THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Beginning WebGL for HTML5

CREATE STUNNING REAL-TIME 3D ON THE WEB AND DEVICES WITHOUT THE NEED FOR PLUGINS

Brian Danchilla

Apress®

Beginning WebGL for HTML5

Brian Danchilla

Apress®

Beginning WebGL for HTML5

ISBN-13 978-1-4302-3996-3

ISBN-13 978-1-4302-3997-0 (eBook)

President and Publisher: Paul Manning
Lead Editor: Ben Renow-Clarke
Technical Reviewer: Massimo Nardone
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Jennifer Blackwell, Anamika Panchoo
Copy Editor: Nancy Sixsmith
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science + Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348–4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code.

For Tressa, as fierce as she is delicate.

Contents at a Glance

Contents

About the Author

Brian Danchilla is a freelance developer and author. He is ZEND PHP Certified and cowrote the book *Pro PHP Programming*. Brian is a contributing author to the book *HTML5 Games Most Wanted*, and was a technical reviewer for *Foundation HTML5 Animation for JavaScript* and *PHP: The Good Parts*. Brian is a seasoned web developer, Java, and OpenGL programmer and has a BA degree as a double major in computer science and mathematics. After several years working within the computer industry, he is now enjoying the flexibility of freelance work. Brian has a strong desire and ability to learn new technologies and APIs and is an avid technical reader. When not programming, Brian likes to play guitar, cook, and travel. He resides in Saskatoon, SK, Canada with his fiancée Tressa.

About the Technical Reviewer

Massimo Nardone holds a Master of Science degree in Computing Science from the University of Salerno, Italy. He works currently as a PCI QSA and Senior Lead IT Security/Cloud Architect for IBM Finland. With more than 19 years of work experience in Cloud Computing, IT Infrastructure, Mobile, Security and WWW technology areas for both national and international projects, Massimo has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, and Software Specialist. He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Helsinki University of Technology TKK became a part of Aalto University) for the course of "Security of Communication Protocols." He holds four international patents (PKI, SIP, SAML and Proxy areas). His beloved baby girl, Neve, was born some days ago and he wants to welcome her from the bottom of his heart.

Acknowledgments

I would like to thank my family: mom, dad, Robert, Karen, Tressa. Longtime friends: Vince, Nick, and Tim. It was not always a smooth journey during this book project and I am very grateful for your support along the way.

Thank you to everyone on the Apress team who was involved with this book. I would especially like to thank Ben Renow-Clarke for getting this book title off the ground, Louise Corrigan for valuable editing advice, Anamika Panchoo for her coordinating skills, and the technical reviewing of Massimo Nardone.

Thank you to my university professor David Mould for introducing me to OpenGL,and my high school art teacher Mrs. Robinson for helping develop my artistic side.

I am thankful for having a good set of headphones and a great music collection that enabled me to zone in and work. Thanks to late, late night *Super Mario Galaxy 2* sessions for inspiration and coffee, lots of coffee.

Several years ago, I first read computer graphics books and thoroughly enjoyed the experience. Now I am blessed to have been able to write my own book about an exciting new technology, WebGL.

Introduction

WebGL (Web-based Graphics Language) is a wonderful and exciting new technology that lets you create powerful 3D graphics within a web browser. The way that this is achieved is by using a JavaScript API that interacts with the Graphics Processing Unit (GPU). This book will quickly get you on your way to demystify shaders and render realistic scenes. To ensure enjoyable development, we will show how to use debugging tools and survey libraries which can maximize productivity.

Audience

Beginning WebGL for HTML5 is aimed at graphics enthusiasts with a basic knowledge of computer graphics techniques. A knowledge of OpenGL, especially a version that uses the programmable pipeline, such as OpenGL ES is beneficial, but not essential. We will go through all the relevant material. A JavaScript background will certainly help.

