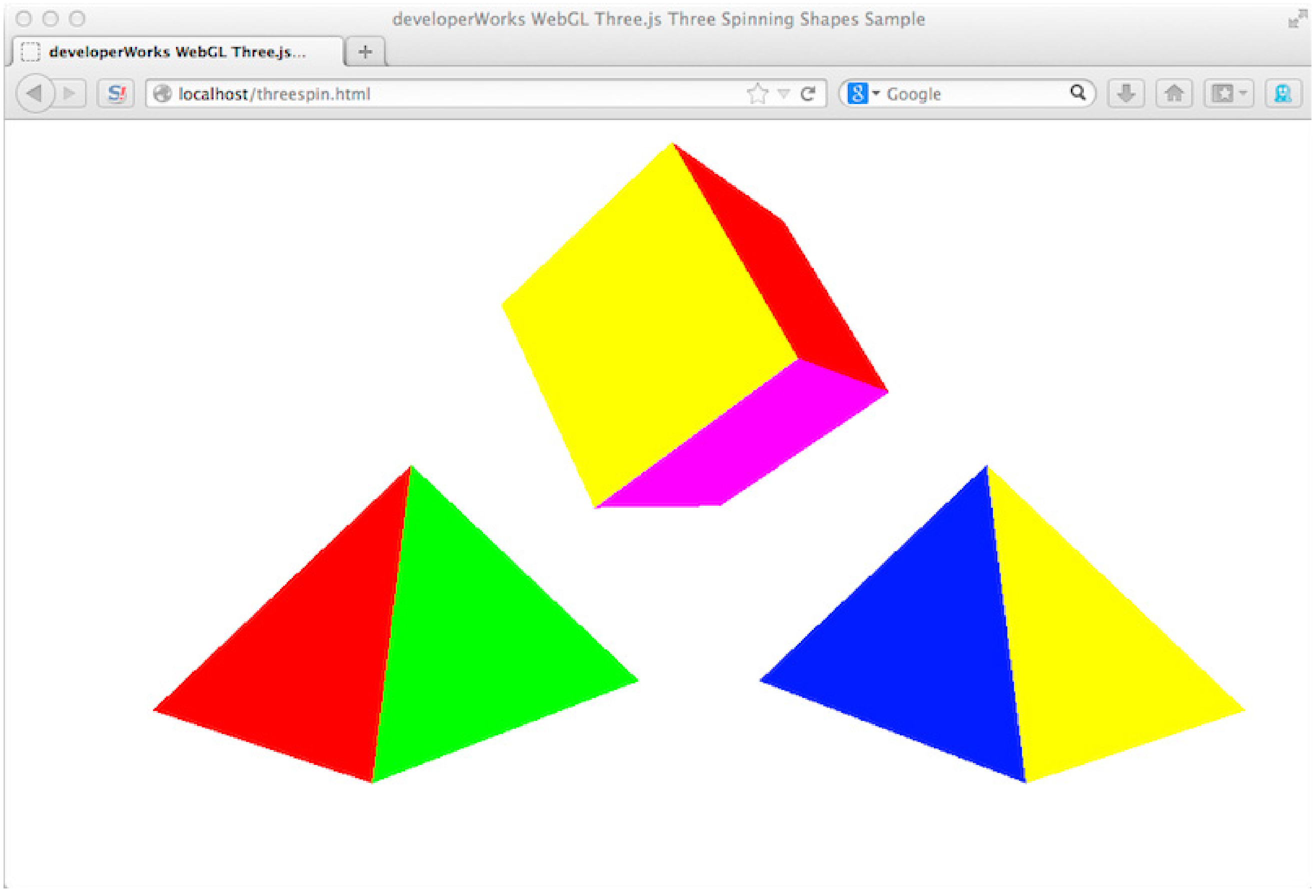
```
pyramid2.position.set(2.5, 1, 0);
...
pyramid2.rotateY(- (Math.PI/ 180));
```

And now you have a scene with two pyramids. Cloning is especially handy for scenes in which you need many almost identical objects.

Building a more complex scene graph

The next example incrementally adds a dissimilar object to the spinning mix. As a scene gets more complicated, the fact that Three.js works with scene graphs when rendering becomes more apparent (see the Scene graphs in 3D rendering sidebar). Load up threespin.html and take a look. Figure 3 shows a snapshot of threespin.html in action.

Figure 3. Adding a rotating cube


developerWorks WebGL Three.js Three Spinning Shapes Sample
developerWorks WebGL Three.js...
localhost/threespin.html
Google

In threespin.html, a spinning cube appears just above the rotating pyramids. The cube's rotation is set around an angled axis, so it appears to be tumbling forward. The key code in threespin.html that adds the cube is highlighted in Listing 4.

Listing 4. Adding a rotating cube

```
function draw3D()  {

    function animate() {
      requestAnimationFrame(animate);
      pyramid1.rotateY(Math.PI/180);
      pyramid2.rotateY(-(Math.PI/180));
      cube.rotateY(Math.PI/180); cube.rotateX(Math.PI/90);
      renderer.render(scene, camera);
    }

    var geo = new THREE.CylinderGeometry(0,2,2,4,1, true);
    var faceColors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00];
    faceColors.forEach( function(color, idx)
       { geo.faces[2 * idx + 1].color.setHex(color);});
    var pyramid1 = new THREE.Mesh(geo,
       new THREE.MeshBasicMaterial({vertexColors: THREE.FaceColors}));
    pyramid1.position.set(-2.5, -1, 0);
    var pyramid2 = pyramid1.clone();
    pyramid2.position.set(2.5, -1, 0);

    geo = new THREE.CubeGeometry(2,2,2);
    faceColors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00, 0x00ffff, 0xff00ff];
    faceColors.forEach( function(color, idx)
       { geo.faces[2 * idx + 1].color.setHex(color);
       geo.faces[2*idx].color.setHex(color);});
    var cube = new THREE.Mesh(geo, new THREE.MeshBasicMaterial({vertexColors:
       THREE.FaceColors}));
    cube.position.set(0, 1, 0);

    var camera = new THREE.PerspectiveCamera(  45, 1024/500,0.1, 100);
    camera.position.z = 7;

    var scene = new THREE.Scene();
    scene.add(pyramid1);
    scene.add(pyramid2);
    scene.add(cube);
    var div = document.getElementById("shapecanvas2");
    var renderer = new THREE.WebGLRenderer();
    renderer.setSize(1024,500);
    div.appendChild( renderer.domElement );
    animate();

}
```

`cube` is a built-in Three.js geometry, so in Listing 4, you create a new `cube` instance by using the `THREE.CubeGeometry` constructor, specifying `2` for each of its three dimensions:

```
geo = new THREE.CubeGeometry(2,2,2);
```

You could have as easily created a rectangular block by specifying different dimensions.

Next, you set the colors of the cube's six square faces. Notice that you must set color for 12 triangles this time because each square face is represented by two triangles:

```
faceColors.forEach( function(color, idx) { geo.faces[2 * idx + 1].color.setHex(color);
   geo.faces[2*idx].color.setHex(color);});
```

The cube is positioned higher on the y-axis within the group of three shapes:

```
cube.position.set(0, 1, 0);
```

To get the tumbling-forward rotation effect, the cube is rotated around the x-axis by 2 degrees and the y-axis by 1 degree for every rAF animate frame:

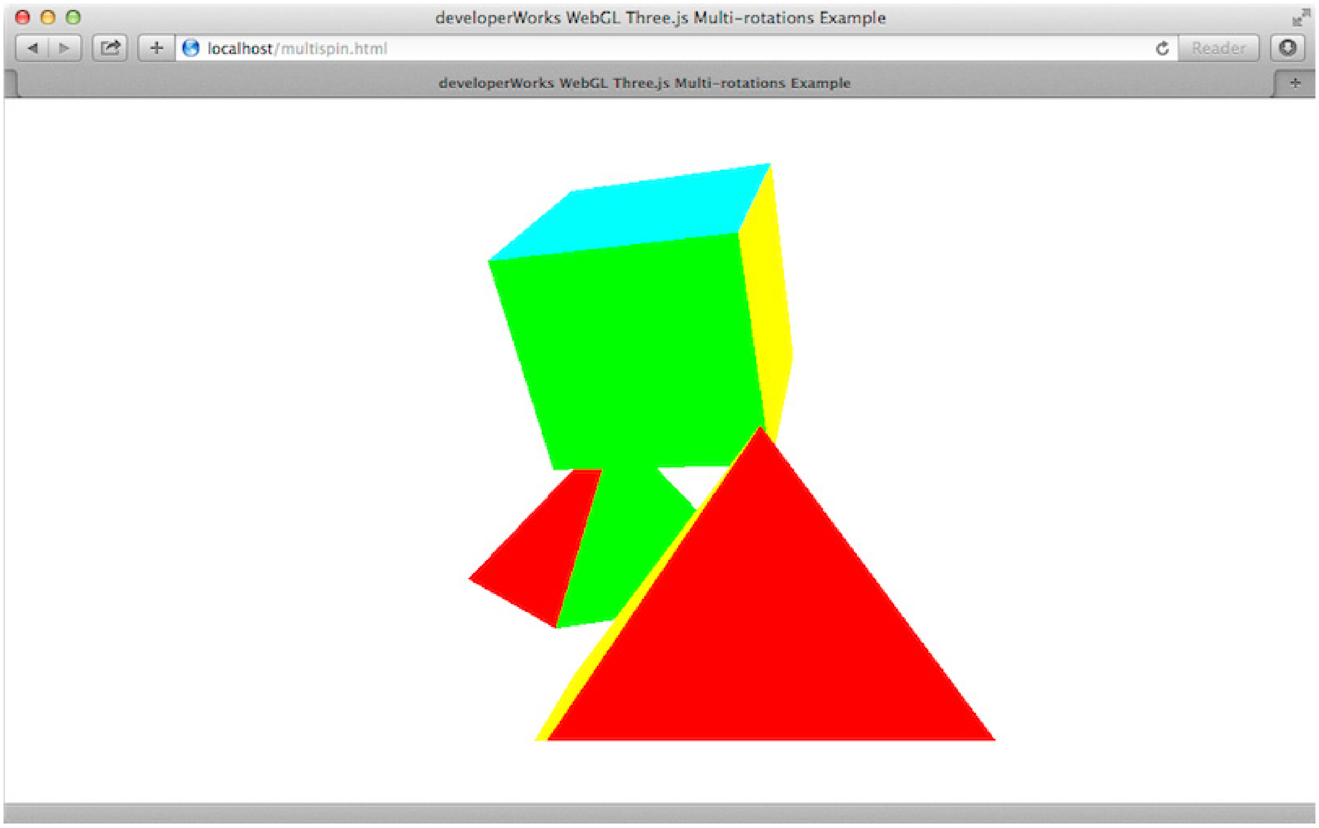
```
cube.rotateY(Math.PI/180); cube.rotateX(Math.PI/90);
```

Scene graph inheritance relationships

Often in 3D work, you'll find yourself wanting to perform manipulations or transformations on a group of related objects. The parent-child relationship between objects in a scene graph, together with the inheritance feature in Three.js (and other frameworks), make group transformations relatively straightforward.

