3D development with WebGL, Part 1: Introducing WebGL

Unleash the power of 3D hardware in your browser applications

Sing Li (westmakaha@yahoo.com)
Consultant
Makawave

10 December 2013

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 first in a three-part series for JavaScript developers who are new to WebGL. In this part, work through a basic example that demonstrates WebGL fundamentals and related 3D graphics concepts.

View more content in this series

3D hardware evolution: A brief history

In the early days of personal computing (circa early 1980s), 3D accelerated graphics hardware was available only on expensive specialized workstations from companies such as Silicon Graphics Computer Systems (later called SGI). These machines catered mostly to scientists and researchers in academia, government, and the defense industry and to entertainment industry pioneers. The graphics hardware in early conventional PCs had very low-resolution 2D capabilities; any 3D computation and drawing had to be performed offline — slowly — by application software.

In the late 1990s, affordable (but still relatively expensive) 3D accelerated graphics adapters from companies such as ATI Technologies Inc. (now AMD) and Nvidia Corp. started to become available to PC enthusiasts. These adapters implemented technological breakthroughs that brought workstation-quality graphics to the PC. But the GPUs in early 3D graphics adapters were limited by low display resolution and low rendering/processing power.

In the 1990s and early 2000s, the capabilities and performance of 3D accelerated adapters for PCs continued to improve, and prices continued to decline. GPU manufacturers' chips were applied not only in PCs but also in dedicated gaming systems and set-top boxes from vendors such as Nintendo and Sony.

By the middle of the last decade, mainstream CPU manufacturers such as Intel and AMD had started a competitive trend to integrate 3D rendering hardware into the main CPUs, or bundling GPU(s) with CPUs on the same chip. This trend dramatically improved the accessibility of 3D graphics to all users. For the first time in the PC's history, the cost of entry

to formerly workstation-grade 3D graphics on the PC was just the cost of downloading a supporting software driver.

We live in a 3D world, yet almost all of our interactions with computers and computerized devices occur over 2D user interfaces. High-speed, fluid, realistic 3D applications — at one time the exclusive domain of computer animators, scientific users, and gaming enthusiasts — were out of reach for mainstream PC users until relatively recently. (See the sidebar: 3D hardware evolution: A brief history.)

Today, all mainstream PC CPUs have built-in 3D graphics acceleration, and gaming PCs have additional dedicated high-performance graphics processing units (GPUs) to handle 3D rendering. This trend is reflected in the reduced instruction set computing (RISC)-based CPUs in phones and tablets. All current mobile CPUs include powerful 3D-capable graphics-acceleration GPUs. The supporting software drivers have also matured, and are now stable and efficient.

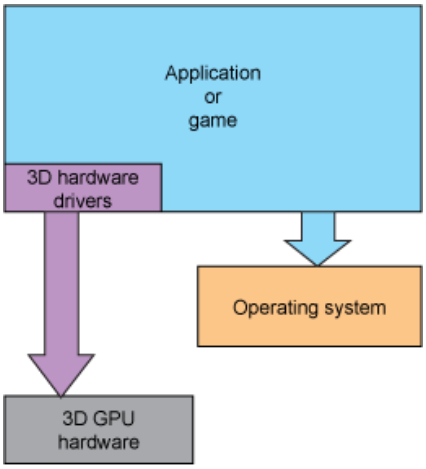
Advances in modern browser technology bring with them hardware-accelerated WebGL, a 3D JavaScript API that runs alongside feature-rich HTML5. JavaScript developers can now create interactive 3D games, applications, and 3D-enhanced UIs. With WebGL integrated into mainstream browsers, 3D application development is finally accessible to a huge population of developers armed simply with a browser and a text editor.

This article, the first in a three-part series, introduces WebGL. It starts with a brief overview of the evolution of the 3D software stack. Then you'll have a chance to experiment with the WebGL API through a hands-on example that covers key aspects of WebGL programming. (See Download for the sample code.) The example is complete yet easy to understand, with some essential 3D graphics concepts explained along the way. (Familiarity with the HTML5 canvas element is assumed.) Part 2 introduces high-level WebGL libraries, and in Part 3, you'll put everything together so you can start creating compelling 3D UIs and apps.

The 3D application software stack

For most of early PC history, 3D hardware drivers were bundled with, or compiled in, the application. This configuration optimizes access to the hardware-accelerated features of the hardware, resulting in the best possible performance. Essentially, you code directly to the hardware's capabilities. Well-designed games or computer-assisted design (CAD) applications can squeeze every ounce of juice out of the underlying hardware. Figure 1 shows this software configuration.

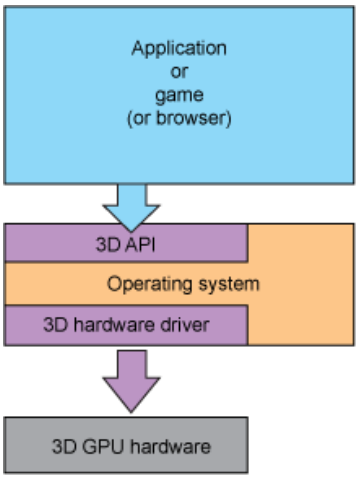
Figure 1. Application with embedded 3D hardware drivers



But the cost of bundling isn't trivial. Once the application is released and installed, the hardware driver is frozen in time — bugs and all. If the graphics-card vendor fixes a bug or introduces an enhanced-performance driver, the application user can't take advantage without installing or upgrading the application. Furthermore, because graphics hardware evolves rapidly, an application or game with a compiled-in 3D driver is prone to instant obsolescence. As soon as new hardware is introduced (with a new or updated driver), the software vendor must make and distribute new releases. This was a major distribution problem prior to the widespread accessibility of high-speed broadband networks

As a solution to the driver-update problem, the operating system took on the role of hosting the 3D graphics driver(s). The application or game calls an API that the OS provides, and the OS in turn converts the call into the primitives that the native 3D hardware driver accepts. Figure 2 illustrates this arrangement.

Figure 2. Application using the operating system's 3D API



In this way (at least in theory) an application can be programmed to the OS's 3D APIs. The application is shielded from changes to the 3D hardware driver and even from the evolution of the 3D hardware itself. For many applications, including all mainstream browsers, this configuration

worked adequately for a long time. The OS acted as a middleman that tried valiantly to cater to all types or styles of applications, and to graphics hardware from competing vendors. But this one-size-fits-all approach takes its toll in terms of 3D rendering performance. Applications that require the best hardware-acceleration performance still must discover the actual installed graphics hardware, implement tweaks to optimize code for each set of hardware, and often be programmed to vendor-specific extensions to the OS's API — again making applications hostage to the underlying drivers or the physical hardware.

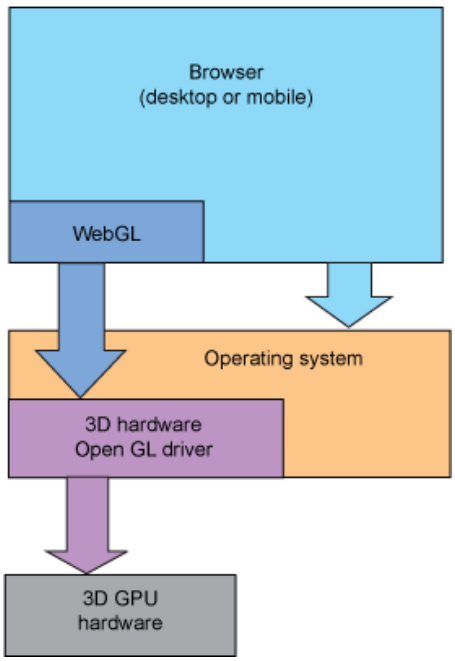
WebGL era

Enter the modern age, with high-performance 3D hardware built into every desktop and mobile device. Applications are increasingly developed with JavaScript to leverage browser capabilities, and web designers and web application developers clamored for faster and better 2D/3D browser support. The result: wide support of WebGL by mainstream browser vendors.