When writing a book of this nature, we unfortunately cannot cover all the prerequisite material. Baseline assumptions about the reader need to be made. The assumptions that I have made are that the reader has a basic knowledge of 2D and 3D computer graphics concepts such as pixels, colors, primitives, and transforms. Appendix B quickly refreshes these concepts. It is also assumed that the reader is familiar (though need not be an expert) with HTML, CSS, and JavaScript. Although much of the book makes use of plain "vanilla" JavaScript, we will use some jQuery. Appendix A discusses newer HTML5 concepts and a quick jQuery crash course that will be essential for properly understanding the text. Appendix D provides a complete reference for further reading on topics that are presented throughout the book.

What You Will Learn

This book presents theory when necessary and examples whenever possible. You will get a good overview of what you can do with WebGL. What you will learn includes the following:

- Understanding the model view matrix and setting up a scene
- Rendering and manipulating primitives
- Understanding shaders and loving their power and flexibility
- Exploring techniques to create realistic scenes
- Using basic physics to simulate interaction
- Using mathematics models to render particle systems, terrain, and fractals
- Getting productive with existing models, shaders, and libraries

- Using the Three.js framework
- Learning about GLGE and philoGL frameworks and a survey of other frameworks available
- Debugging and performance tips
- Understanding other shader uses, such as image processing and nonphotorealistic rendering
- Using an alternate framebuffer to implement picking and shadowmaps
- Learning about current browser and mobile support and the future of WebGL

Book Structure

It is recommended that you start by reading the first two chapters before moving on to other areas of the book. Even though the book does follow a fairly natural progression, you may choose to read the book in order or skip around as desired. For example, the debugging section of Chapter 9 is not strictly essential, but is very useful information to know as soon as possible.

Chapter 1: Setting the Scene

We go through all the steps to render an image with WebGL, including testing for browser support and setting up the WebGL environment, using vertex buffer objects (VBOs), and basic shaders. We start with creating a one color static 2D image, and by the end of the chapter have a moving 3D mesh with multiple colors.

Chapter 2: Shaders 101

Shaders are covered in depth. We show an overview of graphics pipelines (fixed and programmable), give a background of the GL Shading Language (GLSL), and explain the roles of vertex and fragment shaders. Next we go over the primitive types and language details of GLSL and how our WebGL application will interact with our shaders. Finally, we show several examples of GLSL usage.

Chapter 3: Textures and Lighting

We show how to apply texture and simple lighting. We explain texture objects and how to set up and configure them and combine texture lookups with a lighting model in our shader.

Chapter 4: Increasing Realism

A more realistic lighting model—Phong illumination—is explained and implemented. We discuss the difference between flat and smooth shading and vertex and fragment calculations. We show how to add fog and blend objects; and discuss shadows, global illumination, and reflection and refraction.

Chapter 5: Physics

This chapter shows how to model gravity, elasticity, and friction. We detect and react to collisions, model projectiles and explore both the conservation of momentum and potential and kinetic energy.

Chapter 6: Fractals, Height Maps, and Particle Systems

In this chapter we show how to paint directly with the GPU, discuss fractals, and model the Mandlebrot and Julia sets. We also show how to produce a height map from a texture and generate terrain. We also explore particle systems.

Chapter 7: Three.js Framework

The Three.js WebGL framework is introduced. We provide a background and sample usage of the library, including how to fall back to the 2D rendering context if necessary, API calls to easily create cameras, objects, and lighting. We compare earlier book examples to the equivalent Three.js API calls and introduce tQuery, a library that combines Three.js and jQuery selectors.

Chapter 8: Productivity Tools

We discuss the benefits of using frameworks and the merit of learning core WebGL first. Several available frameworks are discussed and the GLGE and philoGL frameworks are given examples. We show how to load existing meshes and find other resources. We list available physics libraries and end the chapter with an example using the physi.js library.

Chapter 9: Debugging and Performance

An important chapter to help identify and fix erroneous code and improve performance by following known WebGL best practices.

Chapter 10: Effects, Tips, and Tricks

Image processing and nonphotorealistic shaders are discussed and implemented. We show how to use offscreen framebuffers that enable us to pick objects from the canvas and implement shadow maps.

Afterword: The Future of WebGL

In the afterword, we will speculate on the bright future of WebGL, the current adoption of it within the browser, and mobile devices and what features will be added next.