Take a look at threespin.html again. Suppose you now want to spin the entire group of three objects around the y-axis while keeping each object rotating independently. You could work out the math and translate/rotate the cube and each of the pyramids within the `animate()`function. But that would take quite a bit of work. You can take advantage of Three.js graph-transformation inheritance instead: Create a parent for the three shapes and spin this parent object around the y-axis. The three child objects inherit the "group spin" automatically in addition to their own rotations. You can observe this combined rotation in your browser if you load up multispin.html, which is captured in action in Figure 4.

Figure 4. Rotating scene graph


developerWorks WebGL Three.js Multi-rotations Example
localhost/multispin.html
Reader
developerWorks WebGL Three.js Multi-rotations Example

In multispin.html, the scene graph is constructed with an instance of `Object3D` named `multi` that's the parent of the cube, sphere, and pyramid. When `multi` is rotated around the y-axis, its children inherit this rotation. Note in particular the cube and the complex composition of transformations that it is undergoing. The cube is still tumbling forward while spinning around the y-axis with the group. In Listing 5, the highlighted code is responsible for creating and rotating the `multi` parent object.

Listing 5. Rotating a group through scene graph inheritance

```
function draw3D()  {
  function animate() {
    requestAnimationFrame(animate);
    pyramid1.rotateY(Math.PI/180);
    pyramid2.rotateY(-(Math.PI/180));
    cube.rotateY(Math.PI/180); cube.rotateX(Math.PI/90);
    multi.rotateY(Math.PI/360);
    renderer.render(scene, camera);
  }

  var geo = new THREE.CylinderGeometry(0,2,2,4,1, true);
  var faceColors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00];
  faceColors.forEach( function(color, idx)
     { geo.faces[2 * idx + 1].color.setHex(color);});
  var pyramid1 = new THREE.Mesh(geo,
     new THREE.MeshBasicMaterial({vertexColors: THREE.FaceColors}));
  pyramid1.position.set(-2.5, -1, 0);
  var pyramid2 = pyramid1.clone();
  pyramid2.position.set(2.5, -1, 0);

  geo = new THREE.CubeGeometry(2,2,2);
  console.log(geo.faces.length);
  faceColors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00, 0x00ffff, 0xff00ff];
  faceColors.forEach( function(color, idx)
     { geo.faces[2 * idx + 1].color.setHex(color);
   geo.faces[2*idx].color.setHex(color);});
  var cube = new THREE.Mesh(geo,
     new THREE.MeshBasicMaterial({vertexColors: THREE.FaceColors}));
  cube.position.set(0,1,0);

  var camera = new THREE.PerspectiveCamera(45, 1024/500,0.1, 100);
  camera.position.z = 7;

  var multi = new THREE.Object3D();
  multi.add(cube);
  multi.add(pyramid1);
  multi.add(pyramid2);
  multi.position.z = 0;

  var scene = new THREE.Scene();
  scene.add(multi);

  var div = document.getElementById("shapecanvas2");
  var renderer = new THREE.WebGLRenderer();
  renderer.setSize(1024,500);
  div.appendChild( renderer.domElement );

  animate();

}
```

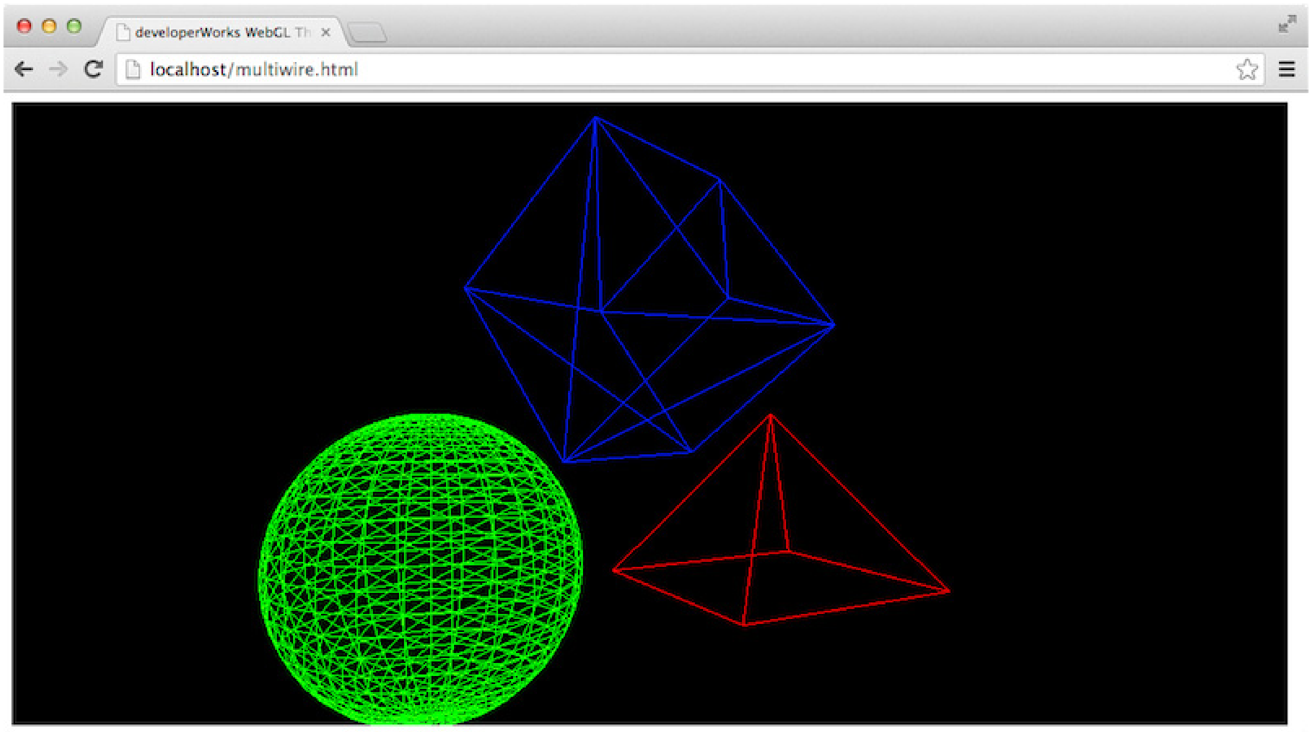
In Listing 5, the `multi` group is rotated PI/360 radians = 0.5 degrees on every rAF animate frame, and each shape self-rotates PI/180 radians = 1 degree at the same time.

Wireframing objects

Thus far, the cube and pyramids are rendered as solids with uniform flat colors. But computer graphics animations often animate a mesh (or *wireframe*) itself.

You know from the raw WebGL example that the mesh wireframe must be explicitly defined (via vertex buffers) before the object can be rendered. Three.js gives you an easy way to render just this frame. Load up multiwire.html and watch the wireframes spin. Figure 5 is a snapshot from multiwire.html.

Figure 5. Wireframe and complexity of a sphere



The background in multiwire.html is set to black to enhance the contrast with the wireframes. You can see clearly that each face of the blue cube is composed of exactly two triangles; that's why you needed to specify each face color twice when setting up the face colors for the cube.

In multiwire.html, I've snuck in a sphere to replace one of the pyramids to help you visualize the ability to model any shape by using enough triangles. The sphere's apparently smooth curvature is rendered by using about 1,200 triangles. You definitely do not want to specify the vertices of each of these triangles manually. Three.js has a built-in geometry generator for spheres. Listing 6 shows the code for multiwire.html.

Listing 6. Rotating wireframe objects

```
function draw3D()  {

    function animate() {
      requestAnimationFrame(animate);
      pyramid1.rotateY(Math.PI/180);
      sphere.rotateY(Math.PI/180);
      cube.rotateX(Math.PI/90);
      multi.rotateY(Math.PI/360);
      renderer.render(scene, camera);
    }

    var geo = new THREE.CylinderGeometry(0,2,2,4,1, true);
    var pyramid1 = new THREE.Mesh(geo,
       new THREE.MeshBasicMaterial({color: 0xff0000, wireframe: true}));
    pyramid1.position.set(-2.5, -1, 0);

    geo = new THREE.SphereGeometry(1, 25, 25);
    var sphere = new THREE.Mesh(geo,
       new THREE.MeshBasicMaterial({color: 0x00ff00, wireframe: true }));
    sphere.position.set(2.5, -1, 0);
    geo = new THREE.CubeGeometry(2,2,2);
    var cube = new THREE.Mesh(geo,
       new THREE.MeshBasicMaterial({color: 0x0000ff, wireframe: true })   );
    cube.position.set(0,1,0);

    var camera = new THREE.PerspectiveCamera(  45, 1024/500,0.1, 100);
    camera.position.z = 7;

    var multi = new THREE.Object3D()
    multi.add(cube);
    multi.add(pyramid1);
    multi.add(sphere);
    multi.position.z = 0;

    var scene = new THREE.Scene();
    scene.add(multi);

    var div = document.getElementById("shapecanvas2");
    var renderer = new THREE.WebGLRenderer();
    renderer.setSize(1024,500);
    renderer.setClearColor(0x000000, 1);
    div.appendChild( renderer.domElement );
    animate();

  }
```

In the highlighted code in Listing 6, you can see how the sphere is created with Three.js built-in `THREE.SphereGeometry` generator. The sphere that's generated has a radius of 1, with 25 rings of 25 segments each.