WebGL is based on OpenGL Embedded System (ES), which is a low-level procedural API for accessing 3D hardware. OpenGL — created in the early 1990s by SGI — is now considered a well-understood and mature API. WebGL gives JavaScript developers near-native-speed access to the 3D hardware on a device for the first time in history. Both WebGL and OpenGL ES are evolving under the auspices of the nonprofit Khronos Group.

WebGL APIs get almost direct access to the underlying OpenGL hardware driver, without the penalty of code being translated first through the browser support libraries and then the OS's 3D API libraries. Figure 3 illustrates this new model.

Figure 3. JavaScript application accessing 3D hardware through WebGL



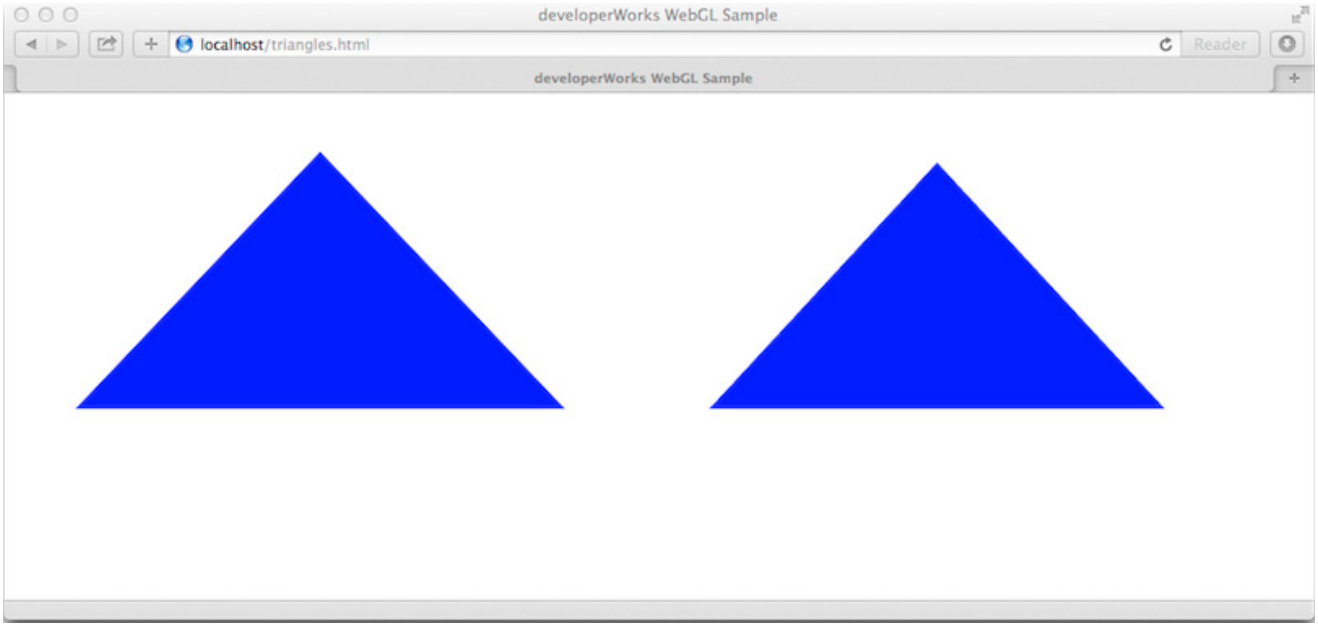
Hardware-accelerated WebGL enables 3D gaming on browsers, real-time 3D data visualization applications, and futuristic interactive 3D UIs — to name just a few possibilities. The

standardization of OpenGL ES ensures that new vendor drivers can be installed without affecting existing WebGL-based applications, delivering on the utopian "any 3D hardware on any platform" promise.

Hands-on WebGL

Now it's time to get hands-on with WebGL. Fire up the latest version of Firefox, Chrome, or Safari and open triangles.html from the code (see Download). The page should look similar to the screen capture in Figure 4, which is from Safari running on OS X.

Figure 4. The triangles.html page

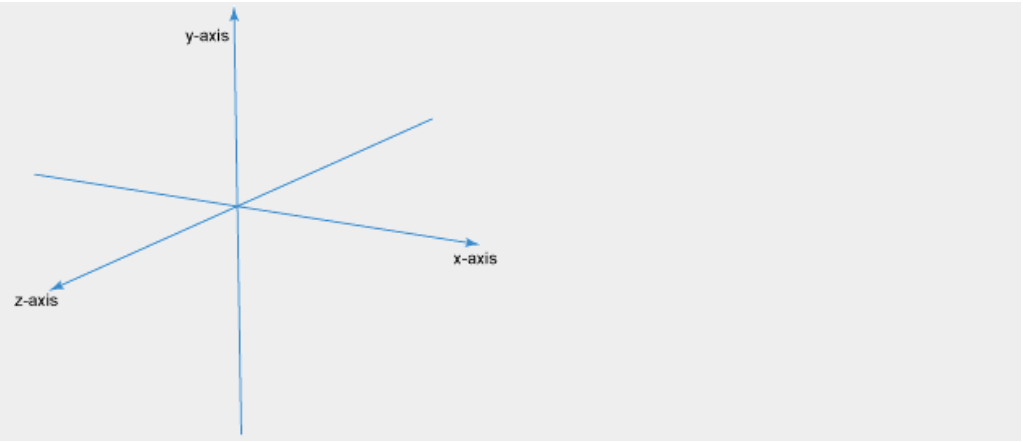


The WebGL 3D axis

The WebGL 3D axis comprises:

- An x-axis going from left (negative) to right (positive)
- A y-axis going from bottom (negative) to top (positive)
- A z-axis going from behind the page (negative) to front of the page (positive)

As you look at the WebGL 3D axis in the image below, imagine yourself standing at the arrowhead of the z-axis pointer and looking down the z-axis to the x-y plane:



Two seemingly identical blue triangles appear on the page. However, not all triangles are created equal. Both triangles are drawn with the HTML5 canvas. But the one on the left is 2D and is drawn in fewer than 10 lines of JavaScript code. The one on the right is a four-sided 3D pyramid object that takes more than 100 lines of JavaScript WebGL code to render.

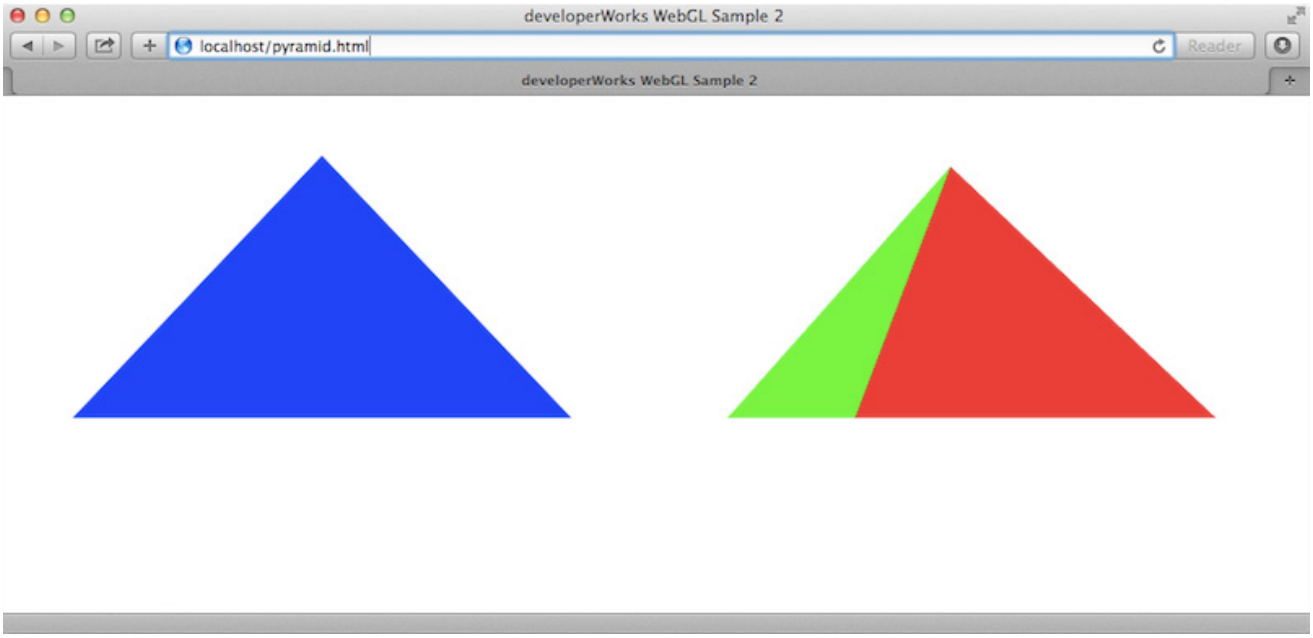
If you view the page's source code, you can confirm that a large volume of WebGL code draws the triangle on the right. However, that triangle certainly doesn't appear to be in 3D. (No, putting on your red-blue 3D glasses won't help.)

