

Professional WebGL Programming

Developing 3D Graphics for the Web

Andreas Anyuru

Contents

Chapter 1: Introducing WebGL

The Basics of WebGL

So Why is WebGL so Great?

Designing a Graphics API

An Overview of Graphics Hardware

Understanding the WebGL Graphics Pipeline

Comparing WebGL to Other Graphics Technologies

Linear Algebra for 3D Graphics

Summary

Chapter 2: Creating Basic Webgl Examples

Drawing a Triangle

Understanding the WebGL Coding Style

Debugging Your WebGL Application

Using the DOM API to Load Your Shaders

Putting It Together in a Slightly More Advanced Example

Summary

Chapter 3: Drawing

WebGL Drawing Primitives and Drawing Methods

Typed Arrays

Exploring Different Ways to Draw

Interleaving Your Vertex Data for Improved Performance

Using a Vertex Array or Constant Vertex Data

A Last Example to Wind Things Up

Summary

Chapter 4: Compact Javascript Libraries and Transformations

Working with Matrices and Vectors in JavaScript

Using Transformations

Understanding the Complete Transformation Pipeline

Getting Practical with Transformations

Understanding the Importance of Transformation Order

A Complete Example: Drawing Several Transformed Objects

Summary

Chapter 5: Texturing

Understanding Lost Context

Introducing 2D Textures and Cubemap Textures

Loading Your Textures

Defining Your Texture Coordinates

Using Your Textures in Shaders

Working with Texture Filtering

Understanding Texture Coordinate Wrapping

A Complete Texture Example

Using Images for Your Textures

Understanding Same-Origin Policy and Cross-Origin Resource Sharing

Summary

Chapter 6: Animations and User Input

Animating the Scene

Event Handling for User Interaction

Applying Your New Knowledge

Summary

Chapter 7: Lighting

Understanding Light

Working with a Local Lighting Model

Understanding the Phong Reflection Model

Understanding the JavaScript Code Needed for WebGL Lighting

Using Different Interpolation Techniques for Shading

Understanding the Vectors that Must be Normalized

Using Different Types of Lights

Understanding the Attenuation of Light

Understanding Light Mapping

Summary

Chapter 8: WEBGL Performance Optimizations

WebGL under the Hood

WebGL Performance Optimizations

A Closer Look at Blending

Taking WebGL Further

Summary

Introduction

Advertisements

Chapter 1

Introducing WebGL

WHAT'S IN THIS CHAPTER?

- The basics of WebGL
- Why 3D graphics in the browser offer great possibilities
- How to work with an immediate-mode API
- The basics of graphics hardware
- The WebGL graphics pipeline
- How WebGL compares to other graphics technologies
- Some basic linear algebra needed for WebGL

In this chapter you will be introduced to WebGL and learn some important theory behind WebGL, as well as general theory behind 3D graphics. You will learn what WebGL is and get an understanding of the graphics pipeline that is used for WebGL.

You will also be given an overview of some other related graphics standards that will give you a broader understanding of WebGL and how it compares to these other technologies.

The chapter concludes with a review of some basic linear algebra that is useful to understand if you really want to master WebGL on a deeper level.

THE BASICS OF WEBGL

WebGL is an application programming interface (API) for advanced 3D graphics on the web. It is based on OpenGL ES 2.0, and provides similar rendering functionality, but in an HTML and JavaScript context. The rendering surface that is used for WebGL is the HTML5 canvas element, which was originally introduced by Apple in the WebKit open-source browser engine. The reason for introducing the HTML5 canvas was to be

able to render 2D graphics in applications such as Dashboard widgets and in the Safari browser on the Apple Mac OS X operating system.

Based on the canvas element, Vladimir Vukićević at Mozilla started experimenting with 3D graphics for the canvas element. He called the initial prototype Canvas 3D. In 2009 the Khronos Group started a new WebGL working group, which now consists of several major browser vendors, including Apple, Google, Mozilla, and Opera. The Khronos Group is a non-profit industry consortium that creates open standards and royalty-free APIs. It was founded in January 2000 and is behind a number of other APIs and technologies such as OpenGL ES for 3D graphics for embedded devices, OpenCL for parallel programming, OpenVG for low-level acceleration of vector graphics, and OpenMAX for accelerated multimedia components. Since 2006 the Khronos Group has also controlled and promoted OpenGL, which is a 3D graphics API for desktops.

The final WebGL 1.0 specification was frozen in March 2011, and WebGL support is implemented in several browsers, including Google Chrome, Mozilla Firefox, and (at the time of this writing) in the development releases of Safari and Opera.



For the latest information about which versions of different browsers support WebGL, visit www.khronos.org/webgl/.

SO WHY IS WEBGL SO GREAT?

In the early days of the web, the content consisted of static documents of text. The web browser was used to retrieve and display these static pages. Over time the web technology has evolved tremendously, and today many websites are actually full-featured applications. They support two-way communication between the server and the client, users can register and log in, and web applications now feature a rich user interface that includes graphics as well as audio and video.

The fast evolution of web applications has led to them becoming an attractive alternative to native applications. Some advantages of web

applications include the following:

- They are cheap and easy to distribute to a lot of users. A compatible web browser is all that the user needs.
- Maintenance is easy. When you find a bug in your application or when you have added some nice new features that you want your users to benefit from, you only have to upgrade the application on the web server and your users are able to benefit from your new application immediately.
- At least in theory, it is easier to have cross-platform support (i.e., to support several operating systems such as Windows, Mac OS, Linux, and so on) since the application is executing inside the web browser.

However, to be honest, web applications also have had (and still have) some limitations compared to native applications. One limitation has been that the user interface of web applications has not been as rich as for their native application counterparts. This changed a lot with the introduction of the HTML5 canvas tag, which made it possible to create really advanced 2D graphics for your web applications. But the initial HTML5 canvas tag only specified a 2D context that does not support 3D graphics.

With WebGL, you get hardware-accelerated 3D graphics inside the browser. You can create 3D games or other advanced 3D graphics applications, and at the same time have all the benefits that a web application has. In addition to these benefits, WebGL also has the following attractive characteristics:

- WebGL is an open standard that everyone can implement or use without paying royalties to anyone.
- WebGL takes advantage of the graphics hardware to accelerate the rendering, which means it is really fast.
- WebGL runs natively in the browsers that support it; no plug-in is needed.
- Since WebGL is based on OpenGL ES 2.0, it is quite easy to learn for many developers who have previous experience with this API, or even for developers who have used desktop OpenGL with shaders.

The WebGL standard also offers a great way for students and others to learn and experiment with 3D graphics. There is no need to download and set up a toolchain like you have to do for most other 3D APIs. To create your

WebGL application, you only need a text editor to write your code, and to view your creation you can just load your files into a web browser with WebGL support.

DESIGNING A GRAPHICS API

There are two fundamentally different ways to design a graphics API:

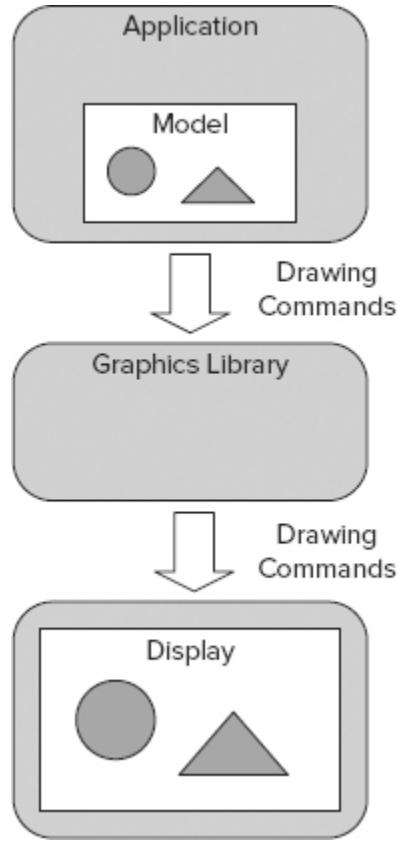
- Using an immediate-mode API
- Using a retained-mode API

WebGL is an immediate-mode API.

An Immediate-Mode API

For an immediate-mode API, the whole scene needs to be redrawn on every frame, regardless of whether it has changed. The graphics library that exposes the API does not save any internal model of the scene that should be drawn. Instead, the application needs to have its own representation of the scene that it keeps in memory. This design gives a lot of flexibility and control to the application. However, it also requires some more work for the application, such as keeping track of the model of the scene and doing initialization and cleanup work. [Figure 1-1](#) shows a simplified diagram of how an immediate-mode API works.

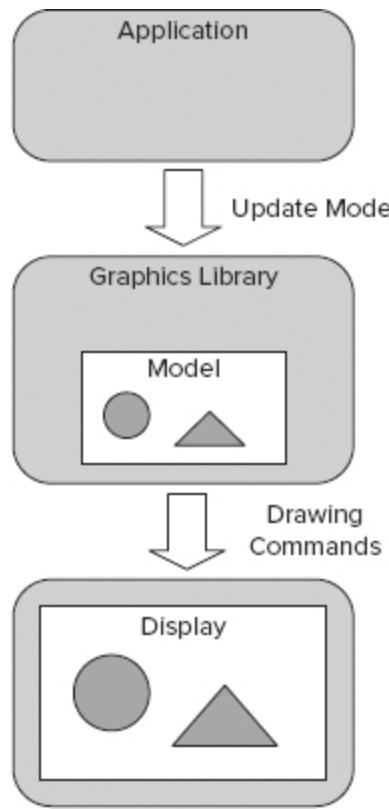
FIGURE 1-1: A diagram of how an immediate-mode API works



A Retained-Mode API

A graphics library that exposes a retained-mode API contains an internal model or scene graph with all the objects that should be rendered. When the application calls the API, it is the internal model that is updated, and the library can then decide when the actual drawing to the screen should be done. This means that the application that uses the API does not need to issue drawing commands to draw the complete scene on every frame. A retained-mode API can in some ways be easier to use since the graphics library does some work for you, so you don't have to do it in your application. [Figure 1-2](#) shows a diagram of how a retained-mode API works. An example of a retained-mode API is Scalable Vector Graphics (SVG), which is described briefly later in this chapter.

FIGURE 1-2: A diagram of how a retained-mode API works

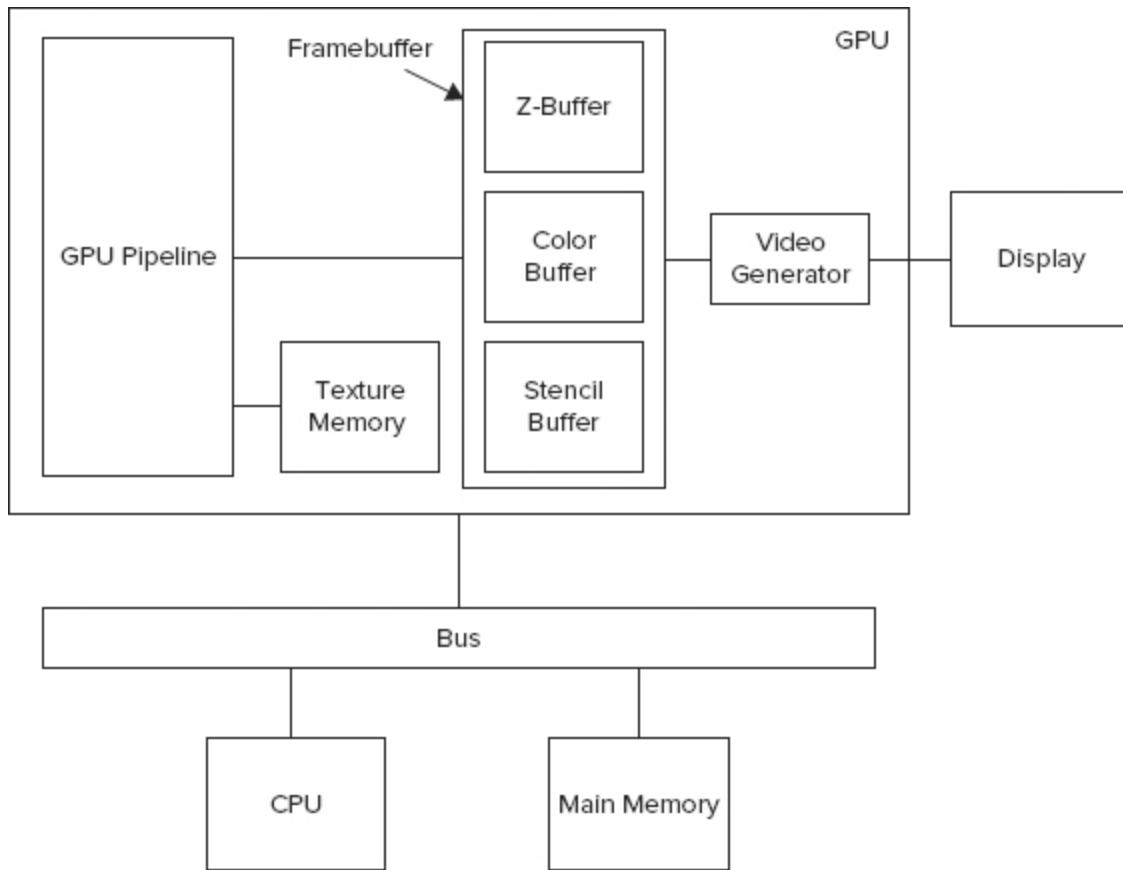


AN OVERVIEW OF GRAPHICS HARDWARE

WebGL is a low-level API and since it is based on OpenGL ES 2.0, it works closely with the actual graphics hardware. To be able to understand the concepts in the rest of this book, it is good to have a basic understanding of graphics hardware and how it works. You probably already know most of this, but to be sure you have the necessary knowledge, this section offers a short overview of the basics.

[Figure 1-3](#) shows a simplified example of a computer system. The application (whether it is a WebGL application that executes in the web browser or some other application) executes on the CPU and uses the main memory (often referred to simply as RAM). To display 3D graphics, the application calls an API that in turn calls a low-level software driver that sends the graphics data over a bus to the graphics processing unit (GPU).

FIGURE 1-3: A simplified diagram of a graphics hardware and its relation to other hardware



GPU

The GPU is a dedicated graphics-rendering device that is specially designed to generate graphics that should be displayed on the screen. A GPU is usually highly parallelized and manipulates graphics data very quickly. The term GPU was first coined and marketed by NVIDIA when they released their GeForce 256 in 1999 as the first GPU in the world.

The GPU is typically implemented as a pipeline where data is moved from one stage in the pipeline to the next stage. Later in this chapter you will learn the different steps of the WebGL graphics pipeline, which consists of conceptual pipeline stages that are then mapped to the physical pipeline stages of the GPU.

Framebuffer

When the graphics data has traversed the complete GPU pipeline, it is finally written to the framebuffer. The *framebuffer* is memory that contains the information that is needed to show the final image on the display. The physical memory that is used for the framebuffer can be located in different places. For a simple graphics system, the framebuffer could actually be allocated as part of the usual main memory, but modern graphics systems normally have a framebuffer that is allocated in special fast graphics memory on the GPU or possibly on a separate chip very close to the GPU.

The framebuffer usually consists of at least three different sub-buffers:

- Color buffer
- Z-buffer
- Stencil buffer

Color Buffer

The color buffer is a rectangular array of memory that contains the color in RGB or RGBA format for each pixel on the screen. The color buffer has a certain number of bits allocated for the colors red, green, and blue (RGB). It may also have an *alpha channel*, which means that it has a certain number of bits allocated to describe the transparency (or opacity) of the pixel in the framebuffer. The total number of bits available to represent one pixel is referred to as the *color depth* of the framebuffer. Examples of color depths are:

- 16 bits per pixel
- 24 bits per pixel
- 32 bits per pixel

A framebuffer with 16 bits per pixel is often used in smaller devices such as some simpler mobile phones. When you have 16 bits per pixel, a common allocation between the different colors is to have 5 bits for red, 6 bits for green, 5 bits for blue, and no alpha channel in the framebuffer. This format is often referred to as RGB565. The reason for selecting green to have an additional bit is that the human eye is most sensitive to green light. Allocating 16 bits for your colors gives $2^{16} = 65,536$ colors in total.

In the same way, a framebuffer with a color depth of 24 bits per pixel usually allocates 8 bits for red, 8 bits for green, and 8 bits for blue. This

gives you more than 16 million colors and no alpha channel in the framebuffer.

A framebuffer with 32 bits per pixel usually has the same allocation of bits as a 24-bit framebuffer — i.e., 8 bits for red, 8 bits for green, and 8 bits for blue. In addition, the 32-bit framebuffer has 8 bits allocated for an alpha channel.

Here you should note that the alpha channel in the framebuffer is not very commonly used. The alpha channel in the framebuffer is usually referred to as the destination alpha channel and is different from the source alpha channel that represents the transparency of the incoming pixels. For example, the process called *alpha blending*, which can be used to create the illusion of transparent objects, needs the source alpha channel but not the destination alpha channel in the framebuffer.



You will learn more about alpha blending in Chapter 8.

Z-Buffer

The color buffer should normally contain the colors for the objects that the viewer of the 3D scene can see at a certain point in time. Some objects in a 3D scene might be hidden by other objects and when the complete scene is rendered, the pixels that belong to the hidden objects should not be available in the color buffer.

Normally this is achieved in graphics hardware with the help of the *Z-buffer*, which is also referred to as the *depth buffer*. The Z-buffer has the same number of elements as there are pixels in the color buffer. In each element, the Z-buffer stores the distance from the viewer to the closest primitive.



You will learn exactly how the Z-buffer is used to handle the depth in the scene in the “Depth Buffer Test” section later in this chapter.

Stencil Buffer

In addition to the color buffer and the Z-buffer — which are the two most commonly used buffers in the framebuffer — modern graphics hardware also contains a stencil buffer. The *stencil buffer* can be used to control where in the color buffer something should be drawn. A practical example of when it can be used is for handling shadows.

Texture Memory

An important operation in 3D graphics is applying a texture to a surface. You can think of texturing as a process that “glues” images onto geometrical objects. These images, which are called *textures*, need to be stored so that the GPU can access them quickly and efficiently. Usually the GPU has a special texture memory to store the textures.



You will learn more about texturing in Chapter 5.

Video Controller

The *video controller* (also called a *video generator*) scans through the color buffer line-by-line at a certain rate and updates the display. The whole display is typically updated 60 times per second for an LCD display. This is referred to as a refresh rate of 60 Hz.

UNDERSTANDING THE WEBGL GRAPHICS PIPELINE

A web application that uses WebGL typically consists of HTML, CSS, and JavaScript files that execute within a web browser. In addition to this classical web application content, a WebGL application also contains source code for its shaders and some sort of data representing the 3D (or possibly 2D) models it displays.

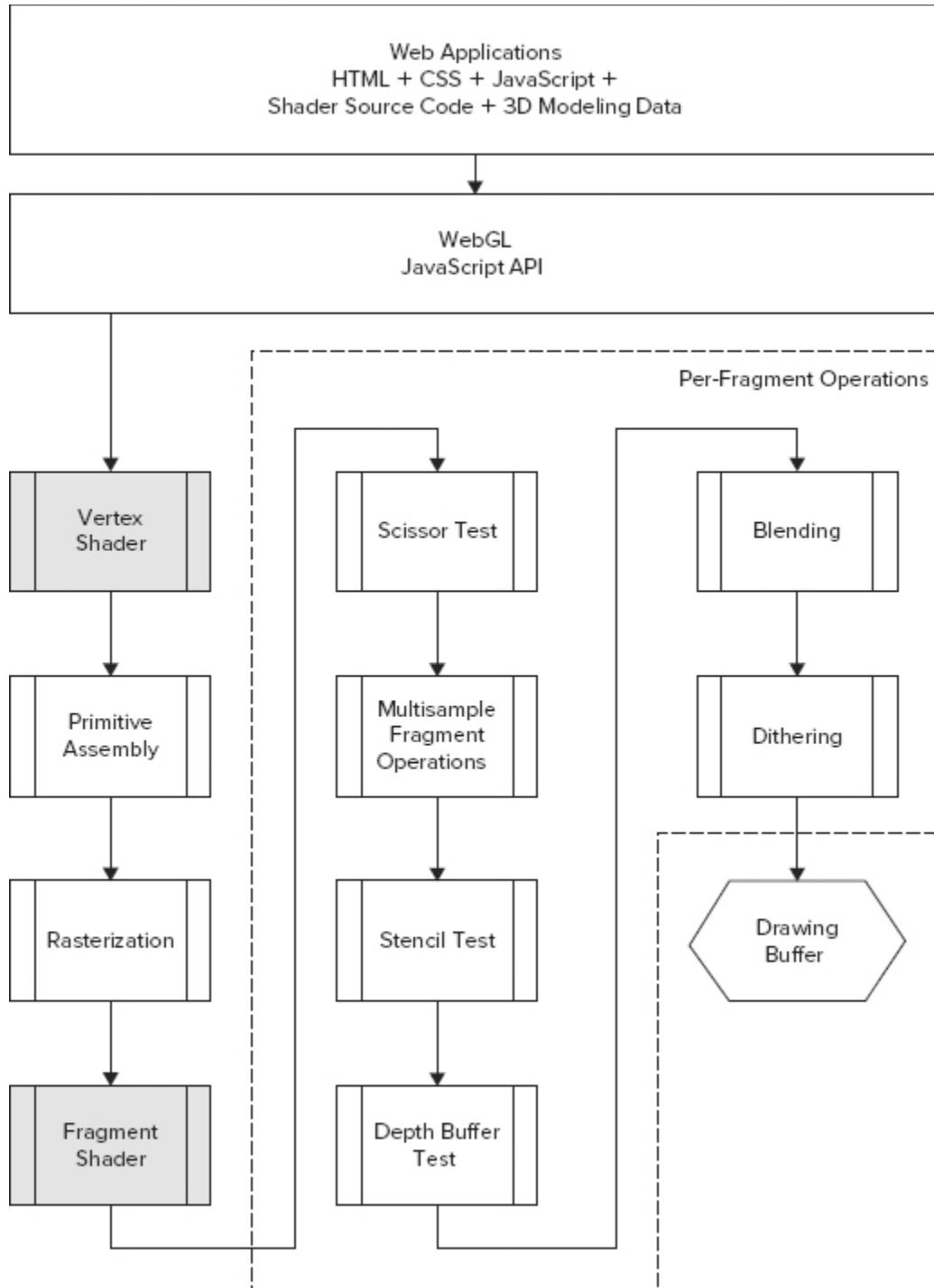
The browser does not require a plug-in to execute WebGL; the support is natively built into the browser. It is the JavaScript that calls the WebGL API

to send in information to the WebGL pipeline for how the 3D models should be rendered. This information consists of not only the source code for the two programmable stages in the WebGL pipeline, the vertex shader, and the fragment shader, but also information about the 3D models that should be rendered.

After the data has traversed the complete WebGL pipeline, the result is written to something that WebGL calls the *drawing buffer*. You can think of the drawing buffer as the framebuffer for WebGL. It has a color buffer, a Z-buffer, and a stencil buffer in the same way as the framebuffer. However, the result in the drawing buffer is composited with the rest of the HTML page before it ends up in the physical framebuffer that is actually displayed on the screen.

In the following sections, you will learn about the different stages of the WebGL pipeline that are shown in [Figure 1-4](#). As shown in the figure, there are several stages in the pipeline. The most important stages for you as a WebGL programmer are the vertex shader and the fragment shader. You will be introduced to several new terms in the following sections. Some will be explained in this chapter, while the explanation for other terms and concepts will come in later chapters. The following section is an introduction to the WebGL graphics pipeline, which means that you do not need to understand every detail presented here.

FIGURE 1-4: An overview of the WebGL graphics pipeline



Vertex Shader

To get a realistic 3D scene, it is not enough to render objects at certain positions. You also need to take into account things like how the objects will look when light sources shine on them. The general term that is used for the

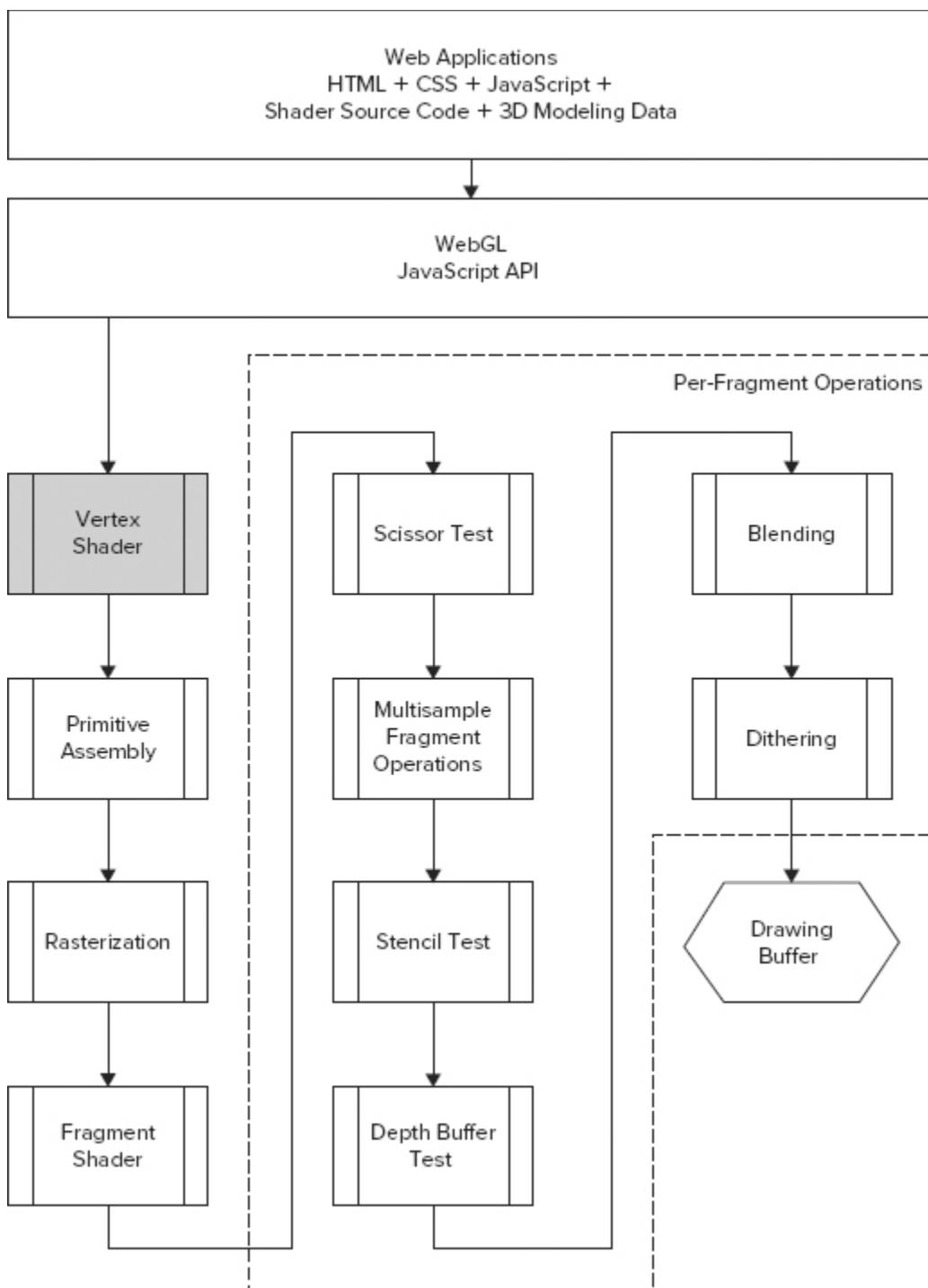
process of determining the effect of light on different materials is called *shading*.

For WebGL, the shading is done in two stages:

- **Vertex shader**
- **Fragment shader**

The first stage is the vertex shader. (The fragment shader comes later in the pipeline and is discussed in a later section in this chapter.) The name *vertex shader* comes from the fact that a 3D point that is a corner or intersection of a geometric shape is often referred to as a vertex (or vertices in plural). The vertex shader is the stage in the pipeline that performs shading for a vertex. [Figure 1-5](#) shows where the vertex shader is located in the WebGL graphics pipeline.

FIGURE 1-5: The location of the vertex shader in the WebGL graphics pipeline



The vertex shader is where the 3D modeling data (e.g., the vertices) first ends up after it is sent in through the JavaScript API. Since the vertex shader is programmable and its source code is written by you and sent in through the JavaScript API, it can actually manipulate a vertex in many ways.

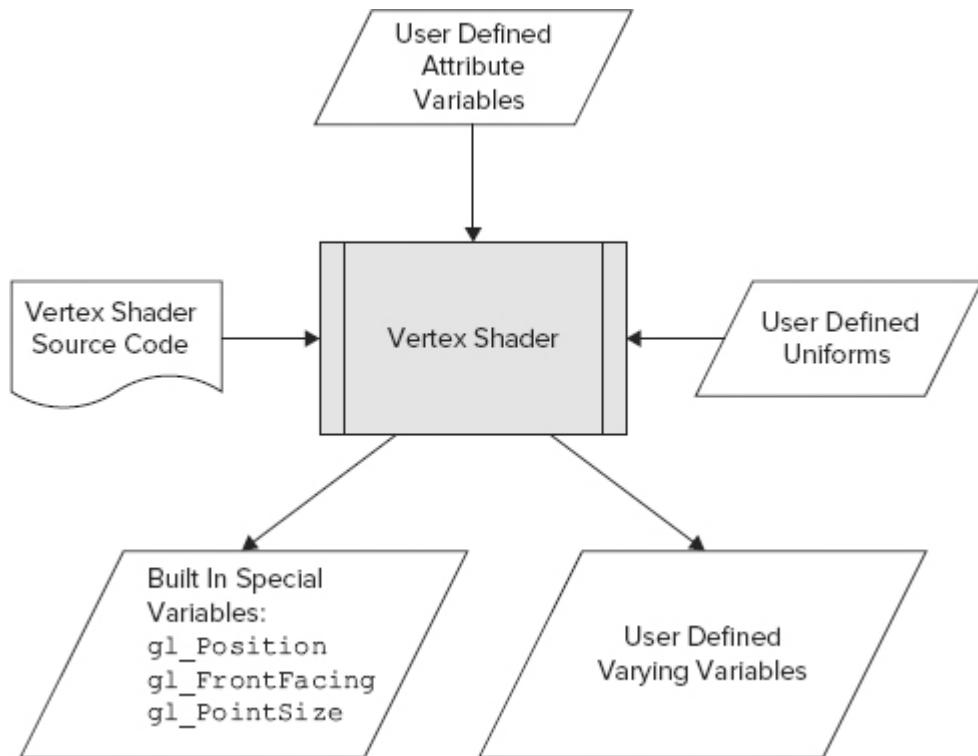
Before the actual shading starts, it often transforms the vertex by multiplying it with a transformation matrix. By multiplying all vertices of an object with the transformation matrix, the object can be placed at a specific position in your scene. You will learn more about transformations later in this chapter and also in Chapter 4, so don't worry if you do not understand exactly what this means now.

The vertex shader uses the following input:

- The actual source code that the vertex shader consists of. This source code is written in OpenGL ES Shading Language (GLSL ES).
- Attributes that are user-defined variables that normally contain data specific to each vertex. (There is also a feature called constant vertex attributes that you can use if you want to specify the same attribute value for multiple vertices.) Examples of attributes are vertex positions and vertex colors.
- Uniforms that are data that is constant for all vertices. Examples of uniforms are transformation matrices or the position of a light source. (As covered later in this chapter, you can change the value for a uniform between WebGL draw calls, so it is only during a draw call that the uniform needs to be constant.)

The output from the vertex shader is shown at the bottom of [Figure 1-6](#) and consists of user-defined varying variables and some built-in special variables.

FIGURE 1-6: An overview of the vertex shader



The varying variables are a way for the vertex shader to send information to the fragment shader. You will take a closer look at the built-in special variables in later chapters. For now, it is enough to understand that the built-in variable `gl_Position` is the most important one, and it contains the position for the vertex after the vertex shader is finished with its job.

The following source code snippet shows an example of a basic vertex shader. Again, you learn more about vertex shaders in later chapters; this source code simply shows you what it looks like:

```

attribute vec3 aVertexPos;
attribute vec4 aVertexColor;

uniform mat4 uMVMMatrix;
uniform mat4 uPMatrix;

varying vec4 vColor;

void main() {
    gl_Position = uPMatrix * uMVMMatrix * vec4(aVertexPos,
1.0);

    vColor = aVertexColor;
}

```

As previously mentioned, this source code is written in OpenGL ES Shading Language. As you can see, the syntax is quite similar to the C programming language. There are some differences in, for example, the supported data types, but if you have programmed using C before, then many things will be familiar to you. Although you don't need to understand every last detail of how this works at this particular time, the following snippets provide a bit more insight.

Starting from the top of the code, the first two lines declare two attribute variables:

```
attribute vec3 aVertexPos;  
attribute vec4 aVertexColor;
```

Once again, the attributes are user-defined variables that contain data specific to each vertex. The actual values for the attribute variables are sent in through the WebGL JavaScript API.

The first variable is called `aVertexPos` and is a vector with three elements. It contains the position for a single vertex. The second variable is named `aVertexColor` and is a vector with four elements. It contains the color for a single vertex.

The next two lines of source code define two uniform variables of the type `mat4`:

```
uniform mat4 uMVMMatrix;  
uniform mat4 uPMatrix;
```

The type `mat4` represents a 4×4 matrix. The two uniform variables in this example contain the transformations that should be applied to each vertex. In the same way as for the attribute variables, the uniforms are set from the WebGL JavaScript API. The difference is that uniforms normally contain data that are constant for all vertices.

The last declaration is the varying variable that is named `vColor`; it contains the output color from the vertex shader:

```
varying vec4 vColor;
```

This varying variable will be input to the fragment shader.

After the declarations of all the variables comes the entry point for the vertex shader:

```
void main() {  
    gl_Position = uPMatrix * uMVMMatrix * vec4(aVertexPos,  
1.0);
```

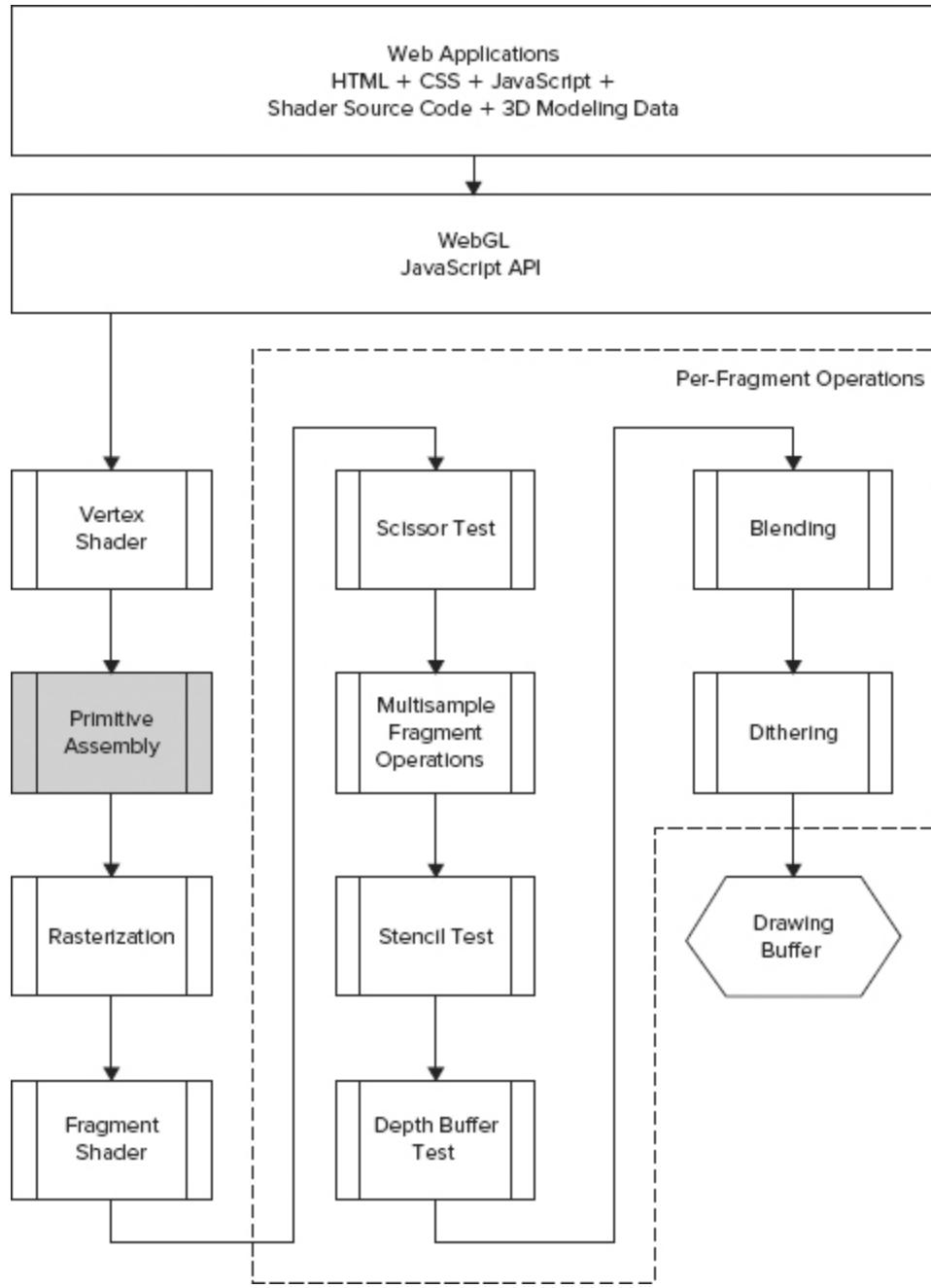
Both for the vertex shader and the fragment shader, the entry point is the `main()` function. The first statement in the `main()` function takes the vertex position `aVertexPos` and multiplies it by the two transformation matrices to perform the transformation. The result is written to the special built-in variable `gl_Position` that contains the position of a single vertex after the vertex shader is finished with it. The last thing the vertex shader does is to take the attribute that contains the color and that was sent in through the WebGL JavaScript API and write this to varying variable `vColor` so it can be read by the fragment shader:

```
vColor = aVertexColor;  
}
```

Primitive Assembly

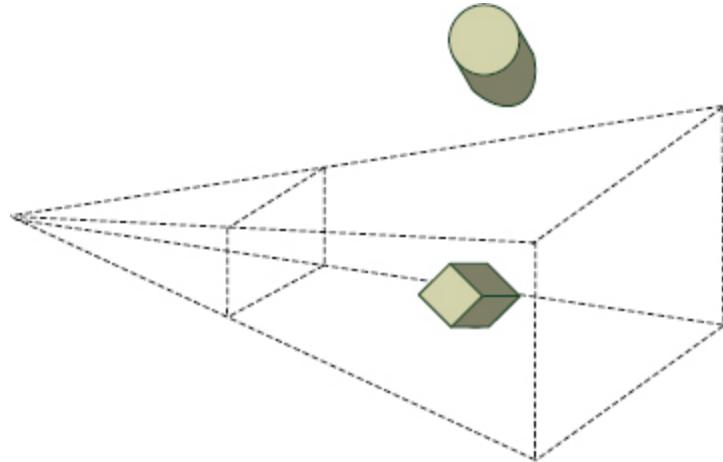
In the step after the vertex shader — known as *primitive assembly* (see [Figure 1-7](#)) — the WebGL pipeline needs to assemble the shaded vertices into individual geometric primitives such as triangles, lines, or point sprites. Then for each triangle, line, or point sprite, WebGL needs to decide whether the primitive is within the 3D region that is visible on the screen for the moment. In the most common case, the visible 3D region is called the *view frustum* and is a truncated pyramid with a rectangular base.

FIGURE 1-7: The location of the primitive assembly stage in the WebGL graphics pipeline



Primitives inside the view frustum are sent to the next step in the pipeline. Primitives outside the view frustum are completely removed, and primitives partly in the view frustum are clipped so the parts that are outside the view frustum are removed. [Figure 1-8](#) shows an example of a view frustum with a cube that is inside the view frustum and a cylinder that is outside the view frustum. The primitives that build up the cube are sent to the next stage in the pipeline, while the primitives that build up the cylinder are removed during this stage.

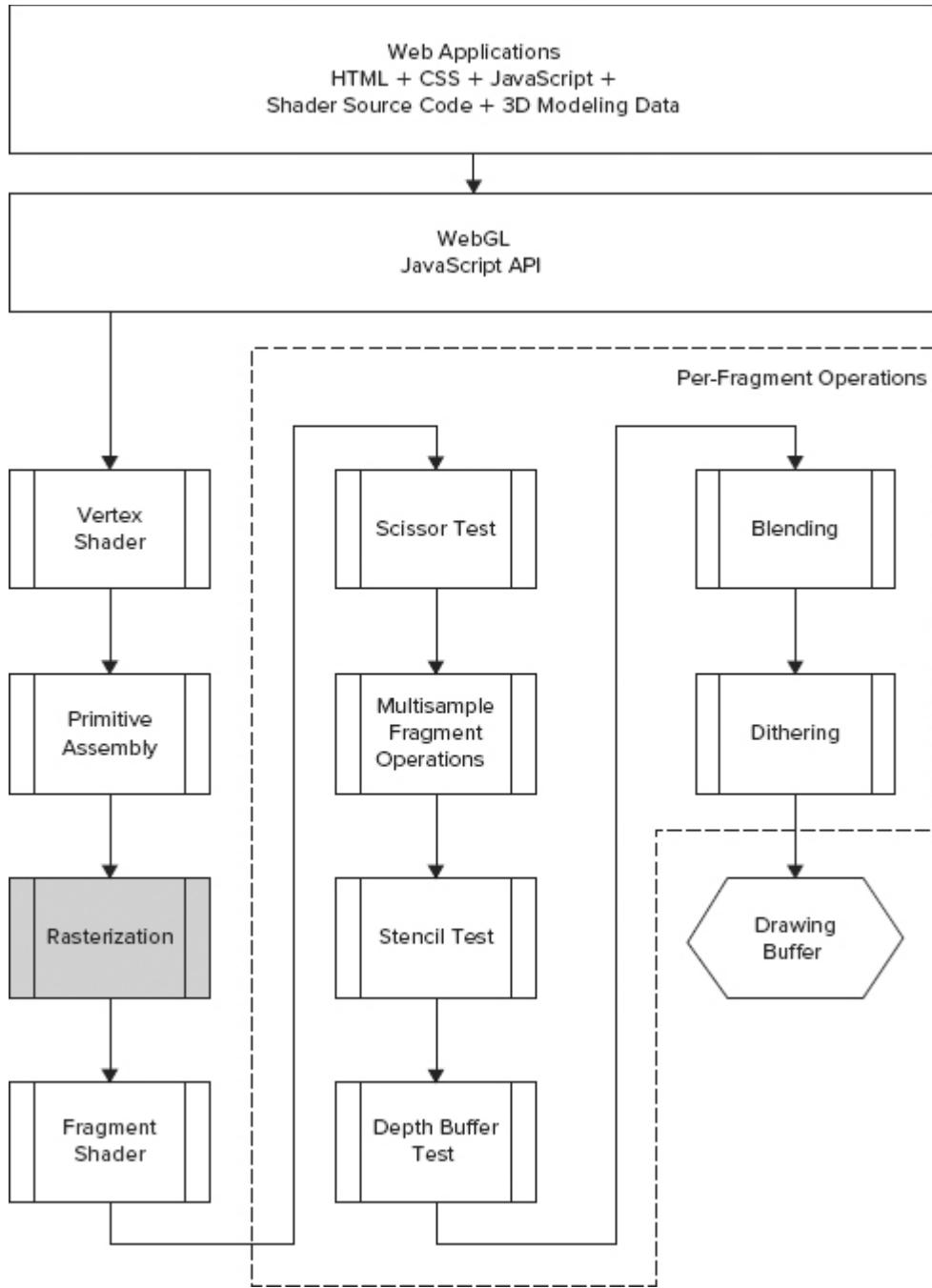
FIGURE 1-8: A view frustum with a cube that is inside the frustum and a cylinder that is outside the frustum



Rasterization

The next step in the pipeline is to convert the primitives (lines, triangles, and point sprites) to fragments that should be sent to the fragment shader. You can think of a fragment as a pixel that can finally be drawn on the screen. This conversion to fragments happens in the rasterization stage (see [Figure 1-9](#)).