In addition, notice that setting `THREE.MeshBasicMaterial`'s `wireframe` property to `true` enables the display of a wireframe for the mesh. For example, the cube is displayed as a wireframe by setting its material's `wireframe` property:

```
var cube = new THREE.Mesh(geo,new THREE.MeshBasicMaterial({color: 0x0000ff, wireframe: true })   );
```

Adding lighting and shadow effects

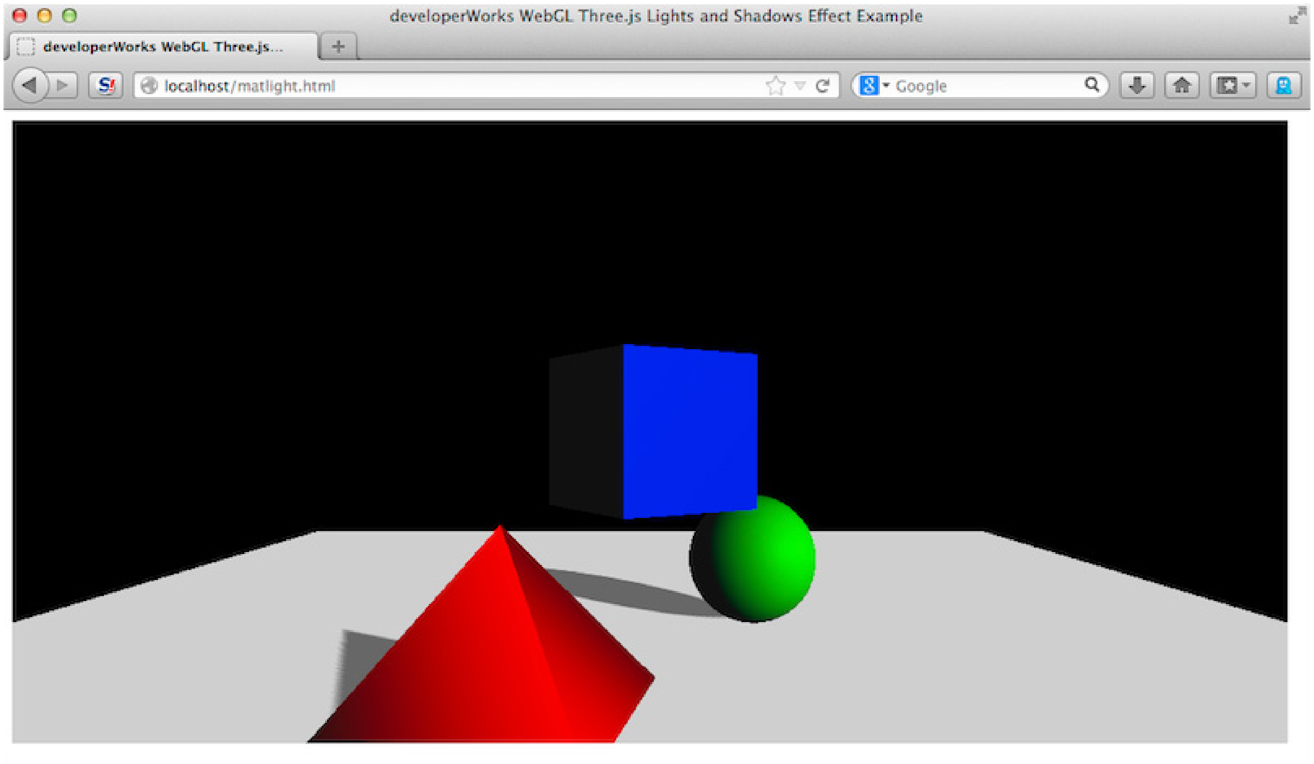
Lights in Three.js

Various lights with different properties are available for customizing the appearance of your Three.js scenes. Use `AmbientLight` to add light uniformly to all objects in a scene. `DirectionalLight` can simulate a light source from far away (casting almost parallel rays) that can cast shadows. `SpotLight` can illuminate and cast a shadow in a controlled direction. `PointLight` has intensity that fades with distance but is omnidirectional from the source.

Up to this point, the objects in the scene are magically illuminated, without the need for you to add lights to the scene. This is the case because the default `THREE.MeshBasicMaterial` requires no light to render.

In a more typical 3D scene, you might want to increase realism through better control of lighting. To do so, you must explicitly add light objects to the scene. Three.js supports several types of lights (see the Lights in Three.js sidebar). Figure 6 shows the group rotating scene with lighting and shadow effects added. Load up matlight.html and see it in action. Notice that this scene is decidedly more realistic than any of the previous examples.

Figure 6. Scene with lights and shadows


developerWorks WebGL Three.js Lights and Shadows Effect Example
developerWorks WebGL Three.js...
localhost/matlight.html
Google

Observe how light is reflected off the pyramid and the sphere as they rotate. See if you can tell where the light is shining from. Listing 7 shows the code for matlight.html.

Listing 7. Lighting and shadow effects

```
function draw3D()  {

    function animate() {
      requestAnimationFrame(animate);

      pyramid1.rotateY(Math.PI/180);
      sphere.rotateY(Math.PI/180);
      cube.rotateY(Math.PI/180);
      multi.rotateY(Math.PI/480);
      renderer.render(scene, camera);
    }

    var geo = new THREE.CylinderGeometry(0,2,2,4,1, true);
    var pyramid1 = new THREE.Mesh(geo, new THREE.MeshPhongMaterial({color: 0xff0000}));
    pyramid1.position.set(-2.5, -1, 0);

    geo = new THREE.SphereGeometry(1, 25, 25);
    var sphere = new THREE.Mesh(geo, new THREE.MeshPhongMaterial({color: 0x00ff00}));
    sphere.position.set(2.5, -1, 0);

    geo = new THREE.CubeGeometry(2,2,2);
    var cube = new THREE.Mesh(geo,new THREE.MeshPhongMaterial({color: 0x0000ff })   );
    cube.position.set(0, 1, 0);

    var camera = new THREE.PerspectiveCamera(  45, 1024/500,0.1, 100);
    camera.position.z = 10;
    camera.position.y = 1;

    var multi = new THREE.Object3D();
    pyramid1.castShadow = true; sphere.castShadow = true;
    multi.add(cube);
    multi.add(pyramid1);
    multi.add(sphere);
    multi.position.z = 0;


    geo = new THREE.PlaneGeometry(20, 25);
    var floor = new THREE.Mesh(geo, new THREE.MeshBasicMaterial({color : 0xcfcfcf}));
    floor.material.side = THREE.DoubleSide;
    floor.rotation.x = Math.PI/2;
    floor.position.y = -2;
    floor.receiveShadow = true;

    var light = new THREE.DirectionalLight(0xe0e0e0);
    light.position.set(5,2,5).normalize();
    light.castShadow = true;
    light.shadowDarkness = 0.5;
    light.shadowCameraRight = 5;
    light.shadowCameraLeft = -5;
    light.shadowCameraTop = 5;
    light.shadowCameraBottom = -5;
    light.shadowCameraNear = 2;
    light.shadowCameraFar = 100;

    var scene = new THREE.Scene();
    scene.add(floor);
    scene.add(multi);
    scene.add(light);
    scene.add(new THREE.AmbientLight(0x101010));


    var div = document.getElementById("shapecanvas2");
    var renderer = new THREE.WebGLRenderer();
    renderer.setSize(1024,500);
    renderer.setClearColor(0x000000, 1);
    renderer.shadowMapEnabled = true;
    div.appendChild( renderer.domElement );
    animate();

  }
```

In Listing 7, a new `floor` object is added to show off the rendered shadow effect. The highlighted code makes use of the plane geometry generator of Three.js to generate a plane that is 20 units wide by 25 units high. This plane is generated on the x-y plane, so it is rotated 90 degrees on the x-axis and moved lower in the scene by translating 2 units down on the y-axis.

The cube, sphere, and pyramid are now made with `THREE.MeshPhongMaterial`, giving them a shiny plastic-like reflectiveness that also enables them to cast shadows. This code changes the material for the cube:

```
var cube = new THREE.Mesh(geo,new THREE.MeshPhongMaterial({color: 0x0000ff }));
```

A directional light is added that points from the upper right toward the origin — more precisely, from (5,2,5) to (0,0,0). In addition to the directional light, an ambient light is added to "fill in" the dark areas of the scene. Otherwise, the area that's not illuminated by the directional light would be completely dark. This ambient light is added by this code:

```
scene.add(new THREE.AmbientLight(0x101010));
```

Rendering shadow effects in Three.js

Adding precise shadows to a scene is a prohibitively expensive exercise that requires computationally intensive ray-tracing or radiosity algorithms. These algorithms are taxing even on today's most capable 3D rendering hardware.

Computationally efficient approximation algorithms exist. Three.js implements the z-buffer shadow mapping technique advanced by Lance Williams in his 1978 paper titled "Casting Curved Shadows on Curved Surfaces." This effective and widely used algorithm renders the scene from the point of view of the light source(s). It then uses the z-buffer (hidden surface removal) information to determine if the light can "see a point in the scene." Any point that's not "seen by the light source" is deemed to be in shadow.

Although shadow effects are common in gaming engines, not all 3D libraries or frameworks support them. (For example, SceneJS — the library that this article examines next — doesn't support shadow effects).

To handle the parallel rays of a directional light source, Three.js requires the definition of an orthographic projection camera (the *shadow camera*) to perform the shadow mapping (to determine what the light can "see"). The code in matlight.html that creates the shadow camera is:

```
light.shadowCameraRight = 5;
light.shadowCameraLeft = -5;
light.shadowCameraTop = 5;
light.shadowCameraBottom = -5;
light.shadowCameraNear = 2;
light.shadowCameraFar = 100;
```

To help the algorithm avoid unnecessary computation, you must specify the objects that can cast shadows and those that can receive shadows. In matlight.html, this code specifies that only the sphere and pyramid cast shadows:

```
pyramid1.castShadow = true; sphere.castShadow = true;
```

And only the white floor can receive shadows:

```
floor.receiveShadow = true;
```

Even though the dynamically changing shadows are rendered through an approximation, they still look quite convincing in the scene.

Staging a 3D scene with a walk-through

Now it's time to put everything you've learned so far into practice by creating a moderately complex scene with two rooms and a door that opens between them. The second room will contain the spinning group of objects. You'll lead viewers from one room, through the door, into the other room, where they'll admire the spinning objects, complete with lighting and shadows. You accomplish the walk-through effect — similar to a familiar "dolly-in" shot frequently used in Hollywood movies — by animating the position of the camera.