WebGL draws 2D views

You see a triangle on the right side of triangles.html because of the orientation of the pyramid. You're looking at one blue side of a multicolored pyramid — analogous to looking directly at one side of a building and seeing only a 2D rectangle. (Take a sneak peek at Figure 5 to see the pyramid in 3D.) This realization reinforces the essence of working with 3D graphics in the browser: The final output is always a 2D view of a 3D scene. Therefore, any static rendering of a 3D scene by WebGL is a 2D image.

Next, load pyramid.html in your browser. On this page, the code to draw the pyramid is almost exactly the same as it is in triangles.html. The one difference is that some code is added to rotate the pyramid along the y-axis continuously. In other words, multiple 2D views of the same 3D scene are drawn (using WebGL) in succession with a time delay. As the pyramid rotates, you can see clearly that the former blue triangle on the right side of triangles.html is indeed one side of a multicolored 3D pyramid. Figure 5 shows a snapshot of pyramid.html in Safari on OS X.

Figure 5. Rotated 3D pyramid on the pyramid.html page



Writing WebGL code

Listing 1 shows the HTML code for the two canvas elements in triangles.html.

Listing 1. HTML code with two canvas elements

```
<html>
<head>
...
</head>
  <body onload="draw2D();draw3D();">
    <canvas id="shapecanvas" class="front" width="500" height="500">
    </canvas>
    <canvas id="shapecanvas2" style="border: none;" width="500" height="500">
    </canvas>
  <br/>
  </body>
</html>
```

The `onload` handler calls two functions: `draw2D()` and `draw3D()`. The `draw2D()` function draws in 2D on the canvas on the left (`shapecanvas`). The `draw3D()` function draws in 3D on the canvas on the right (`shapecanvas2`).

The code that draws the 2D triangle on the left canvas is in Listing 2.

Listing 2. Drawing the 2D triangle on an HTML5 canvas

```
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);        // Top Corner
    c2dCtx.lineTo(450, 250);         // Bottom Right
    c2dCtx.lineTo(50, 250);         // Bottom Left
    c2dCtx.closePath();
    c2dCtx.fill();

}
```

In the straightforward 2D drawing code in Listing 2, a drawing context `c2dCtx` is obtained from the canvas. Then the context's drawing methods are called to create a set of paths that trace the triangle. Finally the closed path is filled with the RGB color `#0000ff` (blue).

Obtaining a 3D WebGL drawing context

Listing 3 shows that the code for obtaining a 3D drawing context from a canvas element is almost the same as in the 2D case. The difference is that the context name to request is `experimental-webgl`, instead of `2d`.

Listing 3. Obtaining a WebGL 3D context from a canvas element

```
function draw3D()  {
      var canvas = document.getElementById("shapecanvas2");

      var glCtx = null;
      var exmsg = "WebGL not supported";
      try
      {
          glCtx = canvas.getContext("experimental-webgl");
      }
      catch (e)
      {
          exmsg = "Exception thrown: " + e.toString();
      }
      if (!glCtx)
      {
          alert(exmsg);
          throw new Error(exmsg);
      }
 ...
```

In Listing 3, the `draw3D()` function shows an alert and raises an error if WebGL isn't supported by the browser. In production applications, you might want to handle this situation by using code that's more application-specific.

Setting the viewport

To tell WebGL where the rendered output should go, you must set the viewport by specifying, in pixels, the area within the canvas that WebGL can draw to. In triangles.html, the entire canvas area is used to render the output:

```
// set viewport
 glCtx.viewport(0, 0, canvas.width, canvas.height);
```

In the next step, you must start creating data to feed to the WebGL rendering pipeline. This data must describe the 3D objects that comprise the scene. In the case of the example, the scene is just a single four-sided multicolored pyramid.

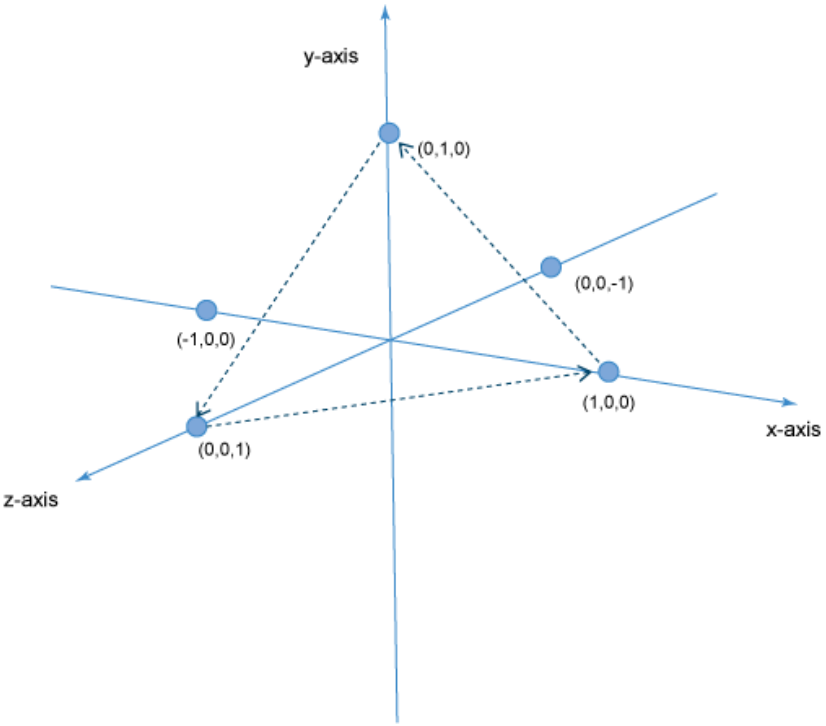
Describing 3D objects

Meshes in 3D rendering

With enough triangles put together, you can approximate an object of any shape. The collection of triangles in three dimensions approximating an object is referred to as a *mesh.* In modern games and 3D applications, it's not unusual to find meshes consisting of thousands of triangles.

To describe a 3D object for WebGL rendering, you must represent the object by using triangles. WebGL can take the description in the form of a set of discrete triangles, or as a strip of triangles with shared vertices. In the pyramid example, the four-sided pyramid is described in a set of four distinct triangles. Each triangle is specified by its three vertices. Figure 6 shows the vertices for one of the pyramid's sides.

Figure 6. Vertices describing one side of the pyramid



In Figure 6, the side's three vertices are `(0,1,0)` on the y-axis, `(0,0,1)` on the z-axis, and `(1,0,0)` on the x-axis. On the pyramid itself, this side is colored yellow and is the side to the right of the visible blue side. Extending the same pattern, you can follow and sketch out the other three sides of the pyramid. The code in Listing 4 defines the four sides of the pyramid in the array named `verts`.