Appendix A: Essential HTML5 and JavaScript

We cover some of the changes between HTML 4 and 5, such as shorter tags, added semantic document structure, the `<canvas>` element, and basic JavaScript and jQuery usage.

Appendix B: Graphics Refresher

This appendix is a graphics refresher covering coordinate systems, elementary transformations and other essential topics.

Appendix C: WebGL Specification Odds and Ends

Contains part of the WebGL specification, available at http://www.khronos.org/registry/webgl/specs/latest/, which were not covered in the book, but are nonetheless important.

Appendix D: Additional Resources

A list of references for further reading about topics presented in the book such as HTML5, WebGL, WebGLSL, JavaScript, jQuery, server stacks, frameworks, demos, and much more.

WebGL Origins

The origin of WebGL starts 20 years ago, when version 1.0 of OpenGL was released as a nonproprietary alternative to Silicon Graphics' Iris GL. Up until 2004, OpenGL used a fixed functionality pipeline (which is explained in Chapter 2). Version 2.0 of OpenGL was released that year and introduced the GL Shading Language (GLSL) which lets you program the vertex and fragment shading portions of the pipeline. The current version of OpenGL is 4.2, however WebGL is based off of OpenGL Embedded Systems (ES) 2.0, which was released in 2007 and is a trimmer version of OpenGL 2.0.

Because OpenGL ES is built for use in embedded devices like mobile phones, which have lower processing power and fewer capabilities than a desktop computer, it is more restrictive and has a smaller API than OpenGL. For example, with OpenGL you can draw vertices using both a `glBegin...glEnd` section or VBOs. OpenGL ES only uses VBOs, which are the most performance-friendly option. Most things that can be done in OpenGL can be done in OpenGL ES.

In 2006, Vladimar Vukićević worked on a Canvas 3D prototype that used OpenGL for the web. In 2009, the Khronos group created the WebGL working group and developed a central specification that helps to ensure that implementations across browsers are close to one another. The 3D context was modified to WebGL, and version 1.0 of the specification was completed in spring 2011. Development of the WebGL specification is under active development, and the latest revision can be found at http://www.khronos.org/registry/webgl/specs/latest/.

How Does WebGL work?

WebGL is a JavaScript API binding from the CPU to the GPU of a computer's graphics card. The API context is obtained from the HTML5 `<canvas>` element, which means that no browser plugin is required. The shader program uses GLSL, which is a C++ like language, and is compiled at runtime.

Without a framework, setting up a WebGL scene does require quite a bit of work: handling the WebGL context, setting buffers, interacting with the shaders, loading textures, and so on. The payoff of using WebGL is that it is much faster than the 2D canvas context and offers the ability to produce a degree of realism and configurability that is not possible outside of using WebGL.

Uses

Some uses of WebGL are viewing and manipulating models and designs, virtual tours, mapping, gaming, art, data visualization, creating videos, manipulating and processing of data and images.

Demonstrations

There are many demos of WebGL, including these:

- http://www.chromeexperiments.com/webgl
- https://code.google.com/p/webglsamples/
- http://aleksandarrodic.com/p/jellyfish/
- Google Body (now http://www.zygotebody.com), parts of Google Maps, and Google Earth
- http://www.ro.me/tech/
- http://alteredqualia.com/

Supported Environments

Does your browser support WebGL? It is important to know that WebGL is not currently supported by all browsers, computers and/or operating systems (OS). Browser support is the easiest requirement to meet and can be done simply by upgrading to a newer version of your browser or switching to a different browser that does support WebGL if necessary. The minimum requirements are as follows:

- Firefox 4+
- Safari 5.1+ (OS X only)
- Chrome 9+
- Opera 12alpha+
- Internet Explorer (IE)—no native support

Although IE currently has no built in support, plugins are available; for example, JebGL (available at http://code.google.com/p/jebgl/), Chrome Frame (available at http://www.google.com/chromeframe), and IEWebGL (http://iewebgl.com/). JebGL converts WebGL to a Java applet for deficient browsers; Chrome Frame allows WebGL usage on IE, but requires that the user have it installed on the client side. Similarly, IEWebGL is an IE plugin.