FIGURE 1-9: The location of rasterization in the WebGL graphics pipeline

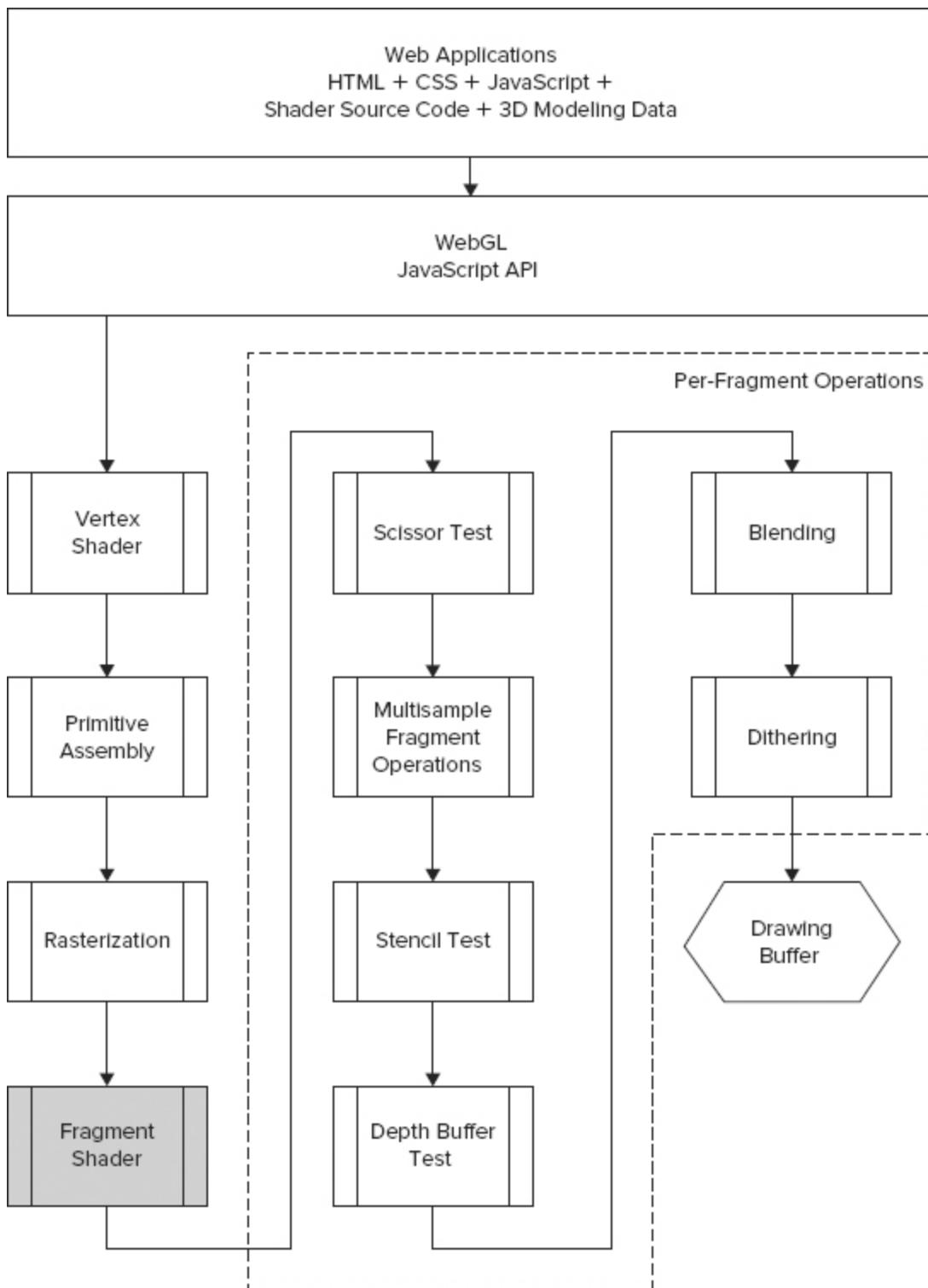


Fragment Shader

The fragments from the rasterization are sent to the second programmable stage of the pipeline, which is the fragment shader (see [Figure 1-10](#)). As mentioned earlier, a fragment basically corresponds to a pixel on the screen. However, not all fragments become pixels in the drawing buffer since the per-fragment operations (which are described next) might discard some

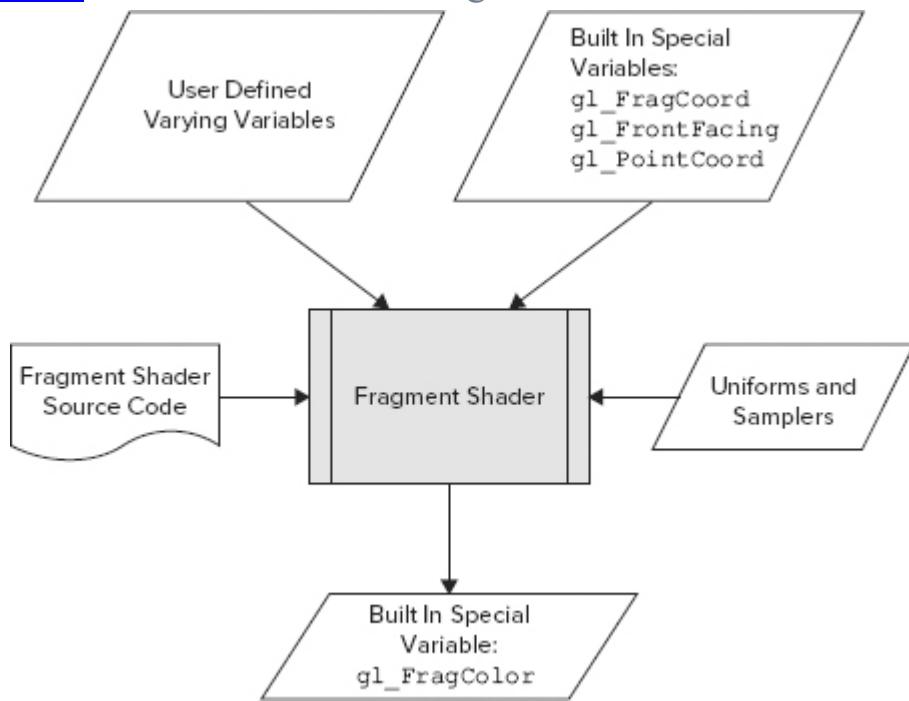
fragments in the last steps of the pipeline. So WebGL differentiates between fragments and pixels. Fragments are called pixels when they are finally written to the drawing buffer.

FIGURE 1-10: The Location of the fragment shader in the WebGL graphics pipeline



In other 3D rendering APIs, such as Direct3D from Microsoft, the fragment shader is actually called a pixel shader. [Figure 1-11](#) shows the input and output of the fragment shader.

FIGURE 1-11: An overview of the fragment shader

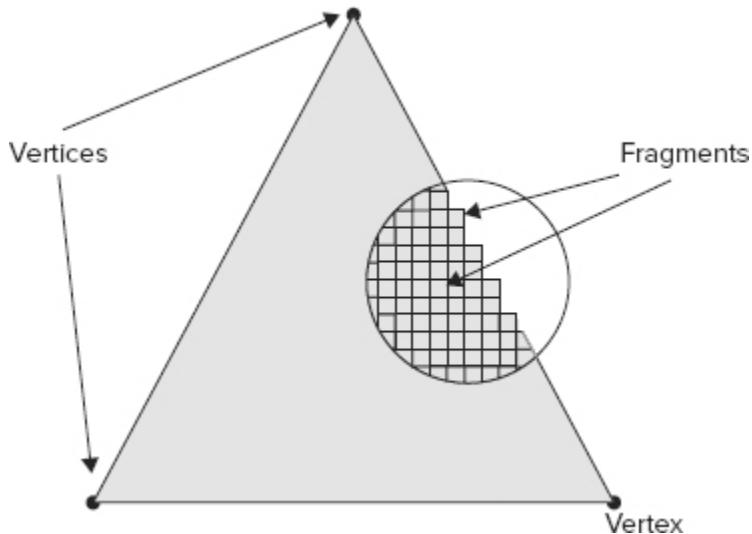


The input to the fragment shader consists of the following:

- The source code for the fragment shader, which is written in OpenGL ES Shading Language
- Some built-in special variables (e.g., `gl_PointCoord`)
- User-defined varying variables, which have been written by the vertex shader
- Uniforms, which are special variables that contain data that are constant for all fragments
- Samplers, which are a special type of uniforms that are used for texturing

As mentioned during the discussion of the vertex shader, the varying variables are a way to send information from the vertex shader to the fragment shader. However, as shown in [Figure 1-12](#), there are generally more fragments than there are vertices. The varying variables that are written in the vertex shader are linearly interpolated over the fragments. When a varying variable is read in the fragment shader, it is the linearly interpolated value that can be different from the values written in a vertex shader. This will make more sense to you in later chapters, after you work more with source code for the shaders.

FIGURE 1-12: The relation between vertices and fragments



The output from the fragment shader is the special built-in variable `gl_FragColor` to which the fragment shader writes the color for the fragment. The following snippet of source code shows a simple fragment shader written in OpenGL ES Shading Language:

```
precision mediump float;

varying vec4 vColor;
void main() {

    gl_FragColor = vColor;

}
```

This fragment shader example starts with what is called a precision qualifier:

```
precision mediump float;
```

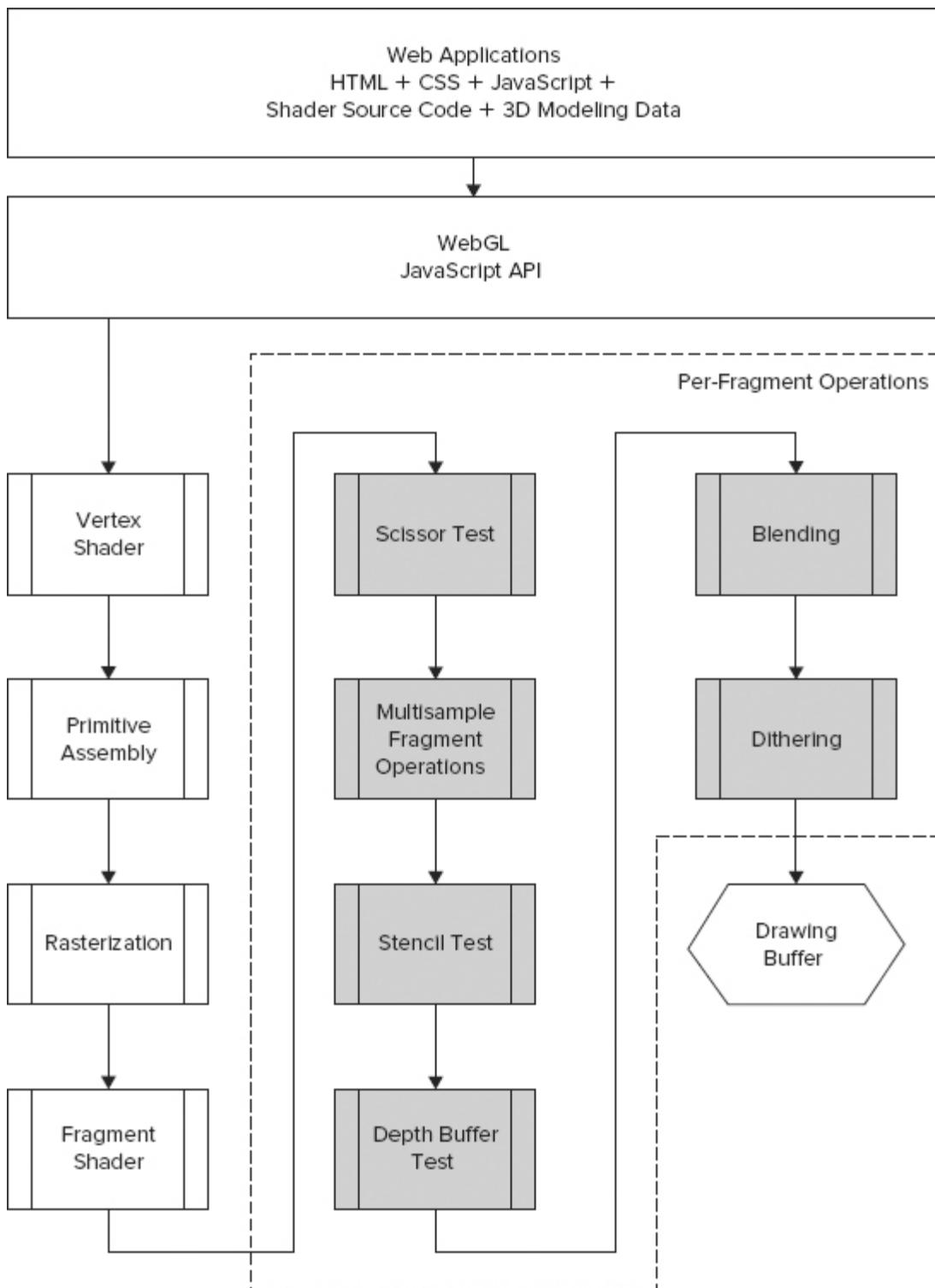
The idea is that the precision qualifier should be a hint to the shader compiler so it knows the minimum precision it must use for a variable or data type. It is mandatory to specify the precision of the float data type in all fragment shaders.

After the precision qualifier, the fragment shader declares the varying variable `vColor`. In the `main()` function, the varying variable `vColor` is written to the built-in special variable `gl_FragColor`. Again note that the value in the varying variable `vColor` is linearly interpolated from the values written to this varying variable in the vertex shader.

Per Fragment Operations

After the fragment shader, each fragment is sent to the next stage of the pipeline, which consists of the per-fragment operations. As the name indicates, this step actually contains several sub-steps. Each fragment output from the fragment shader can modify a pixel in the drawing buffer in different ways, depending on the conditions and results of the different steps in the per-fragment operations. In [Figure 1-13](#), the per-fragment operations are shown inside the dotted line in the right part of the figure. To control the behavior, you can disable and enable functionality in the different steps.

FIGURE 1-13: The location of the per-fragment operations in the WebGL graphics pipeline



Scissor Test

The scissor test determines whether the fragment lies within the scissor rectangle that is defined by the left, bottom coordinate and a width and a

height. If the fragment is inside the scissor rectangle, then the test passes and the fragment is passed to the next stage. If the fragment is outside of the scissor rectangle, then the fragment is discarded and will never reach the drawing buffer.

Multisample Fragment Operations

This step modifies the fragment's alpha and coverage values as a way to perform *anti-aliasing*. In computer graphics, anti-aliasing refers to techniques to improve the appearance of polygon edges so they are not "jagged" — they are instead smoothed out on the screen.

Stencil Test

The stencil test takes the incoming fragment and performs tests on the stencil buffer to decide whether the fragment should be discarded. For example, if you draw a star into the stencil buffer, you can then set different operators to decide whether subsequent painting into the color buffer should affect the pixels inside or outside the star.

Depth Buffer Test

The depth buffer test discards the incoming fragment depending on the value in the depth buffer (also called the Z-buffer). When a 3D scene is rendered in a 2D color buffer, the color buffer can only contain the colors for the objects in the scene that are visible to the viewer at a specific time. Some objects might be obscured by other objects in the scene. The depth buffer and depth buffer test decide which pixels should be displayed in the color buffer. For each pixel, the depth buffer stores the distance from the viewer to the currently closest primitive.

For an incoming fragment, the z-value of that fragment is compared to the value of the depth buffer at the same position. If the z-value for the incoming fragment is smaller than the z-value in the depth buffer, then the new fragment is closer to the viewer than the pixel that was previously in the color buffer and the incoming fragment is passed through the test. If the z-value of the incoming fragment is larger than the value in the Z-buffer, then the new

fragment is obscured by the pixel that is currently in the drawing buffer and therefore the incoming fragment is discarded.

Blending

The next step is blending. Blending lets you combine the color of the incoming fragment with the color of the fragment that is already available in the color buffer at the corresponding position. Blending is useful when you're creating transparent objects.

Dithering

The last step before the drawing buffer is dithering. The color buffer has a limited number of bits to represent each color. Dithering is used to arrange colors in such a way that an illusion is created of having more colors than are actually available. Dithering is most useful for a color buffer that has few colors available.

COMPARING WEBGL TO OTHER GRAPHICS TECHNOLOGIES

To give you a broader understanding of WebGL and other 3D and 2D graphics technologies, the following sections briefly describe some of the most relevant technologies to you and how they compare to WebGL.

This broader understanding is not necessary to impress your friends or your employer; it's intended to help you understand what you read or hear about these technologies. It will also help you understand which of these other technologies can be interesting to look at or read about in other books to transfer concepts or ideas to your WebGL applications.

OpenGL

OpenGL has long been one of the two leading APIs for 3D graphics on desktop computers. (The other API is Direct3D from Microsoft, which is described in a later section.) OpenGL is a standard that defines a cross-platform API for 3D graphics that is available on Linux, several flavors of Unix, Mac OS X, and Microsoft Windows.

In many ways, OpenGL is very similar to WebGL, so if you have previous experience with OpenGL, then it will be easier to learn WebGL. This is especially true if you have experience using OpenGL with programmable shaders. If you have never used OpenGL in any form, don't worry. You will, of course, learn how to use WebGL in this book.

History of OpenGL

During the early 1990s, Silicon Graphics, Inc. (SGI) was a leading manufacturer of high-end graphics workstations. Their workstations were more than regular general-purpose computers; they had specialized hardware and software to be able to display advanced 3D graphics. As part of their solution, they had an API for 3D graphics that they called IRIS GL API.

Over time, more features were added to the IRIS GL API, even as SGI strove to maintain backward compatibility. The IRIS GL API became harder to maintain and more difficult to use. SGI probably also realized that they would benefit from having an open standard so that it would be easier for programmers to create software that was compatible with their hardware. After all, the software is an important part of selling computers.

SGI phased out IRIS and designed OpenGL as a new “open” and improved 3D API from the ground up. In 1992, version 1.0 of the OpenGL specification was introduced and the independent consortium OpenGL Architecture Review Board (ARB) was founded to govern the future of OpenGL. The founding members of OpenGL ARB were SGI, Digital Equipment Corporation, IBM, Intel, and Microsoft. Over the years, many more members joined, and OpenGL ARB met regularly to propose and approve changes to the specification, decide on new releases, and control the conformance testing.

Sometimes even successful businesses can experience a decline. In 2006, SGI was more or less bankrupt, and control of the OpenGL standard was

transferred from OpenGL ARB to the Khronos Group. Ever since then, the Khronos Group has continued to evolve and promote OpenGL.

An OpenGL Code Snippet

A lot of new functionality has been added to OpenGL since the first version was released in 1992. The general strategy that has been used when developing new releases of OpenGL is that they should all be backward compatible. The later releases contain some functionality that has been marked as deprecated in the specification. In this section you can see a very short snippet of source code that would be used to render a triangle on the screen.

The use of `glBegin()` and `glEnd()` is actually part of functionality that has been marked as deprecated in the latest releases, but it is still used by a lot of developers and exists in a lot of literature, in existing OpenGL source code, and in examples on the web.

```
glClear( GL_COLOR_BUFFER_BIT ); // clear the color buffer
glBegin(GL_TRIANGLES); // Begin specifying a triangle
    glVertex3f( 0.0, 0.0, 0.0 ); // Specify the position of
    the first vertex
    glVertex3f( 1.0, 1.0, 0.0 ); // Specify the position of
    the second vertex
    glVertex3f( 0.5, 1.0, 0.0 ); // Specify the position of
    the third vertex
    glEnd(); // We are
            finished with triangles
```

The function `glClear()` clears the color buffer, so it doesn't contain garbage or pixels from the previous frame. Then `glBegin()` is used to indicate that you will draw a triangle with the vertices defined between `glBegin()` and `glEnd()`. The function `glVertex3f()` is called to specify the location of the three vertices that all have the z-value set to zero.

SOME KEY POINTS TO REMEMBER ABOUT OPENGL

Here are some important things to keep in mind about OpenGL:

- OpenGL is an open standard for desktop 3D graphics.
- OpenGL specifications are now maintained and further developed by the Khronos Group, which also maintains the WebGL specification.
- OpenGL is mainly an immediate-mode API.
- The strategy has been to keep new releases backward compatible, with the result that there are generally many ways to do the same thing.
- OpenGL with programmable shaders is quite similar to WebGL, and when you know WebGL well, you should be able to transfer ideas from OpenGL code to WebGL.
- The shaders in OpenGL can be programmed with a high-level language called OpenGL Shading Language (GLSL).

OpenGL ES 2.0

OpenGL ES (OpenGL for Embedded Systems) is a 3D graphics API that is based on desktop OpenGL. Since WebGL is based on OpenGL ES 2.0, this is the API that is most similar to WebGL. The biggest difference is probably that WebGL is used in an HTML and JavaScript context, while OpenGL ES 2.0 is normally used in a C/C++, Objective C, or Java context. But there are also some other differences in how these languages work. For example, the Khronos Group decided to remove some features that are available in OpenGL ES 2.0 when they created the WebGL specification.

History of OpenGL ES 2.0

The first version of OpenGL ES was called OpenGL ES 1.0 and was based on the desktop OpenGL 1.3 specification. The Khronos Group used the functionality of OpenGL 1.3 as a base and removed redundant functionality or features that were not suitable for embedded devices. For example, the desktop OpenGL 1.3 specification contained functionality to specify the primitives that you wanted to draw by calling the function `glBegin` and then calling, for example, `glVertex3f` to specify the vertices for the primitive you wanted to draw, followed by calling `glEnd` (as you saw in the code snippet shown in the earlier section about OpenGL). This functionality was removed in OpenGL ES since it is possible to achieve the same result in a more generic way with what are called vertex arrays.

An OpenGL ES 2.0 Code Snippet

Below, you see a short snippet of OpenGL ES 2.0 source code. Since OpenGL ES 2.0 is fully shader-based (in the same way as WebGL), quite a lot of source code is needed to create even a simple example. The snippet below contains some source code that would be part of an application that draws a single triangle on the screen.

```
GLfloat triangleVertices[] = {0.0f, 1.0f, 0.0f,
                               -1.0f, -1.0f, 0.0f,
                               1.0f, -1.0f, 0.0f};

// Clear the color buffer
glClear(GL_COLOR_BUFFER_BIT);

// Specify program object that contains the linked
shaders
glUseProgram(programObject);

// Load the vertex data into the shader
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
triangleVertices);
 glEnableVertexAttribArray(0);
 glDrawArrays(GL_TRIANGLES, 0, 3);
eglSwapBuffers(myEglDisplay, myEglSurface);
```

First, you see a vector called `triangleVertices` that specifies the three vertices for the triangle. Then the color buffer is cleared with a call to `glClear()`. What is not shown in this short code snippet is how the vertex shader and the fragment shader are compiled and linked to a program object. These steps have to happen before the call to `glUseProgramObject()` that specifies which program object you want to use for rendering. The program object contains the compiled and linked vertex shader and fragment shader.

SOME KEY POINTS TO REMEMBER ABOUT OPENGL ES 2.0

Here are some important things to keep in mind about OpenGL ES 2.0:

- OpenGL ES 2.0 is an open standard for 3D graphics for embedded devices such as mobile phones.
- OpenGL ES 1.x and 2.0 specifications were created by the Khronos Group, which promotes the specifications.
- OpenGL ES 2.0 is an immediate-mode API.
- OpenGL ES 2.0 is not backward compatible with previous releases. This is a different strategy than that used for desktop OpenGL.
- OpenGL ES 2.0 is very similar to WebGL and you can transfer source code and ideas very easily from OpenGL ES 2.0 to WebGL. OpenGL ES Shading Language is also used as the programming language for the shaders in both OpenGL ES 2.0 and WebGL.

Direct3D

DirectX is the name of the Microsoft multimedia and game programming API. One important part of this API is Direct3D for 3D graphics programming. Direct3D can be used for many programming languages, such as C++, C#, and Visual Basic .NET. But even though it has support for several languages, it only works on devices that run the Microsoft Windows operating system.

On a conceptual level, Direct3D has similarities with OpenGL, OpenGL ES, and WebGL since it is also an API to handle 3D graphics. If you have experience with Direct3D, then you probably have a good understanding of 3D graphics concepts such as the graphics pipeline, the Z-buffer, shaders, and texturing. Even though Direct3D uses different naming of some of the concepts, many things will be familiar. However, Direct3D is very different from WebGL when it comes to the details of the APIs.

A Brief History of Direct3D

In 1995 Microsoft bought a company called RenderMorphics, which developed a 3D graphics API called Reality Lab. Microsoft used the expertise from RenderMorphics to implement the first version of Direct3D, which they shipped in DirectX 2.0 and DirectX 3.0. The decision by

Microsoft to not embrace OpenGL but instead create Direct3D as their proprietary API has led to increased fragmentation for 3D graphics on desktops. On the other hand, it has probably also resulted in healthy competition for the 3D graphics industry that has led to innovation for both OpenGL and Direct3D.

An important milestone for Direct3D was when programmable shaders were introduced in version 8.0. The shaders were programmed in an assembly-like language. In Direct3D 9.0, a new shader programming language was released. Called High Level Shading Language (HLSL), it was developed by Microsoft in collaboration with NVIDIA. HLSL for Direct3D corresponds to OpenGL ES Shading Language for WebGL and OpenGL ES 2.0.

A Direct3D Code Snippet

The code snippet that follows shows how a scene is rendered and displayed with Direct3D. Before this code would be executed, you would typically have to set up and initialize Direct3D and create a Direct3D Device. In this code snippet, it is assumed that a pointer to the Direct3D Device is stored in the global variable `g_pd3dDevice`. This pointer is used to access the functions of the API.

```
void render(void) {  
  
    // clear the back buffer  
    g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,  
                          D3DCOLOR_COLORVALUE(0.0f,0.0f,0.0f,1.0f), 1.0f, 0 );  
  
    // Signal to the system that rendering will begin  
    g_pd3dDevice->BeginScene();  
  
    // Render geometry here...  
  
    // Signal to the systme that rendering is finished  
    g_pd3dDevice->EndScene();  
  
    // Show the rendered geometry on the display  
    g_pd3dDevice->Present( NULL, NULL, NULL, NULL );  
}
```

The code starts by calling `Clear()` to clear the color buffer. This call looks a bit more complicated than the corresponding call for OpenGL, OpenGL ES 2.0, or WebGL, since it has more arguments.

The first two arguments are very often zero and `NULL`. You can use them to specify that you want to clear only part of the viewport. Setting zero and `NULL` means that you want to clear the complete color buffer.

The third argument is set to `D3DCLEAR_TARGET`, which means that you clear the color buffer. It would be possible to specify flags to clear the Z-buffer and the stencil buffer in this argument as well.

The fourth argument specifies the RGBA color that you want to clear the color buffer to. In this case, the color buffer is cleared to an opaque black color.

The last two arguments are used to set the values that the Z-buffer and the stencil buffer should be cleared to. Since the third argument only specified that you wanted to clear the color buffer, the values of the last two arguments do not matter in this example.

In Direct3D you have to call `BeginScene()` to specify that you want to begin to draw your 3D scene. You call `EndScene()` to specify that you are finished with the drawing. Between these two calls, you specify all the geometry that you want to render. In the previous code snippet, no code is included to render the geometry.

Finally you would call the function `Present()` to display the back buffer you just rendered on the display.

SOME KEY POINTS TO REMEMBER ABOUT DIRECT3D

Here are some important things to keep in mind about Direct3D:

- Direct3D is a Microsoft proprietary standard for 3D graphics.
- Direct3D only works on devices running Microsoft Windows.
- Direct3D uses a similar graphics pipeline to OpenGL, OpenGL ES 2.0, and WebGL.
- Direct3D uses HLSL to write source code for the shaders. This language corresponds to GLSL for desktop OpenGL, and OpenGL ES Shading Language for OpenGL ES 2.0 and WebGL.
- A pixel shader in Direct3D corresponds to a fragment shader in OpenGL, OpenGL ES 2.0, or WebGL.

HTML5 Canvas

HTML5 is the fifth iteration of Hyper Text Markup Language, or HTML. The HTML5 specification contains a lot of interesting new features for you as a web developer. One of the most interesting ones is the HTML5 canvas.

The HTML5 canvas is a rectangular area of your web page where you can draw graphics by using JavaScript. In the context of WebGL, the HTML5 canvas is especially interesting because it was the basis for the initial WebGL experiments that were performed by Vladimir Vukićević at Mozilla. WebGL is now designed as a rendering context for the HTML5 canvas element. The original 2D rendering context (the `CanvasRenderingContext2D` interface) that is supported by the HTML5 canvas can be retrieved from the canvas element by calling the following code:

```
var context2D = canvas.getContext("2d");
```

A WebGL rendering context (the `WebGLRenderingContext` interface) can be retrieved from the canvas element in the same way, but instead of specifying the string "2d", you specify "webgl" like this:

```
var contextWebGL = canvas.getContext("webgl");
```

After retrieving a rendering context as above, you can use the `context2D` to call functions supported by the original HTML5 canvas (the `CanvasRenderingContext2D`). Conversely, you could use the `contextWebGL` to call functions supported by WebGL (the `WebGLRenderingContext`).

A Brief History of HTML5 Canvas

The Apple Mac OS X operating system contains an application called Dashboard. You are probably familiar with it if you are a Mac user. If not, *Dashboard* is basically an application that hosts mini-applications called widgets. These widgets are based on the same technologies used by web pages, such as HTML, CSS, and JavaScript.

Both Dashboard and the Safari web browser use the WebKit open source browser engine to render web content, and by introducing the `canvas` tag in WebKit in 2004, Apple created a totally new way of drawing 2D graphics in these applications. In 2005, it was implemented by Mozilla Firefox and in

2006 by Opera. The canvas tag was included in the HTML5 specification, and in 2011, Microsoft released Internet Explorer 9, which was the first version of Internet Explorer with canvas support.

An HTML5 Canvas Code Snippet

The code in [Listing 1-1](#) shows how the canvas element is used to draw some simple 2D graphics from JavaScript. All code is embedded within a single HTML file. If you start by looking at the bottom of the listing, you can see that the only element that is within the `<body>` start tag and the `</body>` end tag of the page is the `<canvas>` element. The `<canvas>` defines the rendering surface for the graphics that are drawn with the JavaScript calls to the canvas API. You can also see that the `onload` event handler is defined on the `<body>` tag and that it calls the JavaScript function `draw()`. This means that when the document is fully loaded, the browser automatically triggers the `onload` event handler and the `draw()` function is called.



Available for
download on
[Wrox.com](#)

[LISTING 1-1: An example of an HTML5 canvas tag](#)

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <script type="text/javascript">

    function draw() {

      var canvas = document.getElementById("canvas");

      if (canvas.getContext) {

        var context2D = canvas.getContext("2d");

        // Draw a red rectangle
        context2D.fillStyle = "rgb(255,0,0)";
        context2D.fillRect (20, 20, 80, 80);

        // Draw a green rectangle that is 50% transparent
        context2D.fillStyle = "rgba(0, 255, 0, 0.5)";
        context2D.fillRect (70, 70, 80, 100);

      }

    }

  </script>
</head>
<body onload="draw()">
  <canvas id="canvas" width="100" height="100"></canvas>
</body>
</html>
```

```

        // Draw a circle with black color and linewidth set
        to 5
        context2D.strokeStyle = "black";
        context2D.lineWidth = 5;

        context2D.beginPath();
            context2D.arc(100, 100, 90, (Math.PI/180)*0,
(Math.PI/180)*360, false);

        context2D.stroke();
        context2D.closePath();

        // Draw some text
        context2D.font = "20px serif";
        context2D.fillStyle = "#ff0000";
        context2D.fillText("Hello world", 40,220);

    }

}

</script>

</head>

<body onload="draw();">

<canvas id="canvas" width="300" height="300">

    Your browser does not support the HTML5 canvas
element.

</canvas>

</body>

</html>

```

Going back to the top of the code listing, you have the JavaScript within the `<head>` start tag and the `</head>` end tag. The first thing that happens within the `draw()` function is that the function `document.getElementById()` is used to retrieve a reference to the canvas element that is part of the body. If this succeeds, the reference is then used to obtain a 2D rendering context with the code:

```
var context2D = canvas.getContext("2d");
```

This code line was discussed earlier in this section. As you will see when you look at your first WebGL example in the next chapter, the part of the code that is described so far is very similar to the corresponding initialization code in a WebGL example. [Listing 1-1](#) shows an example of how the HTML5 canvas tag is used.

After you have a `CanvasRenderingContext2D`, you can use it to draw with the API that it exports. The first thing drawn is a red rectangle. The fill color is set to red by setting the property `fillStyle`. There are several ways to specify the actual color that is assigned to `fillStyle`, including the following:

- Use the `rgb()` method, which takes a 24-bit RGB value
`(context2D.fillStyle = rgb(255, 0, 0);).`
- Use the `rgba()` method, which takes a 32-bit color value where the last 8 bits represent the alpha channel of the fill color
`(context2D.fillStyle = rgba(255, 0, 0, 1);).`
- Use a string, which represents the fill color as a hex number
`(context2D.fillStyle = "#ff0000").`

For the first rectangle that is drawn (the red one), the method `rgb()` described under the first bullet above is used to set the fill color. Then the `fillRect()` method is called to draw the actual rectangle and fill it with the current fill color. The method takes the coordinate for the top-left corner of the rectangle and the width and the height of the rectangle:

```
fillRect(x, y, width, height)
```

Here are the lines of code to set the fill color and draw the first rectangle again:

```
// Draw a red rectangle
context2D.fillStyle = "rgb(255, 0, 0)";
context2D.fillRect (20, 20, 80, 80);
```

After the first rectangle is drawn, a second green rectangle is drawn with this code:

```
// Draw a green rectangle that is 50% transparent
context2D.fillStyle = "rgba(0, 255, 0, 0.5)";
context2D.fillRect (70, 70, 80, 100);
```

This second rectangle uses the method `rgba()` to set the `fillStyle` to be green and 50-percent transparent. Since the two rectangles overlap a bit, you

can see the red rectangle through the green rectangle.

The third shape drawn is a circle. To draw a circle, you first need to specify the beginning of a path. A path is one or more drawing commands that are specified together. After all the drawing commands are specified, you can select whether you want to stroke the path or fill it. In this case, when you want to draw a single circle, it might seem complicated to have to specify a path, but if you had a more complicated shape, you would appreciate the way this API is designed.

You specify the beginning of a path with the method `beginPath()`. Then you specify the drawing commands that should build up your path, and finally you specify that your path is finished with `closePath()`.

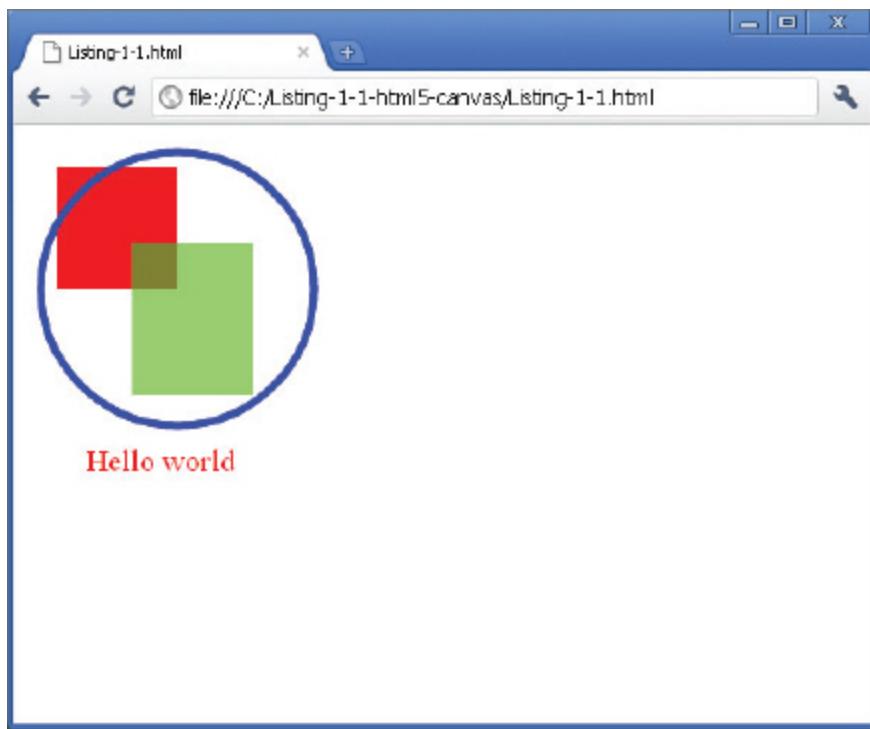
In this case, the only command you want to be part of your path is a command to draw a circle. However, the canvas API does not contain an explicit method to draw a circle. Instead, there are methods that you can use to draw arcs. In this example, the following method was used:

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

The `x` and `y` arguments specify the center of the circle. The `radius` is the radius of the circle that the arc is drawn on, and the `startAngle` and `endAngle` specify where the arc should start and end in radians. The last argument `anticlockwise` is a Boolean that specifies the direction of the arc.

If you load the source code above into your browser, you should see something similar to what's shown in [Figure 1-14](#).

FIGURE 1-14: A simple drawing with HTML5 canvas



SOME KEY POINTS TO REMEMBER ABOUT THE HTML5 CANVAS

Here are some things you should keep in mind about the HTML5 canvas:

- The HTML5 canvas specifies an immediate-mode API to draw 2D graphics on a web page.
- WebGL also draws on the HTML5 canvas, but uses the `WebGLRenderingContext` instead of the original `CanvasRenderingContext2D`.

Scalable Vector Graphics

Scalable Vector Graphics (SVG) is a language used to describe 2D graphics with XML. As the name indicates, it is based on vector graphics, which means it uses geometrical primitives such as points, lines, and curves that are stored as mathematical expressions. The program that displays the vector graphics (for example, a web browser) uses the mathematical expressions to build up the screen image. In this way the image stays sharp, even if the user zooms in on the picture. Of the graphics standards that are described in this chapter, SVG has the least to do with WebGL. But you should know about it

since it is a common graphics standard on the web and is also an example of a retained-mode API.

A Brief History of SVG

In 1998, Microsoft, Macromedia, and some other companies submitted Vector Markup Language (VML) as a proposed standard to W3C. At the same time, Adobe and Sun created another proposal called Precision Graphics Markup Language (PGML). W3C looked at both proposals, took a little bit of VML and a little bit of PGML, and created SVG 1.0, which became a W3C Recommendation in 2001. After this, an SVG 1.1 Recommendation was released, and there is also an SVG 1.2 Recommendation that is still in a Working Draft.

In addition to these “full” SVG specifications, there is also an SVG Mobile Recommendation that includes two simplified profiles for mobile phones. These are called SVG Tiny and SVG Basic. SVG Tiny targets very simple mobile devices, while SVG Basic targets higher-level mobile devices.

An SVG Code Snippet

[Listing 1-2](#) shows a very simple example of SVG code. The code draws a red rectangle, a blue circle, and a green triangle on the screen.



Available for
download on
[Wrox.com](#)

LISTING 1-2: A simple SVG example

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">

<rect x="50" y="30" width="300" height="100"
fill="red" stroke-width="2" stroke="black"/>

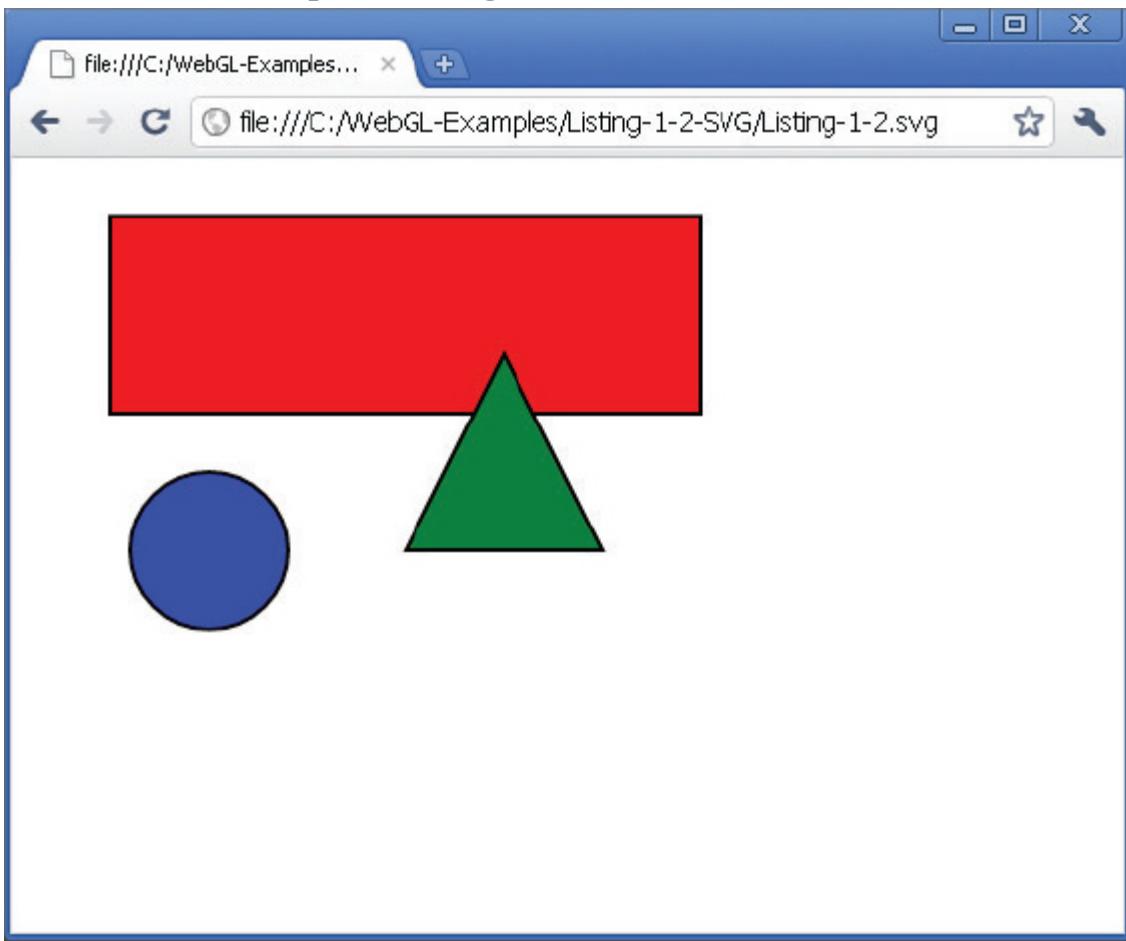
<circle cx="100" cy="200" r="40" stroke="black"
stroke-width="2" fill="blue"/>

<polygon points="200,200 300,200 250,100"
```

```
fill="green" stroke-width="2" stroke="black" />  
</svg>
```

I will not go through all the details of this code, but as you can see, SVG is quite compact. If you view this code in a web browser with SVG support, you will see something like [Figure 1-15](#).

FIGURE 1-15: A simple drawing with SVG



SOME KEY POINTS TO REMEMBER ABOUT SVG

Here are some things you should keep in mind about SVG:

- SVG is an XML-based technology for describing 2D vector graphics.
- Since it is a vector graphics format, SVG can be scaled without degrading the image quality.
- Since SVG is text based, it is easy to copy and paste part of an image. It can also be easily indexed by web crawlers.
- SVG is a retained-mode API that is very different from WebGL.

VRML and X3D

Up to now, this chapter has given you an overview of some of the graphics technologies that are most relevant in the context of WebGL. Two other technologies are less interesting but still deserve to be mentioned. Virtual Reality Markup Language (VRML) and its successor X3D are both XML-based technologies to describe 3D graphics. Neither VRML nor X3D is natively implemented in any of the major browsers. If you want to know more about VRML or X3D, two good places to start are www.web3d.org and www.x3dom.org.

LINEAR ALGEBRA FOR 3D GRAPHICS

Linear algebra is a part of mathematics that deals with vectors and matrices. To understand 3D graphics and WebGL, it is good to have at least a basic understanding of linear algebra. In the following sections, you get an overview of a selected part of linear algebra that is useful to understand 3D graphics and WebGL.