Listing 4. Vertex array describing the set of triangles that makes up the pyramid

```
// Vertex Data
vertBuffer = glCtx.createBuffer();
glCtx.bindBuffer(glCtx.ARRAY_BUFFER, vertBuffer);
var verts = [
0.0, 1.0, 0.0,
-1.0, 0.0, 0.0,
0.0, 0.0, 1.0,

0.0, 1.0, 0.0,
0.0, 0.0, 1.0,
1.0, 0.0, 0.0,

0.0, 1.0, 0.0,
1.0, 0.0, 0.0,
0.0, 0.0, -1.0,

0.0, 1.0, 0.0,
0.0, 0.0, -1.0,
-1.0, 0.0, 0.0

];
glCtx.bufferData(glCtx.ARRAY_BUFFER, new Float32Array(verts),
   glCtx.STATIC_DRAW);
```

Note that the bottom of the pyramid (which is actually a square on the x-z plane) is not included in the `verts` array. Because the pyramid is rotated around the y-axis, the viewer can never see the bottom. It's customary in 3D work not to render the faces of objects that the viewer never sees. Leaving them unrendered can significantly speed up the rendering of complex objects.

JavaScript — Float32Array type array

The *typed array*, such as `Float32Array`, is a relatively new browser-supported JavaScript data type. A typed array represents a contiguous area of memory (a JavaScript `ArrayBuffer`) that contains a sequence of identically formatted binary data elements. A typed array is useful for manipulating raw low-level binary buffered data through JavaScript with great efficiency. It's also useful for passing large volumes of data, such as matrices used in WebGL programming, between JavaScript application code and low-level OpenGL drivers. Essentially, a typed array is a C-style array manipulated through JavaScript APIs. Typed arrays were introduced to enable JavaScript to work directly with formatted binary buffer data for media- and graphics-processing applications.

In Listing 4, the data in the `verts` array is packed into a binary-formatted buffer that can be efficiently accessed by the 3D hardware. This is all done through JavaScript WebGL calls: First, a new zero-sized buffer is created with the WebGL `glCtx.createBuffer()` call and bound to the `ARRAY_BUFFER` target at the OpenGL level with the `glCtx.bindBuffer()` call. Next, the array of data values to be loaded is defined in JavaScript, and the `glCtx.bufferData()` call sets the size of the currently bound buffer and packs the JavaScript data (first converting the JavaScript array into `Float32Array` binary format) into the device-driver-level buffer.

The result is a `vertBuffer` variable that references a hardware-level buffer that contains the required vertex information. The data in this buffer can be directly and efficiently accessed by other processors in the WebGL rendering pipeline.

Specifying the colors of the pyramid's sides

The next low-level buffer that must be set up is referenced by `colorBuffer`. This buffer contains the color information for each of the pyramid's sides. In the example, the colors are blue, yellow, green, and red. Listing 5 shows how `colorBuffer` is set up.

Listing 5. Setup of `colorBuffer` specifying colors for the sides of the pyramid

```
colorBuffer = glCtx.createBuffer();
        glCtx.bindBuffer(glCtx.ARRAY_BUFFER, colorBuffer);
        var faceColors = [
            [0.0, 0.0, 1.0, 1.0], // front  (blue)
            [1.0, 1.0, 0.0, 1.0], // right  (yellow)
            [0.0, 1.0, 0.0, 1.0], // back   (green)
            [1.0, 0.0, 0.0, 1.0], // left   (red)
        ];
        var vertexColors = [];
        faceColors.forEach(function(color) {
            [0,1,2].forEach(function () {
              vertColors = vertColors.concat(color);
            });
        });        glCtx.bufferData(glCtx.ARRAY_BUFFER,
         new Float32Array(vertexColors), glCtx.STATIC_DRAW);
```

In Listing 5, the low-level `colorBuffer` buffer is set up via the `createBuffer()`, `bindBuffer()`, and `bufferData()` calls, which are identical to those used for the `vertBuffer`.

But WebGL has no notion of the "side" of a pyramid. Instead, it works only with triangles and vertices. The color data must be associated with a vertex. In Listing 5, an intermediate JavaScript array named `faceColors` initializes the `vertColors` array. `vertColors` is the JavaScript array used in loading the low-level `colorBuffer`. The `faceColors` array contains four colors — blue, yellow, green, and red — corresponding to the four sides. These colors are specified in red, green, blue, alpha (RGBA) format.

The `vertColors` array contains a color for each vertex of every triangle, in the order that corresponds to their appearance in the `vertBuffer`. Because each of the four triangles has three vertices, the final `vertColors` array contains a total of 12 color entries (each of which is an array of four `float` numbers). A nested `forEach` loop is used to assign the same color to each of the three vertices of each triangle that represents a side of the pyramid.

Understanding OpenGL shaders

A question that might naturally come to mind is how specifying a color for the three vertices of a triangle renders the entire triangle in that color. To answer this question, you must understand the operation of two programmable components in the WebGL rendering pipeline: the *vertex shader* and the *fragment* (*pixel*) *shader*. These shaders can be compiled into code that can be executed on the 3D acceleration hardware GPU. Some modern 3D hardware can execute hundreds of shader operations in parallel for high-performance rendering.

A vertex shader executes for each specified vertex. The shader takes input such as color, location, texture, and other information associated with a vertex. Then the shader computes and transforms the data to determine the 2D location on the viewport where that vertex should be rendered, as

well as the vertex's color and other attributes. A fragment shader determines the color and other attributes of each pixel that comprises the triangle between the vertices. You program both the vertex shader and the fragment shader with OpenGL Shading Language (GLSL) via WebGL.

GLSL

GLSL — a programming language with a syntax similar to ANSI C (with some C++ concepts) — is domain-specific for mapping from the available object shape, location, perspective, color, lighting, texture, and other associated information to the actual color that will be displayed for each 2D canvas pixel where the 3D object will be rendered.

Details on writing your own shader program using GLSL are beyond this article's scope. But you need a minimal understanding of GLSL code to understand the rest of the example program. I'll guide you through the operation of the two trivial GLSL shaders used in the example to help you make sense of all the code that surrounds them.

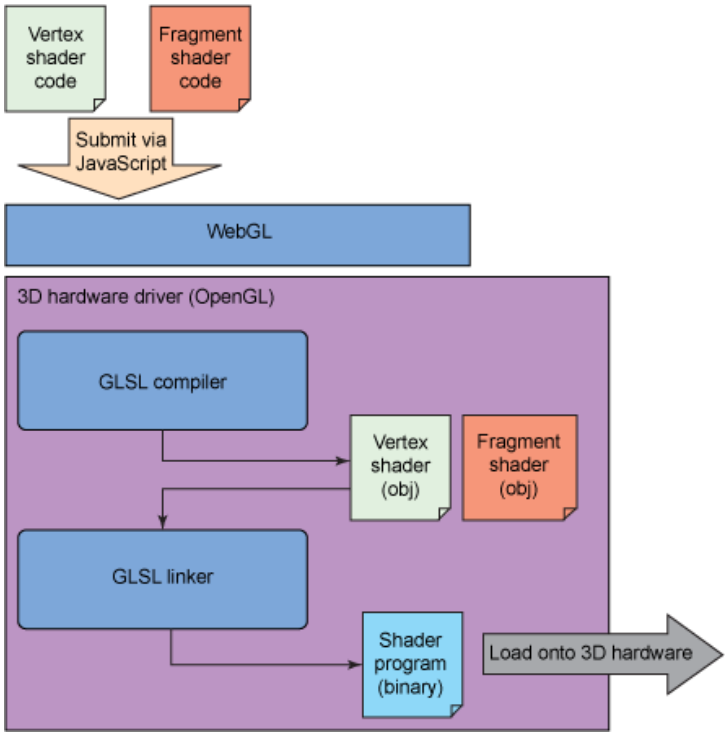
In the next article in this series, you'll learn how to use higher-level 3D libraries and frameworks to work with WebGL. Those libraries and frameworks incorporate GLSL code transparently, so you might never need to write a shader on your own.