Before you plan the shot, you must create the set. This two-room set contains many 3D objects that you can use Three.js APIs to create. It is the most complex scene graph in this article. Table 1 shows a list of the objects included in this scene graph, together with their properties and a brief description of each.

Table 1. Objects (meshes) that comprise the scene in fullscene.html

Mesh/Object	Name	Position	Color/Material	Description
Sphere	`sphere`	(-2.5,-1,0)	Green	Rotating clockwise around y-axis
Cube	`cube`	(0,1,0)	Blue	Rotating clockwise around y-axis
Pyramid	`pyramid1`	(2.5,-1,0)	Red	Rotating clockwise around y-axis
Multiple-object group	`multi`	(0,0,0)	(Not applicable)	Used to rotate sphere, pyramid, and cube together clockwise
Directional light	`light`	Direction from (5,2,5) to (0,0,0)	Pale white	Used to add specular reflection and shadow effects for the spinning geometries
Ambient light	(none)	(Not applicable)	Low-intensity white	Prevents unlit area from going completely dark
Plane (floor)	`floor`	Created on x-y plane, rotated on the x-axis by 90 degrees, then translated on the z-axis by 10 units (outward toward the viewer)	Pale white	20 x 50
Plane (left wall)	`wallleft`	Created on x-y plane rotated on the y-axis by 90 degrees, then translated on the x-axis by -8, and on the z-axis by 12 units	Yellow	50 x 20
Plane (right wall)	`wallright`	Same as `wallleft`, except translated to x=8	Yellow	50 x 20
Shape geometry (wall with door cutout)	`dWall`	Created on x-y plane, then moved to: (-24.5, -2, 8)	Red	50 x 20 with a cutout that is 2 x 3.5 to fit the door
Rectangular block (door)	`door`	Created on x-y as a rectangular cube; then moved to: (-1.5, -0.25, 8)	Gray-blue	2 x 3.5 by 0.2 thick; origin is internally translated to rotate around hinge

Load up fullscene.html and watch the completed scene walk-through for yourself. See how you "walk into the first room," then the door swings open as you head toward the second room with the spinning shapes, and finally you walk into the newly discovered room to admire the spinning shapes. Figure 7 shows fullscene.html as you enter into the first room with the door closed.

Figure 7. Walk-through: Into first room

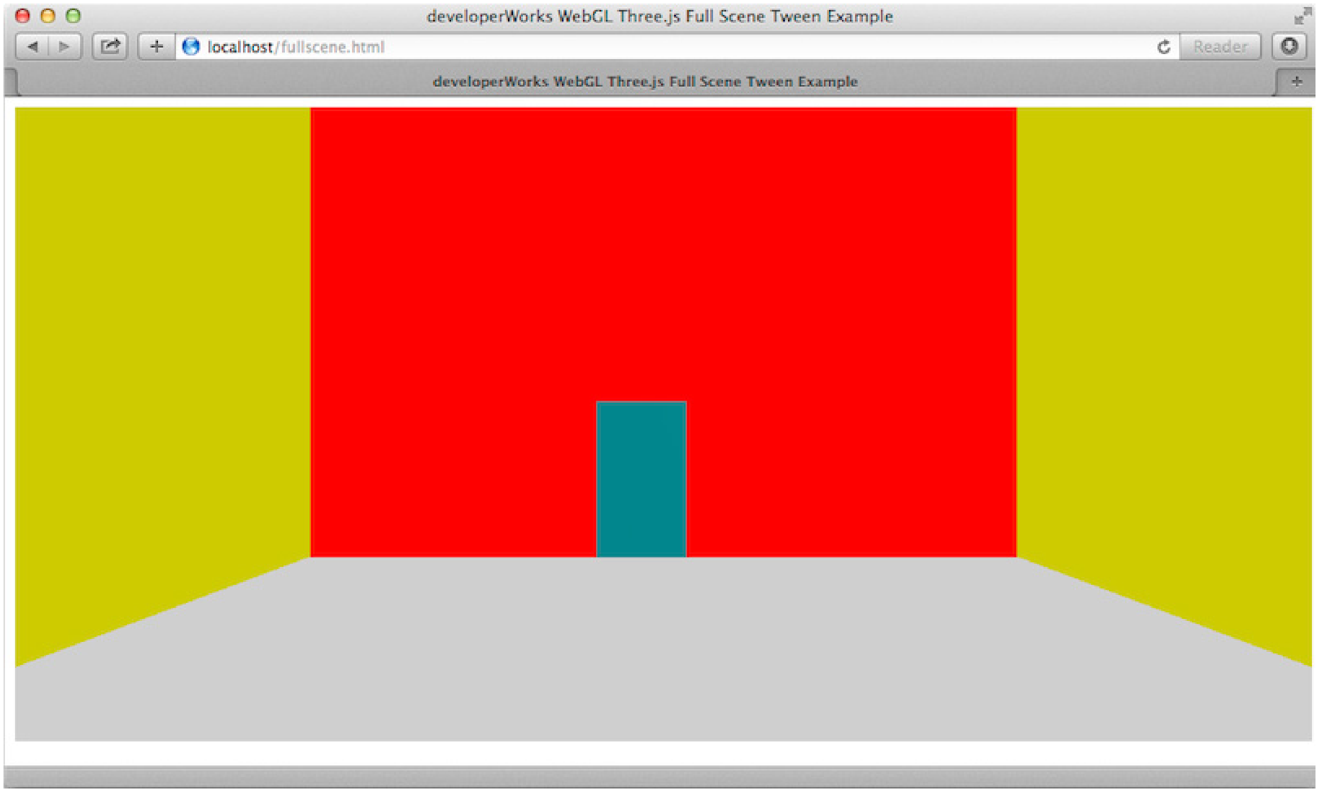

developerWorks WebGL Three.js Full Scene Tween Example
localhost/fullscene.html
Reader
developerWorks WebGL Three.js Full Scene Tween Example

Figure 8 shows fullscene.html, as the door opens to reveal the second room.

Figure 8. Walk-through: Door opening

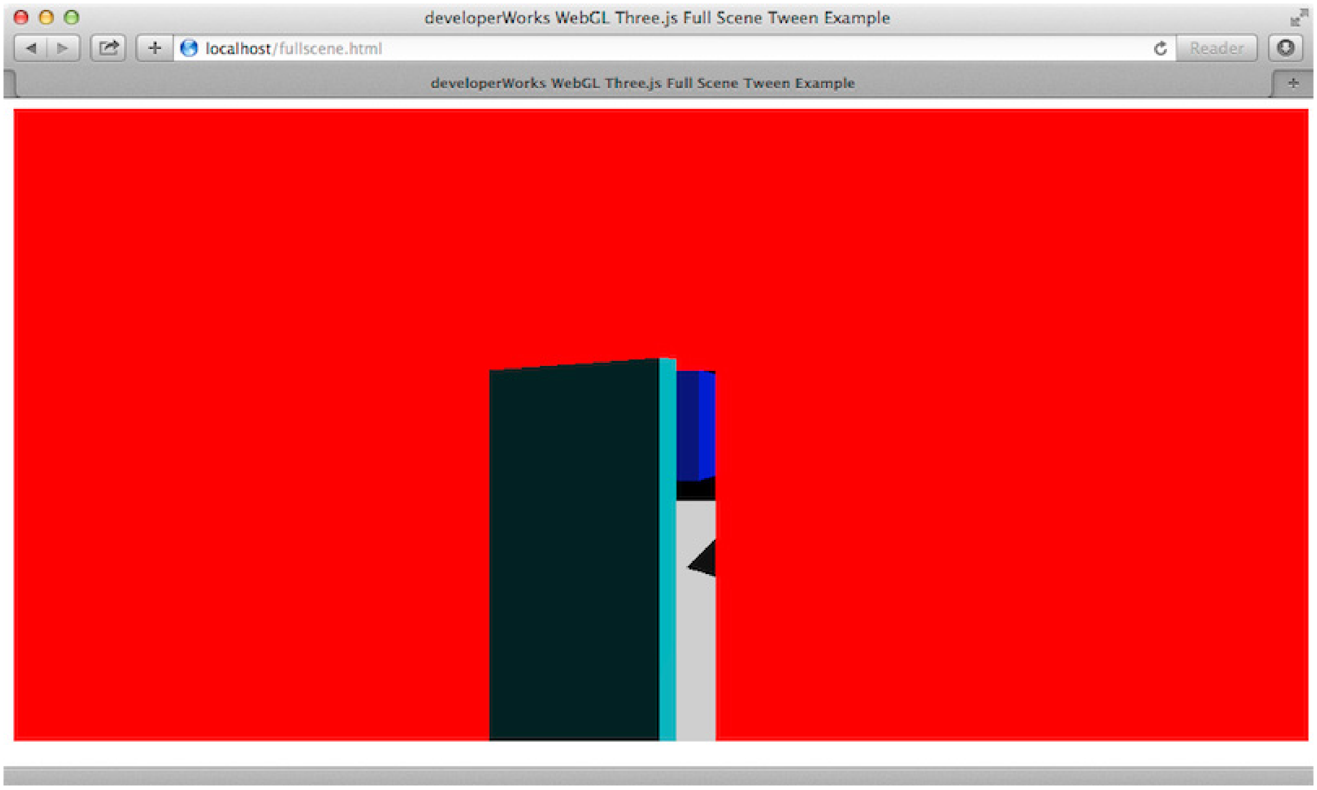
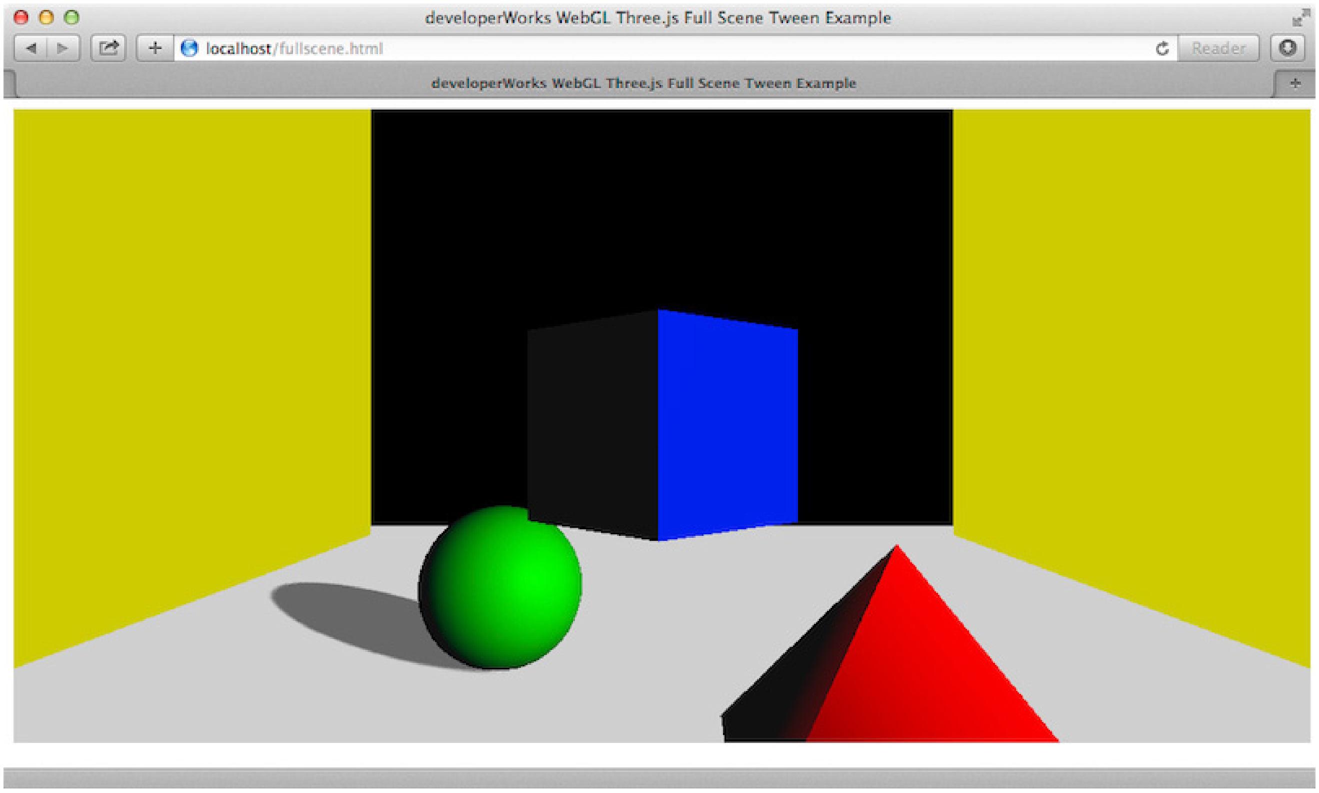