If you feel that you already have enough experience in this topic, please feel free to skip this part. If you are the kind of person who feels that mathematics is difficult or boring, I still recommend that you try to understand the information that is presented here. The text is not that abstract nor as general as these texts often are. Instead, it focuses on the concepts that are important for 3D graphics and WebGL.

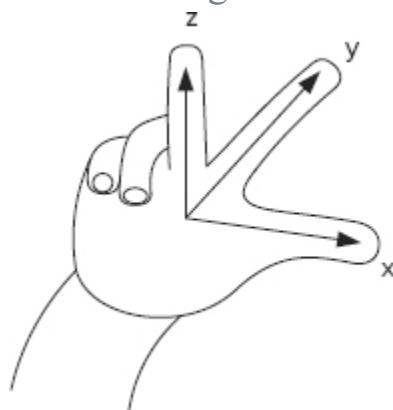
Coordinate System

To be able to specify where you want to draw your shapes with WebGL, you need a coordinate system. A coordinate system is sometimes called a space. There are many different kinds of coordinate systems, but the coordinate system that is used in WebGL is called a three-dimensional, orthonormal, right-handed coordinate system. This sounds a bit more complicated than it is.

Three-dimensional means that it has three axes, which are usually called x , y , and z . Orthonormal means that all the axes are orthogonal (perpendicular) to the other two and that the axes are normalized to unit length. Right-handed refers to how the third axis (the z -axis) is oriented. If the x - and y -axes are positioned so they are orthogonal and meet in the origin, there are two options for how to orient the z -axis so it is orthogonal against both x and y . Depending on which option you choose, the coordinate system is either called right-handed or left-handed.

One way to remember the directions for the axis of a right-handed coordinate system is shown in [Figure 1-16](#). You use your right hand and assign the x -, y -, and z -axes to your thumb, index finger, and middle finger in that order. The thumb indicates the direction of the x -axis, the index finger the direction of the y -axis, and the middle finger the direction of the z -axis.

FIGURE 1-16: A way to visualize a right-handed coordinate system



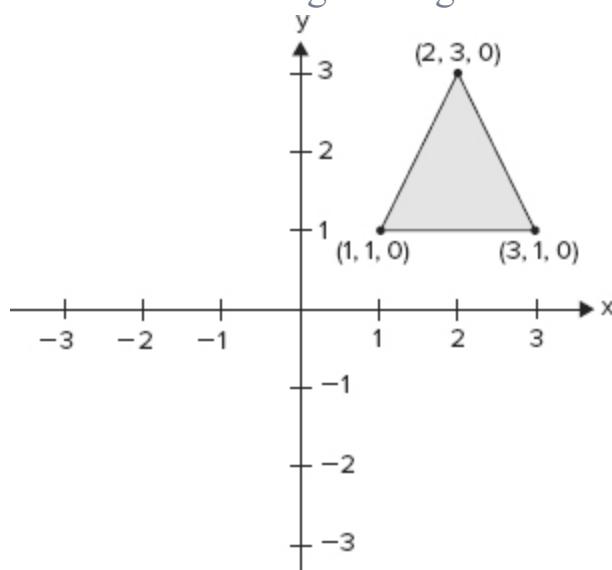
Points or Vertices

A point in a 3D coordinate system is a location that can be defined with an ordered triplet (v_x, v_y, v_z) . The location of this point is found by starting from the origin, where $(v_x, v_y, v_z) = (0, 0, 0)$ and then moving the distance v_x along the x -axis, then the distance v_y along the y -axis, and finally the distance v_z along the z -axis.

In a mathematical context, the point is the most basic building block that is used to build up other geometric shapes. Two points define a line between them and three points define a triangle, where the three points are the corners of the triangle.

When points are used to define other geometric shapes in 3D graphics, they are usually referred to as vertices (or vertex in singular). [Figure 1-17](#) shows an example of three vertices that define a triangle. The three vertices have the coordinates $(1, 1, 0)$, $(3, 1, 0)$, and $(2, 3, 0)$ and are drawn in a coordinate system where the z -axis is not shown but points perpendicular out from the paper. In this figure, the three vertices are marked with a small, filled circle. This is just to show you the location of the vertices in the figure; these circles would not exist if you actually drew a triangle with WebGL.

FIGURE 1-17: Three vertices defining a triangle

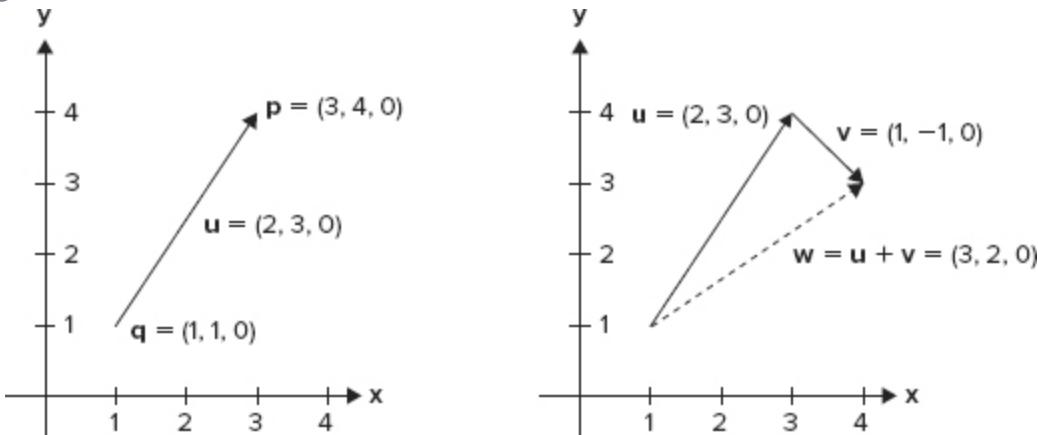


Vectors

Many physical quantities (for example, temperature, mass, or energy) are unrelated to any direction in space and are defined entirely by a numerical magnitude. These quantities are called *scalars*. Other physical quantities (such as velocity, acceleration, or force) need both a numerical magnitude and a direction in space to be completely specified. These quantities are called *vectors*.

A vector \mathbf{v} is a difference between two points. It has no position, but has both a direction and a length. As shown to the left in [Figure 1-18](#), a vector can be graphically illustrated as an arrow that connects the start point with the end point.

FIGURE 1-18: A Graphical visualization of a single vector to the left. To the right, two vectors are added with a new vector as the result



In 3D, a vector can be represented by three coordinates (v_x, v_y, v_z) . This vector has the same direction and length as the vector that starts in the origin and ends at the point specified with coordinates (v_x, v_y, v_z) . To the left in [Figure 1-18](#), you see a vector \mathbf{u} that starts at the point $\mathbf{p} = (1, 1, 0)$ and ends at the point $\mathbf{q} = (3, 4, 0)$. You get this vector by subtracting the start point \mathbf{p} from the end point \mathbf{q} like this:

$$\mathbf{v} = \mathbf{q} - \mathbf{p} = (3, 4, 0) - (1, 1, 0) = (2, 3, 0)$$

You can add vectors together. If two vectors are added, the sum is a new vector, where the different components of the two vectors have been added. This means that:

$$\mathbf{v} + \mathbf{u} = (v_x + u_x, v_y + u_y, v_z + u_z)$$

The addition of two vectors \mathbf{u} and \mathbf{v} can be graphically visualized as shown to the right in [Figure 1-18](#). Draw the “tail” of vector \mathbf{v} joined to the “nose” of vector \mathbf{u} . The vector $\mathbf{w} = \mathbf{u} + \mathbf{v}$ is from the “tail” of \mathbf{u} to the “nose” of \mathbf{v} . In this example, you see how two vectors are added in 3D (but with $z = 0$). Adding $\mathbf{u} = (2, 3, 0)$ with $\mathbf{v} = (1, -1, 0)$ results in the third vector $\mathbf{w} = (3, 2, 0)$.

Vectors can also be multiplied with a scalar (i.e., a number). The result is a new vector where all the components of the vector have been multiplied with the scalar. This means that for the vector \mathbf{u} and the scalar k , you have:

$$k\mathbf{u} = (ku_x, ku_y, ku_z)$$

If you multiply a vector with the scalar $k = -1$, you get a vector with the same length that points in the exact opposite direction from the original

vector you had.

For WebGL, vectors are important — for example, to specify directions of light sources and viewing directions. Another example of when vectors are used is as a *normal vector* to a surface. A normal vector is perpendicular to the surface, and specifying a normal vector is a convenient way to specify how the surface is oriented.

As previously mentioned, vectors are useful in fundamental physical science — for example, to represent velocity. This can be useful if you plan to write a game based on WebGL. Assume that you are writing a 3D game and the velocity of your space ship is 20 meters/second, direction east, parallel with the ground. This could be represented by a vector:

$$\mathbf{v} = (20, 0, 0)$$

The speed would be the length of the vector, and the direction of the space ship would be represented by the direction of the vector.

Dot Product or Scalar Product

If you have two vectors in 3D, you can multiply them in two ways:

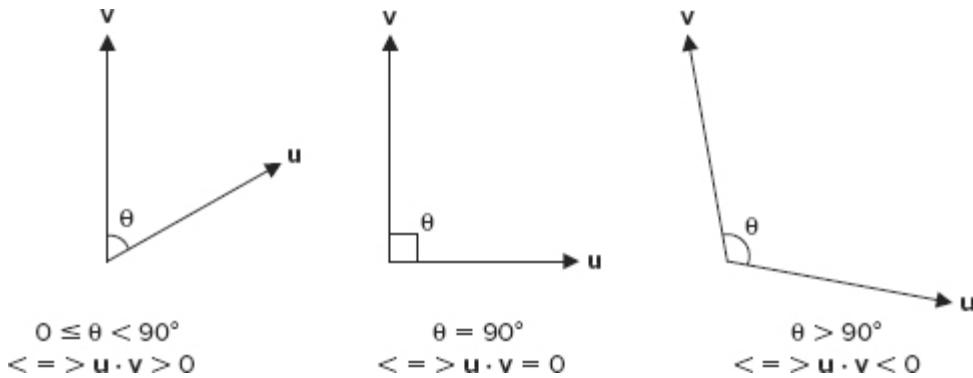
- Dot product or scalar product
- Cross product

In this section you will learn about *dot product*, or *scalar product*. As the latter name indicates, the result of this multiplication is a scalar and not a vector. For two vectors \mathbf{u} and \mathbf{v} , the dot product can be defined in the following way:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta$$

This definition is based on the length of the two vectors and the smallest angle θ between them, as illustrated in [Figure 1-19](#). You can see that since $\cos 90^\circ$ is zero, the dot product of two vectors where the angle between them is 90° equals zero. The reverse is also true — i.e., if you have two vectors and the dot product between them is zero, then you know that the two vectors are orthogonal. Chapter 7 explains how the previous definition of the dot product can be used in WebGL to calculate how light is reflected from a surface.

[**FIGURE 1-19:**](#) The geometry used for the dot product or scalar product



The left side of [Figure 1-19](#) illustrates the case when the smallest angle between the two vectors is less than 90° , which gives a positive result for the dot product. In the middle, the angle is exactly 90° , which gives a dot product of zero. To the right, the angle is greater than 90° , which gives a negative dot product.

The dot product also has a second definition that is algebraic:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z$$

This definition is equivalent but useful in different situations. Later in this chapter, you will see how this definition of the dot product is useful when defining matrix multiplications.

Cross Product

The other way to multiply two vectors is called *cross product*. The cross product of two vectors \mathbf{u} and \mathbf{v} is denoted by:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v}$$

The cross product has this algebraic definition:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$$

The result of a cross product is a new vector \mathbf{w} with the following properties:

- $|\mathbf{w}| = |\mathbf{u}| |\mathbf{v}| \sin \theta$ (where θ is the smallest angle between \mathbf{u} and \mathbf{v}).
- \mathbf{w} is orthogonal against both \mathbf{u} and \mathbf{v} .
- \mathbf{w} is right handed with respect to both \mathbf{u} and \mathbf{v} .

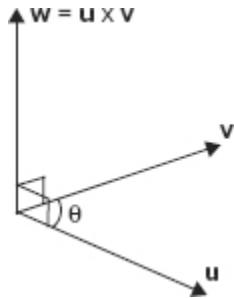
The first bullet above specifies the length of the new vector \mathbf{w} as the length of \mathbf{u} multiplied by the length of \mathbf{v} multiplied by $\sin \theta$, where θ is the smallest angle between \mathbf{u} and \mathbf{v} . Note that it also follows from the first bullet that $\mathbf{u} \times \mathbf{v} = 0$ if and only if \mathbf{u} and \mathbf{v} are parallel, since $\sin 0^\circ = 0$.

The second and third bullets specify the direction of the new vector \mathbf{w} . It follows from the third bullet that the order of \mathbf{u} and \mathbf{v} matters. This means that the cross product is not commutative. Instead, the following relationship is true for a cross product:

$$\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$$

[Figure 1-20](#) shows an illustration of the geometry involved in the cross product.

FIGURE 1-20: The geometry used in the cross product



An important usage of the cross product in 3D graphics is to calculate the normal for a surface such as a triangle. In Chapter 7, you learn how the normal is important when doing lighting calculations in WebGL.

Homogeneous Coordinates

As described earlier in this chapter, you can specify a point or a vector in 3D with three coordinates. However, this can be a bit confusing since both points and vectors are specified in the same way. With *homogeneous coordinates*, a fourth coordinate (called w) is added. For vectors, $w = 0$, and if $w \neq 0$, the homogeneous coordinate specifies a point.

You can easily convert from a homogeneous point (p_x, p_y, p_z, p_w) to a point represented by three coordinates, by dividing all coordinates by p_w . Then the 3D point is defined by the first three coordinates as (p_x, p_y, p_z) . If you have a point represented with three coordinates, you simply add a 1 at the fourth position to get the corresponding point $(p_x, p_y, p_z, 1)$ in homogeneous coordinates.

Aside from differentiating between points and vectors, there is another important reason for the introduction of homogeneous coordinates: If you represent a point with four homogeneous coordinates, it is possible to

represent transformations (such as translations, rotations, scaling, and shearing) with a 4×4 matrix. You will learn about matrices and transformations next.

Matrices

A matrix is composed of several rows and columns of numbers. The numbers in the matrix are called the elements of the matrix. A matrix of m rows and n columns is said to be of dimension $m \times n$.

The most commonly used matrices in WebGL have four rows and four columns — i.e., they are 4×4 matrices like the one shown here:

$$M = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

A matrix with one column (i.e., it has the size $m \times 1$) is called a column vector. A column vector with four elements looks like this:

$$v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

A matrix with one row (i.e., it has the size $1 \times n$) is called a row vector. A row vector with four elements looks like this:

$$v = [v_0 \ v_1 \ v_2 \ v_3]$$

As you will soon learn, column vectors are common in WebGL. They are often used to represent a vertex that is multiplied with a 4×4 matrix to do some kind of transformation on the vertex.

Addition and Subtraction of Matrices

It is possible to add or subtract two matrices if they have the same dimensions. You add two matrices by doing component-wise addition of the elements, very similar to when you add two vectors. It is easiest to understand with an example. If you have two matrices **A** and **B** according to this,

$$A = \begin{bmatrix} 1 & 4 & 2 \\ -1 & 0 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 & 5 \\ 4 & 1 & 2 \end{bmatrix}$$

then when you add the two matrices, you get a matrix **C** shown here:

$$C = A + B = \begin{bmatrix} 2 & 6 & 7 \\ 3 & 1 & 5 \end{bmatrix}$$

Subtraction of matrices is defined in a similar way to addition. With the same matrices **A** and **B** above, you get the matrix **D** as shown here:

$$D = A - B = \begin{bmatrix} 0 & 2 & -3 \\ -5 & -1 & 1 \end{bmatrix}$$

Matrix Multiplication

Matrix multiplication is a very important operation in 3D graphics. Even if you do not have to do this multiplication manually like I do in this section, it is good to know how it actually works. The definition of matrix multiplication is such that two matrices **A** and **B** can only be multiplied together when the number of columns of **A** is equal to the number of rows of **B**. If matrix **A** has m rows and p columns (i.e., it has the format $m \times p$) and matrix **B** has p rows and n columns (i.e., it has the format $p \times n$), then the result of multiplying **A** and **B** creates a matrix with m rows and n columns (i.e., it has the format $m \times n$). You can write this in a symbolic way:

$$[m \times p][p \times n] = [m \times n]$$

So if you multiply a matrix **A** of format $m \times p$ with matrix **B** of format $p \times n$, you get a new matrix **AB** of format $m \times n$. The new matrix **AB** has an element at position ij that is the scalar product of row i of matrix **A** with column j of matrix **B**, that is:

$$a_{i0}b_{0j} + a_{i1}b_{1j} + \dots + a_{ip-1}b_{p-1j} = \sum_{k=0}^{p-1} a_{ik}b_{kj}$$

Looking at an example makes it easier to understand how it works. Assume that you have two matrices **M** and **N**, as shown here:

$$M = \begin{bmatrix} 2 & -1 \\ -2 & 1 \\ -1 & 2 \end{bmatrix} \quad N = \begin{bmatrix} 4 & -3 \\ 3 & 5 \end{bmatrix}$$

Then you get the following result if you multiply **M** and **N**.

$$MN = \begin{bmatrix} 2 \times 4 + (-1) \times 3 & 2 \times (-3) + (-1) \times 5 \\ (-2) \times 4 + 1 \times 3 & (-2) \times (-3) + 1 \times 5 \\ (-1) \times 4 + 2 \times 3 & (-1) \times (-3) + 2 \times 5 \end{bmatrix} = \begin{bmatrix} 5 & -11 \\ -5 & 11 \\ 2 & 13 \end{bmatrix}$$

One thing that is important to mention about matrix multiplication is that the order of the matrices does matter. Just because the product **MN** above is

defined, it does not mean that \mathbf{NM} is defined, and even if both products are defined, the result is generally different. So in general:

$$\mathbf{MN} \neq \mathbf{NM}$$

Another way to express this is that matrix multiplication is not commutative. The most common matrix multiplications in WebGL are to multiply two 4×4 matrices or to multiply a 4×4 matrix with a 4×1 matrix.

Identity Matrix and Inverse Matrix

For scalars, the number 1 has the property that when any other number x is multiplied by 1, the number remains the same. If x is a number, then $1 \times x = x$ is true for all numbers x . The matrix counterpart of the scalar number 1 is the *identity matrix* or *unit matrix*. An identity matrix is always square — i.e., it has the same number of columns as it has rows — and it contains ones in the diagonal and zeroes everywhere else. Here is the 4×4 identity matrix:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If a matrix \mathbf{M} is multiplied by the identity matrix \mathbf{I} , the result is always the matrix \mathbf{M} . This means the following is true:

$$\mathbf{MI} = \mathbf{IM} = \mathbf{M}$$

Now that you know that the identity matrix corresponds to the scalar 1, you want to find a matrix that corresponds. For all numbers except zero, there is an inverse that gives the product 1 if the number is multiplied with this inverse. For example, for any number x , there is another number $1/x$ (which can also be written as x^{-1}) that gives the product 1 if the numbers are multiplied. In a similar way, an inverse of a matrix \mathbf{M} is denoted \mathbf{M}^{-1} and has the property that if it is multiplied by the matrix \mathbf{M} , the result is the identity matrix. This means the following is true:

$$\mathbf{MM}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$$

Note that only square matrices (number of columns equals the number of rows) can have an inverse, however, not all square matrices have an inverse.

Transposed Matrix

The definition of the transpose of a matrix \mathbf{M} is that it is another matrix where the columns become rows and the rows become columns. The notation of the transpose of a matrix \mathbf{M} is \mathbf{M}^T . The transpose of a matrix is defined for any format $m \times n$, but since you will often be using 4×4 matrices in WebGL, you will look at such a matrix as an example. Assume that the matrix \mathbf{M} is defined as shown here:

$$\mathbf{M} = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Then the transposed matrix \mathbf{M}^T is given by the following:

$$\mathbf{M}^T = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$



As you will learn in Chapter 7, transposing and inverting a matrix is useful for doing transformations of normals.

Affine Transformations

On its way to the screen, a 3D model is transformed into several different coordinate systems or spaces. A *transform* is an operation that takes an entity such as a point or a vector and converts it in some way. A special type of transform, called a *linear transform*, preserves vector addition and scalar multiplication. So if the transform is represented by \mathbf{f} and you have two vectors \mathbf{u} and \mathbf{v} , then the transform is only linear if it fulfills the two following conditions:

$$\mathbf{f}(\mathbf{u}) + \mathbf{f}(\mathbf{v}) = \mathbf{f}(\mathbf{u} + \mathbf{v})$$

$$k \mathbf{f}(\mathbf{u}) = \mathbf{f}(k\mathbf{u})$$

The first condition can also be explained in the following way. If you apply the transform on the two vectors and then add the transformed vectors, then you should get the same result as if you first added the vectors and then applied the transform on the sum.

The second condition can be explained like this. If you transform the vector and then multiply the result with a scalar k , then this should give you the same result as if you first multiplied the vector with k and then performed the transformation on the result.

An example of a linear transformation is:

$$f(\mathbf{u}) = 3\mathbf{u}$$

You will now double-check that the two conditions above are true for this transformation. To make it concrete, assume that you have two vectors (\mathbf{p} and \mathbf{q}) as shown here:

$$\mathbf{p} = [0 \ 1 \ 2 \ 3] \quad \mathbf{q} = [4 \ 5 \ 6 \ 7]$$

You start with the first condition, which is that you should get the same result if you first multiply all the elements in the two vectors by 3 and then add the result as if you first added the two vectors and then multiplied the result by 3:

$$\begin{aligned} f(\mathbf{p}) + f(\mathbf{q}) &= 3 \times [0 \ 1 \ 2 \ 3] + 3 \times [4 \ 5 \ 6 \ 7] \\ &= [0 \ 3 \ 6 \ 9] + [12 \ 15 \ 18 \ 21] = [12 \ 18 \ 24 \ 30] \end{aligned}$$

$$\begin{aligned} f(\mathbf{p} + \mathbf{q}) &= 3 \times ([0 \ 1 \ 2 \ 3] + [4 \ 5 \ 6 \ 7]) \\ &= 3 \times [4 \ 6 \ 8 \ 10] = [12 \ 18 \ 24 \ 30] \end{aligned}$$

You get the same result in both cases, so the first condition is clearly met. Now consider the second condition, which was that $k f(\mathbf{u}) = f(k\mathbf{u})$. To make it concrete, use the $k = 2$ and $\mathbf{u} = \mathbf{p} = [0 \ 1 \ 2 \ 3]$, as shown here:

$$k f(\mathbf{u}) = 2 \times (3 \times [0 \ 1 \ 2 \ 3]) = 6 \times [0 \ 1 \ 2 \ 3] = [0 \ 6 \ 12 \ 18]$$

$$f(k\mathbf{u}) = 3 \times (2 \times [0 \ 1 \ 2 \ 3]) = 6 \times [0 \ 1 \ 2 \ 3] = [0 \ 6 \ 12 \ 18]$$

Also, the second condition is met and the transform $f(\mathbf{u}) = 3\mathbf{u}$ is clearly a linear transform.

Examples of linear transformations are scaling, rotation, and shearing. A linear transform of points or vectors in 3D can be represented with a 3×3 matrix. However, in addition to the linear transforms, there is a very basic but important transform that is called *translation*. A translation cannot be represented with a 3×3 matrix.

An *affine transform* is a transform that performs a linear transformation and then a translation. It is possible to represent an affine transform with a 4

$\times 4$ matrix. If you use homogeneous coordinates for the points or vectors, then you can achieve the transformation by multiplying the 4×4 transformation matrix and the column vector containing the point or vector:

$$Mv = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} m_{00}v_0 + m_{01}v_1 + m_{02}v_2 + m_{03}v_3 \\ m_{10}v_0 + m_{11}v_1 + m_{12}v_2 + m_{13}v_3 \\ m_{20}v_0 + m_{21}v_1 + m_{22}v_2 + m_{23}v_3 \\ m_{30}v_0 + m_{31}v_1 + m_{32}v_2 + m_{33}v_3 \end{bmatrix}$$

The four types of affine transforms are described in the following sections.

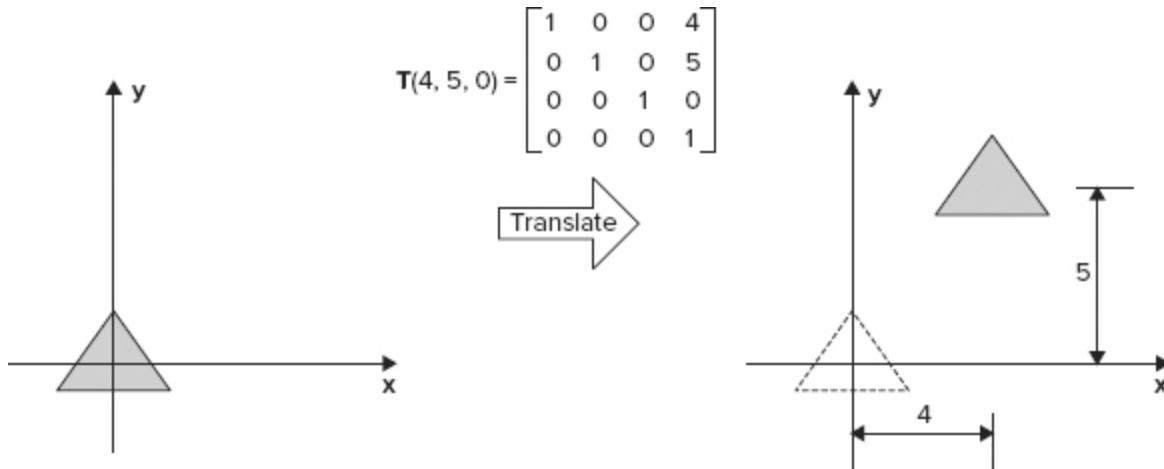
Translation

Translation means that you move every point by a constant offset. The translation matrix is as follows:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The translation matrix shown above translates a point with an offset that is represented by the vector (t_x, t_y, t_z) . [Figure 1-21](#) shows an example of how a triangle is translated by the vector $(t_x, t_y, t_z) = (4, 5, 0)$.

FIGURE 1-21: The translation of a triangle by 4 in the x -direction and 5 in the y -direction



To the left, the triangle is shown before the translation is applied. To the right, the triangle is translated by 4 in the x -direction and 5 in the y -direction. The z -axis is not shown but has a direction perpendicular out from the paper

Now when you know how to perform a matrix multiplication, you can easily see how the translation matrix introduced above affects a single point \mathbf{p} written on homogeneous notation:

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

$$T\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{bmatrix}$$

So it is clear that the result of the multiplication is a new point that is translated by the vector (t_x, t_y, t_z) .

Note that since a vector does not have a position, but just a direction and a size, it should not be affected by the translation matrix. Remember that a vector on homogeneous notation has the fourth component set to zero. You can now verify that the vector \mathbf{v} is unaffected when it is multiplied by the translation matrix:

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

$$T\mathbf{v} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

So for a vector, the multiplication with the translation matrix results in the same vector without any modifications, just as you would expect.

Rotation

A rotation matrix rotates a point or a vector by a given angle around a given axis that passes through the origin. Rotation matrices that are commonly used are \mathbf{R}_x , \mathbf{R}_y , and \mathbf{R}_z , which are used for rotations around the x -axis, y -axis, and z -axis, respectively. These three rotation matrices are shown here:

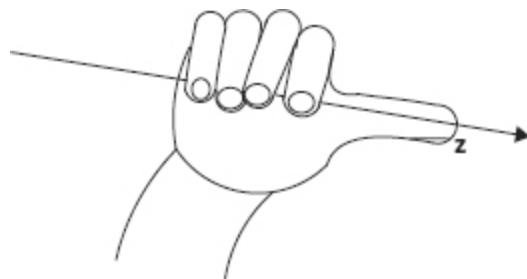
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

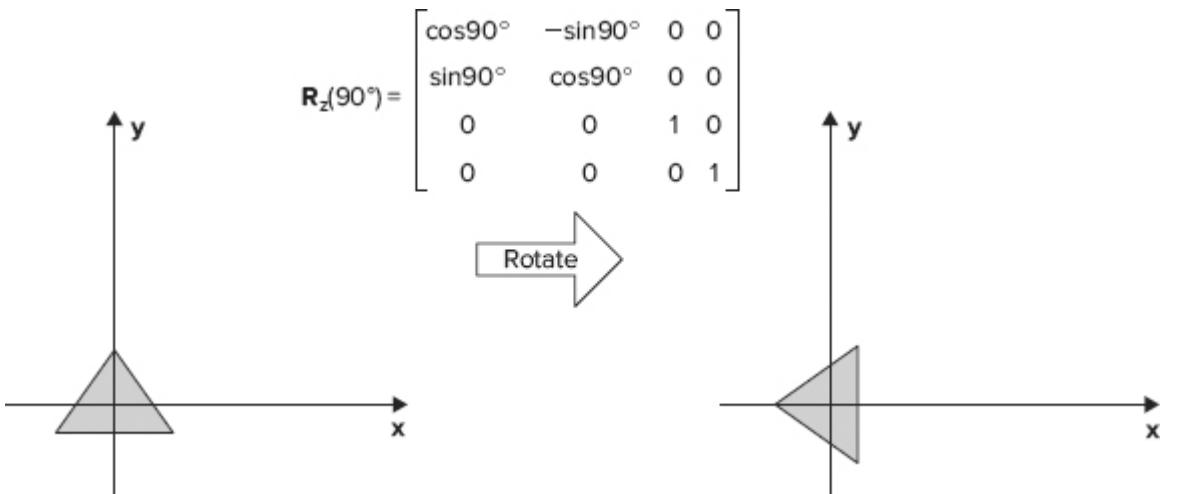
A positive rotation is given by the right-hand grip rule, as shown in [Figure 1-22](#). Imagine that you take your right hand and grip around the axis you want to rotate around so the thumb is pointing in the positive direction of the axis. Then your other fingers are pointing in the direction that corresponds to the positive rotation. This illustration shows a rotation around the z -axis, but the rule works in the same way for any axis.

FIGURE 1-22: The right-hand grip rule helps you to remember the positive direction for rotation around an axis



[Figure 1-23](#) illustrates a triangle that is rotated by 90° around the z -axis by applying the rotation matrix R_z . To the left, the triangle is shown before the rotation. To the right, the triangle is shown after the transformation matrix $R_z(90^\circ)$ has been applied and the triangle has been rotated 90° around the z -axis that is pointing perpendicular out from the paper.

FIGURE 1-23: The rotation of a triangle 90° around the z -axis



It is possible to write down the rotation matrices for rotation around an arbitrary axis, but I will not bore you with these details here. If you are interested, you can have a look in an advanced linear algebra book or look it up on the web.

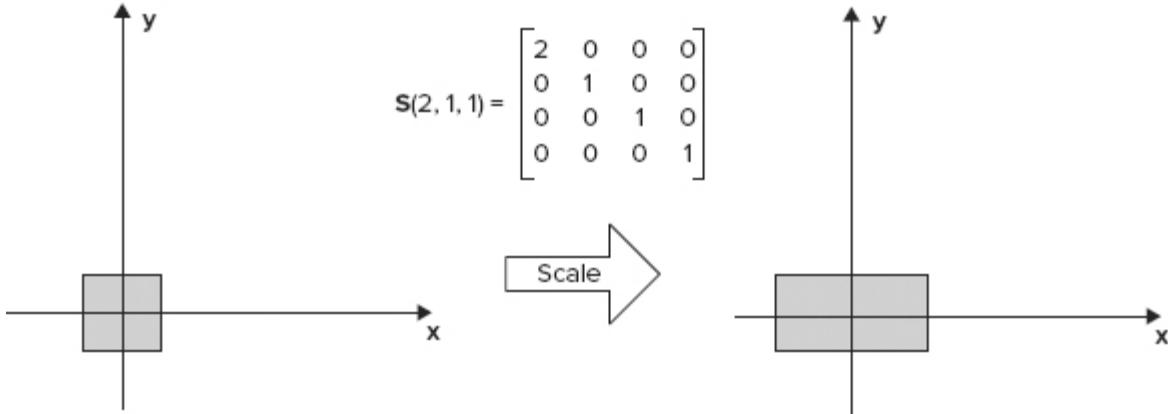
Scaling

Scaling is used to enlarge or diminish an object. The following scaling matrix scales objects with a factor s_x along the x -direction, s_y along the y -direction, and s_z along the z -direction:

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If $s_x = s_y = s_z$, then the transform is called *uniform scaling*. A uniform scaling changes the size of objects, but does not change the shape of objects. [Figure 1-24](#) shows an example of a square that is scaled by two in the x -direction, while the y -direction and the z -direction are left unchanged.

[**FIGURE 1-24:**](#) Scaling by two in the x -direction



To the left, you see a square before the scaling is applied. To the right, you see the square after the scaling matrix $S(2, 1, 1)$ is applied. The square has now been scaled into a rectangle.

Shearing

Shearing is an affine transform that is rarely used in WebGL and other 3D graphics. The main reason to include it in this book is for completeness. However, an example of when the shearing transform could actually be useful in WebGL is if you are writing a game and want to distort the scene.

You obtain a shearing matrix if you start from the identity matrix and then change one of the zeroes in the top-left 3×3 corner to a value that is not zero. There are six zeroes in the top-left 3×3 corner of the identity matrix that could be changed to a non-zero value. This means that there are also six possible basic shearing matrices in 3D. There are different ways to name the shearing matrices, but one common way is to name them something like $H_{xy}(s)$, $H_{xz}(s)$, $H_{yx}(s)$, $H_{yz}(s)$, $H_{zx}(s)$, and $H_{zy}(s)$. In this case, the first subscript indicates which coordinate is changed by the matrix and the second subscript indicates which coordinate performs the shearing. The shearing matrix $H_{xy}(s)$ is shown here:

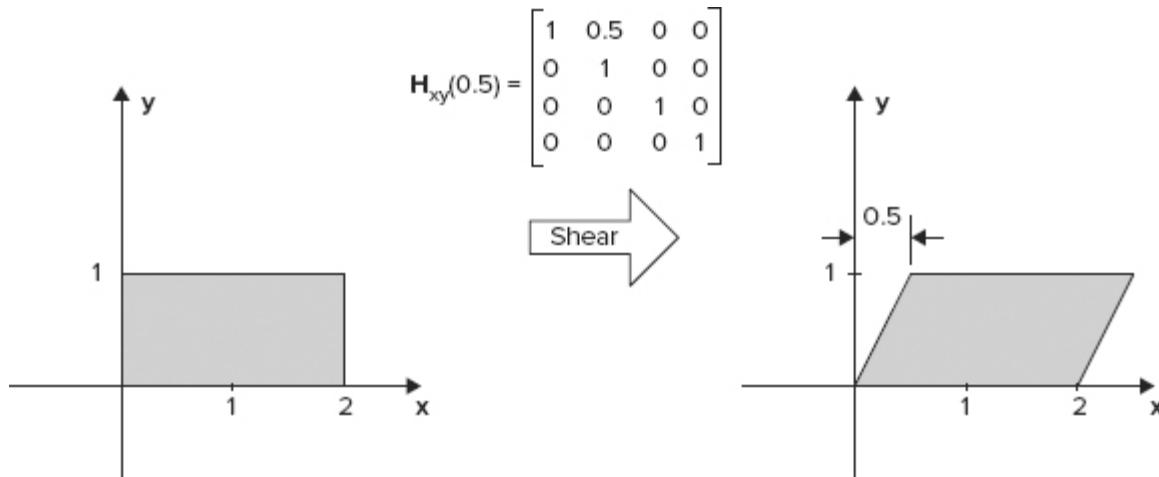
$$H_{xy}(s) = \begin{bmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This shearing matrix changes the x -coordinate when the y -coordinate is changed. If you multiply the shearing matrix $H_{xy}(s)$ with a point p , it is even clearer how the shearing matrix affects the point:

From this result, it is even more obvious what happens. The x -coordinate is sheared to the right when the y -coordinate is increased. An example of shearing is shown in [Figure 1-25](#).

$$H_{xy}(s) = \begin{bmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + sp_y \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

FIGURE 1-25: An example of shearing



SUMMARY

In this chapter you have had an initial look at WebGL, where you learned some background as well as the steps that make up the WebGL pipeline. You also learned a little bit about some other 2D and 3D graphics technologies and how WebGL is related to them.

You have also learned some fundamental linear algebra that is useful in order to understand WebGL or any 3D graphics on a deeper level. Even though you will be able to create many WebGL-based applications without understanding the details of 3D graphics, you will get a much deeper understanding if you really grasp the majority of the content in this section. It will also be very useful if you run into problems and need to debug your application.

Chapter 2

Creating Basic WebGL Examples

WHAT'S IN THIS CHAPTER?

- Create a basic but complete WebGL application
- Create a WebGL context
- Write a simple vertex shader and a fragment shader and use them in your application
- Load your shader source code through the WebGL API
- Compile and link your shaders
- Load your vertex data into the WebGL buffers and use the buffers to draw your scene
- Debug and troubleshoot your application
- Load your shaders with the DOM API

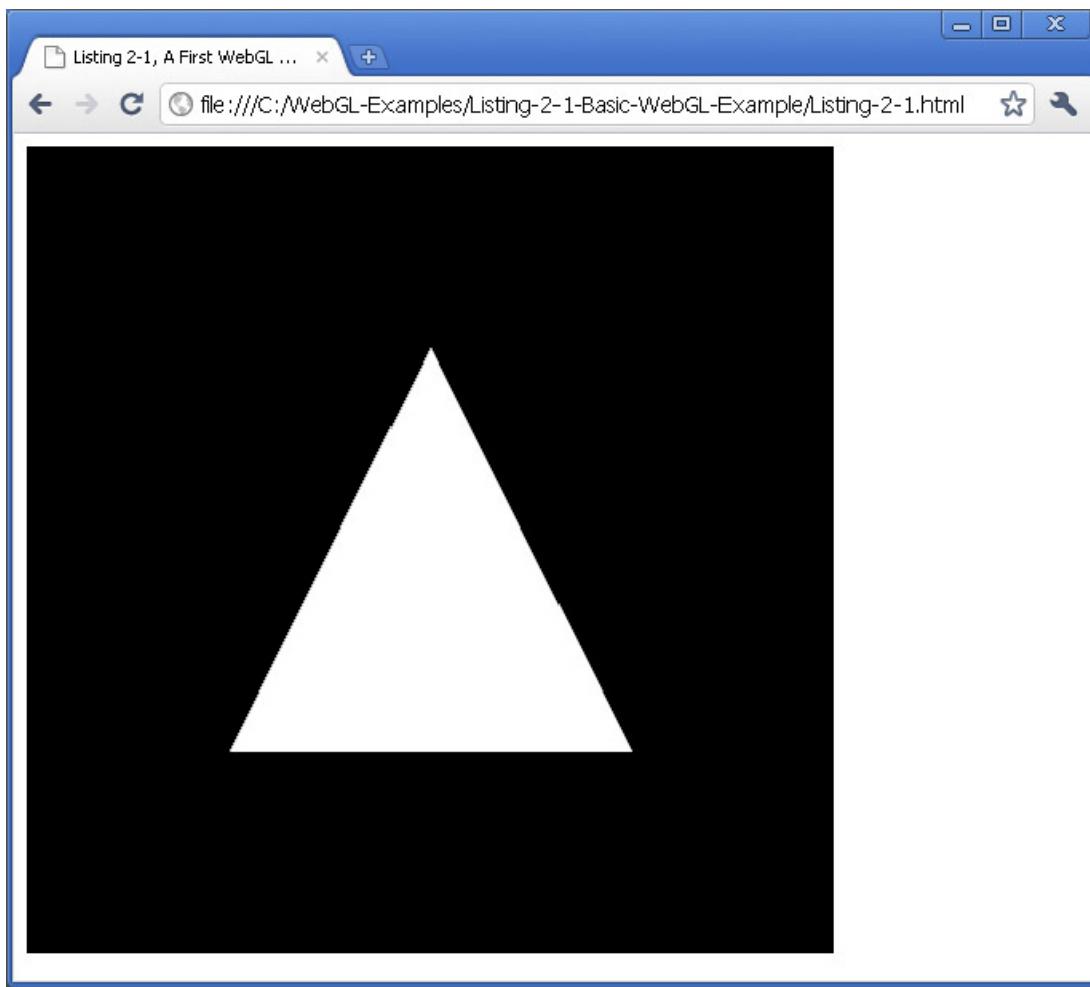
The first chapter was a very theoretic chapter; in this chapter, you will gain more practical knowledge. You will work through a very basic but complete WebGL example. You will start by creating the example as simply as possible. All parts of the example will be explained in detail so that you understand what is going on.

To make the rest of the journey towards learning WebGL a bit easier, you will learn how to debug and troubleshoot an application. You will pick up several valuable tips and tricks on how to find errors and correct them as quickly as possible. Finally, you will learn how to improve some parts of your initial example.

DRAWING A TRIANGLE

The first example will only be in 2D. Even though WebGL is an API that is designed mainly to draw 3D graphics, it is possible to draw in 2D as well. The first basic example involves drawing a white triangle on a black background, as shown in [Figure 2-1](#).

FIGURE 2-1: A triangle drawn with WebGL



In [Listing 2-1](#), you can see that there is a lot of source code just to draw a triangle. One reason for this is that WebGL is fully shader based. This means that you need both a vertex shader and a fragment shader to draw even the most basic objects on the screen.

When you write a WebGL application, both the vertex shader and the fragment shader are written in source code and this source code then needs to be compiled and linked in run time before it can be used as a shader program by the GPU. This is one reason that there are several steps needed to set up and draw even this basic example; the required steps for setting up a basic WebGL application are listed here:

1. Write some basic HTML code that includes a `<canvas>` tag. The `<canvas>` tag provides the drawing area for WebGL. Then you need to write some JavaScript code to create a reference to your canvas so you can create a `WebGLRenderingContext`.
2. Write the source code for your vertex shader and your fragment shader.
3. Write source code that uses the WebGL API to create a shader object for both the vertex shader and the fragment shader. You need to load the source code into the shader objects and compile the shader objects.

4. Create a program object and attach the compiled shader objects to this program object. After this, you can link the program object and then tell WebGL that you want to use this program object for rendering.
5. Set up the WebGL buffer objects and load the vertex data for your geometry (in this case, the triangle) into the buffer.
6. Tell WebGL which buffer you want to connect to which attribute in the shader, and then, finally, draw your geometry (the triangle).

These steps are implicitly included in [Listing 2-1](#), and you will be taken through the different parts of the source code in the following sections.