Processing a shader program in WebGL

A shader program is a linked binary of related shaders (typically, the vertex shader and fragment shader in WebGL) ready to be executed by the hardware GPU. Each shader can range from a nearly trivial one-liner to hundreds of lines of highly complex, multifeatured, parallel code.

Before you can execute a shader with WebGL, the program's GLSL source code must be compiled into binary code and then linked together. The vendor-supplied 3D driver embeds the compiler and linker. You must submit the GLSL code via JavaScript, check for compilation errors, then link the matrices you prepared as parameters. WebGL has an API for all of these operations. Figure 7 illustrates the submission of GLSL code through WebGL.

Figure 7. GLSL shader code compilation and linking through WebGL



The code for obtaining, compiling, and linking the example's GLSL shaders is shown in Listing 6.

Listing 6. Compiling and linking GLSL shader code in WebGL

```
var vertShaderCode = document.getElementById("vertshader").textContent;
var fragShaderCode = document.getElementById("fragshader").textContent;

var fragShader = glCtx.createShader(glCtx.FRAGMENT_SHADER);

glCtx.shaderSource(fragShader, fragShaderCode);
glCtx.compileShader(fragShader);

if (!glCtx.getShaderParameter(fragShader, glCtx.COMPILE_STATUS)) {
   var errmsg = "fragment shader compile failed: "
    + glCtx.getShaderInfoLog(fragShader);
   alert(errmsg);
   throw new Error()
  }

var vertShader = glCtx.createShader(glCtx.VERTEX_SHADER);


glCtx.shaderSource(vertShader, vertShaderCode);
glCtx.compileShader(vertShader);

if (!glCtx.getShaderParameter(vertShader, glCtx.COMPILE_STATUS)) {
   var errmsg = "vertex shader compile failed : "
       + glCtx.getShaderInfoLog(vertShader);
   alert(errmsg);
   throw new Error(errmsg)
  }


// link the compiled vertex and fragment shaders
shaderProg = glCtx.createProgram();
glCtx.attachShader(shaderProg, vertShader);
```

```
glCtx.attachShader(shaderProg, fragShader);
glCtx.linkProgram(shaderProg);
```

In Listing 6, the source code of the vertex shader is stored as a string in `vertexShaderCode`, and the fragment shader source code is stored in `fragmentShaderCode`. Both sets of source code are extracted from `<script>` elements in the DOM by means of the `document.getElementById().textContent` property.

A vertex shader is created by `glCtx.createShader(glCtx.VERTEX_SHADER)`. A fragment shader is created by `glCtx.createShader(glCtx.FRAGMENT_SHADER)`.

The source code is loaded into the shader using `glCtx.shaderSource()`, then the source code is compiled via `glCtx.compileShader()`.

After compilation, the `glCtx.getShaderParameter()` is called to ensure that compilation was successful. Compilation errors can be fetched from the compiler log via `glCtx.getShaderInfoLog()`.

After both the vertex shader and fragment shader are compiled successfully, they're linked together to form an executable shader program. First, `glCtx.createProgram()` is called to create the low-level program object. Then the compiled binaries are associated with the program using `glCtx.attachShader()` calls. Finally, the binaries are linked together by a call to `glCtx.linkProgram()`.

Vertex and fragment shader GLSL code

The vertex shader operates on input data buffers that are prepared earlier in the JavaScript `vertBuffer` and `colorBuffer` variables. Listing 7 shows the GLSL source code for the vertex shader.

Listing 7. Vertex shader GLSL source code

```
attribute vec3 vertPos;
attribute vec4 vertColor;
uniform mat4 mvMatrix;
uniform mat4 pjMatrix;
varying lowp vec4 vColor;
void main(void) {
  gl_Position = pjMatrix * mvMatrix * vec4(vertPos, 1.0);
  vColor = vertColor;
}
```

In Listing 7, the `attribute` keyword is a storage qualifier that specifies the linkage between WebGL and a vertex shader for per-vertex data. In this case, `vertPos` contains a vertex position from the `vertBuffer` each time the shader is executed. And `vertColor` contains the color of that vertex as specified in the `colorBuffer` you set up earlier.

The `uniform` storage qualifier specifies values set in JavaScript and used as read-only parameters within the shader code. The values in these buffers can be changed (especially during animation) by the JavaScript code, but they're never changed by any of the shader code. To put it another way: They can be changed only by the CPU and never by the rendering GPU. After they're set,

`uniform` values are the same for every vertex processed by the vertex shader code. In this case, the `mvMatrix` contains the *model view matrix* setup from JavaScript, and the `pjMatrix` contains the *projection matrix*. (I cover model view matrix and projection matrix in the next section.)

Interpolation as a pattern in 3D development

Interpolation is a conceptual pattern you will see again and again in 3D graphics work and animation. The idea is to specify attributes or behaviors at fixed points in time or space, then let some mathematical computation take care of filling in all the intermediate points. The developer or animator can then control continuous change of attributes and behaviors — which otherwise usually requires the specification of a huge quantity of data points — with very few data points.

The `lowp` keyword is a precision qualifier. It specifies that the `vColor` variable is a low-precision `float` number, which is adequate for describing a color in the WebGL colorspace. `gl_position` is the transformed output value of the shader, used internally by the 3D rendering pipeline for further processing.

The `vcolor` variable has a `varying` storage qualifier. `varying` indicates that this variable is used to interface between the vertex and the fragment shader. Although `vcolor` has a unique single value per vertex, its value in the fragment shader is interpolated among the vertices. (Recall that fragment shader is executed for pixels that are between the vertices.) In this article's GLSL example, the `vColor` variable is set to be the color specified in the `colorBuffer` for each vertex in the vertex shader. Listing 8 shows the fragment shader code.

Listing 8. The GLSL fragment shader source code

```
varying lowp vec4 vColor;
 void main(void) {
   gl_FragColor = vColor;
 }
```

The fragment shader is trivial. It takes the interpolated `vColor` value from the vertex shader and uses it as the output. Because the `vColor` value is set to be the same for each of the three vertices of each side of the pyramid, the interpolated color used by the fragment remains the same color.

You can see the effect of interpolation by ensuring that at least one vertex of each triangle has a different color. Try modifying pyramid.html with the code in bold shown in Listing 9.