developerWorks WebGL Three.js Full Scene Tween Example
localhost/fullscene.html
Reader
developerWorks WebGL Three.js Full Scene Tween Example

Figure 9 shows fullscene.html, after you step through the open door and enter the second room.

Figure 9. Walk-through: In the second room


developerWorks WebGL Three.js Full Scene Tween Example
localhost/fullscene.html
Reader
developerWorks WebGL Three.js Full Scene Tween Example

You should be able to understand this complex scene's construction by going through each row in Table 1 and locating the corresponding object's construction and placement code in fullscene.html. Listing 8 shows the code for fullscene.html.

Listing 8. Creating the scene and animation in fullscene.html

```
function draw3D()  {

   function setup() {

        var tweenOpenDoor = new TWEEN.Tween( door.rotation )
            .to( { y: door.rotation.y - Math.PI }, 3000 );

        var tweenWalkUp = new TWEEN.Tween(camera.position)
            .to({z: camera.position.z - 25}, 8000);

        var tweenWalkIn = new TWEEN.Tween(camera.position)
            .to({z: camera.position.z - 32}, 5000);

          tweenOpenDoor.chain(tweenWalkIn);
          tweenWalkUp.chain(tweenOpenDoor);
          tweenWalkUp.start();

    }

    function animate() {
      requestAnimationFrame( animate );
      ... code to rotate objects ...
      TWEEN.update();
      renderer.render(scene, camera);
    }

   // Code for setting up the three spinning shapes skipped for brevity
    // floor
    geo = new THREE.PlaneGeometry(20, 50);
    var floor = new THREE.Mesh(geo, new THREE.MeshBasicMaterial({color: 0xcfcfcf}));
    floor.material.side = THREE.DoubleSide;
    floor.rotation.x = Math.PI/2;
    floor.position.y = -2;  floor.position.z = 10;
    floor.receiveShadow = true;

    // left wall
    geo = new THREE.PlaneGeometry(50,20);
    var wallleft = new THREE.Mesh(geo ,new THREE.MeshBasicMaterial({color : 0xcccc00}));
    wallleft.material.side = THREE.DoubleSide;
    wallleft.rotation.y = Math.PI/2;
    wallleft.position.x = -8;
    wallleft.position.z = 12;

    // right wall
    var wallright = wallleft.clone();
    wallright.position.x = 8;

    // door
    geo = new THREE.CubeGeometry(2, 3.5, 0.2);
    geo.applyMatrix( new THREE.Matrix4().makeTranslation( 1, 0, 0 ) );  // move to hinge
    var door = new THREE.Mesh(geo, new THREE.MeshPhongMaterial({ color: 0x00c0ce}));
    door.position.set(-1.5, -0.25, 8);

    // wall with door
    var doorWall = new THREE.Shape();
    doorWall.moveTo(  0, 0 );
    doorWall.lineTo(  23, 0 );
    doorWall.lineTo( 23, 3.5 );
    doorWall.lineTo( 25, 3.5 );
    doorWall.lineTo( 25, 0);
    doorWall.lineTo( 50, 0);
    doorWall.lineTo(50, 20)
    doorWall.lineTo(0,20);
    doorWall.lineTo(0,0);
    geo = new THREE.ShapeGeometry(doorWall);
    var dWall = new THREE.Mesh(geo, new THREE.MeshBasicMaterial({color: 0xff0000}));
    dWall.material.side = THREE.DoubleSide;
    dWall.position.set(-24.5,-2, 8);

    // lights
    var light = new THREE.DirectionalLight(0xe0e0e0);
    light.position.set(5,2,5).normalize();
    light.castShadow = true;
    light.shadowDarkness = 0.5;
    light.shadowCameraRight = 5;
    light.shadowCameraLeft = -5;
    light.shadowCameraTop = 5;
    light.shadowCameraBottom = -5;
    light.shadowCameraNear = 2;
    light.shadowCameraFar = 100;

    var scene = new THREE.Scene();
    scene.add(floor)
    scene.add(wallright);
    scene.add(wallleft);
    scene.add(dWall);
    scene.add(door);
    scene.add(light);
    scene.add(multi);
    scene.add(new THREE.AmbientLight(0x101010));

    var camera = new THREE.PerspectiveCamera(  45, 1024/500,0.1, 100);
    camera.position.z = 40;  // 20
```

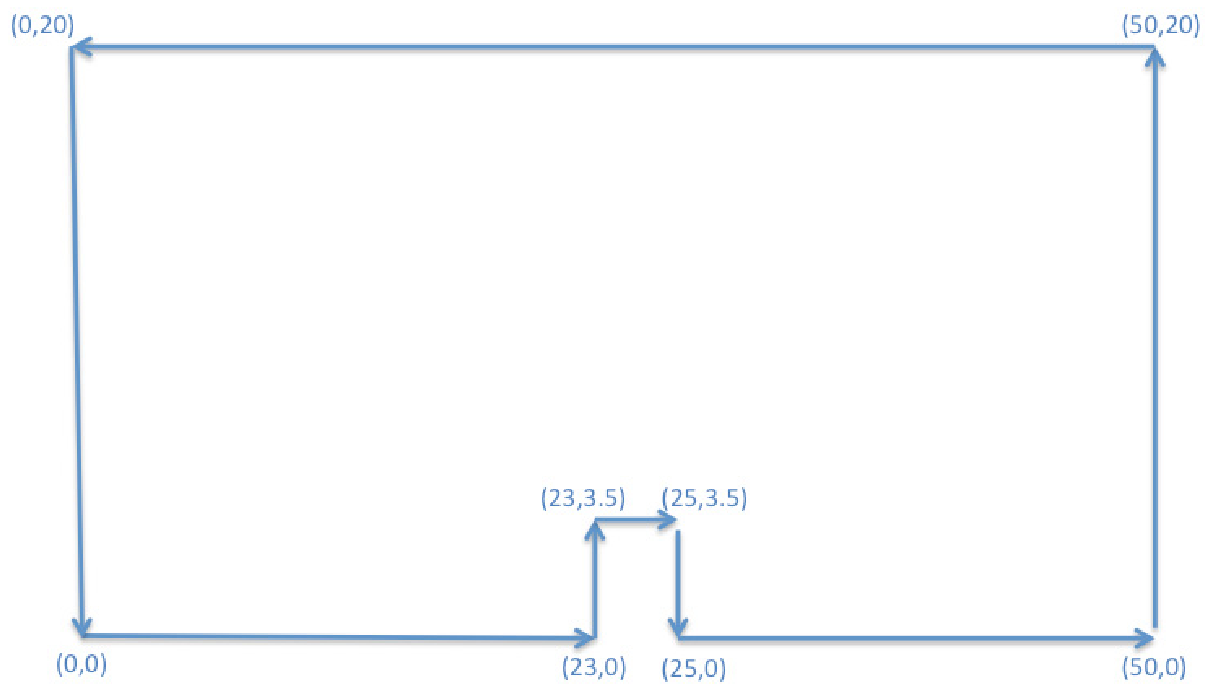
```
    camera.position.y = 1;

    var div = document.getElementById("shapecanvas2");
    var renderer = new THREE.WebGLRenderer();
    renderer.setSize(1024,500);
    renderer.setClearColor(0x000000, 1);
    renderer.shadowMapEnabled = true;

    div.appendChild( renderer.domElement );


    setup();
    animate();

}
```

Most of the object creation and placement code in fullscene.html should already be familiar to you. I'll focus on the techniques introduced in Listing 8 for creating the door and the wall that it's attached to.