In addition to a current browser, you need a supported OS and newer graphics card. There are also several graphics card and OS combinations that have known security vulnerabilities or are highly prone to a severe system crash and so are blacklisted by browsers by default.

Chrome supports WebGL on the following operating systems (according to Google Chrome Help (http://www.google.com/support/chrome/bin/answer.py?answer=1220892):

- Windows Vista and Windows 7 (recommended) with no driver older than 2009–01
- Mac OS 10.5 and Mac OS 10.6 (recommended)
- Linux

Often, updating your graphics driver to the latest version will enable WebGL usage. Recall that OpenGL ES 2.0 is based on OpenGL 2.0, so this is the version of OpenGL that your graphics card should support for WebGL usage. There is also a project called ANGLE (Almost Native Graphics Layer Engine) that ironically uses Microsoft Direct X to enhance a graphics driver to support OpenGL ES 2.0 API calls through conversions to Direct X 9 API calls. The result is that graphics cards that only support OpenGL 1.5 (OpenGL ES 1.0) can still run WebGL. Of course, support for WebGL should improve drastically over the next couple of years.

Testing for WebGL Support

To check for browser support of WebGL. there are several websites such as http://get.webgl.org/, which displays a spinning cube on success; and http://doesmybrowsersupportwebgl.com/, which gives a large "Yay" or "Nay" and specific details if the webgl context is supported. We can also programmatically check for WebGL support using modernizr (http://www.modernizr.com).

Companion Site

Along with the Apress webpage at http://www.apress.com/9781430239963, this book has a companion website at http://www.beginningwebgl.com. This site demonstrates the examples found in the book, and offers an area to make comments and add suggestions directly to the author. Your constructive feedback is both welcome and appreciated.

Downloading the code

The code for the examples shown in this book is available on the Apress website, http://www.apress.com. A link can be found on the book's information page, http://www.apress.com/9781430239963, under the Source Code/ Downloads tab. This tab is located underneath the Related Titles section of the page. Updated code will also be hosted on github at https://github.com/bdanchilla/beginningwebgl.

Contacting the Author

If you have any questions or comments—or even spot a mistake you think I should know about—you can contact the author directly at bdanchilla@gmail.com or on the contact form at http://www.beginningwebgl.com/contact.

CHAPTER 1

Setting the Scene

In this chapter we will go through all the steps of creating a scene rendered with WebGL. We will show you how to

- obtain a WebGL context
- create different primitive types in WebGL
- understand and create vertex buffer objects (VBOs) and attributes
- do static two-dimensional rendering
- create a program and shaders
- set up the view matrices
- add animation and movement
- render a three-dimensional model

A Blank Canvas

Let's start by creating a HTML5 document with a single `<canvas>` element (see Listing 1-1).

Listing 1-1. A basic blank canvas

```
<!doctype html>
<html>
         <head>
              <title>A blank canvas</title>
              <style>
                        body{ background-color: grey; }
                        canvas{ background-color: white; }
                </style>
        </head>
        <body>
                <canvas id="my-canvas" width="400" height="300">
                        Your browser does not support the HTML5 canvas element.
                </canvas>
        </body>
</html>
```

The HTML5 document in Listing 1-1 uses the shorter `<!doctype html>` and `<html>` declaration available in HTML5. In the `<head>` section, we set the browser title bar contents and then add some basic styling that will

change the <body> background to gray and the <canvas> background to white. This is not necessary but helps us to easily see the canvas boundary. The content of the body is a single canvas element. If viewing the document with an old browser that does not support the HTML 5 canvas element, the message *"Your browser does not support the HTML5 canvas element."* will be displayed. Otherwise, we see the image in Figure 1-1.

Figure 1-1. *A blank canvas*

■ **Note** If you need a refresher on HTML5, please see Appendix A. Additional reference links are provided in Appendix D.