Available for
download on
Wrox.com

[LISTING 2-1:](#) A WebGL example that draws a white triangle on a black background

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 2-1, A First WebGL Example</title>
<meta charset="utf-8">
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function createGLContext(canvas) {
    var names = ["webgl", "experimental-webgl"];
    var context = null;
    for (var i=0; i < names.length; i++) {
        try {
            context = canvas.getContext(names[i]);
        } catch(e) {}
        if (context) {
            break;
        }
    }
    if (context) {
        context.viewportWidth = canvas.width;
        context.viewportHeight = canvas.height;
    } else {
        alert("Failed to create WebGL context!");
    }
    return context;
}

function loadShader(type, shaderSource) {
    var shader = gl.createShader(type);
    gl.shaderSource(shader, shaderSource);
```

```

gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert("Error compiling shader" + gl.getShaderInfoLog(shader));
    gl.deleteShader(shader);
    return null;
}
return shader;
}

function setupShaders() {
var vertexShaderSource =
"attribute vec3 aVertexPosition;          \n" +
"void main() {                            \n" +
"    gl_Position = vec4(aVertexPosition, 1.0); \n" +
"}"

var fragmentShaderSource =
"precision mediump float;                \n" +
"void main() {                            \n" +
"    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); \n" +
"}"

var vertexShader      = loadShader(gl.VERTEX_SHADER,
vertexShaderSource);
var fragmentShader    = loadShader(gl.FRAGMENT_SHADER,
fragmentShaderSource);

shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
}

gl.useProgram(shaderProgram);

shaderProgram.vertexPositionAttribute =
gl.getAttributeLocation(shaderProgram, "aVertexPosition");
}

function setupBuffers() {
vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
var triangleVertices = [
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0
];
}

```

```

        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
        gl.STATIC_DRAW);
        vertexBuffer.itemSize = 3;
        vertexBuffer.numberOfItems = 3;
    }

function draw() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                          vertexBuffer.itemSize, gl.FLOAT, false, 0,
    0);

    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}

function startup() {
    canvas = document.getElementById("myGLCanvas");
    gl = createGLContext(canvas);
    setupShaders();
    setupBuffers();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    draw();
}
</script>

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>

</html>

```

Creating the WebGL Context

To keep this example as simple as possible, the code is embedded within a single HTML file. Within this file, almost all of the code is embedded in the `<head>`.

If you start from the end of the source code in [Listing 2-1](#), you see that the `onload` event handler is defined on the `<body>` tag to call the JavaScript function `startup()` like this:

```

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>

</html>

```

The browser triggers the `onload` event when the user enters the web page, and the document and all external content is fully loaded. If you put the call to `startup()` in the `onload` event handler, you know that the document is fully loaded and has reached a stable state. The function `startup()` is the entry point into your WebGL application.

Within the `<body>`, the only thing that is available is an HTML5 `<canvas>` tag. By giving the `<canvas>` tag an id, you can easily access it from the JavaScript code. The `<canvas>` tag is the drawing surface that is used for WebGL, and this is where your WebGL graphics will end up within the web page.

Now if you look at the source code within the `<head>`, you see the JavaScript that is needed for this WebGL example. At the end of this JavaScript code, the function `startup()` is defined. When the `onload` event is triggered and `startup()` is called, the first thing this method does is to use the method `document.getElementById()` to get a reference to the `<canvas>` tag that is defined within the body of the page:

```
function startup() {  
    var canvas = document.getElementById("myGLCanvas");  
    var gl = createGLContext(canvas);
```

The method `document.getElementById()` is not specific to WebGL, but is part of what is called the Document Object Model (DOM) API. This is an API that defines how to access the objects within a document and is heavily used in JavaScript that runs inside the browser.

After a reference to the `canvas` is retrieved with `document.getElementById()`, the homemade JavaScript function `createGLContext()` is called with the `canvas` as argument. Inside this function, a `WebGLRenderingContext` object is created by calling the important method `canvas.getContext()` with a standard context name.

The context name "experimental-webgl" was supposed to be a temporary name for the `WebGLRenderingContext` for use during development of the WebGL specification, while the name "webgl" was supposed to be used once the specification was finalized. However, to increase the possibility that your browser will understand the context name, it is a good idea to try both context names.

The `WebGLRenderingContext` is the interface that basically presents the complete WebGL API. In all the source code examples in this book, the `WebGLRenderingContext` is put in a variable that is named `gl`. As a result, when you see a call to a method that is in the format `gl.someMethod()`, it means that the method `someMethod()` is part of the `WebGLRenderingContext` interface. Or simply put, the method is part of the WebGL API.

When a method (or any other member) of the WebGL API is described in this book, it is normally referred to with the prefix `gl`. For example, `gl.createShader()` refers to the method `createShader()` that is part of the `WebGLRenderingContext`.

INITIALIZING WEBGL

The examples in this book use a very simple way to test if it's possible to create a WebGL context. The two context names "webgl" and "experimental-webgl" are tried. If `canvas.getContext()` is not successful with any of these context names, a JavaScript alert is shown that informs the user that it is not possible to create a WebGL context.

A slightly more sophisticated way is to distinguish between failures that are a result of the browser not recognizing WebGL, and failures that can be attributed to other reasons. You can check if the browser does not recognize WebGL with the following code snippet:

```
if (!window.WebGLRenderingContext) {  
    // The browser does not know what WebGL is.  
  
    // Present a link to "http://get.webgl.org" where the user  
    // can get info about how to upgrade the browser  
  
}
```

If the browser recognizes WebGL but the call to `canvas.getContext()` fails, you could instead send users to <http://get.webgl.org/troubleshooting>. Sending users to these Khronos URLs helps them get the latest information about which browsers support WebGL and how to upgrade their browsers.

If you prefer to not write this code yourself, there is a small JavaScript library called `webgl-utils.js` that performs this initialization and checking for you, and then sends your users to the correct URL if a failure occurs during initialization. The library is available from the Khronos WebGL Wiki (www.khronos.org/webgl/wiki) even though the easiest way to find it might be to search for it in a search engine.

Creating the Vertex Shader and the Fragment Shader

All WebGL programs must have both a vertex shader and a fragment shader. The function `setupShaders()` contains the source code for the vertex shader and the fragment shader as two strings. Both shaders are kept as simple as possible in this initial example. First take a look at the vertex shader:

```
var vertexShaderSource =  
    "attribute vec3 aVertexPosition;          \n" +  
    "void main() {                          \n" +  
    "    gl_Position = vec4(aVertexPosition, 1.0); \n" +  
    "}
```

The source code is assigned to the JavaScript variable `vertexShaderSource` as a string. The `+` operator in JavaScript is used to concatenate each row of the source code to one string. This works fine for smaller examples like this one, but if you have bigger shaders, it is sometimes more convenient to define your shaders at a global level within their own `<script>` tag and then access the source code through the DOM API.



Later in this chapter, you learn how to include your shader source code in your WebGL application and then access it through the DOM API. For now, it is enough to understand that there is another way to define the source code for your shaders that looks slightly different.

The first line of the vertex shader defines a variable named `aVertexPosition` of type `vec3`, which means it is a vector with three components. In front of the type, it's also specified that this variable is an attribute. Attributes are special input variables that are used to pass per vertex data from the WebGL API to the vertex shader.

The `aVertexPosition` attribute in this example is used to pass in the position of each vertex that WebGL should use for drawing the triangle. To make the vertices go through the API and end up in the `aVertexPosition` attribute, you also have to set up a buffer for the vertices and connect the buffer to the `aVertexPosition` attribute. These two steps (which you will look at soon) occur later in the functions `setupBuffers()` and `draw()`.

The next line of the vertex shader declares a `main()` function, which is the entry point for the execution of the vertex shader. The body of the `main()` function is very simple and just assigns the incoming vertex to a variable called `gl_Position`. All vertex shaders must assign a value to this predefined variable. It contains the position of the vertex when the vertex shader is finished with it, and it is passed on to the next stage in the WebGL pipeline.

Since `gl_Position` is defined to be of the type `vec4` — i.e., it is using a homogeneous coordinate — a fourth component set to 1 is added to go from a 3D point to a homogeneous point. If you forget what homogeneous coordinates are, you can go back to Chapter 1 and review them.

The fragment shader in this example is also very simple:

```
var fragmentShaderSource =
  "precision mediump float;          \n"+
  "void main() {                      \n"+
    " gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); \n"+
  " }                                \n";
```

The fragment shader is also a string that is concatenated with the `+` operator. The first line of the fragment shader uses what is called a precision qualifier to declare that the precision used for floats in the fragment shader should be of medium precision.

In the same way as for the vertex shader, the `main` function defines the entry point of the fragment shader. The body of the `main()` function writes a `vec4` representing the

color white into the built-in variable `gl_FragColor`. This variable is defined as a four-component vector that contains the output color in RGBA format that the fragment has when the fragment shader is finished with the fragment.

Compiling the Shaders

To create a WebGL shader that you can upload to the GPU and use for rendering, you first need to create a shader object, load the source code into the shader object, and then compile and link the shader. This section describes loading and compiling the source code. The next section describes linking.

The homemade helper function `loadShader()` can create either a vertex shader or a fragment shader, depending on which arguments are sent to the function. The function is called once with the type argument set to `gl.VERTEX_SHADER` and once with the type argument set to `gl.FRAGMENT_SHADER`. The function `loadShader()` is shown here:

```
function loadShader(type, shaderSource) {  
    var shader = gl.createShader(type);  
    gl.shaderSource(shader, shaderSource);  
    gl.compileShader(shader);  
  
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {  
        alert("Error compiling shader" + gl.getShaderInfoLog(shader));  
        gl.deleteShader(shader);  
        return null;  
    }  
    return shader;  
}
```

First a shader object is created with the method `gl.createShader()`. The argument to this method is the type of the shader that you want to create and is either `gl.VERTEX_SHADER` or `gl.FRAGMENT_SHADER`. The source code is then loaded into the shader object with the method `gl.shaderSource()`. This method takes the created shader object as the first argument and the shader source code as the second argument.

When the source code is loaded, the shader is compiled by calling the method `gl.compileShader()`. After compiling the shader, the status of the compilation is checked with `gl.getShaderParameter()`. If there was a compilation error, the user is notified with a JavaScript alert and the shader object is deleted. Otherwise, the compiled shader is returned.

Creating the Program Object and Linking the Shaders

The second part of the function `setupShaders()` creates a program object, attaches the compiled vertex shader and fragment shader, and links everything together to a shader

program that WebGL can use. Here is the source code again:

```
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
}

gl.useProgram(shaderProgram);

        shaderProgram.vertexPositionAttribute      =
gl.getAttribLocation(shaderProgram,
    "aVertexPosition");
}
```

To create the program object, a method named `gl.createProgram()` is used. The method `gl.attachShader()` is used to attach first the compiled vertex shader and then the compiled fragment shader to the program object. After this, the method `gl.linkProgram()` is used to do the linking. If the linking succeeds, you have a program object and you can call `gl.useProgram()` to tell WebGL that this program object should be used for the rendering.

After the linking, the WebGL implementation has bound the attributes used in the vertex shader to a generic attribute index. The WebGL implementation has a fixed number of “slots” for attributes and the generic attribute index identifies one of these “slots.” You need to know the generic attribute index for each attribute in the vertex shader, since during the draw process the index is used to connect the buffer that contains the vertex data with the correct attribute in the vertex shader. There are two strategies that you can use to know the index:

- Use the method `gl.bindAttribLocation()` to specify which index you want to bind your attributes to before you do the linking. Since you specify the index, you can use the same index during drawing.
- Let WebGL decide which index it should use for a specific attribute, and when the linking is done, use the method `gl.getAttribLocation()` to ask which generic attribute index has been used for a certain attribute.

This example uses the second approach. After the linking, the method `gl.getAttribLocation()` is called to find the index that it has bound the attribute `aVertexPosition` to.

The index is saved in the `shaderProgram` object as a new property that is named `vertexPositionAttribute`. In JavaScript an object is basically just a hash map, and a new property of an object can be created simply by assigning a value to it. So the object `shaderProgram` does not have a predefined property called `vertexPositionAttribute`, but this property is created by assigning a value to it.

Later in the `draw()` function, the index that is saved in this property will be used to connect the buffer containing the vertex data to the attribute `aVertexPosition` in the vertex shader.



It can be convenient to add new properties to objects that are returned from the WebGL API (as shown in this section), and you will probably see this in existing WebGL code around the web as well. But this isn't a good strategy if you need to create more robust applications that handle what is referred to as lost context in WebGL. You will learn more about lost context and how to change your code so it is more robust in Chapter 5. Until then, the examples will not be designed to take care of lost context.

Setting Up the Buffers

After you have the shaders in place, the next step is to set up the buffers that will contain the vertex data. In this example, the only buffer you need is the one for the vertex positions for the triangle. This buffer is created and set up in the function named `setupBuffers()`.

```
function setupBuffers() {
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var triangleVertices = [
        0.0, 0.5, 0.0,
        -0.5, -0.5, 0.0,
        0.5, -0.5, 0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
    gl.STATIC_DRAW);
    vertexBuffer.itemSize = 3;
    vertexBuffer.numberOfItems = 3;
}
```

The function starts by calling `gl.createBuffer()` to create a `WebGLBuffer` object that is assigned to the global variable `vertexBuffer`. Then it binds the created `WebGLBuffer` object as the current array buffer object. This tells WebGL that from now on, you are working with this buffer object. The vertices for the triangle are specified in the JavaScript array named `triangleVertices`.

In this example, the vertex shader does not perform any transformations on the vertices. As you remember, it just assigned the incoming vertex to the built-in special variable `gl_Position` without performing any matrix multiplications.

This means that the default right-handed coordinate system for WebGL will be used. The default coordinate system has its origin with coordinates $(0, 0, 0)$ in the middle of the viewport. The x -axis is horizontal and pointing to the right, the y -axis is pointing upwards, and the z -axis is pointing out of the screen towards you. All three axes stretch from -1 to 1 . The lower-left corner of the viewport will then have the coordinates $x = -1$ and $y = -1$, and the upper-right corner of the viewport will have the coordinates $x =$

1 and $y = 1$. Since this example draws a 2D triangle, all the z -values are set to zero so the triangle will be drawn in the xy -plane at $z = 0$.

Because all of the vertices that are specified in the array `triangleVertices` have an x - and y -coordinate between -0.5 and $+0.5$, the corners of the drawn triangle are some distance from the edge of the viewport. If you change the values in the array from -0.5 to -1.0 and from 0.5 to 1.0 and then reload the page, you will see how the corners of the triangle are now on the edge of the viewport.

Next, a `Float32Array` object is created based on the JavaScript array that contains the vertices. You will look more closely at the `Float32Array` in Chapter 3. For now, it is enough to understand that it is used to send in the vertex data to WebGL.

The call to `gl.bufferData()` writes the vertices data to the currently bound `WebGLBuffer` object. This call tells WebGL which data it should place in the buffer object that was created with `gl.createBuffer()`.

The last thing that is done in the function `setupBuffers()` is to add two new properties with information that will be needed later to the `vertexBuffer` object. The first is the property `itemSize`, which specifies how many components exist for each attribute. The second is the property `numberOfItems`, which specifies the number of items or vertices that exist in this buffer. The information in these two properties is needed when the scene is drawn, so even though you learned from the note in the last section that this is not the best method when you later need to handle lost context, this information is added as properties to the `vertexBuffer` close to the location where the vertex data is specified. This will make it easier to remember to update the two properties when you update the structure of the vertex data.

Drawing the Scene

You have finally come to the point where everything should be drawn. You place the code for drawing the actual scene in a function with the simple name `draw()`.

```
function draw(gl) {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
        vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}
```

The first thing that you do in this function is to specify the viewport. The viewport defines where the rendering results will end up in the drawing buffer. When a WebGL context is created, the viewport is initialized to a rectangle with its origin at $(0, 0)$ and a

width and height equal to the canvas width and height. This means that the call to `gl.viewPort()` does not actually modify anything in this example. It is included here anyway since it is a basic WebGL method that is good to be familiar with.

The method `gl.clear()` with the argument `gl.COLOR_BUFFER_BIT` tells WebGL to clear the color buffer to the color that was previously specified with `gl.clearColor()`. This example uses `gl.clearColor()` to specify black as the clear color. So when `gl.clear()` is now called, the color buffer will be cleared to the color black.

In the function `setupBuffers()`, a `WebGLBuffer` object was already created and bound to the `gl.ARRAY_BUFFER` target with the method `gl.bindBuffer()`. The vertex data was sent to this bound buffer with the method `gl.bufferData()`. But what has not been done yet is to tell WebGL which attribute in the vertex shader should take its input from the bound buffer object. In this example, there is only one buffer object and one attribute in the vertex shader, but normally there would be several buffers and attributes, and so these have to be connected somehow.

The WebGL method `gl.vertexAttribPointer()` assigns the `WebGLBuffer` object currently bound to the `gl.ARRAY_BUFFER` target to a vertex attribute passed in as index in the first argument. The second argument of this method is the size or the number of components per attribute. The number of components per attribute is 3 in this example (since there is an *x*, *y*, and *z* coordinate for each vertex position) and you stored this value in the method `setupBuffers()` as a property named `itemSize` of the `vertexBuffer` object.

The third argument specifies that the values in the vertex buffer object should be interpreted as floats. If you send in data that does not consist of floats, the data needs to be converted to floats before it is used in the vertex shader. The fourth argument is called the normalized flag, and it decides how non-floating point data should be converted to floats. In this example, the values in the buffer are floats, so the argument will not be used anyway. The fifth argument is called stride, and specifying zero means that the data is stored sequentially in memory. The sixth and last argument is the offset into the buffer, and since the data starts at the start of the buffer, this argument is set to zero as well.

UNDERSTANDING THE WEBGL CODING STYLE

If you are an experienced programmer, you are probably familiar with what a coding style guide is. A coding style guide can, for example, include guidelines for how you should name your variables and functions, how you should indent your source code, what your loops should look like, and where you should put your braces. Sometimes there are also guidelines for which features of a programming language you should use.

The idea is that if you (and your team if you are several programmers) are consistent and follow the same coding style throughout all the code you write, it is easier to read and maintain the code. Since it is easier to read and maintain, it should also contain fewer bugs.

This book does not try to follow or enforce a complete coding style, but it tries to be as consistent as possible for naming conventions. However, the code examples in this book have been written and organized in a way that reflects the needs of the book (and the reader). They are not necessarily optimized for real-world applications.

For example, the JavaScript source code in this book is often embedded directly in the `<head>` of the HTML page instead of having an external JavaScript file that is included with the following code:

```
<script type="text/javascript" src="filename.js"></script>
```

Including the source code within the HTML page makes it easy to present it in this book, and it is simple to view the source code in a browser by selecting the View Source Code command after you have opened a web page. However, if you were writing a real application, it might be better to write your JavaScript source code in an external file and include it in the HTML file.

The simple naming conventions used in this book are as follows:

- The `WebGLRenderingContext` that is retrieved by calling `canvas.getContext()` is always put in a global variable named `g1`. This means that if the method `drawArrays()` of the `WebGLRenderingContext` is called, the call looks like `g1.drawArrays()`.
- In general, the `camelCase` naming convention is used for variables and functions both in the JavaScript code and in the shader source code written in OpenGL ES Shading Language. This means that the name starts with a lowercase letter and for each new word in the name, the first letter is capitalized. As an example, the JavaScript function that sets up the buffers in [Listing 2-1](#) was named `setupBuffers()`. As you can see (for example, from `g1.drawArrays()`), the public WebGL API also follows `camelCase` for its methods.
- The shader source code uses the prefix “`a`” for attributes, “`v`” for varying variables, and “`u`” for uniforms. This makes it easy to recognize that `aVertexPosition` is an attribute, `vColor` is a varying variable, and `uMVMatrix` is a uniform.

Even though there are many different coding styles for existing WebGL code, these conventions make the code similar to a lot of existing code. When you start to write your own code, you can, of course, follow whatever style you (or your team) decide on.

DEBUGGING YOUR WEBGL APPLICATION

When you are programming traditional compiled languages such as C, C++, or Java, the compiler tells you when you have made mistakes such as missing a semicolon at the end of a statement or misspelling a variable name. Even though most JavaScript implementations today actually compile the JavaScript code to native machine code before they execute it, you as an end user will normally not be informed about mistakes in the JavaScript code in the same way as with a traditional compiler.

This means that if you write a snippet of HTML and JavaScript code and load it into your browser, even very minor mistakes in the JavaScript code that a traditional compiler would normally have caught and informed you about, can cause your application to behave differently than expected. If you tried to manually write the code for [Listing 2-1](#) in an editor and then loaded it into a browser, there would be a risk that you'd encounter this problem.

Fortunately, help is available. All the browsers that have WebGL support have some sort of development tools. If you have previously done any other web development, you are probably familiar with one or more of these tools. Here is a list of some of the popular ones:

- **Chrome Developer Tools** — Built into Google Chrome
- **Firebug** — An extension to Mozilla Firefox
- **Web Inspector** — Built into Safari
- **Dragonfly** — Built into Opera

This book primarily uses Chrome Developer Tools. You will also get a short overview of how Firebug works. If for some reason you prefer another browser or development tool, you should be able to find the documentation for them on the web and then easily follow most of the instructions in this book.

Using Chrome Developer Tools

Chrome Developer Tools is a powerful web development feature that is integrated into the Google Chrome Browser by default. It is based on a tool called the Web Inspector that is developed by the WebKit open source community. This Web Inspector is the same tool that the Safari browser has built in for web development.

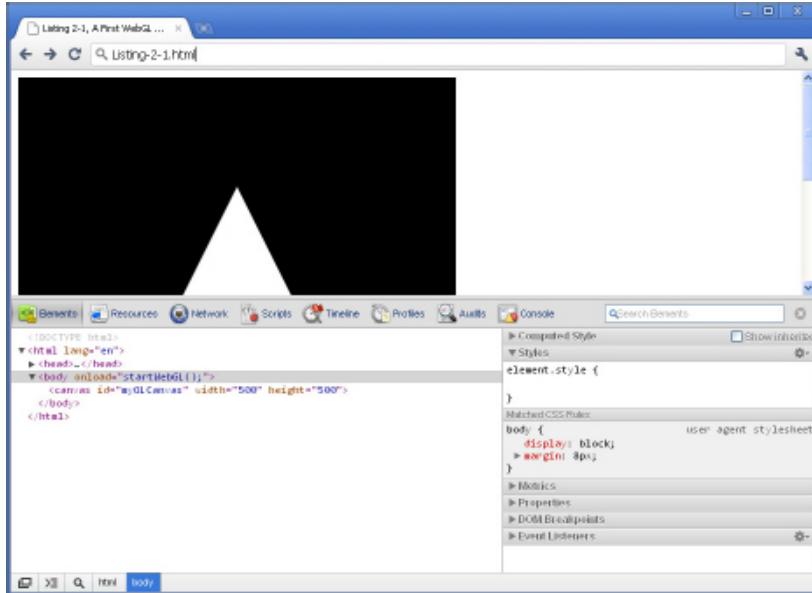
When you have loaded your web page into the Chrome browser, you can start Chrome Developer Tools in several different ways:

- At the top-right corner of your browser window, you can select the wrench menu and then select Tools → Developer Tools.

- On Windows and Linux, you can use the handy shortcut Ctrl+Shift+I. The corresponding shortcut on a Mac keyboard is Command+Option+I.
- You can right-click an element in the web page and select Inspect Element in the pop-up menu.

When you open up Chrome Developer Tools, you see an interface similar to [Figure 2-2](#).

FIGURE 2-2: The User Interface of Chrome Developer Tools when it is docked to the main browser window



In [Figure 2-2](#), the Chrome Developer Tools window is docked to the bottom part of the main window, and you can see some of your web content in the background. At the top of the Chrome Developer Tools window, you can see a toolbar with eight icons. These icons correspond to the eight panels that Chrome Developer Tools consists of:

- Elements
- Resources
- Network
- Scripts
- Timeline
- Profiles
- Audits
- Console

By clicking an icon, you can select the corresponding panel. The following section contains a brief description of the different panels and the functionality that they provide.



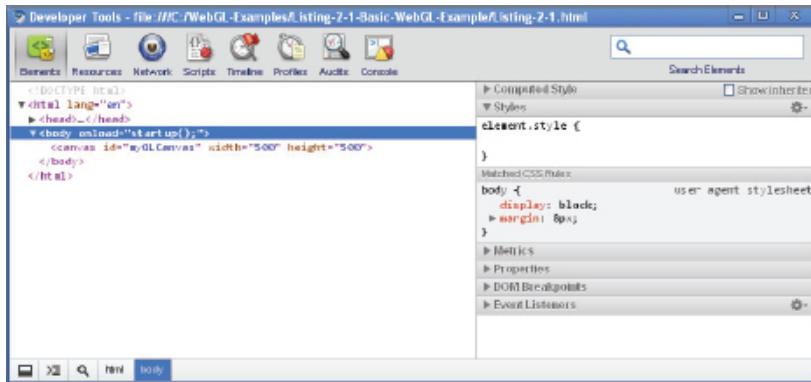
This section gives a brief overview of Chrome Developer Tools. In addition, it focuses on how Chrome Developer Tools can be useful when you develop and debug a WebGL application.

On the web you will find excellent documents and videos from Google about how Chrome Developer Tools works. You should both read the documentation and watch some of the videos. The easiest way to find this information is to use a search engine.

The Elements Panel

The Elements panel gives you an overview of the DOM tree that your web page consists of. You can see it as a much more sophisticated alternative to the View Source Code command that most browsers provide. You can see how the Elements panel looks when it is undocked in [Figure 2-3](#).

FIGURE 2-3: The Elements Panel of Chrome Developer Tools



If you are viewing a web page and want to know details of an element on the page, you can right-click the element and select Inspect Element. This opens the Elements panel with the element highlighted. In [Figure 2-3](#), the inspected element is selected for the canvas. In the sidebar to the right, you can find information about styles, metrics, properties, and event listeners.

A WebGL application does not often use such a big and complicated DOM tree. A lot of the functionality will instead be in the JavaScript and in the shaders. Of course, there are exceptions, and the Elements panel can be useful even for web applications with small DOM trees.

One feature in the Elements panel that can be very useful is the ability to change the HTML, DOM, or CSS in real time. In Chapter 8, you will learn about WebGL optimizations and how different changes affect the performance of your WebGL application. With the Elements panel, you can, for example, change the width and height of the `<canvas>` in real time and see how that affects the performance. The result of this test can quickly give you valuable information about where you might have a performance bottleneck in your WebGL application.

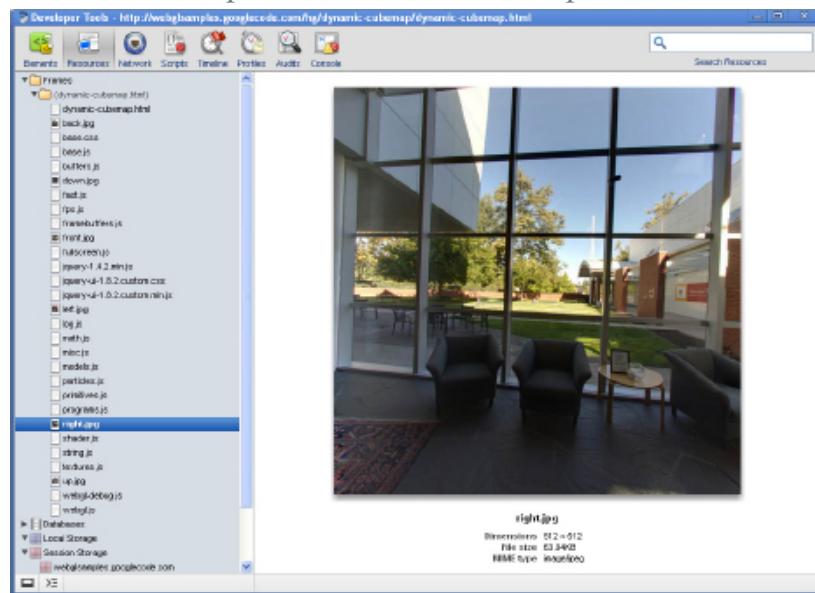
The Resources Panel

The Resources panel lets you look at all the resources that a web application consists of. You get a quick overview of all the used resources including HTML, JavaScript, CSS, and image files that make up the web application. This is very useful when you find a WebGL application that you think looks really good and you want to understand how it has been built.

The Resources panel quickly gives you an idea of how big the application is and whether any JavaScript libraries have been used for it. This panel is also a good way to search through all the resources available in the application. For example, if you want to know how an application uses textures, you can search for the word “texture” in the Resources panel. When you get a result, you can press the Enter key to search for the next occurrence of the word.

When you scroll through the resources in the left sidebar and you select an image file, the corresponding image displays in the right part of the panel. This can help you to understand how the application uses different textures. In [Figure 2-4](#), a resource that is an image file for a texture has been selected in the sidebar to the left. To the right, you see what the actual image looks like. This particular example is from the WebGL demo “Dynamic Cubemap” (<http://webglsamples.googlecode.com/hg/dynamic-cubemap/dynamic-cubemap.html>), which is developed by Gregg Tavares at Google.

FIGURE 2-4: The Resources panel of Chrome Developer Tools

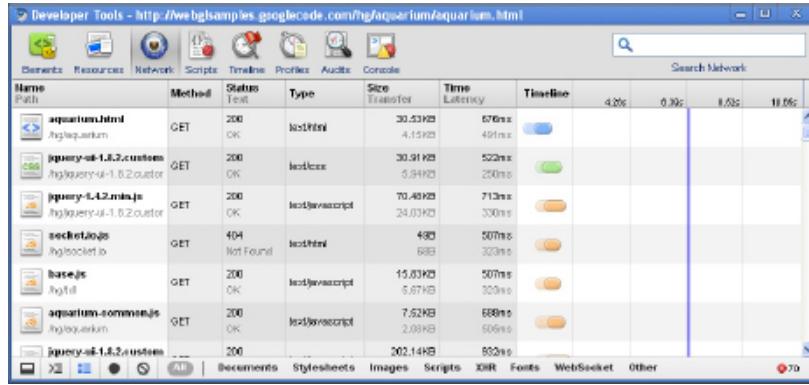


The Network Panel

The Network panel (see [Figure 2-5](#)) lets you investigate which resources are downloaded over the network. This panel helps you to minimize the startup time for your WebGL application. For example, you get a better understanding of the order in which

the resources are downloaded and whether you can make the downloading of your resources more efficient.

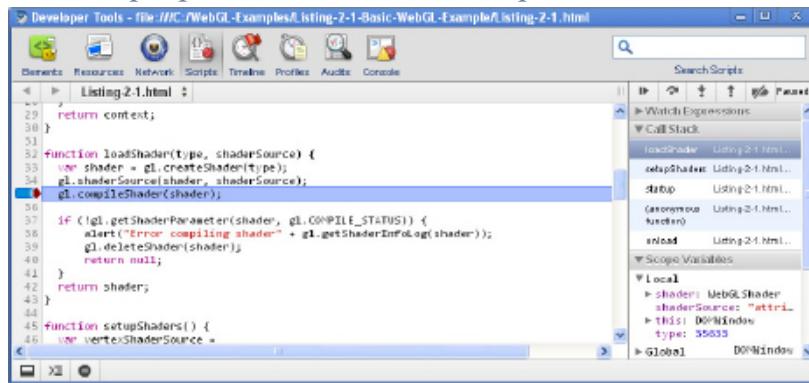
FIGURE 2-5: The Network panel of Chrome Developer Tools



The Scripts Panel

The Scripts panel (see [Figure 2-6](#)) provides a powerful debugger for your JavaScript code. You can set breakpoints in the code, single step, look at the stack trace, and inspect the values of the variables. This is one of the panels where you will probably spend most of your time when developing and debugging WebGL applications with Chrome Developer Tools.

FIGURE 2-6: The Scripts panel of Chrome Developer Tools

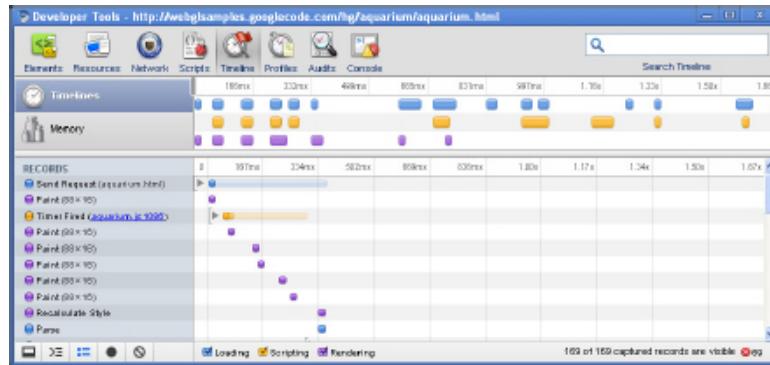


[Figure 2-6](#) shows what the Scripts panel looks like when the execution has stopped at a breakpoint line. On the right side, you can see the call stack. In the top-right corner, you have buttons to continue the execution, single step over a function, single step into a function, step out of a function, and disable all breakpoints.

The Timeline Panel

The Timeline panel (see [Figure 2-7](#)) gives you an overview of different events that happen while you load your web application. You will probably not use it very often when you focus on the WebGL part of your development.

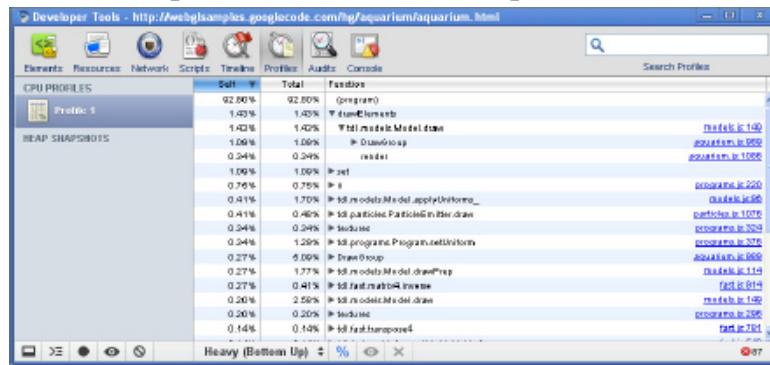
FIGURE 2-7: The Timeline panel of Chrome Developer Tools



The Profiles Panel

With the Profiles panel, you can profile CPU and memory usage. If you have a lot of physics calculations or many matrix multiplications or any other logic that you perform in JavaScript on the CPU and you want to find which functions take up the most time, the Profiles panel can help you a lot. [Figure 2-8](#) shows what it looks like when a profile has been recorded.

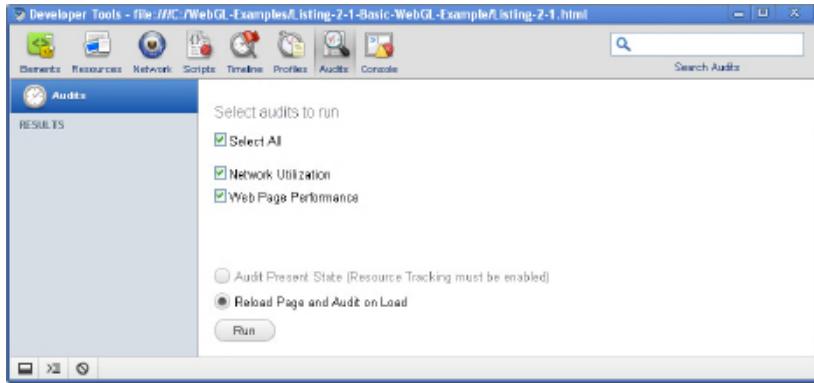
FIGURE 2-8: The Profiles panel of Chrome Developer Tools



The Audits Panel

If you want to improve or optimize your web application but you are not sure how to do it, the Audits panel can help you. It gives you tips about what you can improve, such as to combine your JavaScript files or improve caching. [Figure 2-9](#) shows the Audits panel before the audit has been started.

FIGURE 2-9: The Audits panel of Chrome Developer Tools



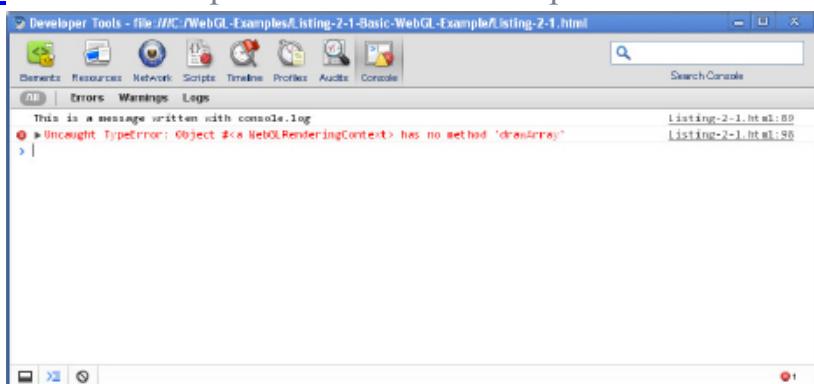
The Console Panel

The Console panel is special in that it is available as a separate panel, but you can also bring it up in any of the other panels by pressing the Escape key.

Here you will find logs that are printed with `console.log()` in the JavaScript code. You will also be notified about syntax and runtime errors in the JavaScript code. If you develop your WebGL application and load it into the browser but it does not work as expected, the Console panel is one of the first panels you should check. Initially you will probably use the Console panel mostly to look at output, but this panel also supports a command line API that you can use to write commands directly in the Console panel. You should look at online documentation for Chrome Developer Tools if you think this sounds interesting.

In [Figure 2-10](#), the first line is a message written from the JavaScript code with `console.log()`. The second line is the result from a JavaScript exception that is due to the fact that the JavaScript tried to call `gl.drawArray()` instead of `gl.drawArrays()`. There is no method called `gl.drawArray()` in the WebGL API, so you get an exception that is shown in the Console panel.

FIGURE 2-10: The Console panel of Chrome Developer Tools



Using Firebug

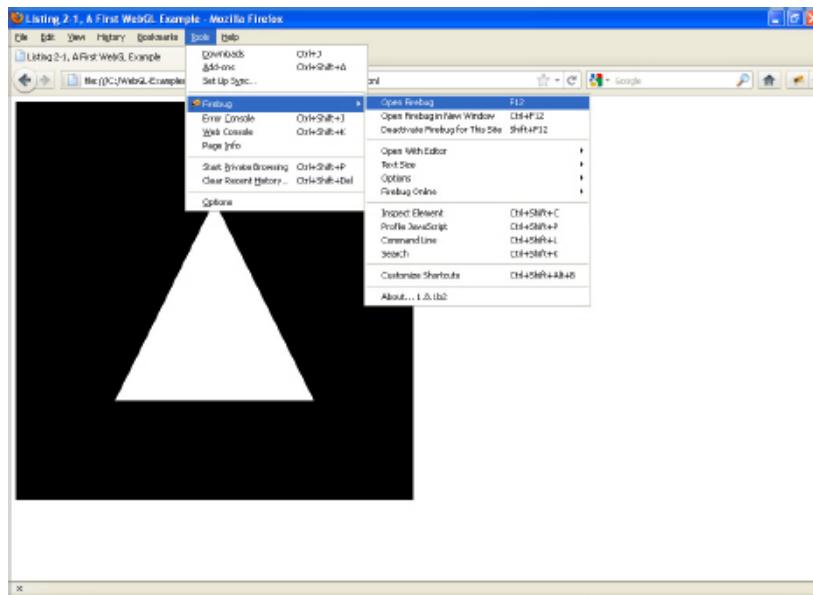
Even though most of the examples in this book use the Google Chrome browser and Chrome Developer Tools, this book also briefly describes how to debug your WebGL application with Firefox and Firebug. After all, Firebug is a great tool that has been around for a longer time than Chrome Developer Tools, and it has many loyal users.

Firebug is an extension to Firefox that was initially developed by Joe Hewitt, who is also one of the original programmers behind Firefox. It has basically the same functionality that is available in Chrome Developer Tools. Since Firebug is an extension to Firefox, you have to download and install it separately. You can do this easily from the URL, <http://getfirebug.com/>.

Once you have installed Firebug, you can start it in a few different ways:

- Use the Firefox menu by selecting Tools → Firebug → Open Firebug, as shown in [Figure 2-11](#).
- Use the keyboard shortcut F12 to both open and close Firebug.
- Right-click an element on a web page and select Inspect Element.

FIGURE 2-11: After Firebug is installed, you can open it from the Tools menu in Firefox

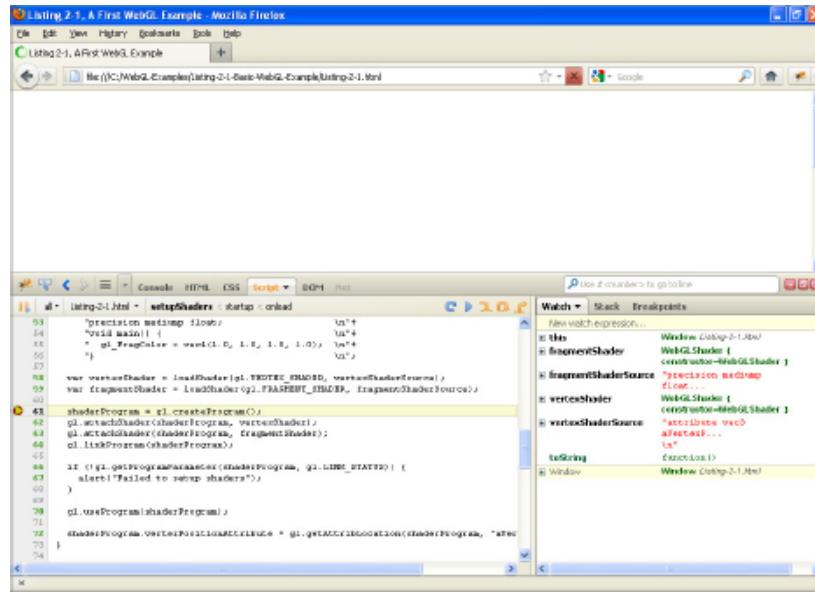


When you open Firebug, you see that it is organized in different panels in a similar way to the Chrome Developer Tools interface. This is because the developers for Chrome Developer Tools were inspired by Firebug, which already existed and was a popular and proven concept when they began to develop Chrome Developer Tools.

The fact that the two tools are quite similar makes it easier for you as a developer to switch between them. [Figure 2-12](#) shows the Scripts panel of Firebug, which contains a similar functionality to the Scripts panel in Chrome Developer Tools. You can set breakpoints in the JavaScript code, single step, look at the call stack, and so on. [Figure](#)

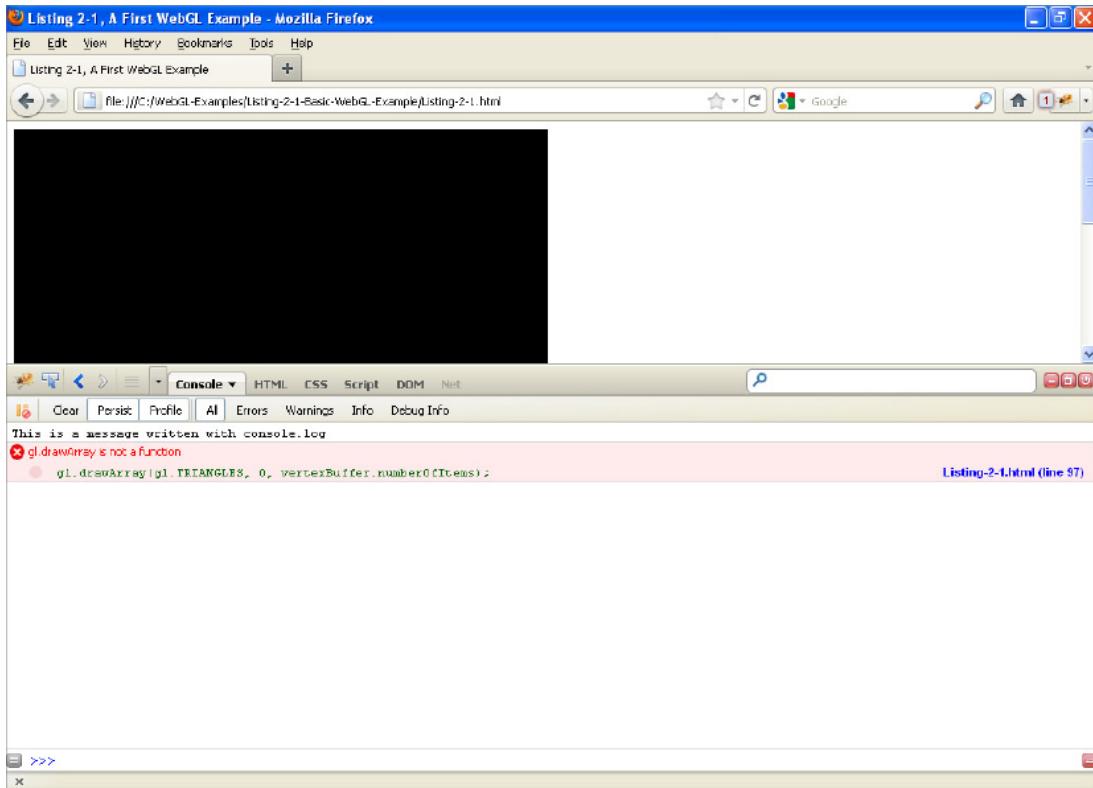
[2-12](#) displays what the panel looks like when the execution of a JavaScript has stopped at a breakpoint.

FIGURE 2-12: The Scripts panel in Firebug



The last Firebug panel described here is the Console panel, which again is very similar to the Console panel in Chrome Developer Tools. [Figure 2-13](#) shows the panel; the first line shows output from `console.log()` and the second line informs you that there is a runtime error in the JavaScript code. You can compare this figure with [Figure 2-10](#), where the corresponding information was shown for Chrome Developer Tools.

FIGURE 2-13: The Console panel in Firebug



If you learn to debug your WebGL applications with either Chrome Developer Tools or Firebug, it will be easy to switch between the two. The rest of this book uses Chrome Developer Tools simply because one of the tools has to be chosen and Chrome Developer Tools is included by default in the Chrome browser.

WebGL Error Handling and Error Codes

When WebGL detects that an error has happened, it generates an error code that is recorded. When an error is recorded, no other errors are recorded until the WebGL application calls the method:

```
gl.getErrors()
```

This method is used to query for the recorded error. It returns the current error code (see [Table 2-1](#) for a list of error codes) and resets the current error to `gl.NO_ERROR`. After this, new errors are recorded.

TABLE 2-1

WEBGL ERROR CODE	DESCRIPTION
<code>gl.NO_ERROR</code>	There has not been a new error recorded since the last call to <code>gl.getError()</code> .
<code>gl.INVALID_ENUM</code>	An argument of the type <code>GLenum</code> is out of range. An example is if you call <code>gl.drawArrays()</code> and send in <code>gl.COLOR_BUFFER_BIT</code> as the first argument instead of <code>gl.TRIANGLES</code> or one of the other valid <code>enum</code> values that this method expects.

WEBGL ERROR CODE	DESCRIPTION
gl.INVALID_VALUE	An argument of a numeric type is out of range. An example is if you call <code>gl.clear()</code> and send in <code>gl.POINTS</code> instead of <code>gl.COLOR_BUFFER_BIT</code> or one of the other valid values that the method expects.
gl.INVALID_OPERATION	A method is called that is not allowed in the current WebGL state. This error can result when you have forgotten to call the correct methods before you call the method that generates this error. An example is if you call <code>gl.bufferData()</code> without first calling <code>gl.bindbuffer()</code> .
gl.OUT_OF_MEMORY	There is not enough memory to execute a specific method.

You can have two strategies for using `gl.getError()` in your WebGL applications:

- You could include `gl.getError()` in your code all the time as a way of writing defensive code. If you get an error, you try to handle it gracefully. This is not recommended because calling `gl.getError()` can hinder the performance of your application considerably.
- The other strategy that is more common for WebGL is to only use `gl.getError()` for debugging purposes. You include it in your code when you develop and try to fix problems. When your code is ready, it doesn't contain any calls to `gl.getError()`. This second strategy is recommended, especially for code (where performance is important).

To be sure exactly which call generates an error, you need to call `gl.getError()` after each WebGL call. This strategy can be very burdensome and the code can be more difficult to read. Another way is to use a library that wraps your `WebGLRenderingContext` and calls `gl.getError()` behind the scenes for you after each call. The authors of Chromium (which is the open-source web browser project from which Google Chrome draws its source code) have developed a small JavaScript library that is very useful even though it only consists of one file. To use it in your WebGL application:

1. Download the JavaScript library from
<https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/debug/webgl-debug.js>

2. Place it together with your WebGL source code and include it with the following code:

```
<script src="webgl-debug.js"></script>
```

3. When you create the `WebGLRenderingContext`, you wrap it with a call that is available in the JavaScript library:

```
gl=WebGLDebugUtils.makeDebugContext(canvas.getContext("webgl"));
```

After you perform these steps, all generated WebGL errors that you would get by using `gl.getError()` are printed in the JavaScript console.

To give you a more complete picture of how easy it is to use `webgl-debug.js`, the following snippet of code shows some parts from [Listing 2-1](#) after the JavaScript library `webgl-debug.js` is used.