Turning 2D shapes into 3D geometries

The wall in the center of the scene, with the door, is not a simple plane geometry. It is created using Three.js's ShapeGeometry API. Essentially, you create a shape by using familiar 2D canvas-like drawing APIs (`moveTo()` and `lineTo()`). Then you ask Three.js to convert the shape into an irregularly shaped 3D plane. (Note that you can also "extrude" the shape into a object that has thickness by using Three.js; see the Three.js documentation on ExtrudeGeometry.) The shape that's used for this wall is shown in Figure 10.

Figure 10. The shape used for creating the middle wall



You can track the segment-by-segment creation of the shape through the lines drawn, starting from the origin at `(0,0)` and ending back at `(0,0)`. The corresponding code to create this wall with door is shown in Listing 9.

Listing 9. Creating `ShapeGeometry` in Three.js

```
var doorWall = new THREE.Shape();
doorWall.moveTo( 0, 0 );
doorWall.lineTo( 23, 0 );
doorWall.lineTo( 23, 3.5 );
doorWall.lineTo( 25, 3.5 );
doorWall.lineTo( 25, 0);doorWall.lineTo( 50, 0);
doorWall.lineTo(50, 20)
doorWall.lineTo(0,20);
doorWall.lineTo(0,0);

geo = new THREE.ShapeGeometry(doorWall);
var dWall = new THREE.Mesh(geo, new THREE.MeshBasicMaterial({color: 0xff0000}));
```

After you draw the 2D shape (`doorWall`), you can create the `ShapeGeometry` in the same way that you create other built-in geometries: via the corresponding constructor, as shown the highlighted code in Listing 9.

Working with off-center rotation

To animate the door opening, the rectangular block that represents the door must be rotated 180 degrees around its hinges. Thus far, you have only rotated objects with their center as the origin. But the hinges are on one side of the door and not through its center.

To move the object away from the axis of rotation, you must pre-translate it by using a matrix. Matrix creation and manipulation APIs are built into Three.js and used heavily under the hood. The `applyMatrix` method of `Object3D` moves the object 1 unit in the *x* direction with respect to its current center:

```
geo.applyMatrix( new THREE.Matrix4().makeTranslation( 1, 0, 0 ) );  // move to hinge
```

Any future rotations will honor the new position of the center, which is now 1 unit to the left, where the conceptual door hinge should be located.

Animation with tweening

Simplifying tweening with tween.js

tween.js is a lightweight library for tweening that's often used in conjunction with WebGL libraries such as Three.js. The simple tween.js syntax is easy to learn for creating and managing multiple tweens in a scene.

In tweening, the generated intermediate values are interpolated among fixed points that you specify. This interpolation is linear by default, but most tweening engines also include an *easing* feature, whereby you can specify a rate-of-change control curve used during interpolation (increasing quadratic interpolation or decreasing quartic interpolation, for example). tween.js offers several dozen easing curves you can select for your tweens.

You can create complex animations by orchestrating the translation and rotation of objects over a period of time. Motions of multiple objects, including cameras and lights, can be coordinated to create complex sequences. And a motion's rate of change (slower, faster) can be varied to achieve dramatic effects.

Tweening is a cornerstone technique in the coding of animation sequences. In tweening, you control the numerical value of an object's property over time (the `position.z` property of a camera, for example) by specifying its desired value at fixed points in time. The tweening engine (code runtime) then generates the intermediate (in-be*tween*) values for you.

tween.js, a versatile tweening library, is used to create the walk-through of the fullscene.html example (see Simplifying tweening with tween.js sidebar). The following tween.js code translates a hypothetical director's instruction to "move into the room and stop short of the door, from 40 units to 15 units in an 8-second dolly-in":

```
var tweenWalkUp = new TWEEN.Tween(camera.position)
       .to({z: camera.position.z - 25}, 8000);
```

Lights, camera, action!

To code the walk-through, first plan the tweens required. In the walk-through sequence, you move the viewer into the scene by animating the camera's position on the z-axis. The viewer's trajectory is interposed by the door opening on its hinge.

The set of directions for this 16-second walk-through sequence is:

1. Walk up to the door, from z=40 to z=15, in 8 seconds.
2. Open the door and swing it around the hinges (180 degrees clockwise) in 3 seconds.
3. Walk into the room, from z=15 to z=8, in 5 seconds.

And the corresponding tweens in fullscene.html are coded as shown in Listing 10.

Listing 10. Corresponding tweens in fullscene.html

```
var tweenOpenDoor = new TWEEN.Tween( door.rotation )
   .to( { y: door.rotation.y - Math.PI }, 3000 );

var tweenWalkUp = new TWEEN.Tween(camera.position)
    .to({z: camera.position.z - 25}, 8000);

var tweenWalkIn = new TWEEN.Tween(camera.position)
    .to({z: camera.position.z - 32}, 5000);

  tweenOpenDoor.chain(tweenWalkIn);
  tweenWalkUp.chain(tweenOpenDoor);
  tweenWalkUp.start();
```

Notice how tweens are connected to one another using their `chain()` methods in the highlighted code. The `start()` method call starts the 16-second sequence.

To update the value on the tweened property before rendering each frame, you must also add a `TWEEN.update()` call in your rAF callback. In this case, it is in the `animate()` function shown in Listing 11.

Listing 11. `animate()` function

```
function animate() {
  requestAnimationFrame( animate );
  pyramid1.rotateY(Math.PI/180);
  sphere.rotateY(Math.PI/180);
  cube.rotateY(Math.PI/180);
  multi.rotateY(Math.PI/480);
  TWEEN.update();
  renderer.render(scene, camera);
}
```

Adding textures to objects in the scene

Texture mapping

Texture mapping (or *texturing*) is the art of applying bitmap graphics to geometries' surfaces to achieve visual effects. For example, the PNG photo of a piece of granite can be "mapped" — interpolated — onto the surface of a sphere (via an associated GLSL shader) to create what appears to be a granite sphere. Texturing is used extensively in 3D graphics to create photo-realistic objects. Three.js also supports more specialized texturing techniques such as *light mapping* and *bump mapping.* Light mapping gives you fine-grained control over the lighting level of surfaces of static objects. With bump mapping, you can render minute elevations and depressions on the surface of a geometry (think golf-ball dimples or mountain ranges on a globe's surface).

The flat colored walls, floor, and door are adequate for prototyping the scene. Final work often requires more realism. For instance, you might want to make the floor a wooden one and add some wallpaper to the walls. You'll do this with Three.js by using *texture mapping* (see the Texture mapping sidebar).

To see the texture-mapped scene with the walk-through, load up fulltexturescene.html in a browser. You'll notice a nice new sofa on the right. In fulltexturescene.html, the walk-through tween is extended with a pause for looking around the room and viewing the sofa. The new 28-second sequence is:

1. Walk up to the door, from z=40 to z=22 in 4 seconds.
2. Turn and look to the right (where the sofa is), turning 45 degrees in 3 seconds (initially fast, and slow a the end).
3. Turn and look to the left, turning 90 degrees in 6 seconds (initially slow, speed up at the end).
4. Turn back to the right 45 degrees in 3 seconds.
5. Continue to move close to the door, from z=22 to 15 in 4 seconds.
6. Open the door, swing it around the hinges (180 degrees clockwise) in 3 seconds.
7. Walk into the room, from z=15 to 8 in 5 seconds.

To see the code that renders this textured scene, view the fulltexturescene.html source code. Notice that to look around the room, the rotation property of the camera is modified by the `tweenLookAround`, `tweenLookAround2`, and `tweenLookAround3` tweens. For steps 2 (`tweenLookAround2`) and 3 (`tweenLookAround3`) in the sequence, tween.js's support for easing is used to create a turn that changes in speed over time. Figure 11 shows the fully textured scene as you enter the room.