Getting Context

When we draw inside of a canvas element, we have more than one option of how we produce our image. Each option corresponds to a different application programming interface (API) with different available functionality and implementation details and is known as a particular context of the canvas. At the moment there are two canvas contexts: "2D" and "webgl". The canvas element does not really care which context we use, but it needs to explicitly know so that it can provide us with an appropriate object that exposes the desired API.

To obtain a context, we call the canvas method getContext. This method takes a context name as a first parameter and an optional second argument. The WebGL context name will eventually be "webgl", but for now, most browsers use the context name "experimental-webgl". The optional second argument can contain buffer settings and may vary by browser implementation. A full list of the optional WebGLContextAttributes and how to set them is shown in Appendix C.

Listing 1-2. Establishing a WebGL context

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

```
<head>
        <title>WebGL Context</title>
        <style>
                body{ background-color: grey; }
                canvas{ background-color: white; }
        </style>
        <script>
                window.onload = setupWebGL;
                var gl = null;

                function setupWebGL()
                {
                        var canvas = document.getElementById("my-canvas");
                        try{
                                gl = canvas.getContext("experimental-webgl");
                        }catch(e){
                        }

                        if(gl)
                        {
                                //set the clear color to red
                                gl.clearColor(1.0, 0.0, 0.0, 1.0);
                                gl.clear(gl.COLOR_BUFFER_BIT);
                        }else{
                                alert( "Error: Your browser does not appear to support
                        WebGL.");
                        }
                }
        </script>
</head>
<body>
        <canvas id="my-canvas" width="400" height="300">
                Your browser does not support the HTML5 canvas element.
        </canvas>
</body>
</html>
```

In Listing 1-2, we define a JavaScript setup function that is called once the window's Document Object Model (DOM) has loaded:

```
window.onload = setupWebGL;
```

We initiate a variable to store the WebGL context with `var gl = null`. We use `gl = canvas.getContext("experimental-webgl");` to try to get the `experimental-webgl` context from our canvas element, catching any exceptions that may be thrown.

■ **Note** The name `"gl"` is conventionally used in WebGL to refer to the context object. This is because OpenGL and OpenGL ES constants begin with `GL_` such as `GL_DEPTH_TEST;` and functions begin with `gl`, such as `glClearColor`.

WebGL does not use these prefixes, but when using the name `"gl"` for the context object, the code looks very similar: `gl.DEPTH_TEST` and `gl.clearColor`

This similarity makes it easier for programmers who are already familiar with OpenGL to learn WebGL.

On success, `gl` is a reference to the WebGL context. However, if a browser does not support WebGL, or if a canvas element has already been initialized with an incompatible context type, the `getContext` call will return `null`. In Listing 1-2, we test for `gl` to be non-null; if this is the case, we then set the clear color (the default value to set the color buffer) to red. If your browser supports WebGL, the browser output should be the same as Figure 1-1, but with a red canvas now instead of white. If not, we output an alert as shown in Figure 1-2. You can simulate this by misspelling the context, to `"zzexperimental-webgl"` for instance.