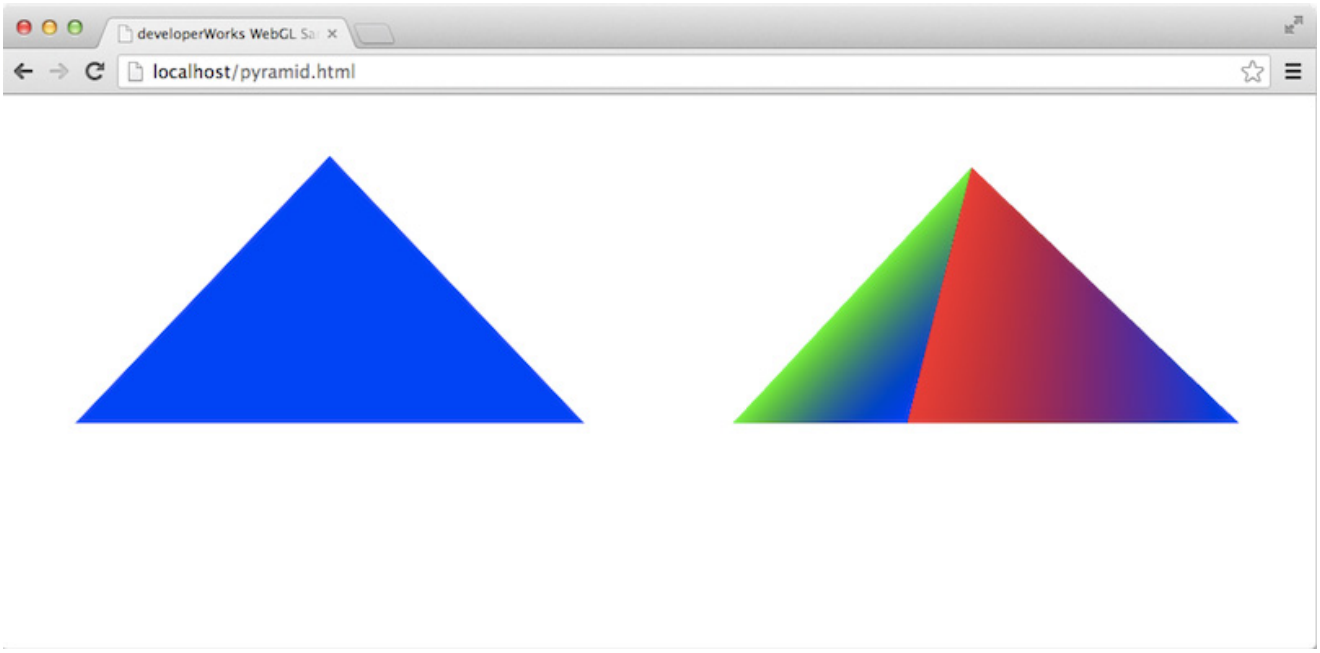
Listing 9. Modifying pyramid.html to show fragment shader interpolation

```
var vertexColors = [];
faceColors.forEach(function(color) {
  [0,1].forEach(function () {
      vertColors = vertColors.concat(color);
   });
   vertColors = vertColors.concat(faceColors[0]);
     });
glCtx.bufferData(glCtx.ARRAY_BUFFER,
  new Float32Array(vertexColors), glCtx.STATIC_DRAW);
```

This modification ensures that one vertex in each triangle has the color blue. Load the modified pyramid.html in a browser. Now you see a color gradient (interpolated color) on all of the pyramid

sides, except for the blue side (whose vertices are still all blue). Figure 8 shows the pyramid sides with interpolated color gradients, in Chrome on OS X.

Figure 8. pyramid.html with pyramid sides modified with interpolated color gradients



Model view and projection matrix

Fast JavaScript matrix manipulation

Working with 3D transformations frequently requires multiplication of 4x4 matrices, which requires skillful coding and tuning if it's to be done efficiently in JavaScript. Fortunately, optimized JavaScript matrix libraries written for WebGL are available. The example uses the `glMatrix` library for matrix manipulation.

To control the transformation of the 3D scene rendered on the canvas, you specify two matrices: the model view and the projection matrix. Earlier, you saw the vertex shader uses them to determine how to transform each 3D vertex.

The model view matrix combines the transformation of the model (the pyramid in this case) and the view (the "camera" through which you view the scene). Basically, the model view matrix controls where to place the objects in the scene and the viewing camera. This code sets up the model view matrix in the example, placing the pyramid three units away from the camera:

```
modelViewMatrix = mat4.create();
mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -3]);
```

You know from `vertBuffer` setup that the pyramid is two units wide, so the preceding code enables the pyramid to "fill the frame" of the viewport.

The projection matrix controls the transformation of the 3D scene through the camera's view onto the 2D viewport. The projection matrix setup code from the example is:

```
projectionMatrix = mat4.create();
mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width / canvas.height, 1, 100);
```

The camera is set to have a `Math.PI / 4` (pi radians divided by 4, or 180 degrees / 4 = 45-degree) field of view. The camera can see things as close as 1 unit away and as far as 100 units away while maintaining a perspective (undistorted) view.

Rendering the 3D scene in the viewport

With all the setup complete, the code in Listing 10 renders the 3D scene as a 2D view in the viewport. This code is contained in the `draw()` function called by the `update()` wrapper. The `update()` wrapper makes it easier to convert the code for pyramid rotation later in pyramid.html.

Listing 10. The `draw()` function to render the scene

```
function draw(ctx) {
  ctx.clearColor(1.0, 1.0, 1.0, 1.0);
  ctx.enable(ctx.DEPTH_TEST);
  ctx.clear(ctx.COLOR_BUFFER_BIT  | ctx.DEPTH_BUFFER_BIT);
  ctx.useProgram(shaderProg);
  ctx.bindBuffer(ctx.ARRAY_BUFFER, vertBuffer);
  ctx.vertexAttribPointer(shaderVertexPositionAttribute, 3 ,
     ctx.FLOAT, false, 0, 0);
  ctx.bindBuffer(ctx.ARRAY_BUFFER, colorBuffer);
  ctx.vertexAttribPointer(shaderVertexColorAttribute, 4 ,
     ctx.FLOAT, false, 0, 0);
  ctx.uniformMatrix4fv(shaderProjectionMatrixUniform,
      false, projectionMatrix);
  mat4.rotate(modelViewMatrix, modelViewMatrix,
      Math.PI/4, rotationAxis);
  ctx.uniformMatrix4fv(shaderModelViewMatrixUniform, false,
      modelViewMatrix);
  ctx.drawArrays(ctx.TRIANGLES, 0, 12 /* num of vertex */);
}
```

Object placement in 3D space

To be proficient in 3D development, you need to get used to placing objects in 3D space by specifying their x-y-z coordinates. In this article's simple example, you only need to place the camera and the pyramid (mesh). In more-complex 3D projects, you need to place multiple 3D objects (meshes) to create a *scene*. You might also place one or more *lights* in the scene to add dramatic effects. (This article uses defaults to keep the math and the code simple.) For animation, you might need to place one or more *paths* into the scene to control the motion of the 3D meshes.

In Listing 10, the `ctx.clearColor()` call clears the viewport to white and `ctx.enable(ctx.DEPTH_TEST)` ensures that the depth buffer (*z-buffer*) is enabled. The call to `ctx.clear(ctx.COLOR_BUFFER_BIT | ctx.DEPTH_BUFFER_BIT)` clears the color and depth buffers.

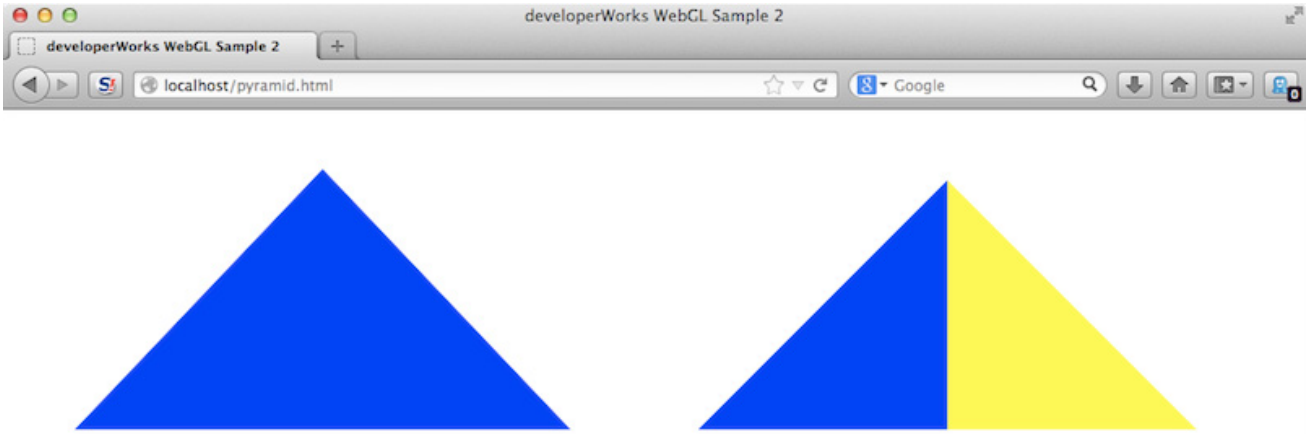
The `shaderProg`, consisting of the `vertShader` and `fragShader` you compiled and linked earlier is loaded for GPU execution by the `ctx.useProgram()` call. Next, the low-level data buffers (`vertBuffer` and `colorBuffer`) you set up earlier in JavaScript are now bound to the attributes of the GLSL shader program through a series of `ctx.bindBuffer()` and `ctx.vertexAttribPointer()` calls. (This workflow is similar in concept to stored procedures in SQL programming, whereby parameters are bound at runtime and the prepared statement can be re-executed.) The

`ctx.uniformMatrix4fv()` calls sets up the model view and the projection matrix for read-only access by the vertex shader.