Figure 11. Textured first room in the walk-through

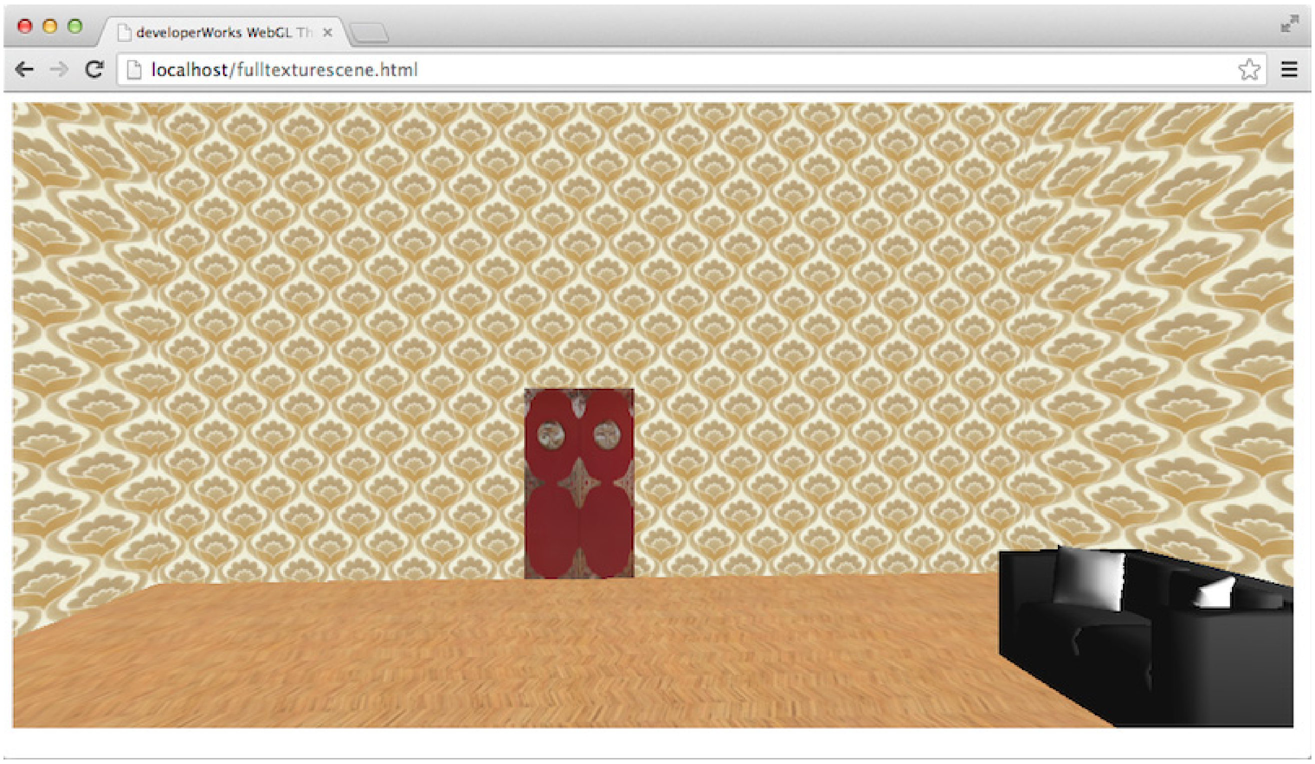
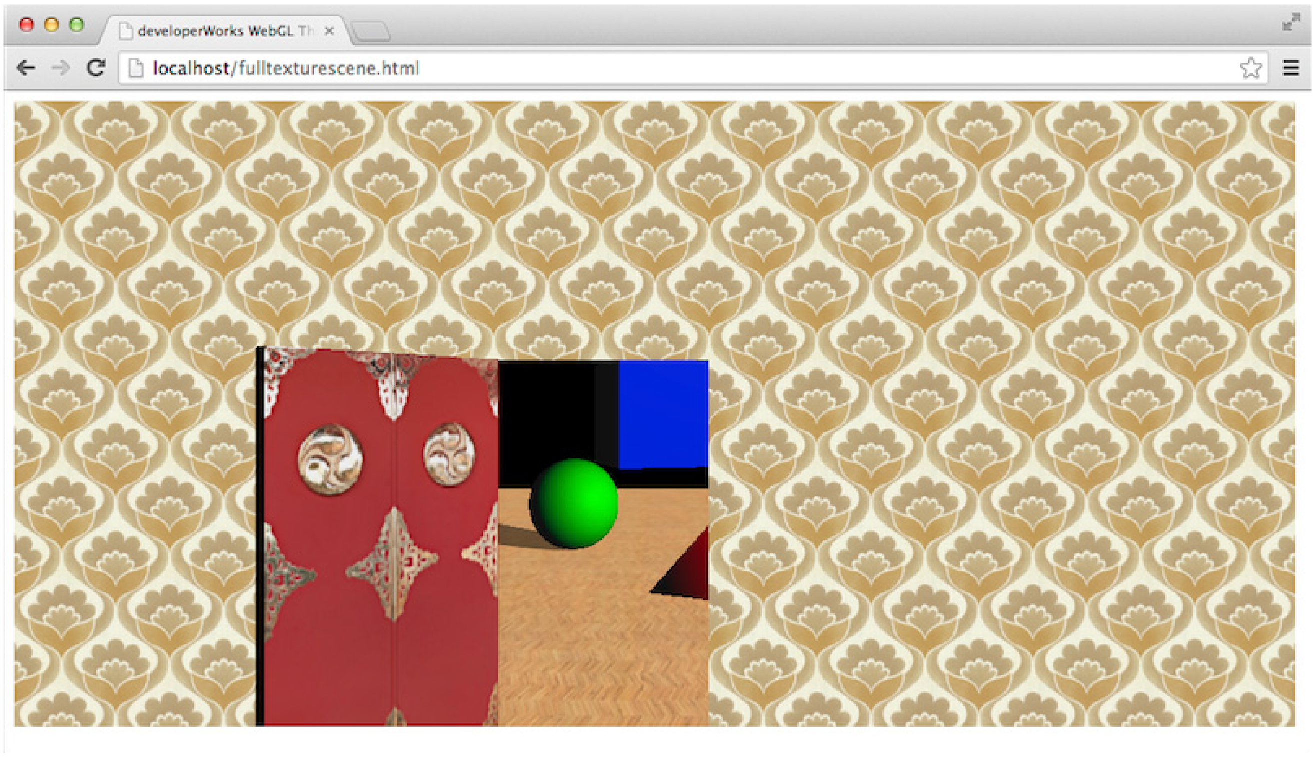


Figure 12 shows the walk-through as you approach the second room containing the spinning shapes, as the door is opening.

Figure 12. Opening the door, with view of the second room



Adding texture to the door

In fulltexturescene.html, the door is textured with the PNG file of a red door (door.jpg), which is a modified texture originally from CGTextures.com (see the 3D assets and repositories sidebar). The code that creates the texture and then texture maps the door:

```
doortex = THREE.ImageUtils.loadTexture('door.jpg');
...
geo = new THREE.CubeGeometry(2, 3.5, 0.2);
door = new THREE.Mesh(geo, new THREE.MeshPhongMaterial({map : doortex}));
```

3D assets and repositories

You'll eventually accumulate many custom textures, geometries, shapes, models, and tweens. All of these *assets* can be modified and reused for new 3D projects. In addition to creating your own 3D assets, you can acquire assets from online stores, warehouses, or repositories. Trimble 3D Warehouse is a popular warehouse for prefabricated 3D models. A popular membership-based asset repository specializing in textures is CGTextures.com.

First, you create the texture by using `THREE.ImageUtils.loadTexture()` to load the door.jpg file. Note that the loading is asynchronous (see the Asysnchronous loading with Three.js LoadingManager sidebar). When you select a bitmap for Three.js texturing, it's best to ensure that its dimensions are powers of 2 (such as 64, 128, 256, or 512); supported dimensions depend on the underlying WebGL GPU.

After the door texture is loaded, you set `THREE.MeshPhongMaterial()`'s `map` property, which creates the texture map. This property interpolates (stretches out) the bitmap to fit each side of the rectangular block. The door has six sides: the front, the back, and four edges. The texture on front and back looks fine. Because the same bitmap is used to texture the edge, if you look carefully, you can see that it's distorted. I've left it this way to keep the code simple.

Asynchronous loading with the Three.js LoadingManager

Browser loading of images and Three.js loading of models are both asynchronous nonblocking activities. Rendering fails if textures aren't fully loaded. Three.js's `LoadingManager` class manages asynchronous loading. An instance of `LoadingManager` can manage multiple asynchronous loaders by calling `onProgress()` for each item loaded and the `onLoad()` method after all pending loading is complete. A `THREE.DefaultLoadingManager` instance is used when a loader is instantiated without a specified `LoadingManager`.

Texturing a plane by tiling a bitmap

The texture used for the floor (floor.jpg) is a modified wood panel texture originally from CGTextures.com. The texture is repeated multiple times across the surface of the plane that represents the floor. The code to texture the floor:

Listing 12. Code to texture the floor

```
floortex = THREE.ImageUtils.loadTexture('floor.jpg');
...
floortex.wrapS = THREE.RepeatWrapping;
floortex.wrapT = THREE.RepeatWrapping;
floortex.repeat.x = 10;
floortex.repeat.y = 10;
```

Both side walls and the middle wall are all textured with the wallpaper texture (wall.jpg) using the same technique.

Loading prefabricated meshes or scenes

3D modeling tools and file formats

Professional 3D modelers work with software tools that enable them to create, texture, and animate 3D objects. Each tool typically saves models in a format optimized for its own operation. Often, tool vendors also offer converters that convert models from proprietary formats to other formats. For example, Wavefront saves in OBJ and MTL (for material) formats, and Trimble SketchUp can export models in Collada format. Three.js comes with a suite of optional model loaders that cater to different needs (see the js/loaders directory of the Three.js distribution). Three.js also supports the saving and loading of models in its own documented JSON format.

When you create 3D scenes, you'll often need to include prefabricated models or sub-scenes. They might be assets that other production team members created, ones that you reuse from previous projects, or ones that you acquire from a public repository or purchase from an asset store.

Three.js comes with a set of model loaders that can load models created by external 3D modeling tools such as Alias Wavefront or Trimble SketchUp (see the 3D modeling tools and file formats sidebar).

The sofa in the first room is a 3D model created by Bilal Hameed from the Trimble 3D Warehouse. This model was cleaned up in Trimble SketchUp and then exported in Collada format, in a file named sofawork.dae.

In fulltexturescene.html, the code in the `setup()` function loads the sofa, waits for it to be completely loaded, and places it into the scene, before starting the tweens. Listing 13 shows the relevant portion of the `setup()` function.

Listing 13. Loading and placing the sofa

```
function setup() {
    var cloader = new THREE.ColladaLoader();
    cloader.options.convertUpAxis = true;
    cloader.load( './sofawork.dae', function ( collada ) {
        sofa = collada.scene;
        sofa.position.set(5, -2, 16);
        scene.add(sofa);
        var newlight = new THREE.DirectionalLight(0xffffff, 0.5);
        newlight.position.set(5, 5,  16);
        scene.add(newlight);
        ...
```

The highlighted code in Listing 13 adds a directional light that points at the sofa to brighten up the model.

Another WebGL 3D Library: SceneJS

You're now acquainted with the major features of Three.js and understand the fundamental support APIs that a WebGL library provides. Not every WebGL library has the same APIs as Three.js, but almost all include similar fundamental support. For comparison, now you'll take a quick look at another popular WebGL library — SceneJS.

Comparing Three.js to SceneJS is like comparing apples to oranges in a sense. Both are time-tested, capable WebGL libraries, but they take completely different approaches to solving the data-management sub-problem for 3D rendering.

SceneJS takes a totally data-centric approach. You create a complete 3D scene in SceneJS by supplying a JSON-compatible JavaScript object that represents an acyclic scene graph tree. Listing 14 shows the scene graph tree (of SceneJS `nodes`) that represents the spinning cube, sphere, and pyramid.