```

<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Snippet demonstrating webgl-debug library</title>
<meta charset="utf-8">
<script src="webgl-debug.js"></script>
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

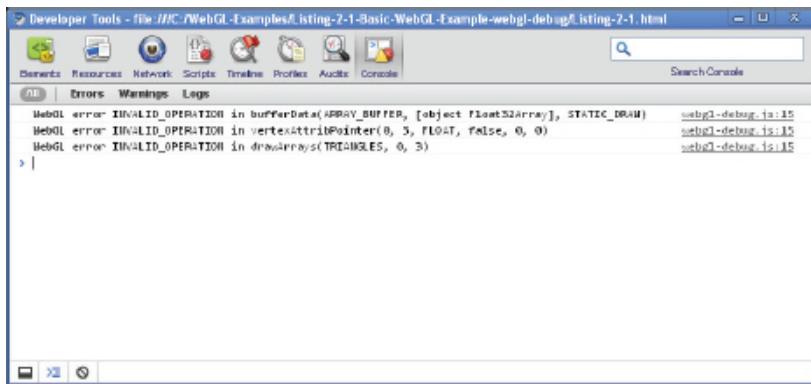
function createGLContext(canvas) {
    var names = ["webgl", "experimental-webgl"];
    var context = null;
    for (var i=0; i < names.length; i++) {
        try {
            context = canvas.getContext(names[i]);
        } catch(e) {}
        if (context) {
            break;
        }
    }
    if (context) {
        context.viewportWidth = canvas.width;
        context.viewportHeight = canvas.height;
    } else {
        alert("Failed to create WebGL context!");
    }
    return context;
}
. . .

function startup() {
    canvas = document.getElementById("myGLCanvas");
    gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
    setupShaders();
    setupBuffers();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    draw();
}

```

[Figure 2-14](#) displays the Console panel of Chrome Developer Tools when the debug library is used and the call to `gl.bindBuffer()` is intentionally removed to generate a WebGL error. When `gl.bindBuffer()` is not called, WebGL generates a `gl.INVALID_OPERATION` when `gl.bufferData()` is called. Since the debug library calls `gl.getError()`, it clears the recorded error and the execution continues, but without any data loaded into the buffer. This means that you see two more logs related to `gl.INVALID_OPERATION` in the Console panel: one when `gl.vertexAttribPointer()` is called and the other when `gl.drawArrays()` is called.

FIGURE 2-14: The Console panel of Chrome Developer Tools when `webgl-debug.js` is used to log three WebGL errors



When you use the debug library, a line number is written to the right of each row in the Console panel that corresponds to a WebGL error. However, this line number is not the line where the method that generated the error was called. Instead, this line number is the line in the `webgl-debug.js` where the `console.log()` call is located. However, if you want to know the line on which you have the method that caused the error, it's easy to find. Open the Scripts panel and put a breakpoint on the line where the library calls the `console.log()`. Then run your application again; when you hit the breakpoint, you can easily see which WebGL method caused the error by looking at the call stack in the Scripts panel.



Note that since the library `webgl-debug.js` uses the method `gl.getError()`, which has a negative impact on the performance, you should not use `webgl-debug.js` in production code.

WebGL Inspector

Chrome Developer Tools and Firebug are great tools that will help you tremendously when you are developing and debugging WebGL applications. However these tools are general web development tools that are not designed specifically for WebGL.

A very useful tool that is developed specifically for WebGL is the WebGL Inspector. This is a WebGL debugger that provides similar features as PIX does for Direct3D, and gDEBugger for OpenGL. WebGL Inspector is open source and is available as a Chrome extension originally written by Ben Vanik. You can download it from <http://benvanik.github.com/WebGL-Inspector/>.

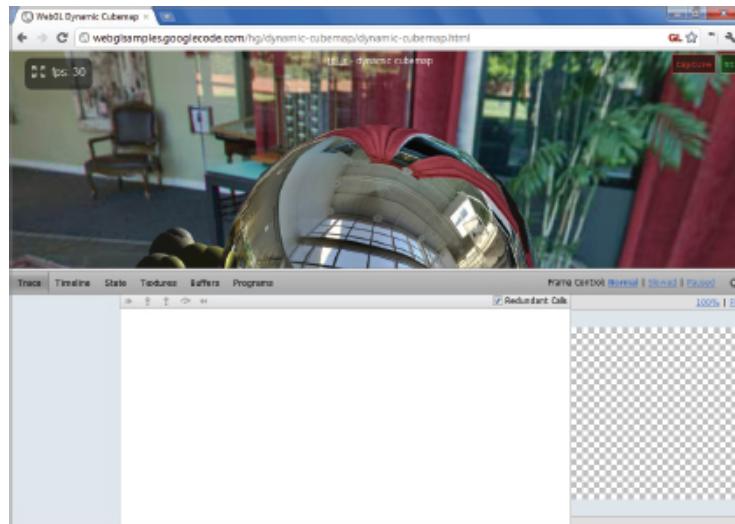
Once you have installed WebGL Inspector and you browse to a web page that contains WebGL content, a small red icon with the text “GL” is shown to the far right of the address bar (sometimes called the *omnibox*) of Google Chrome.



In this section, the demo Dynamic Cubemap (<http://webglsamples.googlecode.com/hg/dynamic-cubemap/dynamic-cubemap.html>) created by Gregg Tavares at Google is used to illustrate how WebGL Inspector is working.

If you click on the GL icon, two buttons named Capture and UI appear on the top right of the web page, just below the GL-icon. If you click the UI button, the full UI of the WebGL Inspector is shown at the bottom of Google Chrome. You will see something that looks similar to [Figure 2-15](#). Initially the UI will not contain any recorded information. If you click UI again, the full UI disappears, leaving you with only Capture and UI on the screen.

FIGURE 2-15: The user interface of the WebGL Inspector before anything has been captured



When you click Capture, one frame of your WebGL application is recorded. After a frame is recorded, you can use the WebGL Inspector to look at a lot of interesting information about the WebGL application you debug.

The WebGL Inspector has six toolbar items at the top of its window:

- Trace
- Timeline
- State
- Textures
- Buffers
- Programs

Each toolbar item corresponds to a panel that is shown when you click on the toolbar item. In this way, the user interface is organized similarly to Chrome Developer Tools and Firebug. The WebGL Inspector contains many useful features and the best way to

learn the details of the tool is to start experimenting with it yourself. In the following sections, a few selected features are described.



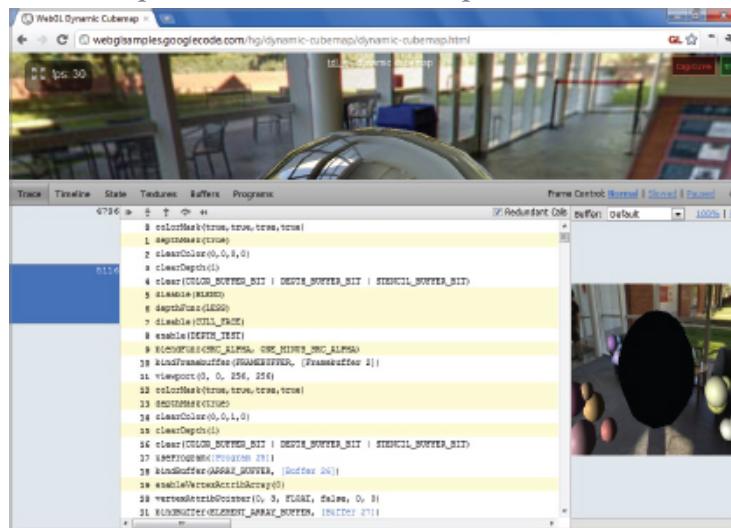
When you now try the WebGL Inspector, there is probably some information that the tool gives you that you do not understand. Even the discussion of the WebGL Inspector in the following sections mentions some concepts that you have not learned about.

You learn much more about WebGL in this book and you should use the WebGL Inspector as you are reading the book. This will help you develop a stronger understanding of this valuable tool.

The Trace Panel

The Trace panel (see [Figure 2-16](#)) shows all WebGL calls that have been recorded during the captured frame. You can step through the recorded commands. To the right of the Trace panel, you will see how the WebGL scene is constructed when you step through the recorded commands.

FIGURE 2-16: The Trace panel of the WebGL Inspector



To step though the recorded commands, you can either use the small buttons with arrows on them at the top of the Trace panel, or you can use a function key that works as a shortcut for the action:

- **F8**—Single step forward
- **F9**—Playback the entire scene
- **F6**—Step backward one call
- **F7**—Skip to the next draw call
- **F10**—Restart from the beginning of the frame

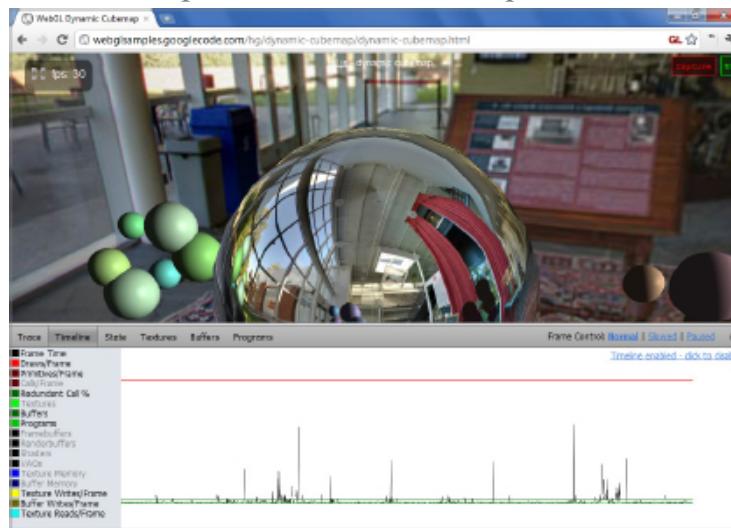
The Trace panel shows when the WebGL application is binding a specific buffer or loading a shader program. When you click on the buffers or shader programs, a new panel displays with more information about the item you clicked on.

A very useful feature in the Trace panel is that WebGL Inspector can highlight redundant calls in yellow. The redundant calls are calls to the WebGL API that do not change any meaningful state of WebGL. For example, if you set a uniform to the same value it already has, this is regarded as a redundant call and is highlighted in the Trace panel. When you need to optimize the performance of a WebGL application, you can see if it is possible to optimize away some of these redundant calls.

The Timeline Panel

The Timeline Panel shows you real-time statistics about your WebGL application. [Figure 2-17](#) shows what the Timeline panel looks like.

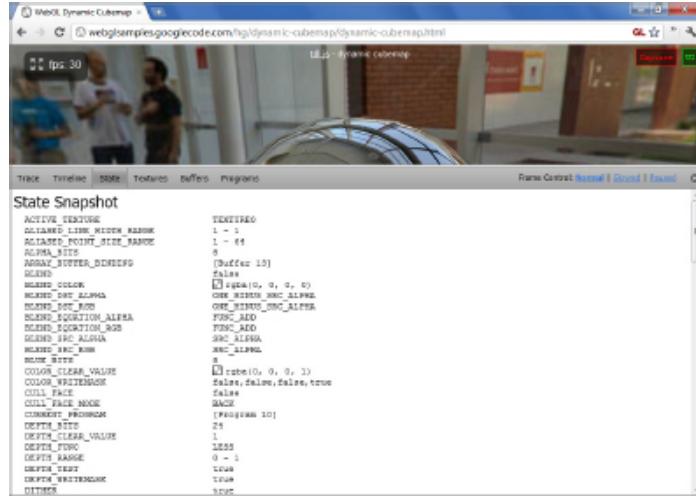
FIGURE 2-17: The Timeline panel of the WebGL Inspector



The State Panel

The State panel shows a snapshot of the state for your WebGL application. [Figure 2-18](#) shows what the State panel looks like.

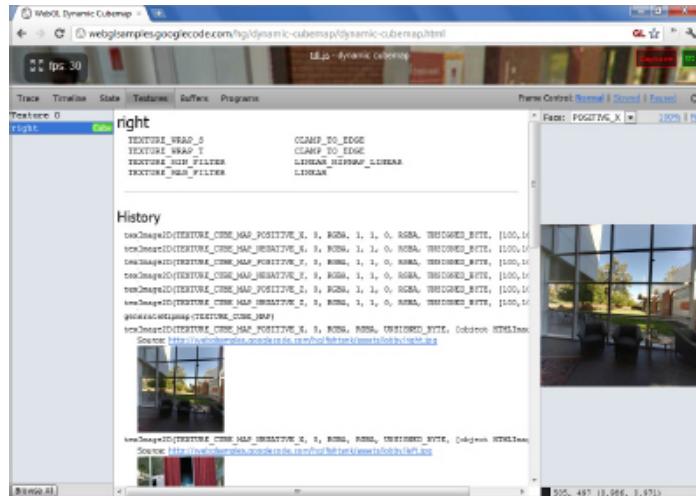
FIGURE 2-18: The State panel of the WebGL Inspector



The Textures Panel

The Textures panel (see [Figure 2-19](#)) shows you a lot of valuable information about the textures that are used in your WebGL application. You can see images for the textures and information identifying them as a 2D texture or a cubemap texture . This panel also includes information about texture wrap mode, texture filtering, and from which URLs the textures are loaded. You will learn more about textures in Chapter 5.

FIGURE 2-19: The Texture panel of the WebGL Inspector

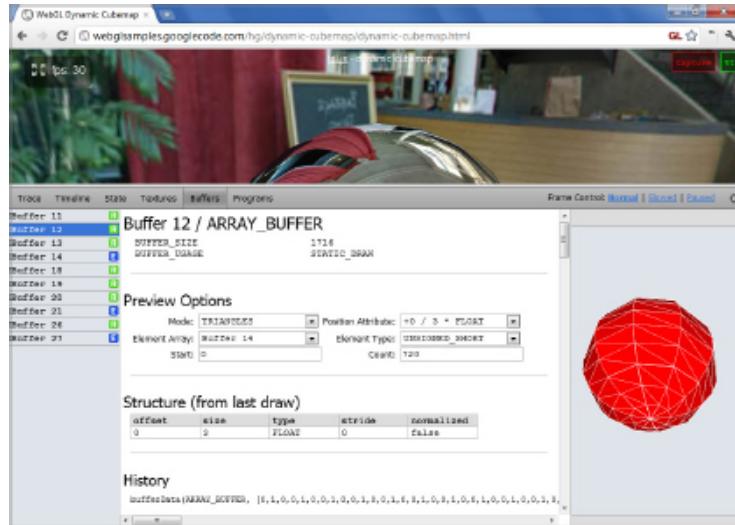


The Buffers Panel

The Buffers panel displays information about the different buffers that are used in the WebGL application. The left of the Buffers panel displays the different buffers that are used in your WebGL program, including information about the content and the size of the buffers and also where the different buffers are bound. The right of the Buffers panel displays the geometry to which the selected buffer corresponds.

In [Figure 2-20](#), the geometry for the selected buffer is a sphere. You can use the mouse to rotate and zoom the geometry, which is really useful when you have more complicated geometry.

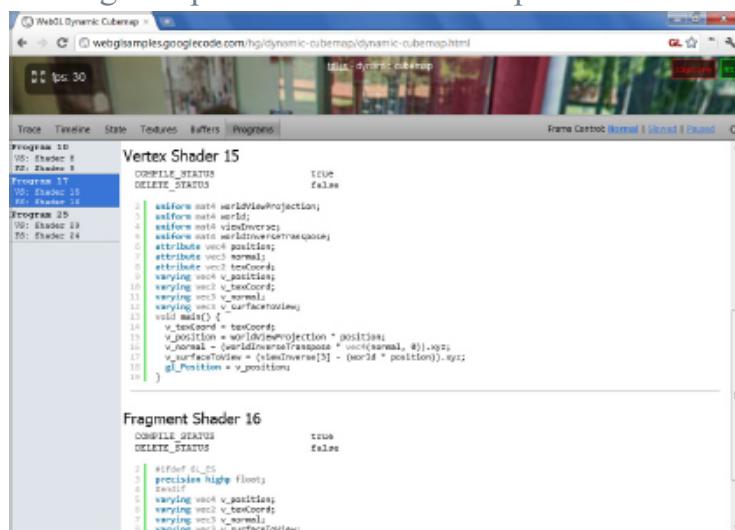
FIGURE 2-20: The Buffers panel of the WebGL Inspector



The Programs Panel

The Programs panel provides you with information about all shader programs that the WebGL application uses. You can see which uniforms and attributes the program uses, and you can also see the source code for both the vertex shader and the fragment shader. Figure 2-21 shows a part of the Programs panel.

FIGURE 2-21: The Programs panel of the WebGL Inspector



Troubleshooting WebGL

There are several things that can go wrong with a WebGL application. To be able to troubleshoot and fix any problems in your application as quickly and easily as possible, it helps to understand what these problems are. Here is a list of the different categories of errors that you may encounter:

- **JavaScript syntax error**—You have made a typo or similar mistake, so the JavaScript engine does not accept the JavaScript source code as valid JavaScript. If you have a syntax error in your JavaScript, your application does not execute at all.
- **Runtime error**—You have made a mistake that the JavaScript engine cannot find before the execution starts. But when you execute your JavaScript, the problem is found and an error is thrown. There are several different runtime errors.
- **Compilation error in the shader**—Something is wrong in your vertex shader or fragment shader source code, so when you try to compile it with `gl.compileShader()`, the compilation fails.
- **Linking error in the shader program**—You have made a mistake in your shaders, so the call to `gl.linkProgram()` fails. For example, this can happen if your fragment shader tries to use a varying variable that is not defined in your vertex shader.
- **WebGL-specific error**—You do something wrong when you call one of the WebGL methods, and so WebGL generates one of the errors listed in [Table 2-1](#). You can retrieve the recorded error with `gl.getError()`.
- **Logical error**—You have committed a logical error in the JavaScript or OpenGL ES Shading Language source code. Your WebGL application consists of valid source code, but due to the logical error, it does not behave as intended.

So now you know that there are many different things that can go wrong when you load your WebGL application into the browser. You also understand the difference between the different problems.

A Troubleshooting Checklist

So what do you do when you load something in your browser and you don't see what you expected? Here is a list of things to check:

1. Does your browser support WebGL? This may be a naïve question, but it's still valid. If you have a browser that does not support WebGL or where WebGL for some reason is not enabled, the call to `canvas.getContext()` returns `null`. Usually this is very obvious to you because most applications that use WebGL have a JavaScript `alert()` or some other mechanism to inform the end user that WebGL is not supported.
2. Check the Console panel of the Chrome Developer Tools to see if you have any syntax errors in the JavaScript code.
3. Check the Console panel of the Chrome Developer Tools to see if you have any runtime errors in the JavaScript code.

4. Check to see if you have any compilation errors or linking errors in your shaders. Normally the code would check whether the calls to `gl.compileShader()` and `gl.linkProgram()` are successful by calling `gl.getShaderParameter()` and `gl.getProgramParameter()`, and then show an `alert()` or inform the user in another way if compilation or linking fails.

5. Check whether any calls to WebGL fail and generate an error that can be retrieved with `gl.getError()`. If you use a debug library such as `webgl-debug.js` that logs these errors with `console.log()`, you see these errors in the Console panel. Start by fixing the first of the errors, since the others might be a result of the first error.

6. Check that your WebGL imaginary camera is pointing in the correct direction. WebGL does not have any built-in support for a camera, but you will learn how to simulate a camera with transformations in Chapter 4. (This information is included here so you can refer to this checklist when you simulate a camera in your WebGL applications.)

7. When you have checked and corrected all the above points and your application still does not behave as you want, it's probably time to start the Scripts panel of the Chrome Developer Tools and debug the code. There are some things that you could actually find by just reviewing your code, but they can be difficult to find, so using the debugger is sometimes easier:

- Check that you did not misspell a name of an object property that you read or write. For example, if you add new properties to the `WebGLBuffer` object to store the `itemSize` and `numberOfItems` that you have in the buffer (as was shown in [Listing 2-1](#)), and then use these properties in the call to `gl.drawArrays()` or `gl.drawElements()`, it is important to make sure that you don't write to a property named `numberOfItems` and then read a property `nbrOfItems`. The reason to check property names very carefully is that JavaScript does not allow you to read uninitialized variables, but it does allow you to read uninitialized properties of an object. This can lead to bugs that are really difficult to find.
- Check that you have spelled all properties of the `WebGLRenderingContext` correctly. This is very similar to the previous point. For example, if you erroneously send in `gl.TRIANGLE` instead of `gl.TRIANGLES` as an argument to `gl.drawArrays()`, JavaScript does not warn you, since it just means that you read an uninitialized property of an object.

There are, of course, many more errors that you can encounter during your WebGL development. But checking this list will hopefully help you to find and correct some of the more irritating problems a bit faster.

USING THE DOM API TO LOAD YOUR SHADERS

In the first WebGL example presented in [Listing 2-1](#), both the vertex shader and the fragment shader were included as strings in the JavaScript code and were concatenated with the JavaScript + operator like this:

```
var vertexShaderSource =
    "attribute vec3 aVertexPosition;          \n" +
    "void main() {                           \n" +
    "    gl_Position = vec4(aVertexPosition, 1.0); \n" +
    "}
```

During the discussion of [Listing 2-1](#), it was indicated that there was another more convenient way to include your shaders in your WebGL application. The following is the same vertex shader source code, but within a `<script>` tag instead. Here the JavaScript + operator is not used to concatenate the strings.

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    void main() {
        gl_Position = vec4(aVertexPosition, 1.0);
    }
</script>
```

For a small example like this, you might feel that there is not a big difference between the two methods; however, when you have bigger shaders with a lot of source code, it will look cleaner and the source code will be easier to read if you use the second approach.

Also, since the shaders in WebGL are written in OpenGL ES Shading Language, there are a lot of example shaders on the Internet and in literature that are written for OpenGL ES 2.0 that you can use for WebGL as well. There are also shader development tools, such as RenderMonkey from AMD, that can output OpenGL ES Shading Language. With the second approach, you can easily include these shaders between the `<script>` tags in your WebGL application.

One thing to remember is that the source code that you write for your shaders between the `<script>` tags is not JavaScript. Since this source code is written in OpenGL ES Shading Language, you cannot use JavaScript syntax or JavaScript methods within this code.

When you include the shader source code between the `<script>` tags instead of directly assigning it to a JavaScript variable, you need a way to retrieve it and concatenate it to one JavaScript string that you can then send in through the WebGL API. A common pattern that is used in a lot of existing WebGL code is shown in the following code snippet. It is difficult to say where this idea originally came from, but early WebGL examples from Mozilla include the pattern.



A slightly different version of the following source code is available from http://learningwebgl.com/cookbook/index.php>Loading_shaders_from_HTML_script_tags, where it is credited to Vladimir Vukićević.

```
function loadShaderFromDOM(id) {
    var shaderScript = document.getElementById(id);

    // If we don't find an element with the specified id
    // we do an early exit
    if (!shaderScript) {
        return null;
    }

    // Loop through the children for the found DOM element and
    // build up the shader source code as a string
    var shaderSource = "";
    var currentChild = shaderScript.firstChild;
    while (currentChild) {
        if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
            shaderSource += currentChild.textContent;
        }
        currentChild = currentChild.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
        shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
        shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
        return null;
    }

    gl.shaderSource(shader, shaderSource);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert(gl.getShaderInfoLog(shader));
        return null;
    }
    return shader;
}
```

The function `loadShaderFromDOM()` takes an `id` attribute as an argument. It uses this `id` to find the corresponding element within the DOM tree. For example, if you give the `<script>` tag that you put your shader source code in the `id "shader-vs"`, you should send in `"shader-vs"` as argument to the function `loadShaderFromDOM()`.

The function uses the DOM API to find the element from the id you send in, and then loops through all the element's children and builds up a text string with the source code.

Finally, the function looks at the type property of the found element and, based on that, creates either a fragment shader or a vertex shader. It loads the source code into the created shader object and compiles it. If the compilation is successful, the shader object is returned to the caller of the function.

PUTTING IT TOGETHER IN A SLIGHTLY MORE ADVANCED EXAMPLE

Up to now, you have seen how to load your shaders with help from the `document.getElementById()` of the DOM API and also learned how to debug and troubleshoot your WebGL application. It is now time to look at a complete example that uses your new knowledge.

[Listing 2-2](#) is based on [Listing 2-1](#) from earlier in this chapter. However, it uses the DOM API to load the vertex shader and the fragment shader that are included within `<script>` tags in the `<head>` of the page. In addition, this example uses the `webgl-debug.js` debug library that was also described earlier in this chapter.



Available for
download on
Wrox.com

LISTING 2-2: A complete example that loads the shaders via the DOM API and includes the debug library `webgl-debug.js`

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 2-2, Load Shaders From DOM</title>
<meta charset="utf-8">
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;

    void main() {
        gl_Position = vec4(aVertexPosition, 1.0);
    }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
    precision mediump float;

    void main() {
```

```

        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
</script>

<script src="webgl-debug.js"></script>
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function createGLContext(canvas) {
    var names = ["webgl", "experimental-webgl"];
    var context = null;
    for (var i=0; i < names.length; i++) {
        try {
            context = canvas.getContext(names[i]);
        } catch(e) {}
        if (context) {
            break;
        }
    }
    if (context) {
        context.viewportWidth = canvas.width;
        context.viewportHeight = canvas.height;
    } else {
        alert("Failed to create WebGL context!");
    }
    return context;
}

function loadShaderFromDOM(id) {
    var shaderScript = document.getElementById(id);

    // If we don't find an element with the specified id
    // we do an early exit
    if (!shaderScript) {
        return null;
    }

    // Loop through the children for the found DOM element and
    // build up the shader source code as a string
    var shaderSource = "";
    var currentChild = shaderScript.firstChild;
    while (currentChild) {
        if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
            shaderSource += currentChild.textContent;
        }
        currentChild = currentChild.nextSibling;
    }
}

```

```

var shader;
if (shaderScript.type == "x-shader/x-fragment") {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
} else if (shaderScript.type == "x-shader/x-vertex") {
    shader = gl.createShader(gl.VERTEX_SHADER);
} else {
    return null;
}

gl.shaderSource(shader, shaderSource);
gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
}
return shader;
}

function setupShaders() {
    vertexShader = loadShaderFromDOM("shader-vs");
    fragmentShader = loadShaderFromDOM("shader-fs");

    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Failed to setup shaders");
    }
}

gl.useProgram(shaderProgram);

            shaderProgram.vertexPositionAttribute =
gl.getAttribLocation(shaderProgram,
    "aVertexPosition");
}

function setupBuffers() {
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var triangleVertices = [
        0.0, 0.5, 0.0,
        -0.5, -0.5, 0.0,
        0.5, -0.5, 0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
    gl.STATIC_DRAW);
    vertexBuffer.itemSize = 3;
}

```

```

    vertexBuffer.numberOfItems = 3;
}

function draw() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                          vertexBuffer.itemSize, gl.FLOAT, false, 0,
                          0);

    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}

function startup() {
    canvas = document.getElementById("myGLCanvas");
    gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
    setupShaders();
    setupBuffers();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    draw();
}
</script>

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>

</html>

```

The source code should be familiar and easy to understand if you understood the previous sections in this chapter. If you concentrate on the highlighted parts of the code, you see the vertex shader and the fragment shader. Just below the shaders, the debug library is included as an external JavaScript file. The function `createGLContext()` is not changed from [Listing 2-1](#).

The next highlighted part contains the function `loadShaderFromDOM()` that you now recognize as well. In the function `setupShaders()`, the JavaScript strings that contained the source code for the shaders are now removed. Instead, `setupShaders()` calls the function `loadShaderFromDOM()`, once for the vertex shader and once for the fragment shader. Finally, if you look at the function `startup()`, you see that it wraps the call to `createGLContext()` within the method `WebGLDebugUtils.makeDebugContext()` to use the debug library.

Experimenting with Code

If you have not already started to change the code and experiment to see what happens, you should do so now. Changing the code will give you a greater understanding, and it will help you to remember the concepts in this chapter. Below are a few suggestions for changing the code. You should understand how to make these changes if you understood the contents of this chapter.

- Change the color of the triangle from white to red.
- Change the background from black to green.
- Change the z-coordinates for the vertices of the triangle so the triangle disappears from the screen. When does this happen?
- Make the triangle look smaller on the screen without changing the vertex data. (Tip: Try the viewport.)
- Intentionally change the code to include the following errors, and look at them in Chrome Developer Tools or in Firebug:
 - JavaScript syntax error
 - JavaScript runtime error
 - The WebGL error `gl.INVALID_OPERATION`

SUMMARY

In this chapter you looked at a basic WebGL example that involved drawing a 2D triangle on the screen. You went through all the source code and gained an understanding of all the basic steps that are needed to create a WebGL application.

You also learned about basic debugging and troubleshooting of your WebGL application. You got an introduction to Chrome Developer Tools and learned how it is useful for working with WebGL. You also had a brief look at Firebug and saw that the two tools were quite similar in many ways. In addition, you learned about the WebGL Inspector and how useful this tool is when you're debugging WebGL applications.

Finally, you learned how to include your vertex shader and fragment shader within the `<script>` element and use the DOM API to retrieve it. Using this technique, it is easy to reuse shader source code that is written or generated for OpenGL ES 2.0.

Chapter 3

Drawing

WHAT'S IN THIS CHAPTER?

- How to draw triangles, lines, and point sprites with WebGL
- Learn about the different primitives
- Understand the importance of triangle winding order
- How to use the drawing methods of WebGL
- Learn what typed arrays are and how they are used in WebGL
- The drawing methods you should use and how to get the best possible performance
- How to interleave different types of vertex data in the same array

In Chapter 2 you learned how you draw a single triangle using WebGL. You used the method `gl.drawArrays()` and sent in `gl.TRIANGLES` as the first argument to tell WebGL that it was a triangle that your vertices specified. As you can imagine, this is not the only way to draw in WebGL. This chapter shows you the different options that are available when you want to draw something using WebGL.

Since the drawing is based on vertex data that you set up in your `WebGLBuffer` objects, you will also take a closer look at the different options you have when you set up and organize your vertex data. The chapter will conclude with an example that uses some of your new knowledge.

WEBGL DRAWING PRIMITIVES AND DRAWING METHODS

The method `gl.drawArrays()` is one of the two available ways to draw your primitives. The other method is `gl.drawElements()`. You will get a better understanding of both of these methods in this chapter. But first, you will take a closer look at the different primitives that your vertices can specify. The primitives described in the following sections are what you can send in as the first argument to either `gl.drawArrays()` or `gl.drawElements()`.

Primitives

You can use WebGL to create complex 3D models. However, these 3D models all have to be built up using the following three basic geometric primitives:

- Triangles
- Lines
- Point sprites

The following sections provide you with the details about each of these primitives.

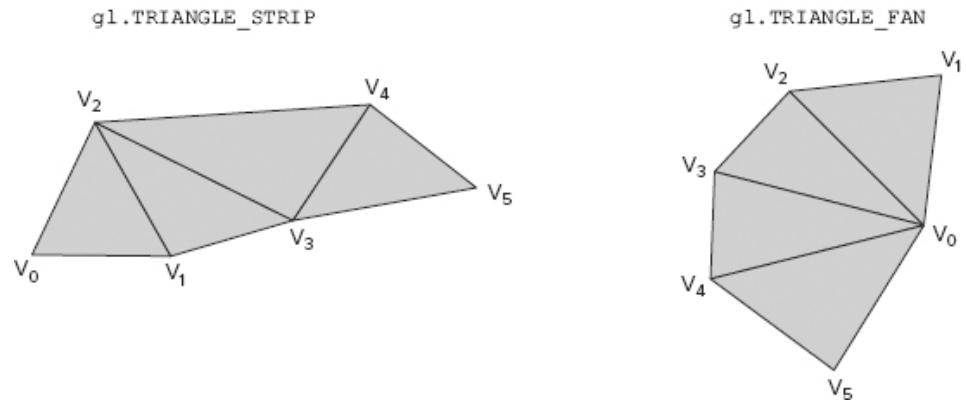
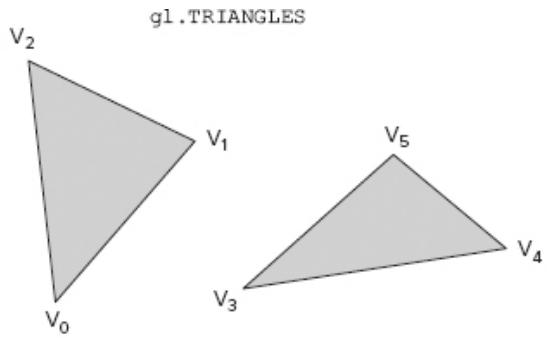
Triangles

In Chapter 1 you learned that points are the most basic building blocks that are used to build up other geometric objects. From a mathematical point of view this is true, but when you consider 3D graphics hardware, it is actually the triangle that is the most basic building block. A 3D point is of course important since you need three points (or vertices) to define a triangle. But most 3D graphics hardware today is highly optimized to draw triangles. The different triangle primitives that you can use with WebGL are part of the `WebGLRenderingContext`:

- `gl.TRIANGLES`
- `gl.TRIANGLE_STRIP`
- `gl.TRIANGLE_FAN`

[Figure 3-1](#) shows an example of what the different primitives could look like. The details of these triangle primitives are described in the following sections.

FIGURE 3-1: The different triangle primitives in WebGL



Independent triangles

`gl.TRIANGLES` is the most basic triangle primitive. If you use `gl.drawArrays()`, you must specify three vertices for each triangle, and if you want to draw several triangles, they are drawn as separate triangles (i.e., vertices from one triangle are not reused in another triangle).

At the top of [Figure 3-1](#) the vertices (V_0, V_1, V_2) specify the first triangle, and the vertices (V_3, V_4, V_5) specify the second triangle. You can determine the number of triangles that are drawn using the following formula:

$$\text{Number of drawn triangles} = \frac{\text{count}}{3}$$

where *count* is the number of vertices that you have when you are using `gl.drawArrays()` or the number of indices that you have if you are using `gl.drawElements()`.

Triangle strips

For many geometric objects you need to draw several connected triangles. `gl.TRIANGLE_STRIP` draws a strip of connected triangles. If you have a lot of connected triangles, the advantage of using a triangle strip over independent triangles is that a triangle strip reuses vertices between the triangles. This allows you to specify fewer vertices when building up the triangles for your geometric models. Fewer vertices mean

less data to process, and this also means less data to transfer from the memory to the GPU.

At the lower-left of [Figure 3-1](#), the first triangle in the strip is defined by the vertices (V_0, V_1, V_2). The next vertex, which is V_3 , together with the two previous vertices, specifies the second triangle. So the second triangle is specified by (V_2, V_1, V_3). Note the order of the vertices here. For a triangle strip (and also for a triangle fan as you will see shortly), the order that is used to build up the first triangle must be followed for the rest of the triangles that make up the same strip.

Since the vertices for the first triangle were specified in counterclockwise order, this order must be followed for the second, third, and fourth triangle as well. This means that the second triangle is specified by (V_2, V_1, V_3) in that order, the third triangle is specified by (V_3, V_2, V_4), and the fourth triangle is specified by (V_4, V_3, V_5).

This pattern is followed for all triangle strips. After the first triangle, each new vertex creates a new triangle with the two previous vertices. For the new triangle, one vertex is added and one vertex is removed (compared to the vertices that build up the previous triangle in the strip). For every second triangle, the first two vertices are reversed.

It is sometimes easier to see the pattern if you have the triangles and vertices in a table. [Table 3-1](#) shows the vertices for the triangles in a triangle strip. In the Corner 3 column, you can easily see that there is one new vertex for each triangle. You can also see that for triangle number 2 and triangle number 4, the vertices for the first and second corners are reversed — i.e., V_2 comes before V_1 for triangle number 2, and V_4 comes before V_3 for triangle number 4.

TABLE 3-1: Pattern of Vertices for Triangle Strip

TRIANGLE NUMBER	CORNER 1	CORNER 2	CORNER 3
1	V_0	V_1	V_2
2	V_2	V_1	V_3
3	V_2	V_3	V_4
4	V_4	V_3	V_5

The number of triangles that are actually drawn when you use `gl.TRIANGLE_STRIP` and `gl.drawArrays()` will be two less than the number of vertices that you specify. (If you instead use `gl.drawElements()`, the number of drawn triangles will be two less than the number of indices.) You can write this in a simple formula:

$$\text{Number of drawn triangles} = \text{count} - 2$$

where *count* is the number of vertices that you have when you are using `gl.drawArrays()` or the number of indices that you have if you are using `gl.drawElements()`.

Triangle fans

If you want to draw several connected triangles, an alternative to the triangle strip is to draw a triangle fan using `gl.TRIANGLE_FAN`.

For a triangle fan, the first three vertices create the first triangle. The first vertex is also used as the origin for the fan, and after the first triangle every other vertex that is specified forms a new triangle with the previous vertex and the origin.

You can see this at the lower right of [Figure 3-1](#) where (V_0, V_1, V_2) creates the first triangle. The second triangle is created with (V_0, V_2, V_3) , the third triangle with (V_0, V_3, V_4) , and the fourth triangle with (V_0, V_4, V_5) .

In the same way as for triangle strips, the triangle fan needs fewer vertices (or indices for `gl.drawElements()`) than the independent triangles to create the same number of triangles. The number of triangles that are drawn when you use `gl.TRIANGLE_FAN` is given by the following formula:

$$\text{Number of drawn triangles} = \text{count} - 2$$

where *count* is the number of vertices that you have when you are using `gl.drawArrays()` or the number of indices that you have if you are using `gl.drawElements()`.

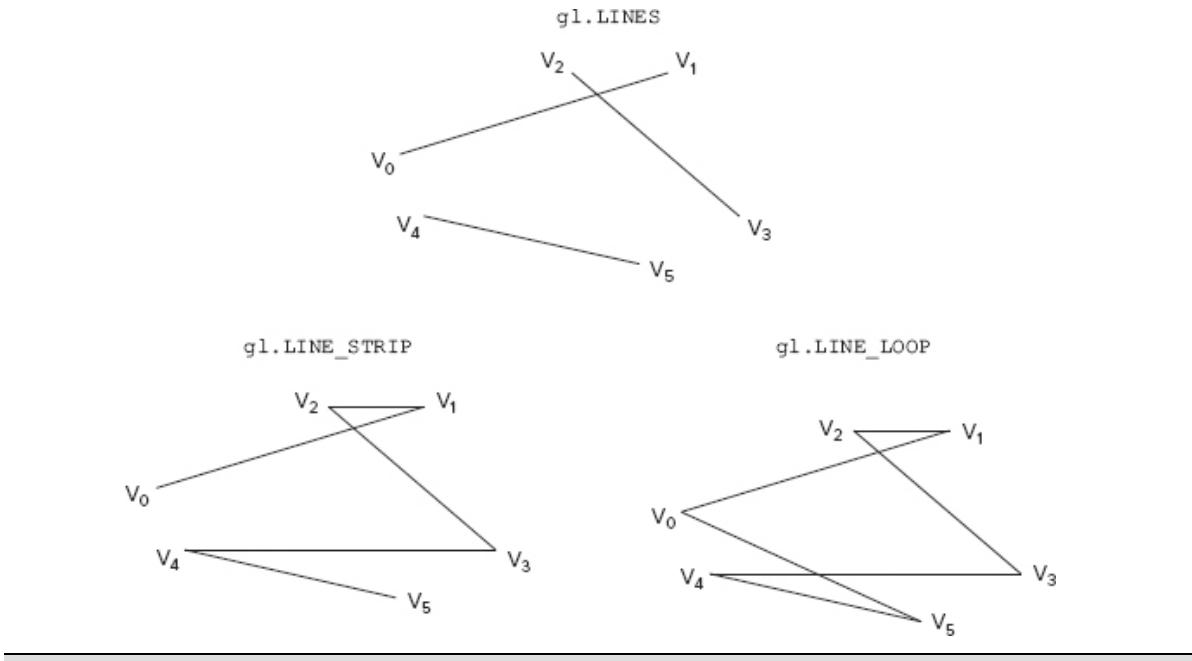
Lines

Even though 3D graphics are concerned mainly with drawing triangles, it is sometimes convenient to be able to draw lines as well. Here are the line primitives that you can use with WebGL:

- `gl.LINES`
- `gl.LINE_STRIP`
- `gl.LINE_LOOP`

[Figure 3-2](#) shows an example of the different primitives. Each line primitive is described in detail in the following sections.

FIGURE 3-2: The different line primitives in WebGL



When you draw a triangle with one of the triangle primitives discussed in the previous section, the triangle is filled by default. If, for some reason, you want to draw a geometric shape that isn't filled (sometimes referred to as a wireframe model), one of the line primitives described in this section might be a better choice.

Independent lines

In the same way that independent triangles drawn with `gl.drawArrays()` do not reuse any vertices between the triangles, independent lines do not reuse any vertices between the lines. You draw independent lines with `g1.LINES`. At the top of [Figure 3-2](#) three separate lines are drawn between (V_0, V_1) , (V_2, V_3) , and (V_4, V_5) .

Two vertices will be used for each line and therefore the number of drawn lines is given by this formula:

$$\text{Number of drawn lines} = \frac{\text{count}}{2}$$

where *count* is the number of vertices that you have when you are using `gl.drawArrays()` or the number of indices that you have if you are using `gl.drawElements()`.

Line strips

If you want to draw several connected lines so one line starts at the same point as another line ends, you can draw line strips using `g1.LINE_STRIP`. At the lower left of [Figure 3-2](#) a series of line strips are drawn between (V_0, V_1) , (V_1, V_2) , (V_2, V_3) , (V_3, V_4) , and (V_4, V_5) . The number of drawn lines is given by this formula:

$$\text{Number of drawn lines} = \text{count} - 1$$

where *count* is the number of vertices that you have when you are using `gl.drawArrays()` or the number of indices that you have if you are using `gl.drawElements()`.

Line loops

A line loop is drawn using `gl.LINE_LOOP` and works very similar to a line strip. The only difference is that for a line loop there is one line drawn from the last to the first vertex specified. At the lower right in [Figure 3-2](#) lines are drawn between (V_0, V_1) , (V_1, V_2) , (V_2, V_3) , (V_3, V_4) , (V_4, V_5) , and (V_5, V_0) . For a line loop the relation between the number of drawn lines and the number of vertices is given by this formula:

$$\text{Number of drawn lines} = \text{count}$$

where *count* is the number of vertices that you have when you are using `gl.drawArrays()` or the number of indices that you have if you are using `gl.drawElements()`.

Point Sprites

The last primitive that you can use in WebGL is the point sprite, which is rendered by specifying `gl.POINTS`. When you render point sprites, one point sprite is rendered for each coordinate that you specify in your vertex array. When you use a point sprite, you should also set the size of the point sprite in the vertex shader. You do this by writing the size in pixels to the built-in special variable `gl_PointSize`.

The supported size for a point sprite is hardware-dependent and the value that you write to `gl_PointSize` is clamped to the supported point size range. The maximum point size supported must be at least one. In the following example of a vertex shader, the point size is set to have a diameter of 5 pixels:

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPos;

    void main() {
        gl_Position = vec4(aVertexPos, 1.0);

        gl_PointSize = 5.0;
    }

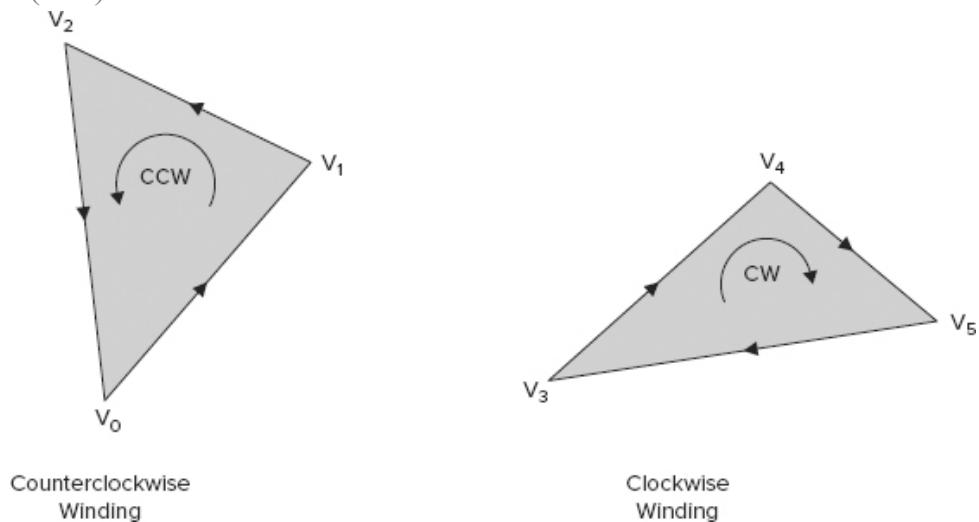
</script>
```

Point sprites are used most often in WebGL to render particle effects. Particle effects are often used to create realistic natural phenomena in real time. Examples of such phenomena are explosions, fire, smoke, and dust. As you can probably guess from these examples, particle effects are often used in 3D graphics games. You will not learn about particle effects in this book, but if you are interested in them, a web search will provide you with a lot of interesting results that will get you started.