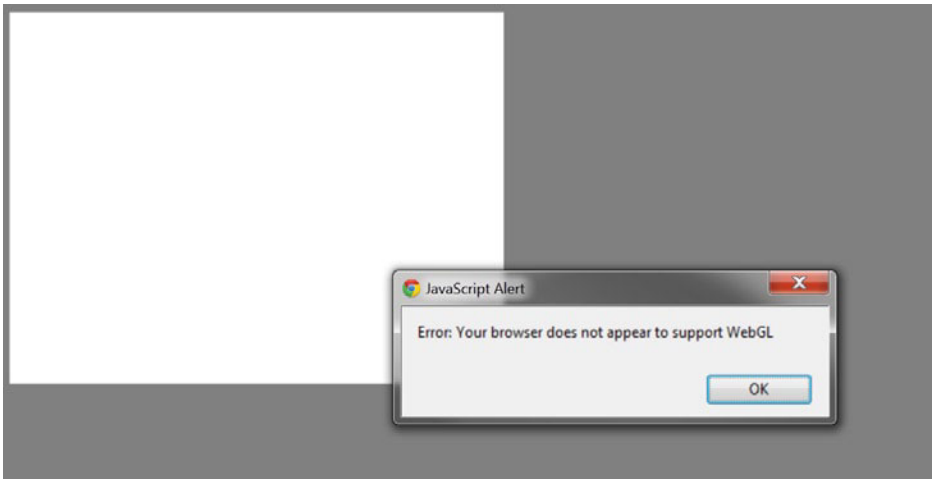

Figure 1-2. *Error alert if WebGL is not supported*

Being able to detect when the WebGL context is not supported is beneficial because it gives us the opportunity to program an appropriate alternative such as redirecting the user to http://get.webgl.org or falling back to a supported context such as `"2D"`. We show how to do the latter approach with Three.js in Chapter 7.

■ **Note** There is usually more than one way of doing things in JavaScript. For instance, to load the `setupWebGL` function in code Listing 1-2, we could have written the `onload` event in our HTML instead:

```
<body onload="setupWebGL();">
```

If we were using jQuery, we would use the document ready function:

```
$(document).ready(function(){ setupWebGL(); });
```

We may make use of these differing forms throughout the book.

With jQuery, we can also shorten our canvas element retrieval to: `var canvas = $("#my-canvas").get(0);`

WebGL Components

In this section we will give an overview of the drawing buffers, primitive types, and vertex storage mechanisms that WebGL provides.

The Drawing Buffers

WebGL has a color buffer, depth buffer, and stencil buffer. A *buffer* is a block of memory that can be written to and read from, and temporarily stores data. The color buffer holds color information—red, green, and blue

values—and optionally an alpha value that stores the amount of transparency/opacity. The depth buffer stores information on a pixel's depth component (z-value). As the map from 3D world space to 2D screen space can result in several points being projected to the same `(x,y)` canvas value, the z-values are compared and only one point, usually the nearest, is kept and rendered. For those seeking a quick refresher, Appendix B discusses coordinate systems.

The stencil buffer is used to outline areas to render or not render. When an area of an image is marked off to not render, it is known as masking that area. The entire image, including the masked portions, is known as a stencil. The stencil buffer can also be used in combination with the depth buffer to optimize performance by not attempting to render portions of a scene that are determined to be not viewable. By default, the color buffer's alpha channel is enabled and so is the depth buffer, but the stencil buffer is disabled. As previously mentioned, these can be modified by specifying the second optional parameter when obtaining the WebGL context as shown in Appendix C.

Primitive Types

Primitives are the graphical building blocks that all models in a particular graphics language are built with. In WebGL, there are three primitive types: points, lines and triangles and seven ways to render them: `POINTS`, `LINES`, `LINE_STRIP`, `LINE_LOOP`, `TRIANGLES`, `TRIANGLE_STRIP`, and `TRIANGLE_FAN` (see Figure 1-3).

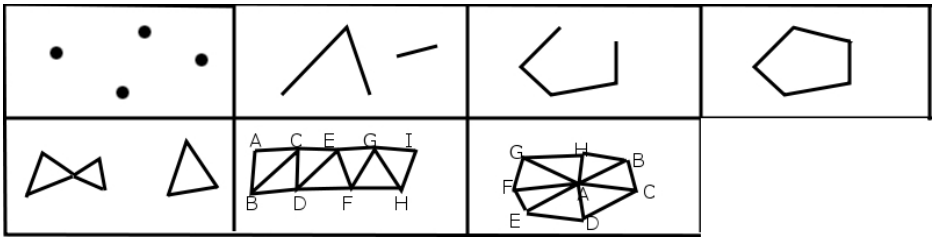

Figure 1-3. *WebGL Primitive Types (top row, l—r: POINTS, LINES, LINE_STRIP, and LINE_LOOP; bottom row, l—r: TRIANGLES, TRIANGLE_STRIP, and TRIANGLE_FAN)*

`POINTS` are vertices (spatial coordinates) rendered one at a time. `LINES` are formed along pairs of vertices. In Figure 1-3 two of the lines share a common vertex, but as each line is defined separately, it would still require six vertices to render these three lines. A `LINE_STRIP` is a collection of vertices in which, except for the first line, the starting point of each line is the end point of the previous line. With a `LINE_STRIP`, we reuse some vertices on multiple lines, so it would take just five vertices to draw the four lines in Figure 1-3. A `LINE_LOOP` is similar to a `LINE_STRIP` except that it is a closed off loop with the last vertex connecting back to the very first. As we are again reusing vertices among lines, we can produce five lines this time with just five vertices.