Last but not least, the `ctx.drawArrays()` call renders the set of four triangles — a total of 12 vertices — to the viewport.

You might notice the `mat4.rotate(modelViewMatrix, modelViewMatrix, Math.PI/4, rotationAxis)` call that appears immediately before the model view matrix for vertex shader access is set up. This call rotates the pyramid by `Math.PI/4` or 45 degrees around the y-axis just before rendering. If you look back at Figure 6, the reason will become obvious. Notice that the setup of the pyramid's sides places an edge of the pyramid on the z-axis. Imagine the camera on the z-axis pointing into the screen; what it will see without rotating the pyramid is half of the blue side and half of the yellow side. Rotating the pyramid by 45 degrees ensures that only the blue side is visible. You can easily view this effect by commenting out the `mat4.rotate()` call and loading the page again. Figure 9 shows the result (note that the vertex color interpolation code change has been reverted in this version), in Firefox on OS X.

Figure 9. pyramid.html with two pyramid sides showing before rotation



Animating the pyramid's rotation

rAF support across browsers

The latest versions of Chrome and Firefox support rAF out of the box. Safari supports rAF only if you use a special vendor prefix. It is possible to create a shim that reverts to `setTimeout` for browser versions that do not support rAF. This article's example uses a polyfill script solution that enables a consistent calling syntax for rAF across supporting browsers.

Being a low-level API, WebGL contains no intrinsic support for animation.

To show the rotation animation, the example depends on the browser's support for animation via the `requestAnimationFrame()` function (rAF). `requestAnimationFrame()` takes a callback function

as an argument. The browser calls back the function before the next screen update, typically up to 60 times per second. In the callback, you must call `requestAnimationFrame()`again so that it's called before the next screen update.

The code in pyramid.html that calls `requestAnimationFrame()` is shown in Listing 11.

Listing 11. Calling `requestAnimationFrame` for screen updates

```
function update(gl) {
        requestAnimationFrame(function() { update(gl); });
        draw(gl);
}
```

In Listing 11, the `update()` function is supplied as the callback, and `update()` must also call `requestAnimationFrame()` for the next frame. Each time `update()` is called, the `draw()` function is also called.

Listing 12 shows how the `draw()` function from triangles.html is modified to rotate the pyramid incrementally around the y-axis. The additional or modified code is in boldface.

Listing 12. Animating the pyramid's rotation

```
var onerev = 10000; // ms
var exTime = Date.now();

  function draw(ctx) {
        ctx.clearColor(1.0, 1.0, 1.0, 1.0);
        ctx.enable(ctx.DEPTH_TEST);
        ...
        ctx.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);
        var now = Date.now();
        var elapsed = now - exTime;
        exTime = now;
        var angle = Math.PI * 2 * elapsed/onerev;
        mat4.rotate(modelViewMatrix, modelViewMatrix, angle, rotationAxis);
        ctx.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);
        ctx.drawArrays(ctx.TRIANGLES, 0, 12 /* num of vertex */);
  }
```

In Listing 12, the rotation angle per frame is calculated by dividing the elapsed time `elapsed` by the time for one complete revolution (`onerev` = 10 seconds in the example).

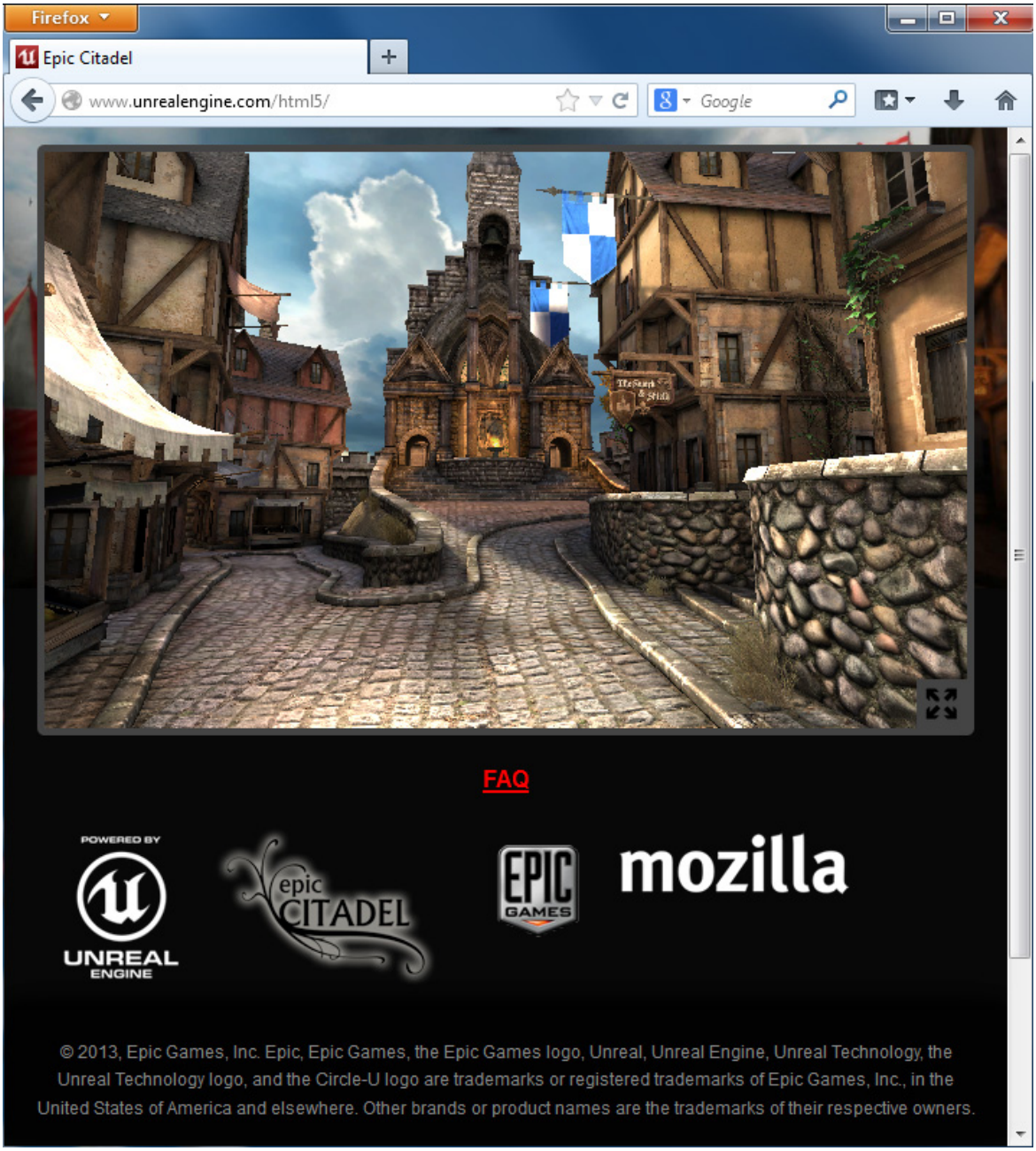
You will see more use of rAF in upcoming articles in this series.