Understanding the Importance of Winding Order

An important property for a triangle in WebGL is called the *winding order*. The winding order for a triangle can be either counterclockwise (CCW) or clockwise (CW). Counterclockwise winding order occurs when the vertices build up the triangle in counterclockwise order, as shown on the left in [Figure 3-3](#). Clockwise winding order occurs when the vertices build up the triangle in clockwise order, as shown on the right in [Figure 3-3](#).

FIGURE 3-3: The winding order for a triangle is either counterclockwise (CCW) or clockwise (CW).



The winding order is important because it is used to decide whether or not the triangles face the viewer. Triangles facing the viewer are called *front-facing*, and triangles that are not facing the viewer are called *back-facing*. In many cases there is no need for WebGL to rasterize triangles that are back-facing. For example, if you have a scene with objects whose back the viewer cannot see, you can tell WebGL to cull the faces that cannot be seen. You can easily do this by calling the following three methods:

```
gl.frontFace(gl.CCW);  
gl.enable(gl.CULL_FACE);  
gl.cullFace(gl.BACK);
```

The first method tells WebGL that triangles with CCW winding are front-facing. This is actually the default used if you don't call `gl.frontFace()` at all. The second method enables the culling for faces. Culling for faces is disabled by default, so you need to call `gl.enable(gl.CULL_FACE)` to enable it. The third method tells WebGL to cull the back-facing triangles, which is also the default if you don't call `gl.cullFace()` at all.

You can use these methods in a few different ways. For example, you can tell WebGL that it is the triangles with CW winding that are front-facing by calling using the following code:

```
gl.frontFace(gl.CW);
```

You can also specify that you instead want to cull the front-facing triangles by calling using the following code:

```
gl.cullFace(gl.FRONT);
```



If you have a scene that consists of objects whose back side the user cannot see, it is a good idea to enable back-face culling. This can increase the performance of your WebGL application since the GPU will not need to rasterize triangles that are not visible.

WebGL's Drawing Methods

In WebGL, there are three methods that you can use to update the drawing buffer:

- `gl.drawArrays()`
- `gl.drawElements()`
- `gl.clear()`

However when you draw your geometry, you must use one of the first two methods. The third method, `gl.clear()`, can only be used to set all the pixels to the color that has been previously specified using the method `gl.clearColor()`. You have already used `gl.drawArrays()`, so now it's time to look at it more in detail. After that, you will learn about `gl.drawElements()`.

The `gl.drawArrays()` Method

The method `gl.drawArrays()` draws the primitives that you specify as the first argument to the method, based on the vertex data in the enabled `WebGLBuffer` objects that are bound to the `gl.ARRAY_BUFFER` target.

This means that before you can call `gl.drawArrays()`, you must do the following:

- Create a `WebGLBuffer` object with `gl.createBuffer()`.
- Bind the `WebGLBuffer` object to the target `gl.ARRAY_BUFFER` using `gl.bindBuffer()`.
- Load vertex data into the buffer using `gl.bufferData()`.
- Enable the generic vertex attribute array using `gl.enableVertexAttribArray()`.
- Connect the attribute in the vertex shader with correct data in the `WebGLBuffer` object by calling `gl.vertexAttribPointer()`.

The prototype for `gl.drawArrays()` is as follows:

```
void drawArrays(GLenum mode, GLint first, GLsizei count)
```

This is what the arguments mean:

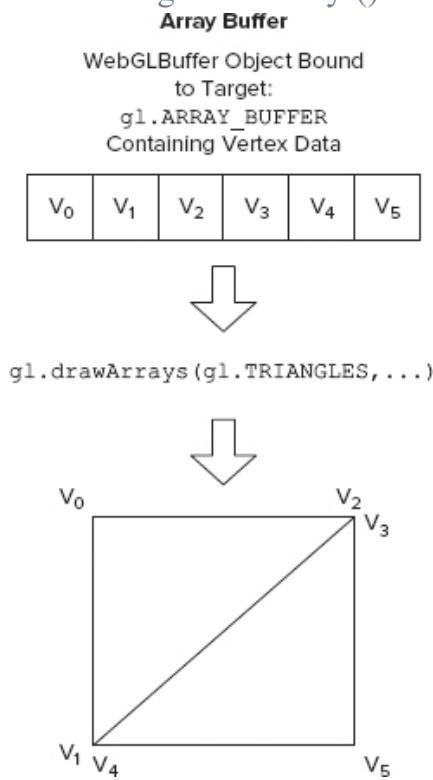
- `mode` specifies the primitive you want to render. It can contain one of the following values:

- `gl.POINTS`
- `gl.LINES`

- `gl.LINE_LOOP`
- `gl.LINE_STRIP`
- `gl.TRIANGLES`
- `gl.TRIANGLE_STRIP`
- `gl.TRIANGLE_FAN`
- `first` specifies which index in the array of vertex data should be used as the first index.
- `count` specifies how many vertices should be used.

To sum up, the method specifies which primitives should be drawn with the `mode` argument. It uses `count` number of consecutive vertices, with the first one taken from the index given by `first`. [Figure 3-4](#) shows a conceptual view of how `gl.drawArrays()` works.

[FIGURE 3-4](#): Conceptual view of how `gl.drawArrays()` works



The way `gl.drawArrays()` is designed makes it mandatory to have the vertices for the primitives that should be rendered in correct order. The method is simple and fast if you don't have any shared vertices. However, if you have an object that consists of a mesh of triangles representing the object's surface, each vertex is often shared by several triangles in the mesh. In this case it might be better to use the method `gl.drawElements()`.

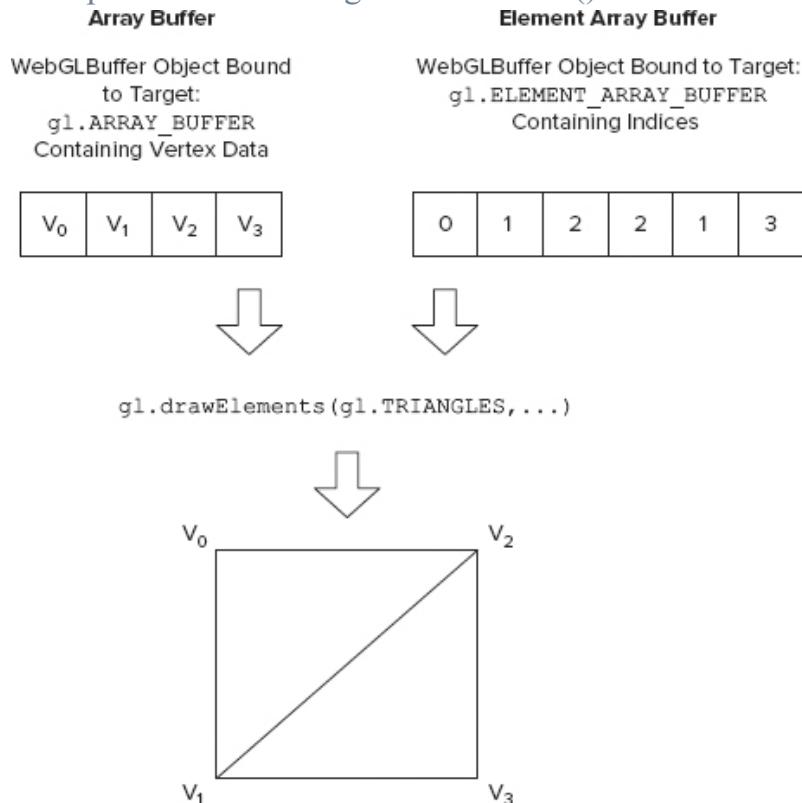
The `gl.drawElements()` Method

You have already seen how the primitives `gl.TRIANGLE_STRIP` and `gl.TRIANGLE_FAN` can increase the reuse of vertices compared to the primitive `gl.TRIANGLES` if you have many shared vertices.

The method `gl.drawElements()` is sometimes referred to as *indexed drawing* and you could see it as a way to increase the reuse of the vertices even further. You saw previously that `gl.drawArrays()` directly uses one or several *array buffers* (`WebGLBuffer` objects bound to the target `gl.ARRAY_BUFFER`) containing your vertex data in the correct order. The method `gl.drawElements()` also uses the array buffers that contain the vertex data, but in addition, it uses an *element array buffer* (`WebGLBuffer` object that is bound to the `gl.ELEMENT_ARRAY_BUFFER` target). This element array buffer contains the indices into the array buffer with vertex data.

This means that the vertex data can be in any order in the array buffer since it is the indices in the element array buffer object that decide the order in which `gl.drawElements()` uses the vertices. (However, as you will see later in this chapter, it is best from a performance point of view if the vertices are read as sequentially as possible.) In addition, if you want to share vertices, this is easy to achieve by letting several items in the element array buffer point to the same index in the array buffer. [Figure 3-5](#) shows an overview of how `gl.drawElements()` works.

FIGURE 3-5: Conceptual view of how `gl.drawElements()` works



Before you can call `gl.drawElements()`, you need to set up the array buffers in the same way as was listed for `gl.drawArrays()` earlier in this chapter. In addition, you

need to set up the element array buffer by doing the following:

- Create a `WebGLBuffer` object using `gl.createBuffer()`.
- Bind the `WebGLBuffer` object to the target `gl.ELEMENT_ARRAY_BUFFER` using `gl.bindBuffer()`.
- Load indices that decide the order in which the vertex data is used into the buffer using `gl.bufferData()`.

The prototype for `gl.drawElements()` is as follows:

```
void drawElements(GLenum mode, GLsizei count, GLenum type, GLintptr offset)
```

The arguments are described here:

- `mode` specifies the primitive you want to render. It can contain the same values as for `gl.drawArrays()`:
 - `gl.POINTS`
 - `gl.LINES`
 - `gl.LINE_LOOP`
 - `gl.LINE_STRIP`
 - `gl.TRIANGLES`
 - `gl.TRIANGLE_STRIP`
 - `gl.TRIANGLE_FAN`
- `count` specifies how many indices you have in the buffer bound to the `gl.ELEMENT_ARRAY_BUFFER` target.
- `type` specifies the type for the element indices that are stored in the buffer bound to `gl.ELEMENT_ARRAY_BUFFER`. The types you can specify are either `gl.UNSIGNED_BYTE` or `gl.UNSIGNED_SHORT`.
- `offset` specifies the offset into the buffer bound to `gl.ELEMENT_ARRAY_BUFFER` where the indices start.



Normally it's a good thing to reuse vertices for your shapes. But you should note that just because your models have triangles that have the same vertex positions, you do not necessarily want to share other vertex data such as colors and normals.

If you take a simple example like a cube, there are actually only eight unique vertex positions that define the cube. But if you want to specify a different color for each face of the cube and you use `gl.TRIANGLE_STRIP` or `gl.TRIANGLE_FAN`, you will need to specify four vertices for each of the six faces. That is a total of 24 vertices. If you use `gl.TRIANGLES`, you need six vertices for each of the six faces, which total 36 vertices.

Degenerate Triangles

From a performance point of view, you should always try to have as few calls as possible to `gl.drawArrays()` or `gl.drawElements()`. It is more efficient to have one call to

`gl.drawArrays()` or `gl.drawElements()` with an array that contains 200 triangles than to have 100 draw calls that each draws two triangles.

If you are using independent triangles (i.e., the primitive `gl.TRIANGLES`), this is straightforward to achieve. However, if you are using `gl.TRIANGLE_STRIP`, it is less obvious how you can combine different strips when there is a discontinuity between the strips.

The solution to create this discontinuity or jump between strips is to insert extra indices (or vertices if you are using `gl.drawArrays()`) that result in *degenerate triangles*. A degenerate triangle has at least two indices (or vertices) that are the same, and therefore the triangle has zero area. These triangles are easily detected by the GPU and destroyed.



It is possible to use degenerate triangles with both `gl.drawElements()` and `gl.drawArrays()`. However, using degenerate triangles with `gl.drawArrays()` requires that you duplicate vertex data. This is more costly, from both a memory and performance perspective, than to duplicate indices in the element array buffer.

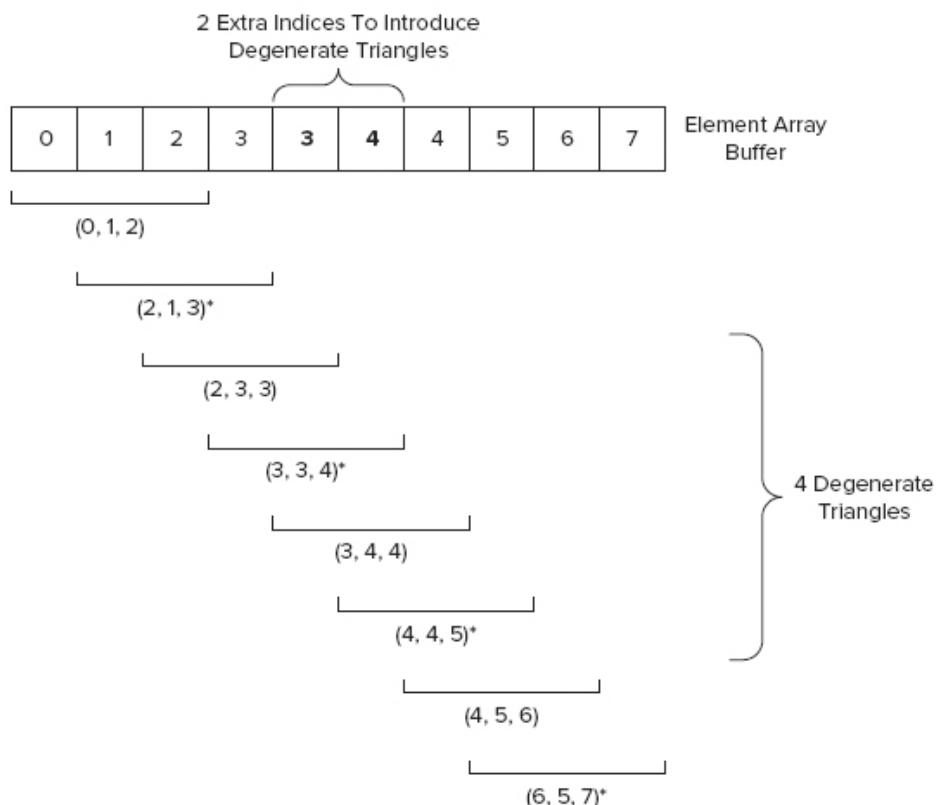
This is because vertex data occupies more memory than the element indices. In addition, duplicating several vertices is bad since the post-transform vertex cache in the GPU can usually only cache indexed vertices. This means that the duplicated vertices need to be transformed by the vertex shader every time they appear in a triangle strip.

The number of extra indices that are needed to connect two triangle strips depends on the number of indices that are used for the first strip. This is because the winding order of the triangles matters. Assuming that you want to keep the same winding order for the first and the second strip, you have two cases to consider:

- The first strip consists of an even number of triangles. To connect the second strip, you need to add two extra indices.
- The first strip consists of an odd number of triangles. To connect the second strip, you need to add three extra indices if you want to keep the winding order.

An example of the first bullet (i.e., you have a first strip that contains an even number of triangles) is shown in [Figure 3-6](#). The first strip consists of the two triangles (V_0, V_1, V_2) and (V_2, V_1, V_3) , which corresponds to the element indices $(0, 1, 2, 3)$ in the element array buffer. The second strip consists of the two triangles (V_4, V_5, V_6) and (V_6, V_5, V_7) , which corresponds to the element indices $(4, 5, 6, 7)$ at the end of the element array buffer.

FIGURE 3-6: An illustration of how two triangle strips are connected with degenerate triangles when the first strip consists of an even number of triangles



To connect the two triangle strips and create the needed degenerate triangles, two extra indices are added between the two strips. These added indices are the index **3** and the index **4** in boldface type. The rule here is to duplicate the last index of the first strip and the first index of the last strip.

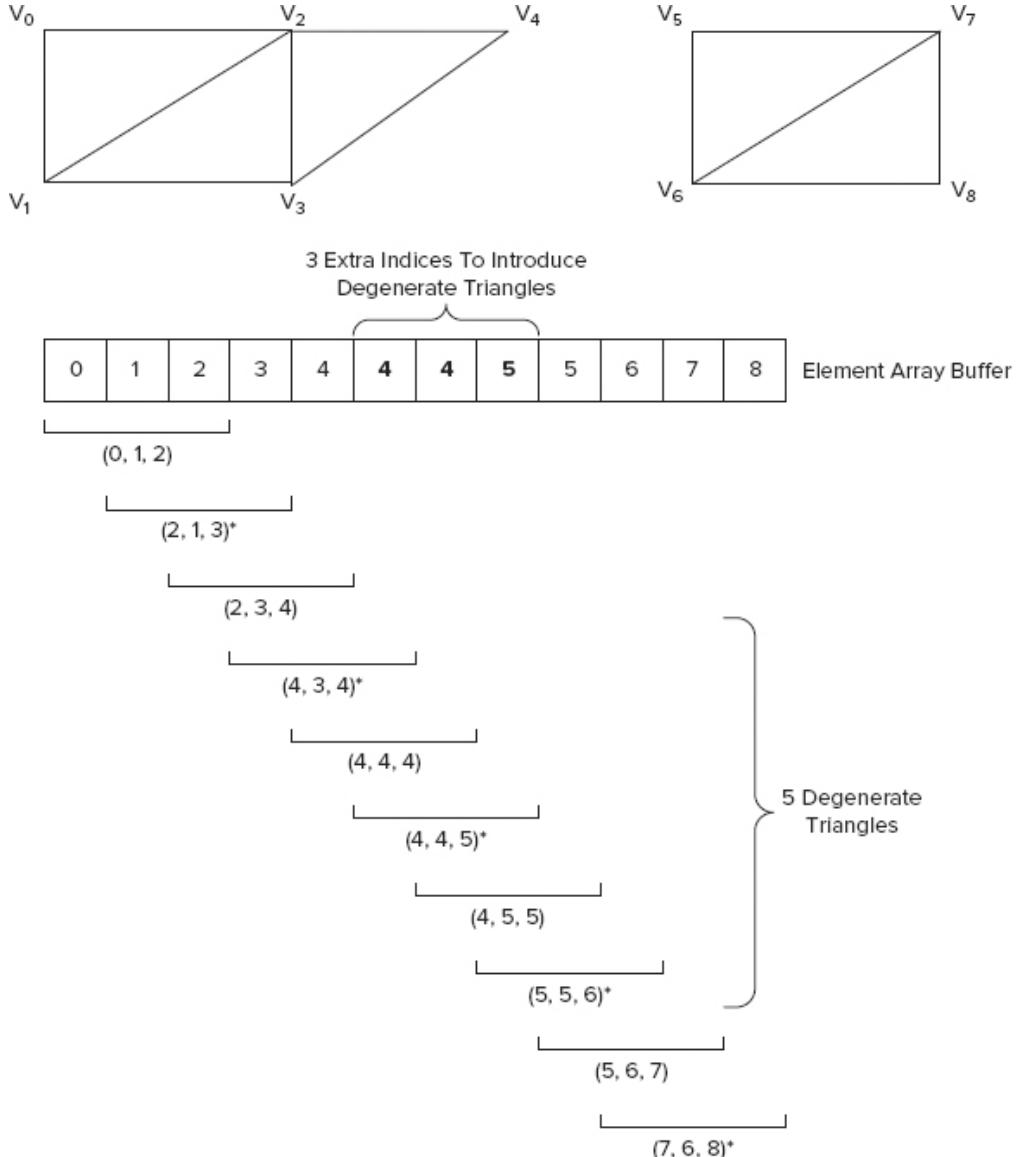
Below the element array buffer in [Figure 3-6](#), you can see how the indices will be interpreted as triangles in the triangle strip. Every second triangle has a star (*) after its parenthesis to indicate that the triangle is built up by reversing the first two indices in the triplet. This is exactly the same thing you learned in the section about triangle strips earlier in this chapter.

First you have the two triangles of the first triangle strip with index (0, 1, 2) and (2, 1, 3), then you have the four degenerate triangles that will be discarded by the GPU, and finally you have the two triangles of the last strip with indices (4, 5, 6) and (6, 5, 7).

[Figure 3-7](#) shows the corresponding illustration in the case when the first strip consists of an odd number of triangles. In this example, the first strip consists of the three triangles (V_0, V_1, V_2), (V_2, V_1, V_3), and (V_2, V_3, V_4), which corresponds to the element indices (0, 1,

2, 3, 4) in the element array buffer. The second strip again consists of the two triangles (V_5, V_6, V_7) and (V_7, V_8, V_8), which corresponds to the element indices (5, 6, 7, 8) at the end of the element array buffer.

FIGURE 3-7: An Illustration of how two triangle strips are connected with degenerate triangles when the first strip consists of an odd number of triangles



If you want to keep the winding order between the strips in this case, you can add an extra index in addition to the two indices you added for the case when the first strip had an even number of triangles. In total, this means that you add the three indices **4**, **4**, and **5** in boldface type in the element array buffer.

Below the element array buffer shown in [Figure 3-7](#), you can see how the indices will be interpreted as triangles in the triangle strip. First you have the three triangles of the first triangle strip with index (0, 1, 2), (2, 1, 3), and (2, 3, 4). Then you have the five

degenerate triangles that will be removed by the GPU. Finally, you have the two triangles of the last strip with index (5, 6, 7) and (7, 6, 8).

TYPED ARRAYS

In the complete examples of WebGL provided thus far, `Float32Array` has been used to upload the vertices through the WebGL API to a buffer using code similar to the following lines:

```
vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
var triangleVertices = [
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
gl.STATIC_DRAW);
```

In this section, you will learn what a `Float32Array` really is and why it is used.

In programming languages such as C and C++, it is not uncommon to handle binary data, and there is good support for manipulation of binary data in these languages. In JavaScript, however, it has traditionally been less common to handle binary data, and so the same support has not been built into the language. The method used by developers to manipulate binary data if needed is to represent the data as a string and then use the JavaScript method `charCodeAt()`, together with the bit manipulation operators (such as `&`, `|`, `<<`, and `>>`).

The method `charCodeAt()` returns the Unicode of a character at the index you specify as argument. Here is an example of how you can obtain a specific byte from a certain position in your data that is represented as a string:

```
var oneByte = str.charCodeAt(index) & 0xFF;
```

Using combinations of this technique, it is possible to read binary data as, for example, 32-bit integers or floats. But it is quite costly, and for WebGL, when speed is often critical, this is not a viable solution. Instead, the JavaScript typed array was introduced, and it provides a much more efficient way to handle binary data.

Buffer and View

To handle binary data, the typed array specification has a concept of a buffer and one or several views of the buffer. The buffer is a fixed-length binary data buffer that is represented by the type `ArrayBuffer`. For example, you can create an 8-byte buffer using the following call:

```
var buffer = new ArrayBuffer(8);
```

After this call, you will have a buffer with 8 bytes. However, it is not possible to manipulate the data in the buffer directly. To do this, you need to have a view of the `ArrayBuffer`. There are several different views that you can create of an `ArrayBuffer`, and the `Float32Array` that you have already seen is one such view. You can create a `Float32Array` from the `ArrayBuffer` with the name `buffer` using the following code:

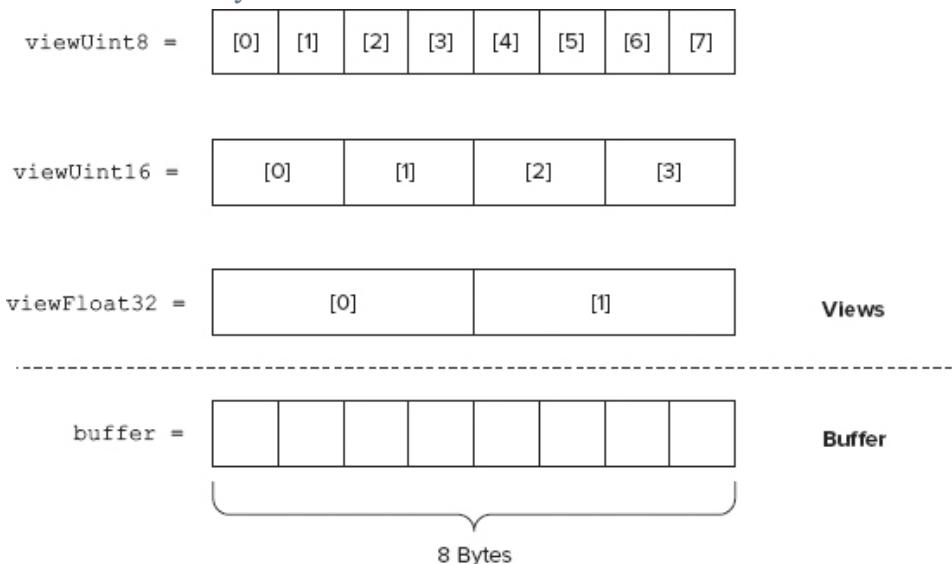
```
var viewFloat32 = new Float32Array(buffer);
```

If you want, you can then create additional views to the same buffer. For example, you can create a `Uint16Array` and a `Uint8Array` as two additional views to the same buffer:

```
var viewUint16 = new Uint16Array(buffer);
var viewUint8 = new Uint8Array(buffer);
```

After you have created an `ArrayBuffer` and one `Float32Array`, one `Uint16Array`, and one `Uint8Array` view based on this buffer, you have a layout that looks like [Figure 3-8](#). You can see how the different indices in the different views can be used to access different bytes in the `ArrayBuffer`. For example, `viewFloat32[0]` refers to bytes 0 to 3 in the `ArrayBuffer` interpreted as a 32-bit floating point number, and `viewUint16[2]` refers to bytes 4 and 5 interpreted as an unsigned 16-bit integer.

FIGURE 3-8: The structure when you have created one 8-byte `ArrayBuffer` and three different views to the `ArrayBuffer`



Supported View Types

You have already seen examples of three kinds of typed array views. [Table 3-2](#) shows the supported view types together with their sizes in bytes.

TABLE 3-2: Different Kinds of Typed Array Views

VIEW TYPE	DESCRIPTION	ELEMENT SIZE IN BYTES
<code>Uint8Array</code>	8-bit unsigned integer	1 byte
<code>Int8Array</code>	8-bit signed integer	1 byte

VIEW TYPE	DESCRIPTION	ELEMENT SIZE IN BYTES
Uint16Array	16-bit unsigned integer	2 bytes
Int16Array	16-bit signed integer	2 bytes
Uint32Array	32-bit unsigned integer	4 bytes
Int32Array	32-bit signed integer	4 bytes
Float32Array	32-bit IEEE floating point	4 bytes
Float64Array	64-bit IEEE floating point	8 bytes

All the view types have the same constructors, properties, constants, and methods. The typed array specification uses the generic term *TypedArray* to refer to one of the view types listed in [Table 3-2](#).

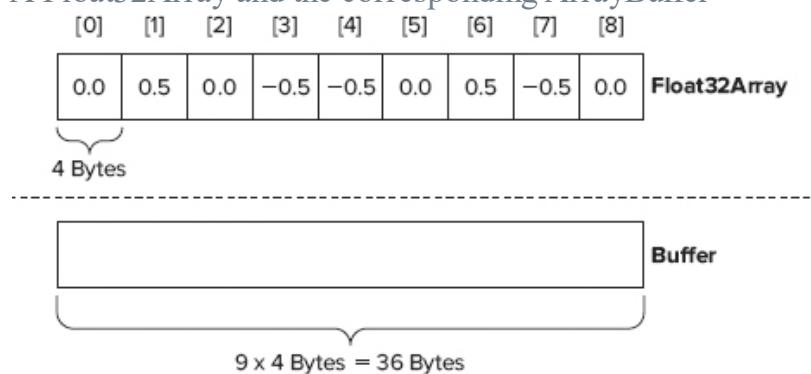
This section will use the `Float32Array` as an example, but you have the same members for the other view types. In the previous section, which explained buffers and views, the example code first created a buffer and then explicitly created the view afterwards. However, if you remember the code that was used to upload the vertices through the WebGL API, it looked like this:

```
vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
var triangleVertices = [
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
    gl.STATIC_DRAW);
```

This example uses a constructor for the `Float32Array` that takes a JavaScript Array as argument.

This constructor creates a new `ArrayBuffer` with enough space to hold the data in the JavaScript Array `triangleVertices`. But it also directly creates a `Float32Array` view that is bound to the buffer. The vertices in the `triangleVertices` are uploaded into the `ArrayBuffer`. [Figure 3-9](#) shows an illustration of the `Float32Array` and the corresponding `ArrayBuffer` that is created and sent into `gl.bufferData()` with the snippet of source code shown previously.

FIGURE 3-9: A `Float32Array` and the corresponding `ArrayBuffer`



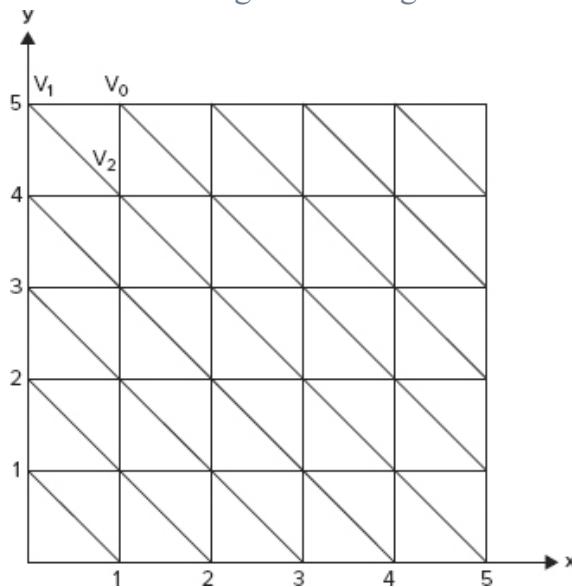
EXPLORING DIFFERENT WAYS TO DRAW

Now that you have learned a bit about the different primitives (e.g., `gl.TRIANGLES` and `gl.TRIANGLE_STRIP`) and the two methods, `gl.drawArrays()` and `gl.drawElements()`, it is time to make a comparison of the alternatives you have when you are drawing your geometry.

The most common primitives that you will draw with WebGL are probably triangles that you will draw using either `gl.drawArrays()` or `gl.drawElements()`. Generally, you can use `gl.TRIANGLES` or `gl.TRIANGLE_STRIP`. The way the primitive `gl.TRIANGLE_FAN` is specified makes it less general when you want to draw a generic mesh of triangles, so it's not included in this comparison.

Assume that you want to draw the mesh that consists of the 50 triangles in [Figure 3-10](#). What would it mean to do this with the following combination of methods and primitives?

FIGURE 3-10: A simple mesh consisting of 50 triangles



- `gl.drawArrays()` and `gl.TRIANGLES`
- `gl.drawArrays()` and `gl.TRIANGLE_STRIP`
- `gl.drawElements()` and `gl.TRIANGLES`
- `gl.drawElements()` and `gl.TRIANGLE_STRIP`

By going through which array buffers and element array buffers would be needed and how much memory would be needed for the data in these buffers, you will take another look at the information presented so far in this chapter and you will have a chance to get a deeper understanding of the topic.

Note that the primary goal with this comparison is not necessarily to find the best option from a performance point of view. For example, when the mesh in [Figure 3-10](#) is

rendered, it's done in an order that is not necessarily the most efficient. The ten triangles in the first column are processed first, followed by the ten triangles in the second column, and so on.

gl.drawArrays() and gl.TRIANGLES

You can probably say that using `gl.drawArrays()` and `gl.TRIANGLES` is the simplest option. Since `gl.drawArrays()` is used, you don't need an element array buffer. The primitive `gl.TRIANGLES` uses three vertices for every triangle. The total number of vertices needed is given by the following formula:

$$\begin{aligned}\text{Number of needed vertices} &= 3 \times \text{Number of Triangles} \\ &= 3 \times 50 = 150 \text{ vertices}\end{aligned}$$

So you need an array buffer with 150 vertices. If you only consider the vertex position and you use a `Float32Array` for the positions, the memory required to store the vertex data is given by the following formula:

$$\text{Required Memory} = 150 \text{ vertices} \times 3 \times 4 \frac{\text{bytes}}{\text{vertices}} = 1800 \text{ bytes}$$

So there are 150 vertices multiplied by 3 (since each vertex position has an *x*, *y*, and *z* coordinate) and then multiplied by 4 since each element of a `Float32Array` has a size of 4 bytes. This gives you a result of 1800 bytes.

The following code snippet shows how the buffer is set up and how the drawing is done with this option. You can see that several vertices must be duplicated. For example, V_3 is in exactly the same position as V_2 , and V_4 is in the same position as V_1 .

```
function setupBuffers() {  
  
    meshVertexPositionBuffer = gl.createBuffer();  
    gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);  
  
    var meshVertexPositions = [  
        1.0, 5.0, 0.0, //v0  
        0.0, 5.0, 0.0, //v1  
        1.0, 4.0, 0.0, //v2  
  
        1.0, 4.0, 0.0, //v3 = v2  
        0.0, 5.0, 0.0, //v4 = v1  
        0.0, 4.0, 0.0, //v5  
  
        1.0, 4.0, 0.0, //v6 = v3 = v2  
        0.0, 4.0, 0.0, //v7 = v5  
        1.0, 3.0, 0.0, //v8  
  
        1.0, 3.0, 0.0, //v9 = v8  
        0.0, 4.0, 0.0, //v10 = v7 = v5  
        0.0, 3.0, 0.0, //v11  
  
        ...  
    ]
```

```

    5.0,  0.0,  0.0, //v148
    4.0,  1.0,  0.0, //v149
    4.0,  0.0,  0.0  //v150

};

        gl.bufferData(gl.ARRAY_BUFFER,
Float32Array(meshVertexPositions),
gl.STATIC_DRAW);
meshVertexPositionBuffer.itemSize = 3;
meshVertexPositionBuffer.numberOfItems = 150;

gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

}

...
function draw() {
...
gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0,
meshVertexPositionBuffer.numberOfItems);
}

```

gl.drawArrays() and gl.TRIANGLE_STRIP

In this case the method `gl.drawArrays()` is also used, so no element array buffer is needed here, either. However, since the primitive `gl.TRIANGLE_STRIP` is used, fewer vertices per triangle are needed compared to when independent triangles are used. Going back to the formula presented in the section about triangle strips earlier in this chapter:

Number of drawn triangles = $count - 2$

where $count$ is the number of vertices that you have when you are using `gl.drawArrays()`. This means that the number of vertices that are needed for 50 triangles are given by the following formula:

$$\text{Number of needed vertices} = \text{Number of drawn triangles} + 2 = 50 + 2 = 52$$

So you could use an array buffer with 52 vertices. However, since the primitive `gl.TRIANGLE_STRIP` is used, it would in this case require you to do five calls to `gl.drawArrays()` to draw the mesh of triangles. From a performance point of view, you

typically want to do as few calls to `gl.drawArrays()` as possible. The solution is to use degenerate triangles. You need to do a jump at the end of columns 1, 2, 3, and 4. Each jump requires two extra vertices, which means that eight extra vertices are needed for the degenerate triangles. If you add these eight vertices to the 52 vertices you calculated earlier, you get a total of 60 vertices. With 60 vertices, you can draw the complete mesh with the primitive `gl.TRIANGLE_STRIP` and only one call to `gl.drawArrays()`.

You can calculate the memory required for these 60 vertices using the following formula:

$$\text{Required Memory} = 60 \text{ vertex} \times 3 \times 4 \frac{\text{bytes}}{\text{vertex}} = 720 \text{ bytes}$$

There are 60 vertices multiplied by 3 (since you have an x , y , and z coordinate) and then multiplied by 4 since each element of a `Float32Array` has a size of 4 bytes. This gives you a result of 720 bytes. As you can see, using `gl.TRIANGLE_STRIP` instead of `gl.TRIANGLES` can save some memory, even for a relatively small mesh like this one.

Here are some selected snippets of the code needed to draw the triangles with this option:

```
function setupBuffers() {

    meshVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

    var meshVertexPositions = [
        1.0, 5.0, 0.0, //v0
        0.0, 5.0, 0.0, //v1
        1.0, 4.0, 0.0, //v2
        0.0, 4.0, 0.0, //v3
        1.0, 3.0, 0.0, //v4
        0.0, 3.0, 0.0, //v5
        1.0, 2.0, 0.0, //v6
        0.0, 2.0, 0.0, //v7
        1.0, 1.0, 0.0, //v8
        0.0, 1.0, 0.0, //v9
        1.0, 0.0, 0.0, //v10
        0.0, 0.0, 0.0, //v11

        // The 2 vertices below create the jump
        // from column 1 to column 2 of the mesh
        0.0, 0.0, 0.0, //v12, degenerate
        2.0, 5.0, 0.0, //v13 degenerate

        2.0, 5.0, 0.0, //v14
        1.0, 5.0, 0.0, //v15
        2.0, 4.0, 0.0, //v16

        ...
        4.0, 1.0, 0.0, //v58
        5.0, 0.0, 0.0, //v59
    ]
}
```

```

        4.0,  0.0,  0.0  //v60
    ];
                gl.bufferData(gl.ARRAY_BUFFER,
new
Float32Array(meshVertexPositions),
gl.STATIC_DRAW);
meshVertexPositionBuffer.itemSize = 3;
meshVertexPositionBuffer.numberOfItems = 60;

gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

}

...
function draw() {
...
gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLE_STRIP, 0,
meshVertexPositionBuffer.numberOfItems);

}

```

gl.drawElements() and gl.TRIANGLES

When you are using `gl.drawElements()`, you need both an array buffer with the vertices and an element array buffer with indices. Starting with the array buffer, there are actually only 36 unique vertex positions needed for the mesh. The memory required to store these 36 vertices in the array buffer is given by the following formula:

$$\text{Required Memory for array buffer} = 36 \text{ vertices} \times 3 \times 4 \frac{\text{bytes}}{\text{vertex}} = 432 \text{ bytes}$$

To this you should add the memory required for the indices in the element array buffer. Since `gl.TRIANGLES` is used, three indices are needed for each triangle. This gives the formula:

$$\text{Number of needed indices} = 3 \times \text{Number of drawn triangles} = 3 \times 50 = 150 \text{ indices}$$

If the indices are stored in an element array buffer of type `Uint16Array`, each element of the array has a size of 2 bytes. This gives the required memory:

$$\text{Required Memory for element array buffer} = 150 \text{ indices} \times 2 \frac{\text{bytes}}{\text{index}} = 300 \text{ bytes}$$

You calculate the total memory needed by adding the memory for the array buffer, which was 432 bytes, to the 300 bytes required for the indices in the element array buffer. This means that the total memory required is 732 bytes.

The following snippet shows you what the code looks like:

```
function setupBuffers() {
    meshVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

    var meshVertexPositions = [
        1.0, 5.0, 0.0, //v0
        0.0, 5.0, 0.0, //v1
        1.0, 4.0, 0.0, //v2
        0.0, 4.0, 0.0, //v3
        1.0, 3.0, 0.0, //v4
        0.0, 3.0, 0.0, //v5
        1.0, 2.0, 0.0, //v6
        0.0, 2.0, 0.0, //v7
        1.0, 1.0, 0.0, //v8
        0.0, 1.0, 0.0, //v9
        1.0, 0.0, 0.0, //v10
        0.0, 0.0, 0.0, //v11

        ...
        4.0, 1.0, 0.0, //v34
        5.0, 0.0, 0.0, //v35
        4.0, 0.0, 0.0 //v36
    ];

    gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(meshVertexPositions),
    gl.STATIC_DRAW);
    meshVertexPositionBuffer.itemSize = 3;
    meshVertexPositionBuffer.numberOfItems = 36;

    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    meshIndexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

    var meshIndex = [
        0, 1, 2,
        2, 1, 3,
        2, 3, 4,
        4, 3, 5,
        4, 5, 6,
        6, 5, 7,
        ...
        35, 34, 36
    ];

    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(meshIndex),
```

```

    gl.STATIC_DRAW);
meshIndexBuffer.itemSize = 1;
meshIndexBuffer.numberOfItems = 150;
}

...
function draw() {
...
gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
    meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

gl.drawElements(gl.TRIANGLES,
    meshIndexBuffer.numberOfItems,
    gl.UNSIGNED_SHORT, 0);
}

```

gl.drawElements() and gl.TRIANGLE_STRIP

Since you are also using `gl.drawElements()` in this case, you need both an array buffer with the vertices and an element array buffer with indices. The content of the array buffer will look exactly the same as in the previous example when you used `gl.drawElements()` together with `gl.TRIANGLES`. There are still 36 unique vertex positions for the mesh. Each vertex position is specified by an *x*, *y* and *z* coordinate where each component occupies 4 bytes. The required memory is given by:

$$\text{Required Memory for array buffer} = 36 \text{ vertices} \times 3 \times 4 \frac{\text{bytes}}{\text{vertex}} = 432 \text{ bytes}$$

The difference comes with the indices in the element array buffer. Here you again use the connection between the number of triangles and number of vertices:

$$\text{Number of drawn triangles} = \text{count} - 2$$

where *count* is the number of indices that you have when you are using `gl.drawElements()`. This means that the number of indices that are needed for 50 triangles is given by the following formula:

$$\text{Number of needed indices} = \text{Number of drawn Triangles} + 2 = 50 + 2 = 52 \text{ indices}$$

But since you want to have one call to `gl.drawElements()`, you use degenerate triangles and you need eight extra indices to create the jump between the columns. This adds up to 60 indices in the element array buffer. If the element array buffer consists of a `Uint16Array`, this means that the required memory for the element array buffer is as follows:

$$\text{Required Memory for element array buffer} = 60 \text{ indices} \times 2 \frac{\text{bytes}}{\text{index}} = 120 \text{ bytes}$$

The total memory needed in this case is the 432 bytes needed for the array buffer plus the 120 bytes needed for the element array buffer, which totals 552 bytes.