`TRIANGLES` are vertex trios. Like `LINES`, any shared vertices are purely coincidental and the example in Figure 1-3 requires nine vertices, three for each of the three triangles. A `TRIANGLE_STRIP` uses the last two vertices along with the next vertex to form triangles. In Figure 1-3 the triangles are formed by vertices ABC, (BC)D, (CD)E, (DE)F, (EF)G, (FG)H, and (GH)I. This lets us render seven triangles with just nine vertices as we reuse some vertices in multiple triangles. Finally, a `TRIANGLE_FAN` uses the first vertex specified as part of each triangle. In the preceding example this is vertex A, allowing us to render seven triangles with just eight vertices. Vertex A is used a total of seven times, while every other vertex is used twice.

■ **Note** Unlike OpenGL and some other graphics languages, a quad is not a primitive type. Some WebGL frameworks provide it as a "basic" type and also offer geometric solids built in, but at the core level these are all rendered from triangles.

Vertex Data

Unlike old versions of OpenGL or "the '2D' canvas context", you can't directly set the color or location of a vertex directly into a scene. This is because WebGL does not have fixed functionality but uses programmable shaders instead. All data associated with a vertex needs to be streamed (passed along) from the JavaScript API to the Graphics Processing Unit (GPU). With WebGL, you have to create vertex buffer objects (VBOs) that will hold vertex attributes such as position, color, normal, and texture coordinates.

These vertex buffers are then sent to a shader program that can use and manipulate the passed-in data in any way you see fit. Using shaders instead of having fixed functionality is central to WebGL and will be covered in depth in the next chapter.

We will now turn our attention to what vertex attributes and uniform values are and show how to transport data with VBOs.

Vertex Buffer Objects (VBOs)

Each VBO stores data about a particular attribute of your vertices. This could be position, color, a normal vector, texture coordinates, or something else. A buffer can also have multiple attributes interleaved (as we will discuss in Chapter 9).

Looking at the WebGL API calls (which can be found at http://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf or at http://www.khronos.org/registry/webgl/specs/latest/), to create a buffer, you call `WebGLBuffer createBuffer()`and store the returned object, like so:

```
var myBuffer = gl.createBuffer();
```

Next you bind the buffer using `void bindBuffer(GLenum target, WebGLBuffer buffer)` like this:

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, myBuffer);
```

The target parameter is either `gl.ARRAY_BUFFER` or `gl.ELEMENT_ARRAY_BUFFER`. The target `ELEMENT_ARRAY_BUFFER` is used when the buffer contains vertex indices, and `ARRAY_BUFFER` is used for vertex attributes such as position and color.

Once a buffer is bound and the type is set, we can place data into it with this function:

```
void bufferData(GLenum target, ArrayBuffer data, GLenum usage)
```

The usage parameter of the bufferData call can be one of `STATIC_DRAW, DYNAMIC_DRAW,` or `STREAM_DRAW`. `STATIC_DRAW` will set the data once and never change throughout the application's use of it, which will be many times. `DYNAMIC_DRAW` will also use the data many times in the application but will respecify the contents to be used each time. `STREAM_DRAW` is similar to `STATIC_DRAW` in never changing the data, but it will be used at most a few times by the application. Using this function looks like the following:

```
var data = [      1.0, 0.0, 0.0,
                  0.0, 1.0, 0.0,
                  0.0, 1.0, 1.0
           ];
gl.bufferData(gl.ARRAY_BUFFER, data, gl.STATIC_DRAW);
```

Altogether the procedure of creating, binding and storing data inside of a buffer looks like:

```
var data = [      1.0, 0.0, 0.0,
                  0.0, 1.0, 0.0,
                  0.0, 1.0, 1.0
           ];
```