Listing 14. SceneJS JSON of three meshes

```
var threeShapes =  [{
         type:"material",
        color:{ r:0.0, g:0, b:1.0 },

        nodes:[
            {
                type:"translate",
                y: 1,
                nodes: [
    {
                        type:"rotate",
                        id:"myRotate",
                        y:1.0,
                        angle:0,

                nodes:[
    {
                        type:"prims/box",
                        xSize: 1,
                        ySize: 1,
                        zSize: 1
    }
                       ]
                    }
                ]
            }
        ]


    }
    ,

    {
    type:"material",
        color:{ r:0.0, g:1.0, b:0 },

        nodes:[
            {
                type:"translate",
                x: 2.5,
                y: -1,
                nodes: [
                    {
                        type:"rotate",
                        id:"myRotate2",

                        y:1.0,
                        angle:0,

                        nodes:[

                            {
                                type:"prims/sphere",
                                slices: 25,
                                rings: 25,
                                radius: 1
                            }
                        ]
                    }
                ]
            }
        ]
    },

    {
    type:"material",
        color:{ r:1.0, g:0.0, b:0 },

        nodes:[
            {
                type:"translate",
                x: -2.5,
                y: -1,
        nodes: [

    {
                type:"rotate",
                id:"myRotate3",
                y:1.0,
               angle:0,

        nodes:[

                            {
                type:"prims/cylinder",
                radiusTop: 0,
                radiusBottom: 2,
                height: 2,
                radialSegments: 4,
                openEnded: true
                            }
                        ]
                    }
                ]
            }
        ]
    }
];
```

SceneJS core and plugins

To keep the core library source code and runtime footprint small, SceneJS makes extensive use of optional plugins. As part of configuration, you must specify the path to the plugin directory. Items that are primitives in other libraries (such as cube and sphere geometries in Three.js) are implemented as plugins in SceneJS.

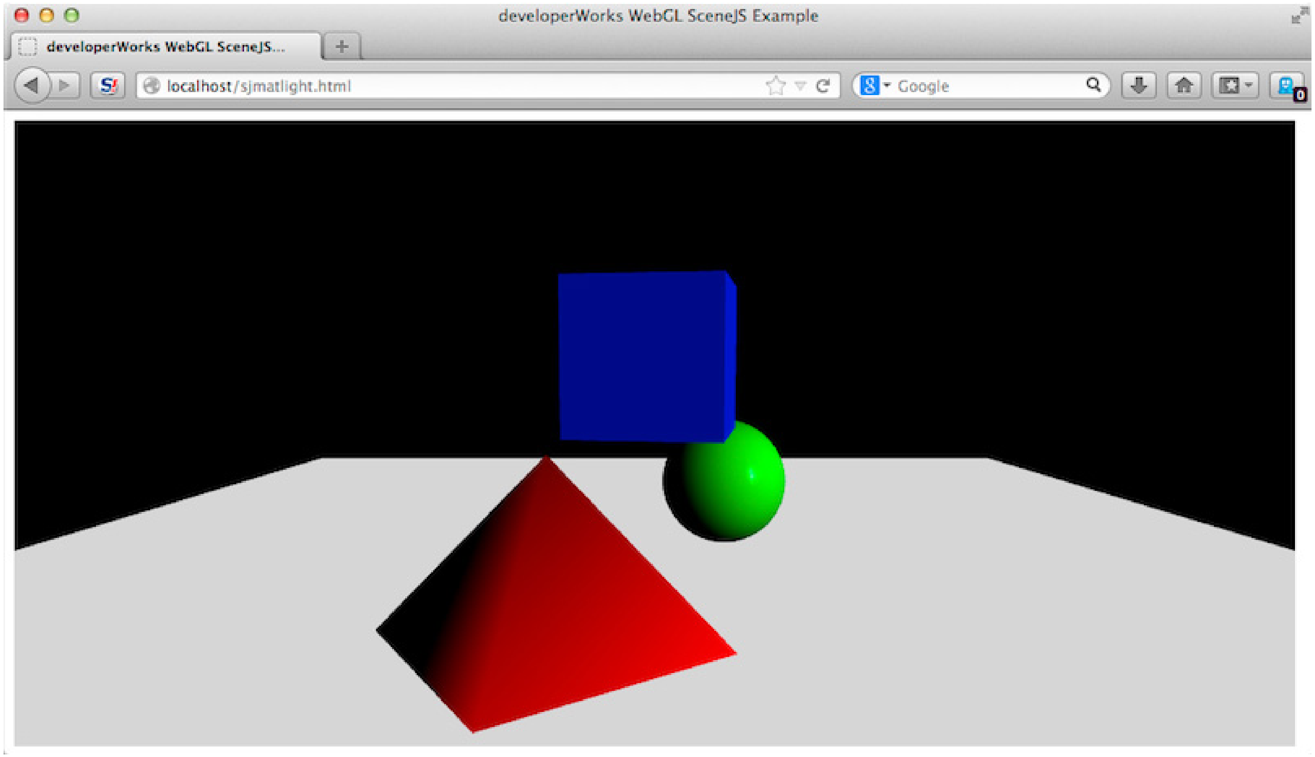
When SceneJS goes to work, it parses the scene graph tree and creates the scene within its runtime. Behaviors associated with each type of object (such as a mesh or a camera) are attached to the internal representation of the loaded tree. Use of *IDs* for each node of the tree enables any pre-labeled node to be addressed and manipulated dynamically. Because nodes are related to one another via parent-child relationships in a tree, changing the properties of upstream nodes can alter the behaviors or appearance of an entire sub-tree of nodes. If all of this sounds a lot like HTML pages being parsed into a browser's DOM and manipulated dynamically with a library such as jQuery, you've grasped the idea behind SceneJS.

By following this data-driven approach, SceneJS gains the ability to deal efficiently with large, complex, dynamic scene graphs that can be generated from one or more back-end data sources. What's more, SceneJS is network-friendly. Scene graphs and sub-graphs can be modified and updated "live" over the network through the transfer of snippets of JSON.

Recreating matlight.html in SceneJS

If you load sjmatlight.html, you'll see a reimplementation of the Three.js matlight.html example that uses SceneJS instead. sjmatlight.html is depicted in Figure 13. Compare it with the original matlight.html or with Figure 6.

Figure 13. sjmatlight.html using SceneJS


developerWorks WebGL SceneJS Example
developerWorks WebGL SceneJS...
localhost/sjmatlight.html
Google

matlight.html and sjmatlight.html operate almost identically. One difference that might be evident is that SceneJS has no intrinsic shadow effect. SceneJS specializes in engineering and medical applications where shadow effect is less important (than it is in, say, games or interior design applications).

Some primitives in SceneJS are loaded via optional plugins (see the SceneJS core and plugins sidebar). Listing 15 shows the configuration of the plugins directory's location.

Listing 15. Including SceneJS core and plugins

```
{
<!DOCTYPE html>
<html lang="en">
<head>
    <title>developerWorks WebGL SceneJS Example</title>
    <meta charset="utf-8">

    <script src="./scenejs.js"></script>

<script>
    SceneJS.setConfigs({
        pluginPath:"./plugins"
    });
    ...
```

Listing 16 shows the code in sjmatlight.html that creates the perspective camera for viewing the scene. Here again, you create a tree of nodes in which the group of three spinning objects is a child node of the camera.

Listing 16. Perspective camera in SceneJS

```
function setUp() {
   scene = SceneJS.createScene({
      canvasId: "shapecanvas2",
      nodes:[
          {
              type:"lookAt",
              eye:{ y:1, z:10 },
              look:{ x:0, y:0, z:0 },
              nodes:
           [
              { type:"camera",
                optics: {
                  type: "perspective",
                  fovy: 45.0,
                  aspect: 1024/500,
                  near: 0.10,
                  far : 100

                },
                nodes: [sceneNodes]
              }

            ]
          }
          ]
      });
```

A convenient function, `rotateObject()`, is defined to rotate the objects. Each rotatable object is tagged with an ID that's used for directly addressing during runtime. Listing 17 shows the `rotateObject()` function.

Listing 17. Object rotation by ID in SceneJS

```
   function rotateObject(id, degreeInc) {
    scene.getNode(id,  function (obj) {
                var angle = 0;
                scene.on("tick",
                  function () {
                        angle = angle + degreeInc;
                        obj.setAngle(angle);
                });
            });
   }
...
   rotateObject("cube1", 1);
   rotateObject("sphere1", 1);
   rotateObject("pyramid1", 1);
   rotateObject("multi", - 0.25);
```

Conclusion

Three.js and SceneJS are both mature and highly capable WebGL libraries, each with its own strengths and shortcomings. Three.js excels in general-purpose 3D development and offers an amazing array of primitives, effects, and model loaders. SceneJS is better suited to creating complex, dynamically changing scene graphs that are data-driven.

To say that libraries such as Three.js and SceneJS greatly simplify WebGL programming is an understatement. From this article and Part 1, it should be clear to you that raw WebGL development without the assistance of a library is an impractical activity. As a JavaScript developer, you are fortunate to have at your fingertips the fruits of the past four decades of 3D hardware and software R&D. A browser and a text editor are all that you need to be creative and productive with 3D and WebGL.

In Part 3, you'll start interacting with the user through 3D scenes, and you'll explore some possible applications.

Downloads

Description	Name	Size
Sample code	WebGL2dl.zip	1570KB

Resources

Learn

- WebGL: Visit the WebGL home page on the Khronos Group site, and read the latest working draft of the WebGL Specification.
- Three.js by mrdoob: Study the documentation, try the getting-started tutorial, and work through a variety of examples.
- SceneJS by Lindsay Kay @xeoLabs: Browse the examples, tutorials, documentation, and more.
- "Casting Curved Shadows on Curved Surfaces" (Lance Williams, 1978): This paper introduces an algorithm that uses z-buffer visible surface computation to display shadows.
- CGTextures.com: This membership-based online textures repository offers a vast selection of images specially prepared for 3D texturing.
- Trimble 3D Warehouse (formerly Google SketchUp 3D Warehouse): This online warehouse has thousands of prefabricated 3D models in a variety of file formats — many of them ready to use in Three.js scenes.
- WebGL Quick Reference: Take advantage of a handy cheat sheet for WebGL API syntax and concepts at a glance.
- Can I use WebGL?: This valuable site tracks up-to-date browser support for WebGL by versions.

Get products and technologies

- Check out Sing Li's 3D development with WebGL, Part 2: Code less do more with WebGL code libraries.
- Three.js: Download the Three.js library.
- sceneJS: Download sceneJS and plugins for it.
- tween.js by sole (Soledad Penadés of Mozilla Corp.): This lightweight tweening library works well with Three.js and also supports many easing functions.
- Trimble SketchUp (formerly Google SketchUp): This excellent 3D modeling tool features user-friendly 3D extrusion tools that are perfect for beginners starting out in 3D modeling.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Sing Li

Sing Li has been a developerWorks author since the site's inception, writing articles and tutorials that cover a variety of web and Java topics. He has more than two decades of system engineering experience, starting with embedded systems, crossing over to scalable enterprise systems, and now back full circle with web-scale mobile-enabled services and "Internet of things" ecosystems.