The code for this example is as follows:

```

function setupBuffers() {
    meshVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

    var meshVertexPositions = [
        1.0, 5.0, 0.0, //v0
        0.0, 5.0, 0.0, //v1
        1.0, 4.0, 0.0, //v2
        0.0, 4.0, 0.0, //v3
        1.0, 3.0, 0.0, //v4
        0.0, 3.0, 0.0, //v5
        1.0, 2.0, 0.0, //v6
        0.0, 2.0, 0.0, //v7
        1.0, 1.0, 0.0, //v8
        0.0, 1.0, 0.0, //v9
        1.0, 0.0, 0.0, //v10
        0.0, 0.0, 0.0, //v11
        // start of column 2
        2.0, 5.0, 0.0, //v12
        1.0, 5.0, 0.0, //v13
        2.0, 4.0, 0.0, //v14

        ...
        4.0, 1.0, 0.0, //v34
        5.0, 0.0, 0.0, //v35
        4.0, 0.0, 0.0 //v36
    ];

    gl.bufferData(gl.ARRAY_BUFFER, new
        Float32Array(meshVertexPositions),
        gl.STATIC_DRAW);
    meshVertexPositionBuffer.itemSize = 3;
    meshVertexPositionBuffer.numberOfItems = 36;

    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    meshIndexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

    var meshIndex = [
        0, 1, 2,
        3, 4, 5,
        6, 7, 8,
        9, 10, 11,
    ];
}

```

```

    11, 12,    // indices for degenerate triangles
    12, 13, 14,
    ...
    34, 35, 36
];

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(meshIndex),
  gl.STATIC_DRAW);
meshIndexBuffer.itemSize = 1;
meshIndexBuffer.numberOfItems = 60;
}

...
function draw() {
  ...
  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
    meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

  gl.drawElements(gl.TRIANGLE_STRIP,
    meshIndexBuffer.numberOfItems,
    gl.UNSIGNED_SHORT, 0);
}

```

Conclusions of the Comparison

Now that you have gone through all four options, [Table 3-3](#) gives you an overview of the results from the previous sections. You can see that the method `gl.drawElements()` together with `gl.TRIANGLE_STRIP` is a combination that can save a lot of memory compared to the method `gl.drawArrays()` and the primitive `gl.TRIANGLES` when you have many shared vertices.

[TABLE 3-3:](#) Summary of Different Drawing Options

METHOD AND PRIMITIVE	TOTAL MEMORY USED	ELEMENT ARRAY BUFFER NEEDED	DEGENERATE TRIANGLES NEEDED
<code>gl.drawArrays()</code> and <code>gl.TRIANGLES</code>	1800 bytes	No	No
<code>gl.drawArrays()</code> and <code>gl.TRIANGLE_STRIP</code>	720 bytes	No	Yes
<code>gl.drawElements()</code> and <code>gl.TRIANGLES</code>	732 bytes	Yes	No
<code>gl.drawElements()</code> and <code>gl.TRIANGLE_STRIP</code>	552 bytes	Yes	Yes

You should also remember that this was only a small mesh with 50 triangles and that only the vertex position was considered. Since only the vertex position was considered, each element in the array buffer has a size of 12 bytes. If you had a bigger mesh and the vertex data also included normals and texture coordinates, the amount of memory you could save by using `gl.drawElements()` and `gl.TRIANGLE_STRIP` would be even greater.

It is important not to consume more memory than necessary, but this is not the only important factor from a performance point of view. The order in which you draw your triangles and how your vertex data is organized can also be important from a performance perspective. You will learn why in the following section.

Pre-Transform Vertex Cache and Post-Transform Vertex Cache

As discussed in Chapter 1, the vertices that you send in through the WebGL API need to be transformed by the vertex shader before they can be passed on in the WebGL pipeline for primitive assembly, rasterization, fragment shading, per-fragment operations, and finally end up as pixels in the drawing buffer. As you have seen, the same vertex is often used in several triangles of a mesh. This means that when you have transformed the vertex once, all subsequent transforms of the same vertex are in some way redundant work.

To address this problem, modern GPUs have a cache that is usually referred to as a *post-transform vertex cache*, which exists to avoid vertex shading being done for the same vertex multiple times. When a vertex is transformed by the vertex shader, it's put in the post-transform vertex cache. This cache is usually quite small and organized as a FIFO (first in, first out) cache, which means that it only caches the result for the most recent transformed vertices.

This means that if the vertices for the triangles are sent in random order, many vertices will miss the cache even though they might have been transformed before. As a measurement of how efficient the post-transform vertex cache is, the ACMR (average cache miss ratio) can be used. The ACMR is defined as the number of cache misses divided by the number of rendered triangles, according to the following formula:

$$\text{ACMR} = \frac{\text{Number of cache misses}}{\text{Number of rendered triangles}}$$

Obviously you want the ACMR to be as low as possible. For a large mesh, the number of triangles is approximately double the number of vertices. This means that in theory, the ACMR could be as low as 0.5 in an optimal case, where each vertex is transformed only once. In a really bad case where all three vertices of every triangle miss the cache, the ACMR will have a value of 3.0.

If the triangles of your mesh are organized so that the post-transform vertex cache is used as efficiently as possible, there can be much to gain from a performance point of view. There has been a lot of research in this area, and there are even some tools (such as Stripe, which was developed at Stony Brook University) that try to organize your vertices in an optimal order.

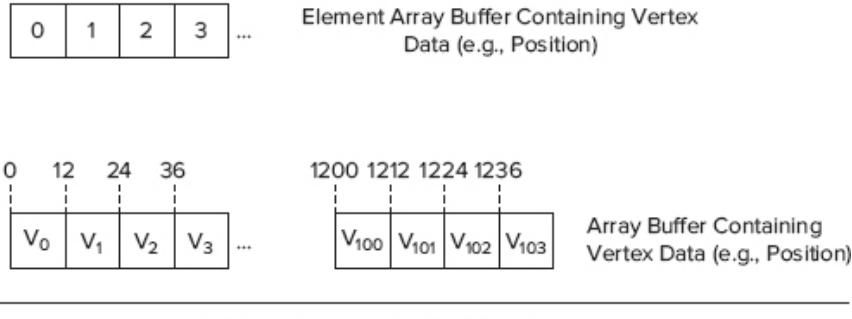
In addition to the post-transform vertex cache, the GPU normally also has some sort of *pre-transform vertex cache*. When a vertex is not found in the post-transform vertex cache, it needs to be read and transformed by the vertex shader.

When a vertex needs to be fetched, a larger consecutive block of vertex data is normally fetched into the cache at the same time. This means that even though you could have the vertex data in your vertex array buffer in any order when you use indexed drawing with `gl.drawElements()`, it is a good idea to have the vertices in the same order that they are used when possible.

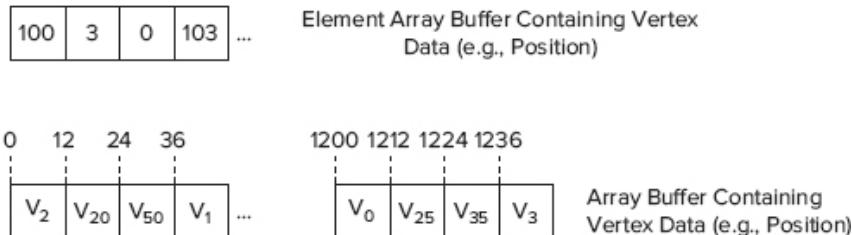
[Figure 3-11](#) shows an example of this. At the top of the figure, the vertex data in the array buffer is organized consecutively in the memory. When V_0 is read, it is likely that some of the subsequent vertices will be read into the pre-transform vertex cache as well. When V_1 , V_2 , and V_3 are subsequently needed, these are already available in the cache and do not need to be read from a slower memory.

FIGURE 3-11: At the top, the data in the array buffer is organized consecutively in memory, which is likely to result in a better hit rate in the pre-transform vertex cache. At the bottom, the data in the array buffer is not organized consecutively, so the hit rate in the pre-transform vertex cache will likely be lower.

Good for Pre-Transform Vertex Cache



Bad for Pre-Transform Vertex Cache



At the bottom of [Figure 3-11](#), the vertices are not organized consecutively and it is less likely that you will get a hit in the pre-transform vertex cache.



It is good to understand that the order in which the triangles are rendered and also the order in which you have the vertices in memory can impact your performance. But don't worry too much about trying to find an optimal order to render your triangles and arranging your vertex data until you have a performance problem and know it is due to your WebGL application being vertex bound.

This said, if you know that you have many shared vertices in your objects, a good rule of thumb is to use indexed drawing with `gl.drawElements()` instead of `gl.drawArrays()` to at least increase the chance that the post-transform vertex cache will be used. If you don't order your triangles for optimal use of the post-transform vertex cache, the primitive `gl.TRIANGLE_STRIP` will normally give you the best performance as long as you don't have to include too many extra vertices for degenerate triangles to connect different strips.

INTERLEAVING YOUR VERTEX DATA FOR IMPROVED PERFORMANCE

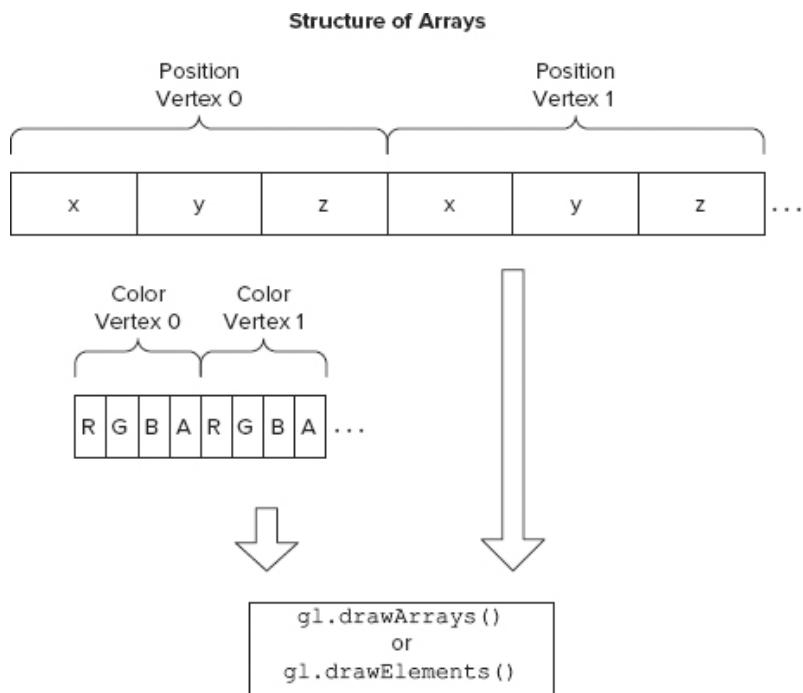
So far, most of the examples you have looked at have only included the vertex position as vertex data. However, the vertex data for a real WebGL application usually consists of more data than the position of the vertex. Data that is commonly used in addition to the position includes the vertex normal, vertex color, and texture coordinates.

When you have more than one type of vertex data, this data can mainly be structured in two ways:

- Store each type of vertex data in a separate array in a `WebGLBuffer` object. This means that you have one array with the vertex positions and another array with vertex normals, and so on. This is often referred to as *structure of arrays*.
- Store all vertex data in one single array in a `WebGLBuffer` object. This means that you interleave the different types of vertex data in the same array. This is often referred to as *array of structures*.

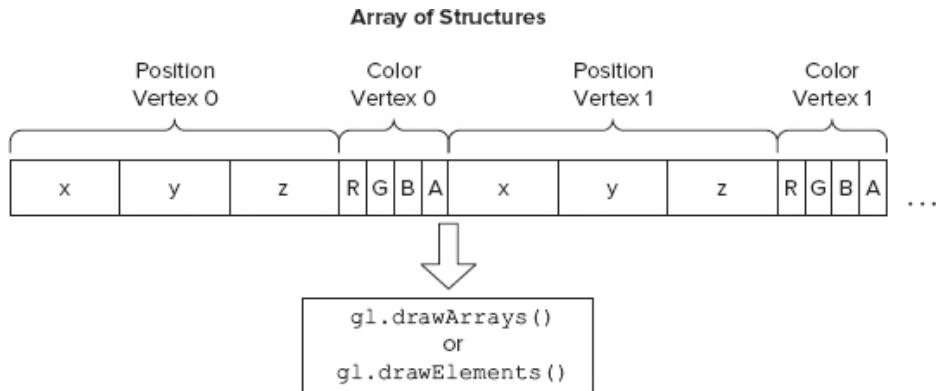
The first option (structure of arrays) is often the most straightforward alternative when it comes to setting up the buffers and loading the data into the buffers. Each type of vertex data has its own vertex array. [Figure 3-12](#) shows an overview of what the structure of arrays alternative looks like when you have vertex data that consists of a position represented by an x , y , and z -coordinate that are all floats and colors in RGBA format, where each of the four color components are represented by an unsigned byte.

FIGURE 3-12: A conceptual diagram of how the non-interleaved vertex arrays (structure of arrays) work



Using the array of structures alternative often requires that you do a little more work when you set up the buffers and load the data. [Figure 3-13](#) shows how the position data and the color data are interleaved in the same vertex array.

FIGURE 3-13: A conceptual diagram of how the interleaved vertex array (array of structures) works



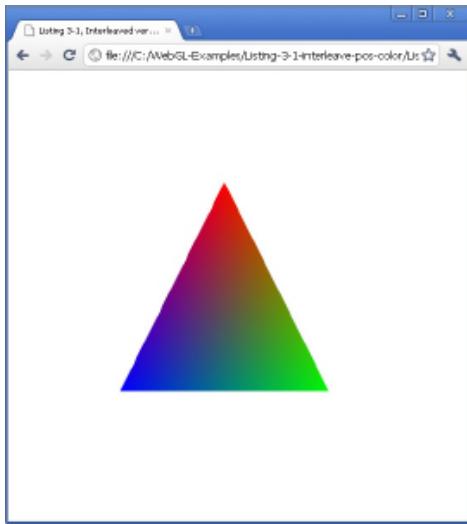
From a performance point of view, the preferred way to organize your vertex data is to interleave the data as an array of structures. This is because you get better memory locality for your vertex data. It is likely that when a vertex shader needs the position for a certain vertex, it will also need the normal and texture coordinates for the same vertex at almost the same time. By keeping these coordinates close together in the memory, it is more likely that when one is read into the pre-transform vertex cache, the others will also be read in the same block and will be available in the pre-transform vertex cache when they are needed by the vertex shader.

Using an Array of Structures

Using an array of structures is not actually difficult once you understand it. But it can be a bit tricky the first time, and therefore this section has a complete example of how to use it so that you can easily load it into your browser and try it out yourself.

The geometry in [Listing 3-1](#) is kept as simple as possible so you can focus on the interleaved vertex array. When you load the source code into your browser, you see a colored triangle that looks similar to [Figure 3-14](#). Since this book is not full-color, you can't see the colors, so you should load the source code into the browser to see it.

FIGURE 3-14: When you load the source code for [Listing 3-1](#) into your browser, you will see a colored triangle.



You are already familiar with most of the code. You should remember from Chapter 2 how the JavaScript function `startup()` is triggered when the page is loaded. You should also know how to set up the WebGL context and how to load, compile, and link your shaders.

Take a look at the source code for the vertex shader and the fragment shader. What is interesting here when you try to understand the interleaved vertex array is that there are two attribute variables, one for the position and one for the color. These two attribute variables get their input from a single vertex array that contains the interleaved vertex data.

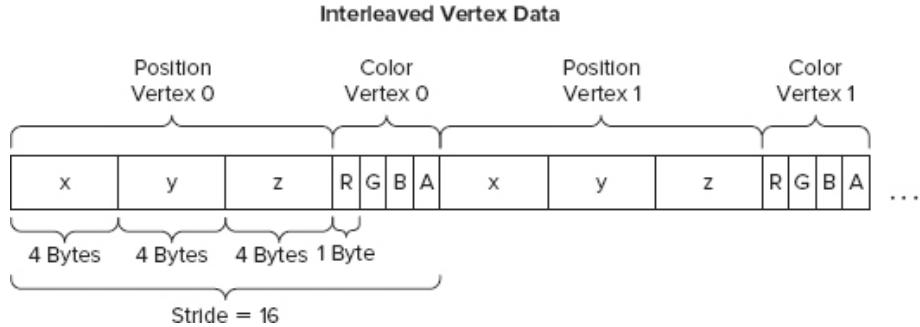
Setting Up the Buffer for Interleaved Vertex Data

Now if you look further down in [Listing 3-1](#), you will find the function `setupBuffers()`, which is one of the more interesting parts in this example. This is where the buffers are set up and loaded with data.

First a `WebGLBuffer` object is created with `gl.createBuffer()` and bound with `gl.bindBuffer()`. Then the vertex data is specified in a typical JavaScript array. The vertex data is already interleaved here, so it first has three position elements (x, y, z), then four color elements (r, g, b, a), and then three position elements again, and so on. This example first has the data in a JavaScript array, but that is mainly to give you a better overview of the vertex data when you read the code. The JavaScript array is then read and the values are loaded into a typed array, so the vertex values could just as well have been loaded from an external file or some other resource.

Previously when you looked at how to load vertex data into a `WebGLBuffer` object, you used a JavaScript array directly as input when you created a typed array. In this example, the interleaved vertex data is of a different type so it's slightly more complicated now. The `WebGLBuffer` object that you prepare now will have the structure that is shown in [Figure 3-15](#).

FIGURE 3-15: The format of the interleaved vertex array used in [Listing 3-1](#)



First there are three position elements (x, y, z), where each element occupies 4 bytes. This means that these three elements occupy 12 bytes in total and this is also the offset from the start of the buffer to the first color component. Each of the four color components (r, g, b, a) occupies 1 byte. The total size of the data for one vertex consists of the three position elements and the four color components, and this is calculated in [Listing 3-1](#) with the following snippet of code:

```
// Calculate how many bytes that are needed for one vertex element
// that consists of (x,y,z) + (r,g,b,a)
var vertexSizeInBytes = 3 * Float32Array.BYTES_PER_ELEMENT +
                      4 * Uint8Array.BYTES_PER_ELEMENT;

var vertexSizeInFloats = vertexSizeInBytes /
                        Float32Array.BYTES_PER_ELEMENT;
```

You can easily see that the `vertexSizeInBytes` will be 16 and the `vertexSizeInFloats` will be four. After these are calculated, an `ArrayBuffer` is allocated that will hold the actual vertex data. To access the position elements, a `Float32Array` is mapped to the `ArrayBuffer`, and to access the color elements, a `Uint8Array` is mapped to it.

```
// Allocate the buffer
var buffer = new ArrayBuffer(nbrOfVertices * vertexSizeInBytes);

// Map the buffer to a Float32Array view to access the position
var positionView = new Float32Array(buffer);

// Map the same buffer to a Uint8Array to access the color
var colorView = new Uint8Array(buffer);
```

Then there is a loop that reads the values from the JavaScript array and populates the `ArrayBuffer`.

```
// Populate the ArrayBuffer from the JavaScript Array
var positionOffsetInFloats = 0;
var colorOffsetInBytes = 12;
var k = 0; // index to JavaScript Array
for (var i = 0; i < nbrOfVertices; i++) {
  positionView[positionOffsetInFloats] = triangleVertices[k];
  // x
  positionView[1+positionOffsetInFloats] = triangleVertices[k+1];
```

```

// Y
    positionView[2+positionOffsetInFloats] = triangleVertices[k+2];
// Z
    colorView[colorOffsetInBytes] = triangleVertices[k+3];
// R
    colorView[1+colorOffsetInBytes] = triangleVertices[k+4];
// G
    colorView[2+colorOffsetInBytes] = triangleVertices[k+5];
// B
    colorView[3+colorOffsetInBytes] = triangleVertices[k+6];
// A

    positionOffsetInFloats += vertexSizeInFloats;
    colorOffsetInBytes += vertexSizeInBytes;
    k += 7;
}

```

The variable `positionOffsetInFloats` is the location in the `Float32Array` that the `x`-coordinate is written to for each of the vertices. In the first iteration of the loop, this is zero, and then it is increased with `vertexSizeInFloats` (which is four) for each of the iterations in the loop. The variable `colorOffsetInBytes` is the location in the `Uint8Array` of the red color component. In the first iteration of the loop, it is set to 12, and then it is increased with `vertexSizeInBytes` (which is 16) for each iteration in the loop.

Drawing Based on Interleaved Vertex Data

When you have set up your buffer for the interleaved data correctly, you need to tell WebGL how it is set up and then draw based on this data.

```

// Bind the buffer containing both position and color
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);

// Describe how the positions are organized in the vertex array
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                      triangleVertexBuffer.positionSize, gl.FLOAT, false, 16,
                      0);

// Describe how colors are organized in the vertex array
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                      triangleVertexBuffer.colorSize, gl.UNSIGNED_BYTE, true, 16,
                      12);

// Draw the triangle
gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);

```

First, bind the buffer that contains your interleaved vertex data. Then you need to call `gl.vertexAttribPointer()` once for each attribute in the vertex shader that you want to connect to vertex data in your bound buffer. The first call to this method in the previous snippet tells WebGL how the position data is organized in the bound `WebGLBuffer` object. The meaning of the arguments is as follows:

- The first argument tells WebGL which generic attribute index the bound buffer should be used as input for. The index that is specified here corresponds to the `aVertexPosition` attribute in the vertex shader.
- The second argument specifies how many elements are used for the position. The position consists of x, y, and z, so `triangleVertexBuffer.positionSize` was set to 3 and this is the value that is sent in as the second argument.
- The third argument is set to `gl.FLOAT` and specifies that the position elements are floats.
- The fourth argument is the normalize flag. It specifies how data items that are not floats should be handled. The position data is of float type so whether the flag is set to `false` does not really matter for the position data.
- The fifth argument is the stride. The stride specifies the number of bytes between the start of one vertex element and the start of the next vertex element of the same type. As you can see in [Figure 3-15](#), the stride is 16 bytes in this example. In previous examples where you had vertex data that was not interleaved, the stride was set to zero.
- The sixth argument is the offset to the first element of the type of vertex data that the call specifies. Since the position is specified before the color data, the offset for the position is zero.

When you understand the first call to `gl.vertexAttribPointer()`, the second call is easy to understand. For the second call, the third argument is set to `gl.UNSIGNED_BYTE` since each color component was placed in one byte in the `ArrayBuffer`.

The fourth argument (the normalize flag) is now important since the vertex data is `gl.UNSIGNED_BYTE` instead of `gl.FLOAT`. All vertex attributes that are used by the vertex shader are single, precision floating point values. If you send in vertex data of another type, the data will be converted to floats before it is used by the vertex shader. The normalize flag specifies how this conversion is done.

Setting the normalize flag to true, as in this example, has the effect that all unsigned values are mapped to the range [0.0, 1.0] and all signed values are mapped to the range [-1.0, 1.0]. Since the color vertex data was provided as `gl.UNSIGNED_BYTE`, it is mapped to the range [0.0, 1.0] by dividing all the color values by 255.

Finally you should note that the last argument for the second call to `gl.vertexAttribPointer()`, which is the offset to the first color component, is set to 12, since this is the size of the three preceding position elements.



A common mistake that you may see in online forums about WebGL is that beginners do not use the stride argument correctly when they try to use `gl.vertexAttribPointer()` on interleaved vertex data. Instead of giving the correct stride, which is the number of bytes between the start of one vertex element and the start of the next vertex element of the same type, they set the stride argument to be the number of bytes between the end of one vertex element and the start of the next vertex element of the same type. This mistake will, of course, result in problems.



Available for
download on
[Wrox.com](#)

LISTING 3-1: Example of an Interleaved Vertex Array

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 3-1, Interleaved vertex data</title>
<meta charset="utf-8">
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec4 aVertexColor;
    varying vec4 vColor;

    void main() {
        vColor = aVertexColor;
        gl_Position = vec4(aVertexPosition, 1.0);
    }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
    precision mediump float;

    varying vec4 vColor;
    void main() {
        gl_FragColor = vColor;
    }
</script>

<script src="webgl-debug.js"></script>

<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var triangleVertexBuffer;

function createGLContext(canvas) {
    var names = ["webgl", "experimental-webgl"];
    var context = null;
    for (var i=0; i < names.length; i++) {
        try {
            context = canvas.getContext(names[i]);
        } catch(e) {}
        if (context) {
            break;
        }
    }
    if (context) {
        context.viewportWidth = canvas.width;
        context.viewportHeight = canvas.height;
    }
}</script>
```

```

    } else {
        alert("Failed to create WebGL context!");
    }
    return context;
}

function loadShaderFromDOM(id) {
    var shaderScript = document.getElementById(id);

    // If we don't find an element with the specified id
    // we do an early exit
    if (!shaderScript) {
        return null;
    }

    // Loop through the children for the found DOM element and
    // build up the shader source code as a string
    var shaderSource = "";
    var currentChild = shaderScript.firstChild;
    while (currentChild) {
        if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
            shaderSource += currentChild.textContent;
        }
        currentChild = currentChild.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
        shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
        shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
        return null;
    }

    gl.shaderSource(shader, shaderSource);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert(gl.getShaderInfoLog(shader));
        return null;
    }
    return shader;
}

function setupShaders() {
    vertexShader = loadShaderFromDOM("shader-vs");
    fragmentShader = loadShaderFromDOM("shader-fs");

    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
}

```

```

gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
}

gl.useProgram(shaderProgram);

shaderProgram.vertexPositionAttribute =
gl.getAttributeLocation(shaderProgram, "aVertexPosition");

shaderProgram.vertexColorAttribute =
gl.getAttributeLocation(shaderProgram, "aVertexColor");

// We enable vertex attrib arrays for both position and color
attribute
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
}

function setupBuffers() {

    triangleVertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
    var triangleVertices = [
        // ( x     y     z ) (r     g     b     a )
        // -----
        0.0,  0.5,  0.0,   255,   0,   0,  255, // v0
        0.5, -0.5,  0.0,    0, 250,   6,  255, // v1
       -0.5, -0.5,  0.0,    0,   0, 255,  255 // v2
    ];
    var nbrOfVertices = 3;

    // Calculate how many bytes that are needed for one vertex element
    // that consists of (x,y,z) + (r,g,b,a)
    var vertexSizeInBytes = 3 * Float32Array.BYTES_PER_ELEMENT +
                           4 * Uint8Array.BYTES_PER_ELEMENT;

    var vertexSizeInFloats      =      vertexSizeInBytes      /
Float32Array.BYTES_PER_ELEMENT;

    // Allocate the buffer
    var buffer = new ArrayBuffer(nbrOfVertices * vertexSizeInBytes);

    // Map the buffer to a Float32Array view to access the position
    var positionView = new Float32Array(buffer);

    // Map the same buffer to a Uint8Array to access the color
    var colorView = new Uint8Array(buffer);
}

```

```

// Populate the ArrayBuffer from the JavaScript Array
var positionOffsetInFloats = 0;
var colorOffsetInBytes = 12;
var k = 0; // index to JavaScript Array
for (var i = 0; i < nbrOfVertices; i++) {
    positionView[positionOffsetInFloats] = triangleVertices[k];
// x
    positionView[1+positionOffsetInFloats] = triangleVertices[k+1];
// y
    positionView[2+positionOffsetInFloats] = triangleVertices[k+2];
// z
    colorView[colorOffsetInBytes] = triangleVertices[k+3];
// R
    colorView[1+colorOffsetInBytes] = triangleVertices[k+4];
// G
    colorView[2+colorOffsetInBytes] = triangleVertices[k+5];
// B
    colorView[3+colorOffsetInBytes] = triangleVertices[k+6];
// A

    positionOffsetInFloats +=vertexSizeInFloats;
    colorOffsetInBytes +=vertexSizeInBytes;
    k +=7;
}

gl.bufferData(gl.ARRAY_BUFFER, buffer, gl.STATIC_DRAW);
triangleVertexBuffer.positionSize = 3;
triangleVertexBuffer.colorSize = 4;
triangleVertexBuffer.numberOfItems = 3;
}

function draw() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Bind the buffer containing both position and color
    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);

    // Describe how the positions are organized in the vertex array
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
        triangleVertexBuffer.positionSize, gl.FLOAT, false, 16,
0);

    // Describe how colors are organized in the vertex array
    gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
        triangleVertexBuffer.colorSize, gl.UNSIGNED_BYTE, true, 16,
12);

    // Draw the triangle
    gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);
}

```

```

}

function startup() {
    canvas = document.getElementById("myGLCanvas");
    gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
    setupShaders();
    setupBuffers();
    gl.clearColor(1.0, 1.0, 1.0, 1.0);

    draw();
}
</script>

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>
</html>

```

USING A VERTEX ARRAY OR CONSTANT VERTEX DATA

In the source code that you have looked at so far, the attributes in the vertex shader have taken input from a vertex array in a `WebGLBuffer` object. The vertex array contains data that is specific to each vertex in the geometry. To let an attribute take its input from a vertex array, you typically do the following:

- Enable the generic attribute index that corresponds to an attribute in the vertex shader with a call to `gl.enableVertexAttribArray()`.
- Create a `WebGLBuffer` object, bind it, and load the vertex data into the buffer.
- Call `gl.vertexAttribPointer()` to connect a generic attribute index that corresponds to an attribute in the vertex shader to the bound `WebGLBuffer` object.

However, if you have vertex data that is constant over all your vertices for a primitive, you don't need to copy the same data into each position of a vertex array.

Instead, you can disable the generic attribute index that corresponds to the attribute in the vertex shader you want to send constant data to. You do this by calling the method `gl.disableVertexAttribArray()`. Then you can set the data that should be constant for all your vertices. To set constant vertex data for an attribute of the type `vec4`, you call the following method:

```
gl.vertexAttrib4f(index, r, g, b, a);
```

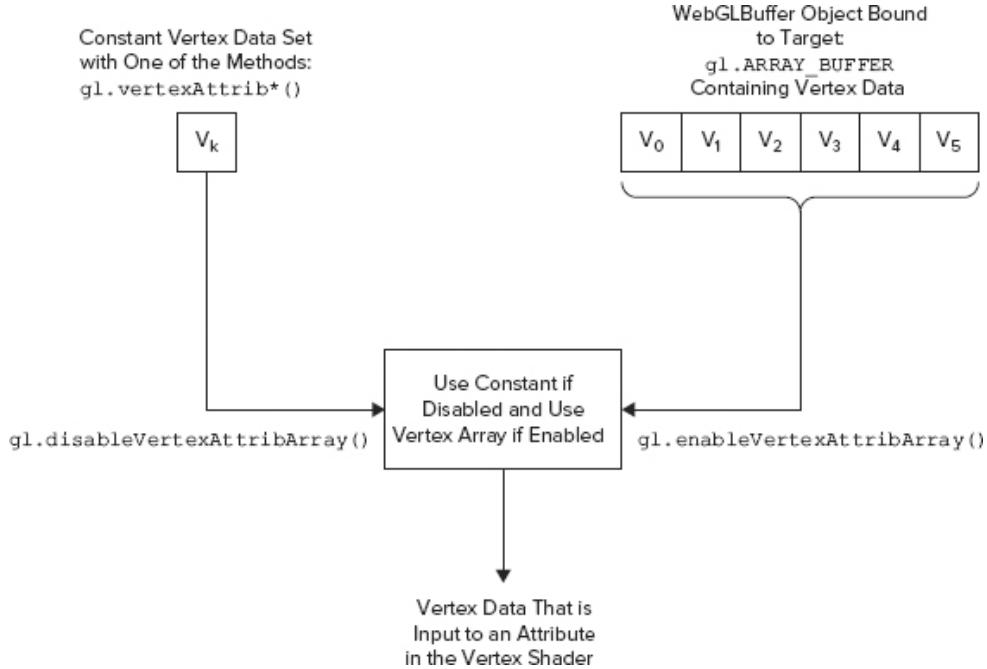
In this call, the argument `index` is the generic attribute index that you want to set your vertex data for, and the arguments `r`, `g`, `b`, and `a` could be the four color components for a

color that you want to set for all your vertices. There are corresponding methods to set three, two, and one float(s):

```
gl.vertexAttrib3f(index, x, y, z);
gl.vertexAttrib2f(index, x, y);
gl.vertexAttrib1f(index, x);
```

[Figure 3-16](#) shows an overview of how the methods `gl.enableVertexAttribArray()` and `gl.disableVertexAttribArray()` work.

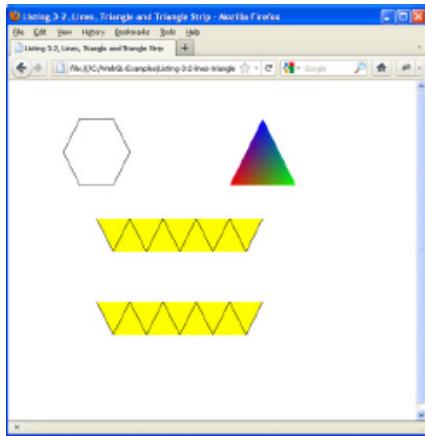
FIGURE 3-16: Conceptual diagram of how the methods `gl.enableVertexAttribArray()` and `gl.disableVertexAttribArray()` work



A LAST EXAMPLE TO WIND THINGS UP

Now that you have learned a lot about how to draw in WebGL and the different options you have for your drawing, you'll look at one final example that uses some of the new concepts introduced in this chapter. [Listing 3-2](#) contains the source code for an example that uses both `gl.drawArrays()` and `gl.drawElements()` together with line and triangle primitives to draw a hexagon, an independent triangle, and a triangle strip. If you load the source code from [Listing 3-2](#) into your browser, you should see something similar to [Figure 3-17](#). Even though you should be able to follow this discussion by looking at this figure, it might make more sense if you actually load the source code into your browser so you can see the colors.

FIGURE 3-17: WebGL geometry that corresponds to [Listing 3-2](#)



You can start by looking at the shader source code, which is fairly straightforward. You can see that the vertex shader has two attribute variables that will take their input from the WebGL API. You will see later in the code that the `aVertexPosition` will take its input from the bound and enabled `WebGLBuffer` objects, while the input for the other attribute `aVertexColor` will alternate between a bound `WebGLBuffer` object and a constant color that is sent in through the WebGL API with `gl.vertexAttrib4f()`.

The `aVertexPosition` is expanded to a `vec4` homogeneous coordinate and then assigned directly to the built-in variable `gl_Position` without any transformation.

The fragment shader takes the varying variable `vColor` and assigns it to the special built-in variable `gl_FragColor`. It is worth mentioning again that the varying variable `vColor` in the fragment shader contains a linear interpolation of the values that is sent in as `vColor` in the vertex shader. This is illustrated in this example with the independent triangle to the top right in [Figure 3-17](#). The three vertices of the triangle have the colors red, green, and blue. The pixels between the three vertices have a color that is a linear interpolation of these three colors. (To get a better feel for the coloring, load the source code into your browser.)

If you continue down in the source code to the location where the JavaScript starts, you can see some global variables:

```
var gl;
var canvas;
var shaderProgram;
var hexagonVertexBuffer;
var triangleVertexBuffer;
var triangleVertexColorBuffer;
var stripVertexBuffer;
var stripElementBuffer;
```

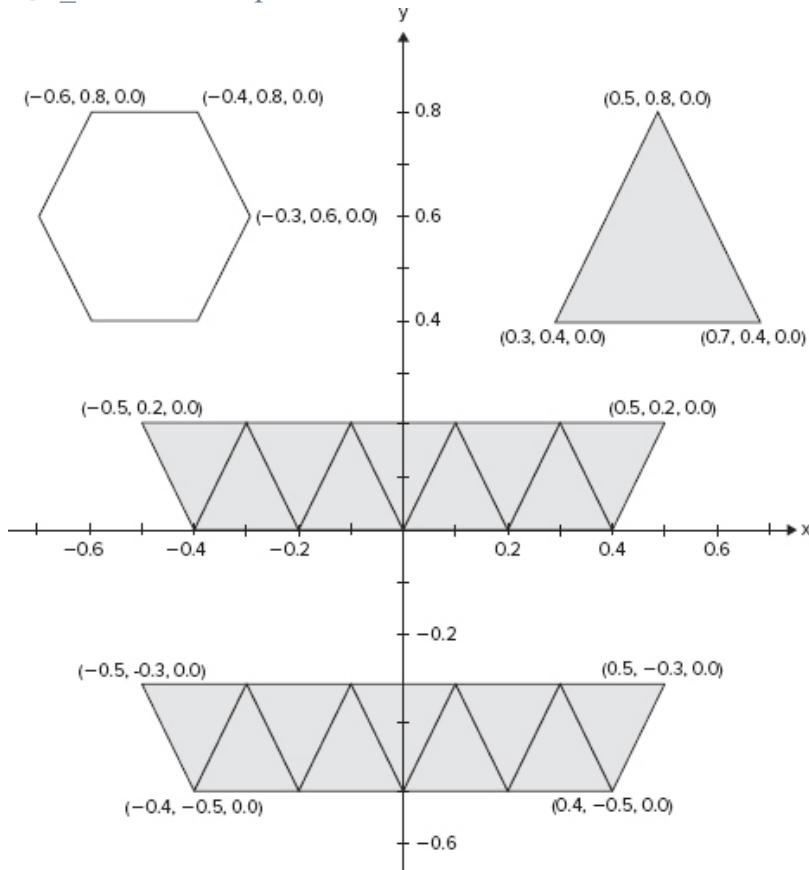
Here the last five variables will be used to hold different `WebGLBuffer` objects. The `hexagonVertexBuffer` contains the vertex positions for the hexagon. The `triangleVertexBuffer` contains the vertex positions for the independent triangle. The `triangleVertexColorBuffer` contains the vertex colors for the triangle. The `stripVertexBuffer` contains the vertex positions for the triangle strip. As you will see,

the triangle strip is drawn with indexed drawing using `gl.drawElements()`, so it needs the element array buffer `stripElementBuffer` in addition to the array buffer.

The color for the hexagon and the triangle strip is constant for all vertices, so it is set with the method `gl.vertexAttribfv()`. Therefore there is no need for buffers that contain the colors for these two shapes.

If you look further down in the code in [Listing 3-2](#), you see the next chunk of highlighted code, which consists of the creation and setup of the buffers that were just discussed. [Figure 3-18](#) shows the vertex positions for the different shapes in this example.

FIGURE 3-18: Positions of the geometry in clip coordinates, which are the coordinates that the variable `gl_Position` expects



Note that since no transformations were done in the vertex shader, the default coordinate system for WebGL is used. This default coordinate system is better known as a *clip coordinate system* or *clip space*. The coordinates are called clip coordinates and these coordinates are what `gl_Position` expects as input. In Chapter 4, you will learn more about the different coordinate systems and how you can use transformations to convert between them.

In clip coordinates, the lower-left corner of the viewport has $x = -1$ and $y = -1$ and the upper-right corner of the viewport has $x = 1$ and $y = 1$. So for your geometry to end up in

the viewport, it needs to have x - and y -coordinates within this range. Since the shapes that are used in this example are only in 2D, the z -coordinate is set to zero.

The source code that creates and loads the vertices should now be familiar to you. However, you should pay some extra attention to how the buffer for the triangle strip is created and populated with vertex data. As shown in [Figure 3-17](#), the triangle strip actually appears as two separate triangle strips that are positioned above each other without any connected geometry. However, you can see in the source code that it is actually only one `WebGLBuffer` object that contains all the vertices for the triangle strip. Degenerate triangles are then used to create the jump from the first strip to the second strip.

Since the first strip consists of an odd number of triangles, three extra indices are inserted to create the jump from the first to the second strip. These extra indices are the 10, 10, and 11 that you can also see on the second line with indices below:

```
stripElementBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);
var indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               10, 10, 11, // 3 extra indices for the degenerate
               triangles
               11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21];

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
gl.STATIC_DRAW);

stripElementBuffer.numberOfItems = 25;
```

The next interesting part of this example is the drawing. First the hexagon is drawn with the code that is also shown in the following snippet:

```
// Draw the hexagon
// We disable the vertex attrib array since we want to use a
// constant color for all vertices in the hexagon
gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0,
0.0, 1.0);

gl.bindBuffer(gl.ARRAY_BUFFER, hexagonVertexBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
hexagonVertexBuffer.itemSize, gl.FLOAT,
false, 0, 0);

gl.drawArrays(gl.LINE_STRIP, 0, hexagonVertexBuffer.numberOfItems);
```

Before the actual call to `gl.drawArrays()`, there is code to set up things correctly. First there is a call to `gl.disableVertexAttribArray()` to use a constant vertex color that is set to opaque black with the call to `gl.vertexAttrib4f()` instead of using colors from a vertex array. Then the buffer is bound to the `gl.ARRAY_BUFFER` target and `gl.vertexAttribPointer` is called to connect the `aVertexPosition` in the vertex

shader to the `hexagonVertexBuffer` that contains the vertex positions for the hexagon. Finally, `gl.drawArrays()` can be called to draw the hexagon.

The next shape to draw is the independent triangle. It is drawn with the following code:

```
// Draw the independent triangle
// For the triangle we want to use per-vertex color so
// we enable the vertexColorAttribute again
gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);

gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
    triangleVertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
    triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);
```

An interesting note regarding this snippet is that now `gl.enableVertexAttribArray()` must be called to enable the vertex attribute array since the triangle strip takes its color from a vertex array and not a constant vertex attribute.

The next shape to draw is the triangle strip:

```
// draw triangle-strip
    // We disable the vertex attribute array for the
vertexColorAttribute
    // and use a constant color again.
    gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
    gl.bindBuffer(gl.ARRAY_BUFFER, stripVertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
        stripVertexBuffer.itemSize, gl.FLOAT, false,
0, 0);

    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 1.0, 1.0,
0.0, 1.0);
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);

                                gl.drawElements(gl.TRIANGLE_STRIP,
stripElementBuffer.numberOfItems,
    gl.UNSIGNED_SHORT, 0);
    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0,
0.0, 1.0);
```

Here the vertex attribute array is disabled again. It is worth noting that both an array buffer and an element array buffer must be bound since indexed drawing with `gl.drawElements()` is used.

Finally, there are two calls to draw the helper lines that show up between the triangles of the triangle strip:

```
// Draw help lines to easier see the triangles
// that build up the triangle-strip
```

```

gl.drawArrays(gl.LINE_STRIP, 0, 11);
gl.drawArrays(gl.LINE_STRIP, 11, 11);

```

The first call draws a line strip that starts at vertex zero and uses 11 vertices in total. The second call draws a strip that starts at vertex 11 and uses 11 vertices in total.

The end of [Listing 3-2](#) includes the `startup()` function that, among other things, contains the three calls to enable back-face culling:

```

<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 3-2, Lines, Triangle and Triangle Strip</title>
<meta charset="utf-8">
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec4 aVertexColor;
    varying vec4 vColor;

    void main() {
        vColor = aVertexColor;
        gl_Position = vec4(aVertexPosition, 1.0);
    }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
    precision mediump float;

    varying vec4 vColor;
    void main() {
        gl_FragColor = vColor;
    }
</script>