Pushing the limits of WebGL

The pyramid example explores important fundamental concepts in WebGL programming, barely scratching the surface of what's possible.

To see some amazing examples of the possibilities — applications that push the limits of current-day WebGL — point your latest Firefox browser, running on a relatively modern machine, to the Unreal Engine 3 WebGL demonstration - Epic Citadel, from Epic Games. Figure 10 shows Epic Citadel in action (running in Firefox on Windows).

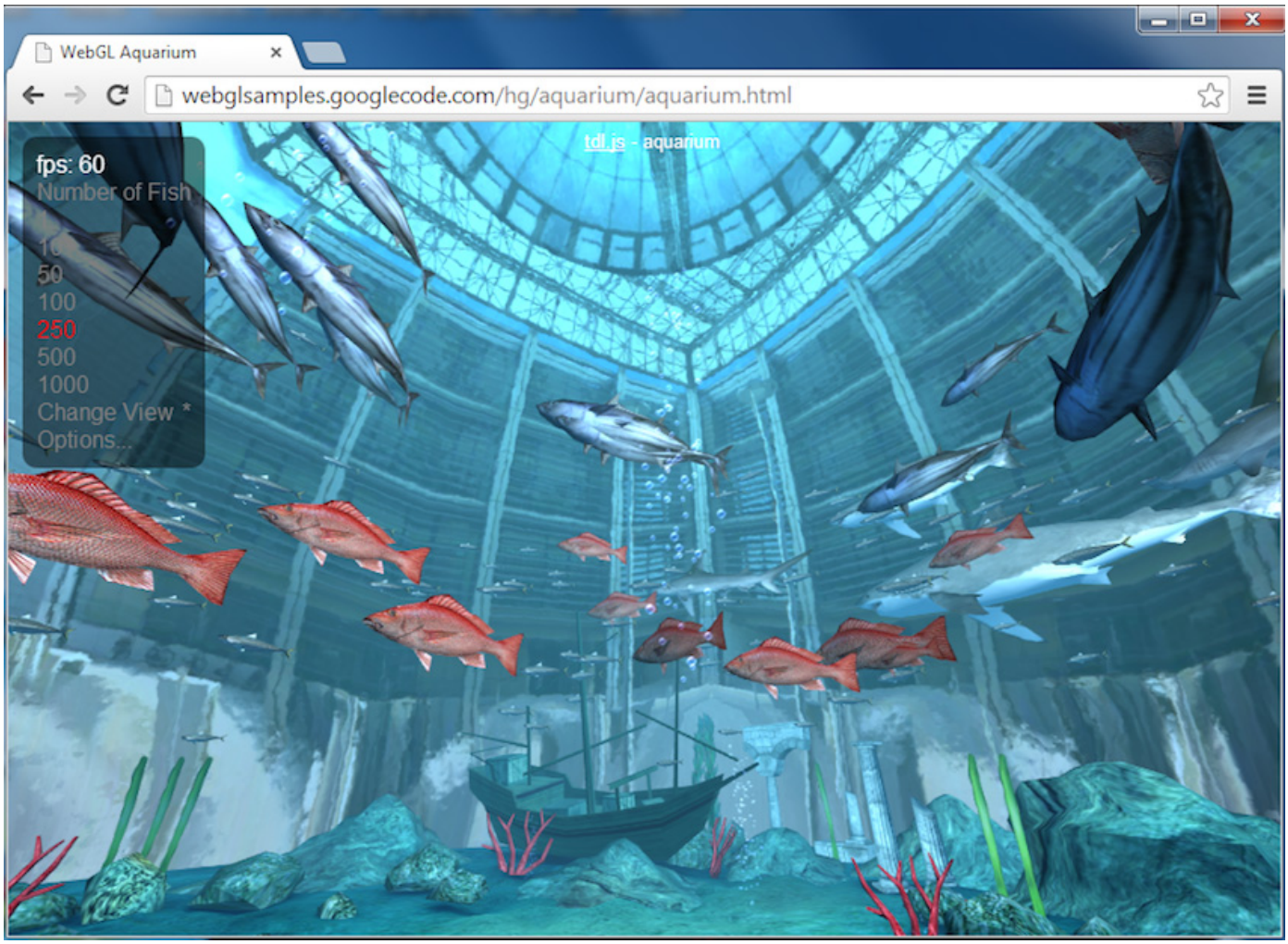
Figure 10. Epic Citadel — Unreal Engine 3 on WebGL demonstration from Epic Games



Epic Citadel is the result of compiling the well-known game engine product (originally written in C/C++) to JavaScript and WebGL. (The compiler technology used is emscripten, and the output JavaScript subset is known as asm.js.) You can interact with and walk through this WebGL-rendered medieval town, complete with cobblestone streets, castles, and animated waterfalls.

Another interesting example is the WebGL Aquarium from Google, by Greggman and Human Engines. Figure 11 shows the aquarium running in Chrome on Windows.

Figure 11. A futuristic glass dome aquarium WebGL sample from Google



With this aquarium application, you can select the number of fish swimming around in a spherical all-glass aquarium. You can also use an option to try out some of the rendering effects, such as reflection, fog, and light rays.

Conclusion

WebGL opens up the raw 3D hardware for JavaScript API access, but the API remains low-level:

- WebGL has no notion of and does not care about what is being displayed within the 3D scene. The simple 3D pyramid object in this article's example isn't known to the WebGL API layer. Instead, you must painstakingly keep track of the vertices data that makes up the object.
- Each call to draw a WebGL scene renders only a 2D picture. Animation itself is not part of WebGL and must be implemented by additional code.
- Movement and interactions between object and environment, such as light reflections and object physics, must be implemented in additional code.
- Handling of events, such as user input, object selection, or object collision, must be implemented in higher-level code.

Writing 100+ lines of code just to spin a pyramid should seem daunting. And it should be evident that the creation of any reasonably complex WebGL application requires the use of a higher-level library or framework. Thankfully, there's no shortage of WebGL libraries/frameworks, many of which are freely available in the open source code community. Part 2 explores the use of WebGL libraries.

Downloads

Description	Name	Size
Sample code	webgl1dl.zip	20KB

Resources

Learn

- WebGL: Visit the WebGL home page on the Khronos Group site and read the latest working draft of the WebGL Specification.
- WebGL Quick Reference: Take advantage of a handy cheat sheet for WebGL API syntax and concepts at a glance.
- OpenGL Rendering Pipeline: Get an overview of the conceptual hardware "pipeline" used in hardware-accelerated 3D rendering.
- "Create great graphics with the HTML5 canvas" (Ken Bluttman, developerWorks, February 2011): If you need a refresher on HTML5 canvas 2D drawing techniques, check out this introductory article.
- Chromium Rendering Stack: See how the open source Chromium browser supports GPU acceleration internally via a dual rendering stack.
- OpenGL Shading Language: Read the latest GLSL documentation for details on writing or experimenting with writing shader code.
- JavaScript Typed Arrays: Get the latest specification from the Khronos Group.
- Can I use WebGL?: This valuable site tracks up-to-date browser support for WebGL by versions.
- Epic Citadel: Walk through a complete 3D-rendered medieval town in this WebGL game engine demonstration from Epic Games. Also check out the emscripten compiler and the asm.js low-level JavaScript subset that this project uses.
- WebGL Aquarium: Start your own futuristic spherical 3D fish aquarium with this WebGL sample from Google code by Greggman and Human Engines.

Get products and technologies

- Check out Sing Li's 3D development with WebGL, Part 1: Introducing WebGL.
- glMatrix JavaScript Matrix and Vector Library: Download the de-facto standard JavaScript library for high-performance matrix handling in WebGL applications.
- requestAnimationFrame polyfill: You can use this polyfill by Erik Moller (based on Paul Irish's original blog post) to use identical syntax for `requestAnimationFrame` calls across browsers.

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.