<script src="webgl-debug.js"></script>

<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var hexagonVertexBuffer;
var triangleVertexBuffer;
var triangleVertexColorBuffer;
var stripVertexBuffer;
var stripElementBuffer;

function createGLContext(canvas) {
    var names = ["webgl", "experimental-webgl"];
    var context = null;
    for (var i=0; i < names.length; i++) {
        try {
            context = canvas.getContext(names[i]);
        }
    }
    if (!context) {
        console.error("Failed to create GL context");
    }
    return context;
}
</script>

```

```

        } catch(e) {}
        if (context) {
            break;
        }
    }
    if (context) {
        context.viewportWidth = canvas.width;
        context.viewportHeight = canvas.height;
    } else {
        alert("Failed to create WebGL context!");
    }
    return context;
}

function loadShaderFromDOM(id) {
    var shaderScript = document.getElementById(id);

    // If we don't find an element with the specified id
    // we do an early exit
    if (!shaderScript) {
        return null;
    }

    // Loop through the children for the found DOM element and
    // build up the shader source code as a string
    var shaderSource = "";
    var currentChild = shaderScript.firstChild;
    while (currentChild) {
        if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
            shaderSource += currentChild.textContent;
        }
        currentChild = currentChild.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
        shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
        shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
        return null;
    }

    gl.shaderSource(shader, shaderSource);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert(gl.getShaderInfoLog(shader));
        return null;
    }
    return shader;
}

```

```

function setupShaders() {
    vertexShader = loadShaderFromDOM("shader-vs");
    fragmentShader = loadShaderFromDOM("shader-fs");

    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Failed to setup shaders");
    }

    gl.useProgram(shaderProgram);

        shaderProgram.vertexPositionAttribute      =
    gl.getAttribLocation(shaderProgram,
        "aVertexPosition");
        shaderProgram.vertexColorAttribute       =
    gl.getAttribLocation(shaderProgram,
        "aVertexColor");

    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}

function setupBuffers() {
    hexagonVertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, hexagonVertexBuffer);
    var hexagonVertices = [
        -0.3,  0.6,  0.0, //v0
        -0.4,  0.8,  0.0, //v1
        -0.6,  0.8,  0.0, //v2
        -0.7,  0.6,  0.0, //v3
        -0.6,  0.4,  0.0, //v4
        -0.4,  0.4,  0.0, //v5
        -0.3,  0.6,  0.0, //v6
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(hexagonVertices),
        gl.STATIC_DRAW);
    hexagonVertexBuffer.itemSize = 3;
    hexagonVertexBuffer.numberOfItems = 7;

    triangleVertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
    var triangleVertices = [
        0.3,  0.4,  0.0, //v0
        0.7,  0.4,  0.0, //v1
        0.5,  0.8,  0.0, //v2
    ];
}

```

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
  gl.STATIC_DRAW);
triangleVertexBuffer.itemSize = 3;
triangleVertexBuffer.numberOfItems = 3;

triangleVertexColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
var colors = [
  1.0, 0.0, 0.0, 1.0, //v0
  0.0, 1.0, 0.0, 1.0, //v1
  0.0, 0.0, 1.0, 1.0 //v2
];
gl.bufferData(gl.ARRAY_BUFFER,      new      Float32Array(colors),
gl.STATIC_DRAW);
triangleVertexColorBuffer.itemSize = 4;
triangleVertexColorBuffer.numberOfItems = 3;

stripVertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, stripVertexBuffer);
var stripVertices = [
  -0.5, 0.2, 0.0, //v0
  -0.4, 0.0, 0.0, //v1
  -0.3, 0.2, 0.0, //v2
  -0.2, 0.0, 0.0, //v3
  -0.1, 0.2, 0.0, //v4
  0.0, 0.0, 0.0, //v5
  0.1, 0.2, 0.0, //v6
  0.2, 0.0, 0.0, //v7
  0.3, 0.2, 0.0, //v8
  0.4, 0.0, 0.0, //v9
  0.5, 0.2, 0.0, //v10
  // start second strip
  -0.5, -0.3, 0.0, //v11
  -0.4, -0.5, 0.0, //v12
  -0.3, -0.3, 0.0, //v13
  -0.2, -0.5, 0.0, //v14
  -0.1, -0.3, 0.0, //v15
  0.0, -0.5, 0.0, //v16
  0.1, -0.3, 0.0, //v17
  0.2, -0.5, 0.0, //v18
  0.3, -0.3, 0.0, //v19
  0.4, -0.5, 0.0, //v20
  0.5, -0.3, 0.0 //v21
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(stripVertices),
  gl.STATIC_DRAW);
stripVertexBuffer.itemSize = 3;
```

```

stripVertexBuffer.numberOfItems = 22;

stripElementBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);
var indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               10, 10, 11, // 3 extra indices for the
               degenerate triangles
               11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21];

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
              gl.STATIC_DRAW);

stripElementBuffer.numberOfItems = 25;
}

function draw() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Draw the hexagon
    // We disable the vertex attrib array since we want to use a
    // constant color for all vertices in the hexagon
    gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0,
                      0.0, 1.0);

    gl.bindBuffer(gl.ARRAY_BUFFER, hexagonVertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                          hexagonVertexBuffer.itemSize, gl.FLOAT, false, 0,
                          0);

    gl.drawArrays(gl.LINE_STRIP, 0, hexagonVertexBuffer.numberOfItems);

    // Draw the independent triangle
    // For the triangle we want to use per-vertex color so
    // we enable the vertexColorAttribute again
    gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);

    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                          triangleVertexBuffer.itemSize, gl.FLOAT, false, 0,
                          0);

    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                          triangleVertexColorBuffer.itemSize, gl.FLOAT,
                          false, 0, 0);

    gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);
}

```

```

// draw triangle-strip
    // We disable the vertex attribute array for the
vertexColorAttribute
    // and use a constant color again.
    gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
    gl.bindBuffer(gl.ARRAY_BUFFER, stripVertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                          stripVertexBuffer.itemSize, gl.FLOAT, false, 0,
0);

    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 1.0, 1.0,
0.0, 1.0);
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);

    gl.drawElements(gl.TRIANGLE_STRIP,
stripElementBuffer.numberOfItems,
gl.UNSIGNED_SHORT, 0);
    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0,
0.0, 1.0);

// Draw help lines to easier see the triangles
// that build up the triangle-strip
    gl.drawArrays(gl.LINE_STRIP, 0, 11);
    gl.drawArrays(gl.LINE_STRIP, 11, 11);
}

function startup() {
    canvas = document.getElementById("myGLCanvas");
    gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
    setupShaders();
    setupBuffers();
    gl.clearColor(1.0, 1.0, 1.0, 1.0);

    gl.frontFace(gl.CCW);
    gl.enable(gl.CULL_FACE);
    gl.cullFace(gl.BACK);

    draw();
}
</script>

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>

</html>
    gl.frontFace(gl.CCW);
    gl.enable(gl.CULL_FACE);
    gl.cullFace(gl.BACK);

```

As the code is written in this example, no faces are actually culled since all triangles have counterclockwise winding and `gl.cullFace()` specifies that only back-facing triangles should be culled. However, having back-face culling enabled is a way to ensure that you include the extra indices for the degenerate triangles correctly. Remember, depending on how these extra indices are included, you can change the winding of the second strip, which would then be culled.



Available for
download on
[Wrox.com](#)

LISTING 3-2: Example that draws a hexagon, a triangle, and a triangle strip

Some Things to Experiment With

Now that you have gone through all the code and a description of the code, you should try to experiment with it and make some changes. Try the following:

- Change the indices for the degenerate triangles so you only use the two extra indices 10 and 11. You also have to update the `stripElementBuffer.numberOfItems`. Note how the lower strip disappears. What is the reason for this?
- Keep the change from above (i.e., only have the two extra indices 10 and 11 to create degenerate triangles) and change the argument to `gl.frontFace()` from `gl.CCW` to `gl.CW`. What happens and why?
- Remove the line that enables culling and see what happens.
- Remove the two lines that call `gl.drawArrays()` to draw the help lines between the triangles in the triangle strip just to see how the triangle strip would normally look.

SUMMARY

In this chapter you learned about the options that are available when you want to draw in WebGL. You learned about the seven different primitives that you can use when you draw with the two drawing methods `gl.drawArrays()` and `gl.drawElements()`:

- `gl.TRIANGLES`
- `gl.TRIANGLE_STRIP`
- `gl.TRIANGLE_FAN`
- `gl.LINES`
- `gl.LINE_STRIP`
- `gl.LINE_LOOP`
- `gl.POINTS`

You now understand that the different triangle primitives are the most basic building blocks that are used for drawing with WebGL. You looked at how the winding of triangles

is used to determine whether or not the triangles are facing the user. You also learned how to cull triangles that are not facing the user.

This chapter also gave you an understanding of what typed arrays are and how they are used in WebGL to send the vertex data through the WebGL API and then further to the GPU.

Finally, you should have developed the fundamental knowledge necessary to select which primitive and which method you should use for your future WebGL applications. You learned how the pre-transform vertex cache and the post-transform vertex cache could affect the performance of your WebGL applications.

Chapter 4

Compact JavaScript Libraries and Transformations

WHAT'S IN THIS CHAPTER?

- How to use a JavaScript library to handle vector and matrix mathematics in JavaScript
- How to use transformations to position and orient your objects in a 3D scene
- An overview of the different kinds of transformations that are important for WebGL
- Learn about the different coordinate systems or spaces that your vertices pass on their way to the screen
- How to use a transformation matrix stack to let different objects have different transformations
- How to position and orient an imaginary camera in WebGL

In Chapter 1 you learned some fundamental linear algebra that is useful for WebGL and, in particular, you learned the mathematics behind some important transformations. In this chapter you will learn how these transformations are used in WebGL. First, however, you will learn about three JavaScript libraries that you can use to handle matrix and vector mathematics in JavaScript; these will be useful for your WebGL applications.

You will also learn about the different coordinate systems or spaces that the vertices go through when they are sent in through the WebGL API and traverse through the WebGL pipeline towards the display. After reading this important chapter, you will understand how to position and orient the objects in the scene in your 3D space by applying different transformations.

WORKING WITH MATRICES AND VECTORS IN JAVASCRIPT

JavaScript does not have any built-in support to handle matrices or vectors. The JavaScript data type that is closest to a matrix or a vector is the built-in `Array` object. You can either write your own functionality to handle matrices and vectors in JavaScript or you can use one of the open source libraries that are available. The following are three libraries that you can find on the web and that are quite commonly used in WebGL applications:

- Sylvester
- WebGL-mjs
- glMatrix

These three libraries are all relatively small and have similar functionality. Another thing they have in common is that they do not abstract away the WebGL API. There are several other larger libraries that are written as abstraction layers on top of WebGL. This book refers to these larger libraries as WebGL frameworks.

These WebGL frameworks contain a variety of functionality, but these frameworks often provide an API where many WebGL-specific details are hidden. You have learned that when you use WebGL, there are many details that need to be handled before you can actually draw something. You need to create a shader object, load the source code into the shader object, compile the shaders, link the shaders, and so on. These tasks are often handled internally by the WebGL framework; the application does not need to worry about all those details.

Although you might appreciate this functionality in many ways, at this point you want to learn the details about WebGL, and so it is better to use one of the smaller libraries that mainly handles the matrix and vector mathematics for you and still lets you see the details of the WebGL API.

Before you start learning about the libraries, there is a practical detail you should know about. Most JavaScript libraries that are put on a web server and used in real web applications have been compressed in some way to minimize the download time. This usually includes removing whitespaces and

comments that the JavaScript engine will not need in order to execute the JavaScript.

Sometimes the JavaScript libraries exist in two versions:

- The compressed version that is supposed to be used by your application when you are finished with the development and publish it on a web server.
- The uncompressed version that is available so you can easily read the source code, debug your application, and understand how it works.

Sometimes the file that contains the uncompressed version of the library has a name that indicates this. As an example, the uncompressed version of Sylvester has the filename `sylvester.src.js`, while the file `sylvester.js` contains the compressed version of the library. But it can also be the other way around, where the compressed version of the file has a name that indicates that it is compressed or minimal. For the JavaScript library glMatrix, the file for the compressed version has the name `glMatrix-min.js` while the uncompressed version simply has the name `glMatrix.js`.

You don't have to think too much about the filenames, though. If you open the files in a text editor, it will be obvious which file contains the compressed version and which file contains the uncompressed version. In the examples in this book, the uncompressed versions of the libraries have been used so that it's easier for you to read the source code and debug the library.

Now it is time for you to get an overview of each of the three libraries. You will find the latest source code and documentation for the libraries if you do a search in your favorite search engine.

Sylvester

Sylvester was written by James Coglan and is a general vector and matrix mathematics library for JavaScript. It is not written specifically for WebGL, so it lacks some functionality that is available in WebGL-mjs and glMatrix. Instead, it is more generic and can handle matrices and vectors of arbitrary sizes. It also includes functionality to handle lines and planes in 3D, something that is not supported in the other two libraries.

You'll now take a closer look at the functionality and notation of Sylvester. To create a new vector, use the following function:

```
Vector.create(elements)
```

This creates a new vector from the argument `elements`, which is a JavaScript array. There is a short alias for this function as well:

```
$V(elements)
```

This means that if you want to create a vector with the three elements (3, 4, and 5), you simply write

```
var v = Vector.create([3, 4, 5]);
```

or with the shorter version,

```
v = $V([3, 4, 5]);
```

If you have two vectors, you can, for example, easily calculate their sum, the dot product, or the cross product between them:

```
var u = Vector.create([1, 2, 3]);
var v = Vector.create([4, 5, 6]);

var s = u.add(v);      // s = [5, 7, 9]
var d = u.dot(v);     // d = 1*4+2*5+3*6 = 32
var c = u.cross(v);   // c = [-3, 6, -3]
```

You'll now look at the notation for handling matrices. You create a new matrix as follows:

```
Matrix.create(elements)
```

In the same way as for `Vector.create()`, there is an alias for `Matrix.create()`:

```
$M(elements)
```

Here the `elements` should be a nested array. As you will see, this is one of the differences between Sylvester and the other two libraries, which use a one-dimensional array to populate a matrix. For Sylvester, the top-level array contains the rows, and each row is an array of elements. So to create a 3×2 matrix, you would write the code as follows:

```
var M = Matrix.create([[ 2, -1],    // first row
                      [-2,  1],    // second row
                      [-1,  2]]); // third row
```

Or with the shorter notation:

```
var M = $M([[ 2, -1],    // first row
            [-2,  1],    // second row
            [-1,  2]]); // third row
```

If you have two matrices (`M` and `N`), you can calculate the product `MN` between them:

```
var MN = M.multiply(N);
```

[Listing 4-1](#) shows the complete code for using some of the functionality of the Sylvester JavaScript library from an HTML file with embedded JavaScript. In addition to the methods that are already described, this example uses the method `inspect()`, which is available for vectors and matrices in Sylvester. It is used to return a string representation of an object and is very handy when you debug your code.



Available for
download on
[Wrox.com](#)

[LISTING 4-1: Vector and matrix mathematics with the Sylvester JavaScript library](#)

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Test of Sylvester JavaScript Library</title>
<meta charset="utf-8">
<script type="text/javascript" src="sylvester.src.js">
</script>
<script type="text/javascript">

function testSylvesterJsLibrary() {
    var u = Vector.create([1,2,3]);
    var v = Vector.create([4,5,6]);

    var s = u.add(v);
    alert(s.inspect()); // s = [5, 7, 9]

    var d = u.dot(v); // d = 1*4+2*5+3*6 = 32
    alert(d); // will alert 32

    var c = u.cross(v);
    alert(c.inspect()); // Will alert [-3, 6, -3]

    var M = Matrix.create([[ 2, -1], // first row
                          [-2, 1], // second row
                          [-1, 2]]); // third row

    var N = Matrix.create([[4, -3], // first row
                          [3, 5]]); // second row

    var MN = M.multiply(N);
    alert(MN.inspect()); // will alert [ 5, -11]
                         // [-5, 11]
```

```

        //          [ 2, 13]

var I = Matrix.I(4); // create identity matrix
alert(I.inspect()); // will alert [1, 0, 0, 0]
                    //           [0, 1, 0, 0]
                    //           [0, 0, 1, 0]
                    //           [0, 0, 0, 1]

}

</script>
</head>
<body onload="testSylvesterJsLibrary();">
    Simple web page to test Sylvester JavaScript library. <br />
    The page shows a couple of alerts with the result
    from vector and matrix maths. <br />
    You need to read the source code with view source or or
    similar
    to understand the results of the alerts.
</body>
</html>
```

If you use Chrome Developer Tools (or Firebug) and set a breakpoint after the matrices have been created and initialized in [Listing 4-1](#), you can see that the matrices in Sylvester are represented as nested JavaScript arrays, as shown in [Figure 4-1](#).

FIGURE 4-1: In Sylvester, matrices are represented by nested arrays

The screenshot shows the Chrome DevTools interface. On the left, the Scripts panel displays the source code for Listing 4-1.html. The code uses the Sylvester library to perform matrix operations and alert the results. On the right, the Call Stack panel shows a stack trace for a function named `testSylvesterLibrary`, which is part of the `Matrix` constructor. The stack trace includes frames for `Matrix`, `Object`, and `Function`.

```

10 // Test of Sylvester JavaScript Library
11 // http://C/WebGL-Examples/Using-4-1-sylvester/Listing-4-1.htm
12
13 Simple web page to test Sylvester JavaScript Library.
14 The page shows a couple of alerts with the result from vector and matrix math.
15 You need to read the source code with view source or similar to understand the results of the alerts.
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53

```

For example, if you take matrix MN from [Listing 4-1](#), you can see that it consists of one outer array of three elements since the matrix has three rows. Then each row is represented by an inner array of two elements since there are two columns in this matrix.



Some key points to remember about Sylvester:

- *Sylvester is not written primarily for WebGL, so if you use it for WebGL, you might want to complement it with some other small library or write a few functions of your own for some WebGL-specific functionality such as setting up the projection matrix or the view transform.*
- *Sylvester uses an object-oriented paradigm.*
- *Sylvester has support for vectors, matrices, lines, and planes.*
- *Matrices in Sylvester are represented as nested JavaScript arrays. This means that they are quite close to how matrices are normally written down in linear algebra. Conversely, the nested array needs to be flattened to a single one-dimensional array before it is sent to the vertex shader through the WebGL API.*

WebGL-mjs

WebGL-mjs was written primarily by Vladimir Vukićević, who should not just be acknowledged for this well-written library, but also for his contributions to WebGL as a technology. In contrast to Sylvester, WebGL-mjs

was written specifically with WebGL in mind; it can only handle matrices of the dimension 4×4 and vectors with three components.

Since WebGL-mjs is written specifically for WebGL, it also contains very useful methods (e.g., `makeScale()`, `makeTranslate()`, and `makeRotate()`) to perform transformations in an easy way. When you use these methods, you don't have to explicitly perform the matrix multiplications in your application. In addition to the functionality to perform transformations, it also contains some nice functionality to, for example, set up the projection matrix that defines your view frustum.

WebGL-mjs does not stuff vectors and matrices into JavaScript objects. Instead, it provides its functionality as a set of functions for performing operations on vectors and matrices, which are actually just represented as arrays in WebGL-mjs.

A new vector with three elements x , y , and z is created with the function,
`V3.$(x, y, z)`

To create a new vector with the components 1, 2, and 3, you simply write the code as follows:

```
var u = V3.$(1, 2, 3);
```

In the following code, two vectors are created and then the sum, the dot product, and the cross product between them are calculated:

```
var u = V3.$(1, 2, 3);
var v = V3.$(4, 5, 6);
var s = V3.add(u, v);    // s = [5, 7, 9]
var d = V3.dot(u, v);   // d = 1*4+2*5+3*6 = 32
var c = V3.cross(u, v); // c = [-3, 6, -3]
```

A 4×4 matrix is created with the function,

```
M4x4.$(m00, m01, m02, m03, // first column
        m04, m05, m06, m07, // second column
        m08, m09, m10, m11, // third column
        m12, m13, m14, m15) // fourth column
```

where `m00..m03` are the elements of the first column in the matrix, `m04..m07` are the elements of the second column, and so on.



*When matrix elements are added and stored column by column, it is often referred to as using **column-major order** matrices.*

Note that this is different from how matrices are normally written down in linear algebra. It is also different from Sylvester, which uses nested arrays to create a matrix.

If you have two matrices M and N , you calculate the product MN as follows:

```
var MN = M4x4.mul(M, N);
```

[Listing 4-2](#) shows a complete example that uses the WebGL-mjs library to create vectors and matrices and do some calculations based on them. As you can see in this example and from the code snippets that were presented here, the functions of WebGL-mjs always take the vectors and matrices that they operate on as arguments. Compared to Sylvester, the notation is less object oriented.



Available for
download on
[Wrox.com](#)

[LISTING 4-2: Vector and matrix mathematics with the WebGL-mjs JavaScript library](#)

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Test of WebGL-mjs JavaScript Library</title>
<meta charset="utf-8">
<script type="text/javascript" src="mjs.js"></script>
<script type="text/javascript">

function convertToString(array) {
    var stringRep = '';
    for (var i=0; i<array.length; i++) {
        var stringRep = stringRep + array[i] + ' ';
    }
    return stringRep;
}

function testMjsJsLibrary() {
    var u = V3.$(1,2,3);
    var v = V3.$(4,5,6);

    var s = V3.add(u,v);           // s = [5,7,9]
    alert(convertToString(s));    // will alert [5,7,9]

    var d = V3.dot(u,v); // d = 1*4+2*5+3*6=32
    alert(d);                  // will alert 32
}
```

```

var c = V3.cross(u,v);
alert(convertToString(c)); //will alert [-3,6,-3]

var M = M4x4.$(1,0,0,0,   // first column
               0,1,0,0,   // second column
               0,0,1,0,   // third column
               2,3,4,1); // fourth column

var I = M4x4.$(1,0,0,0,   // first column
               0,1,0,0,   // second column
               0,0,1,0,   // third column
               0,0,0,1); // fourth column

var MI = M4x4.mul(M,I);
alert(convertToString(MI)); // will alert 1,0,0,0
                           //           0,1,0,0,
                           //           0,0,1,0,
                           //           2,3,4,1

var T = M4x4.makeTranslate3(2,3,4);
alert(convertToString(T)); // will alert 1,0,0,0
                           //           0,1,0,0,
                           //           0,0,1,0,
                           //           2,3,4,1
}

</script>
</head>

<body onload="testMjsJsLibrary();">
  Simple web page to test WebGL-mjs JavaScript library. <br />
  The page shows a couple of alerts with the result
  from vector and matrix maths. <br />
  You need to read the source code with view source or or
  similar
  to understand the results of the alerts.
</body>

</html>

```

[Figure 4-2](#) shows what it looks like if you set a breakpoint in Chrome Developer Tools after the matrices have been created and initialized in [Listing 4-2](#). You can see that a matrix in WebGL-mjs is represented as a

Float32Array with 16 elements numbered from 0 to 15. If you look at the Float32Array that represents the matrix T from [Listing 4-2](#), you can see that it is the elements with index 13, 14, and 15 that corresponds to the translation in x -, y -, and z -direction, respectively.

FIGURE 4-2: In WebGL-mjs, matrices are represented by a `Float32Array` with 16 elements



Some Key Points to Remember about WebGL-mjs:

- *WebGL-mjs* was written specifically with *WebGL* in mind, so it has some nice functionality to perform transformations and to set up the projection matrix and the view transform.
 - *WebGL-mjs* does not use an object-oriented paradigm; instead, the vectors or matrices that the functions operate on are sent as arguments to the functions.
 - *WebGL-mjs* can only handle matrices of 4×4 dimensions and vectors of three components.
 - Matrices in *WebGL-mjs* are represented by a `Float32Array` of 16 elements where the first four elements in the array represent the first column in the matrix, the next four elements in the array represent the second column, and so on.
 - The representation of a matrix as a `Float32Array` makes it possible to send the matrix directly through the *WebGL API* to the vertex shader.

glMatrix

The JavaScript library glMatrix was written by Brandon Jones. It was written primarily for WebGL and is actually quite similar to WebGL-mjs. It has support for three element vectors and 3×3 or 4×4 matrices. However, there are some differences in how you specify the arguments for the functions and how glMatrix handles its optional arguments. But before looking at the optional arguments, you should understand a basic operation. To create a vector with three elements, you use the function,

```
vec3.create(vec)
```

where `vec` is an optional array containing the three elements you want to initialize with. This means that to create a three-component vector with the elements 1, 2, and 3, you write the following code:

```
var u = vec3.create([1, 2, 3]);
```

You can also create a three-component vector without initializing it:

```
var u = vec3.create();
```

Most of the functions in glMatrix that perform operations on vectors or matrices have an optional last argument that is the destination vector or destination matrix for the operation. Functions that operate on a vector have the following form:

```
vec3.operation(srcVec, otherOperands, destVec(optional));
```

If `destVec` is specified, the result of the operation is written to `destVec`, which is also returned from the function. However, if `destVec` is not specified, the result of the operation is instead written to `srcVec` and also returned. This means that the `srcVec` will be altered if you do not specify the `destVec`, which can lead to problems if you do not expect it to be altered.

In the following code snippet, you can see how it works when the optional destination argument is specified. Two vectors `u` and `v` are created and initialized with values. Then a third vector `s` is created to hold the result of the add operation. Since the third vector will only be used to write the result to, you do not need to initialize it. Finally, the two vectors are added and the result is written to the destination vector `s`:

```
var u = vec3.create([1, 2, 3]);
var v = vec3.create([4, 5, 6]);
var s = vec3.create();
vec3.add(u, v, s);           // s = [5, 7, 9] and u is unchanged
```

Since the optional destination vector is specified, the result is written to this vector, and this is the only argument that is changed.

Now if you instead look at the following code snippet, you will see what happens if you don't specify the destination vector.

```
var u = vec3.create([1,2,3]);
var v = vec3.create([4,5,6]);
var s = vec3.add(u,v); // s = [5,7,9] and u = [5,7,9]
```

So when you don't specify the optional destination vector when using `glMatrix`, the result is written to the first operand.

Operations on matrices basically follow the same pattern as for vectors. You can create a 4×4 matrix with a call to the function,

```
mat4.create(mat)
```

where `mat` is an optional array that contains the 16 elements that you want to initialize the matrix with. In the same way as for WebGL-mjs, the first four elements in the array populate the first column of the matrix, the next four elements populate the second column, and so on. If you want to specify the optional initialization array, the code you call would look like this:

```
var N = mat4.create([0,1,2,3,           // first column
                    4,5,6,7,           // second column
                    8,9,0,1,           // third column
                    2,3,4,5]); // fourth column
```

If you have two matrices `M` and `N` and want to calculate the product `MN`, you would call

```
var MN = mat4.multiply(M,N);
```

Note that since the optional destination matrix is not specified for this multiplication, the result would also be written to the matrix `M`.

[Listing 4-3](#) shows the source code for a complete example that uses the `glMatrix` JavaScript library. In addition to the usual vector and matrix manipulation, the example uses the function `vec3.str()` and `mat4.str()` to convert a vector or a matrix to a string. Also note how the function `mat4.translate()` is used at the end of [Listing 4-3](#). It creates a translation matrix that translates two units in *x*-direction, three units in *y*-direction, and four units in *z*-direction. The elements that correspond to the translation have the indices 13, 14, and 15.



Available for
download on
Wrox.com

**LISTING 4-3: Vector and matrix mathematics with the glMatrix
JavaScript library**


```

        2,3,4,1]); // fourth column

var IM = mat4.create();
mat4.multiply(I, M, IM);
alert(mat4.str(IM)); // will alert [1,0,0,0
                      //          0,1,0,0,
                      //          0,0,1,0,
                      //          2,3,4,1]

var T = mat4.create();
mat4.translate(I, [2,3,4],T);
alert(mat4.str(T)); // will alert [1,0,0,0
                      //          0,1,0,0,
                      //          0,0,1,0,
                      //          2,3,4,1]

}

</script>
</head>
<body onload="testglMatrixJsLibrary();">
    Simple web page to test glMatrix JavaScript library. <br />
    The page shows a couple of alerts with the result
    from vector and matrix maths. <br />
    You need to read the source code with view source or or
    similar
    to understand the results of the alerts.
</body>
</html>

```

[Figure 4-3](#) shows what it looks like if you set a breakpoint after the matrices have been created and initialized in [Listing 4-3](#). Similarly to WebGL-mjs, you can see that a matrix in glMatrix is represented as a `Float32Array` with 16 elements numbered from 0 to 15. If you look at the `Float32Array` that represents the matrix `T` from [Listing 4-3](#), you can see that here it is the elements with the indices 13, 14, and 15 that also correspond to the translation in the x -, y -, and z -directions, respectively.

FIGURE 4-3: In glMatrix, matrices are represented by a `Float32Array` with 16 elements

The screenshot shows a browser window with the title "Test of glMatrix JavaScript ...". The address bar shows the URL "file:///C:/WebGL-Examples/Listing-4-3-glMatrix/Listing-4-3.html". The page content is a simple test script for the glMatrix library, displaying comments and code snippets related to matrix operations.

The developer tools sidebar is open, showing the "Paused" state of the debugger. The "Watch Expressions" panel contains a entry for "IM: Float32Array". The "Call Stack" panel shows the call stack for the "testJMatrixLibrary" function. The "Scope Variables" panel shows the local variables "I", "M", "IM", and "T" as "Float32Array" objects, with their respective element values listed.

```

33
34 var I = mat4.create([1,0,0,0, // first column
35   0,1,0,0, // second column
36   0,0,1,0, // third column
37   0,0,0,1]); // fourth column
38
39 var M = mat4.create([1,0,0,0, // first column
40   0,1,0,0, // second column
41   0,0,1,0, // third column
42   2,3,4,1]); // fourth column
43
44 var IM = mat4.create();
45 mat4.multiply(I, M, IM);
46 alert(mat4.str(IM)); // will alert [1,0,0,0
47 // 0,1,0,0,
48 // 0,0,1,0,
49 // 2,3,4,1]
50
51 var T = mat4.create();
52 mat4.translate(I, [2,3,4], T);
53 alert(mat4.str(T)); // will alert [1,0,0,0
54 // 0,1,0,0,
55 // 0,0,1,0,
56 // 2,3,4,1]
57
58 </script>
59 </head>
60 <body onload="testglMatrixJsLibrary();">
61 Simple web page to test glMatrix JavaScript library. <br>
62 The page shows a couple of alerts with the result
63 from vector and matrix maths. <br>
64 You need to read the source code with view source or or similar
65 to understand the results of the alerts.
66 </body>

```



Some Key Points to Remember about glMatrix

- Similar to WebGL-mjs, glMatrix is also written specifically with WebGL in mind, so it has some nice functionality to perform transformations and to set up the projection matrix and the view transform.
- glMatrix does not use an object-oriented paradigm; instead, the vectors or matrices that the functions operate on are sent in as arguments.
- glMatrix can handle matrices with dimensions of 3×3 or 4×4 and vectors with three components.
- Functions in glMatrix often modify the first argument if you do not specify the last optional destination argument.
- Matrices in WebGL-mjs are represented by a `Float32Array` of 16 elements where the first four elements in the array represent the first column in the matrix, the next four elements in the array represent the second column, and so on.
- The representation of a matrix as a `Float32Array` makes it possible to send the matrix directly through the WebGL API to the vertex shader.

Now that you have learned more about these three libraries, it is up to you to choose which one you want to use for your future WebGL projects. You

may find another library not described here that you like even better. For the rest of the examples in this book that need one of these smaller libraries, the `glMatrix` library will be used.

USING TRANSFORMATIONS

In Chapter 1 you learned some of the basic linear algebra behind transformations. As already explained, it is important to understand these basics. However, as you may have noticed when you investigated the JavaScript libraries in the previous section, you don't have to manually perform matrix multiplications to do rotations, translations, or scaling in your WebGL code. There are JavaScript libraries that you can use if you want. With the knowledge you gained from the linear algebra sections in Chapter 1, you can also write your own library if you prefer.

Sometimes the trickiest part about using transformations with WebGL is not to perform the matrix multiplications. Instead, people who are new to 3D graphics often think that the most difficult part is to use a series of different transformations to position objects and the camera in the scene. But don't worry; you will learn what you need to know about transformations in the following sections.

How Transformations Are Used

Listing 3-2 in Chapter 3 showed an example where a hexagon, a triangle, and a triangle strip were drawn on the screen. In this example, no transformations were done in the vertex shader, so the coordinates that were loaded into the `WebGLBuffer` object for these shapes had to correspond to the shapes' final position in the viewport. More specifically, all the coordinates that were stored in the `WebGLBuffer` object had to be clip coordinates, because this is what the built-in variable `gl_Position` expects after the vertex shader is finished with its job.

A smarter and more flexible way that is normally used in WebGL, and in 3D graphics in general, is to model your objects in their own local coordinate system, and then apply different transforms to position and orient them

correctly in a world coordinate system. In WebGL, you generally discuss the following different types of transformations:

- Model transformation
- View transformation
- Modelview transformation
- Projection transformation
- Perspective division
- Viewport transformation

On the lowest level, the first four of these transformations are normally implemented by doing matrix multiplications in the vertex shader. The perspective division and the viewport transformation are a bit different. None of these are matrix multiplications, and they are not handled by the vertex shader but by the primitive assembly stage of the WebGL pipeline. (Refer to Chapter 1 for more information about the primitive assembly stage in the WebGL pipeline.)

Model Transformation

The initial coordinates that the vertices for an object have when they are sent in through the WebGL API and stored in the `WebGLBuffer` object, are called *object coordinates*.

The model transformation is used to position and orient the model in a **world coordinate system**. Each model that you use in your 3D scene can have its own model transformation. When all models are transformed to the world coordinate system, they all exist in the same coordinate system. The model transformation is generally composed of a combination of the following transformations:

- translation
- rotation
- scaling

If you look at the functionality in the JavaScript library `glMatrix`, these transformations correspond to the following functions:

- `mat4.translate()`
- `mat4.rotate()`
- `mat4.scale()`

Assume that you are writing a 3D game in WebGL that needs a space ship. You would first model your space ship in object coordinates. Then a model transform would translate the space ship to a position in the world coordinate system, rotate it into the correct direction, and then possibly scale it to a size that fits in your game.

The nice thing here is that one single 4×4 matrix can store all your combinations of translation, rotation, and scaling. This means that to transform the vertices for your object from object coordinates to world coordinates, you would simply have to multiply the vertices with this 4×4 matrix. You can easily do this multiplication in the vertex shader.

View Transformation

In a 3D scene, WebGL should only render the objects that the camera (also called the viewer or observer) sees. The view transformation is used to position and orient an imaginary camera. Here the word “imaginary” is used because WebGL does not really have explicit methods to move a camera in the scene. Instead, the camera is always (during perspective projection) located in the origin looking down the negative z -axis. What you actually do with the view transformation is to set up a matrix that transforms your vertices for your scene with the inverse of where you want to move your camera. You will understand this better when you read the next section.

The view transformation is usually composed of a combination of the following transformations:

- translation
- rotation

As you saw in the previous section, these transformations can be performed by the `glMatrix` functions:

- `mat4.translate()`
- `mat4.rotate()`

After the view transform has been applied to the vertices, they are in the **eye coordinate system** and the coordinates are called *eye coordinates*.



One strategy to set up your view transformation matrix is to use a combination of calls to mat4.translate() and mat4.rotate(). Another often easier strategy is to use the utility function mat4.lookAt(). This function takes a position for the camera, a viewing direction, and a vector that specifies the “up direction,” and creates a view transformation matrix. You will use mat4.lookAt() in several later examples.

Modelview Transformation

The model transformation and the view transformation are actually the same thing when it comes to how the final scene will appear. What is important is the relation between the camera and the objects in the scene. Imagine that you have both an object and a camera located in the origin. The camera is looking down the negative z -axis. In order to see the object, you have two options:

- **Move the camera backwards** — Since the camera is looking down along the negative z -axis by default, a backwards movement will involve moving the camera along the positive z -axis.
- **Move the object forward into the scene** — This means to move the object along the negative z -axis.

Both options are actually the same thing; they are only different ways to think. Either you move the camera in one direction, or you move all the objects in the opposite direction.

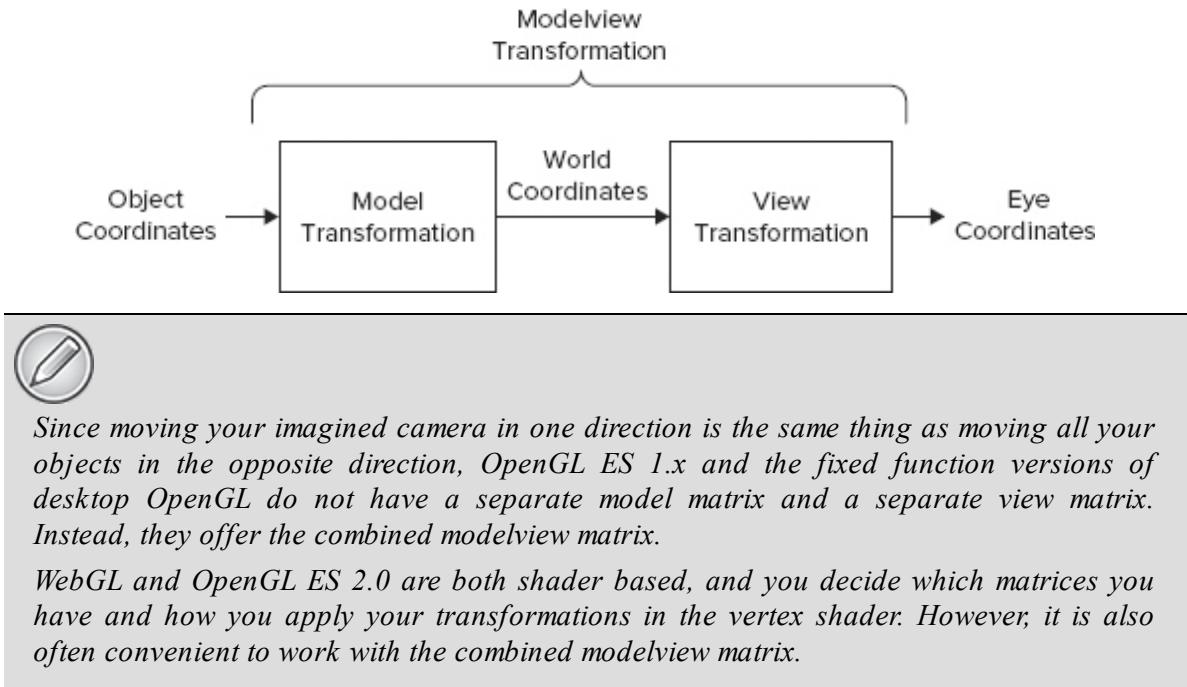
Assume that you want to have 10 units between the camera and the object that both originally were positioned in the origin. Then, no matter which way you think of the problem, the result is that you should multiply your object’s vertices with a matrix where the translation element in the z -direction is -10 . You can set up this matrix by using the glMatrix JavaScript library like this:

```
mat4.translate(modelViewMatrix, [0, 0, -10],  
modelViewMatrix);
```

The term *modelview transformation* means that the model transformation and the view transformation have been combined into a single matrix that is called *modelview matrix*. When you write your WebGL application, it is sometimes convenient to think about your view transform and your model transform as separate transforms, but both can still be stored in the combined modelview matrix. When you multiply your vertices with a combined modelview matrix, it means that the vertices are converted directly from

object coordinates to eye coordinates. An overview of how the modelview transformation is built up from the model transformation and view transformation is shown in [Figure 4-4](#).

FIGURE 4-4: The modelview transformation is a combination of the model transformation and the view transformation



Projection Transformation

The projection transformation is applied after your modelview transformation. You use it to determine how your 3D scene is projected on the screen. The projection transformation also decides how your view volume looks. What the projection transformation actually does is to take your view volume and transform it to into a unit cube that ranges from -1 to $+1$ along the three axes.

The `gl_Position` variable that is one of the outputs from the vertex shader expects all vertices that should be displayed on the screen to be within this unit cube. To work with coordinates that are outside this unit cube, you have to use a projection matrix in the vertex shader and transform the position of the vertices so they are within this unit cube before they are written to the `gl_Position` variable. Vertices and primitives that are outside this unit cube will be clipped.

In the examples that you have looked at so far, no projection transform has been used, and therefore you had to position your shapes within this cube for them to be visible on the screen.

An analogy that is sometimes used in 3D graphics is to compare the transforms with using a camera and taking a photo of some objects. In this case, the model transform corresponds to positioning your objects in the scene, the view transform corresponds to positioning and pointing your camera, and the projection transform corresponds to selecting a lens for your camera. The lens of a camera affects the field of view and to some extent how objects look.

There are two types of projection transforms:

- Orthographic projection
- Perspective projection

Both projections are described in the following sections.

Orthographic Projection

Orthographic projection is sometimes referred to as parallel projection, and a property of this projection is that parallel lines remain parallel after the projection. The most important characteristic of this projection is probably that it does not affect the size of the objects depending on how close or far way they are from the viewer. All objects keep their relative size after the projection. This is usually not what you want when you draw a scene that should look realistic, but it can be useful, for example, in CAD (Computer Aided Design) applications or if you want to draw 2D graphics with WebGL.

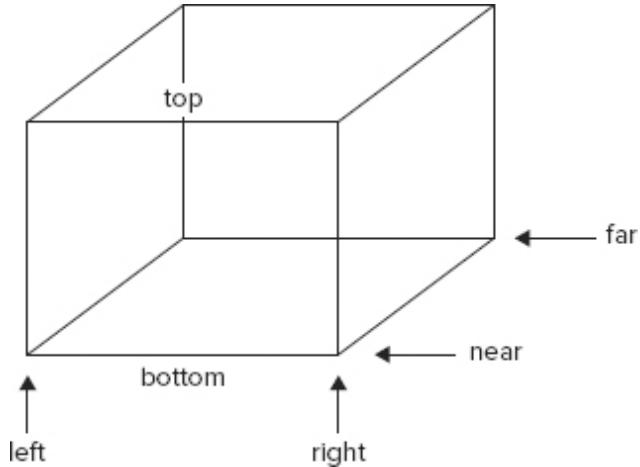
The viewing volume for orthographic projection is a rectangular box. All primitives that are inside this rectangular box are sent forward in the WebGL pipeline, and primitives outside the box are clipped. You can set up the projection matrix for orthographic projection with the `glMatrix` function `mat4.ortho()` like this:

```
mat4.ortho(left,      right,      bottom,      top,      near,      far,  
projectionMatrix);
```

This creates a projection matrix that has a rectangular view volume that looks like [Figure 4-5](#). The left bounds of the box is specified with the argument `left`, the right bounds with `right`, the bottom bounds with `bottom`, the top bounds with `top`, the near bounds with `near`, and the far bounds with

`far`. The viewer is infinitely far away in front of the box. The created projection matrix is stored in the last argument, `projectionMatrix`.

FIGURE 4-5: The view volume for orthographic projection



Perspective Projection

When you are using perspective projection, objects that are far away from the camera look smaller than objects that are close to the camera. This type of projection often gives you a more realistic scene than orthographic projection since it is closer to how your eyes work. The view volume for perspective projection is a *frustum* (a truncated pyramid).

There are two functions in `glMatrix` that you can use to set up your matrix for perspective projection:

- `mat4.perspective()`
- `mat4.frustum()`

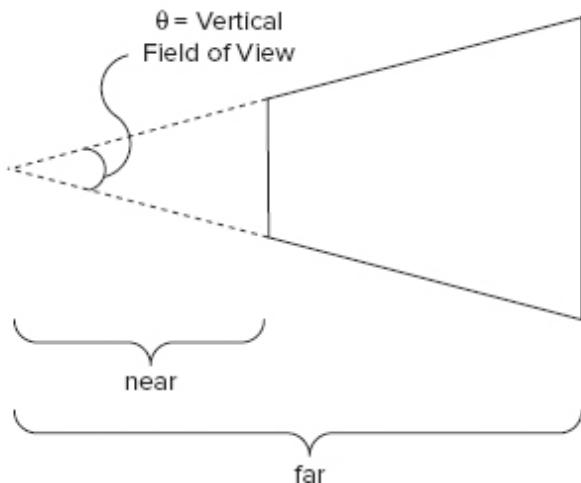
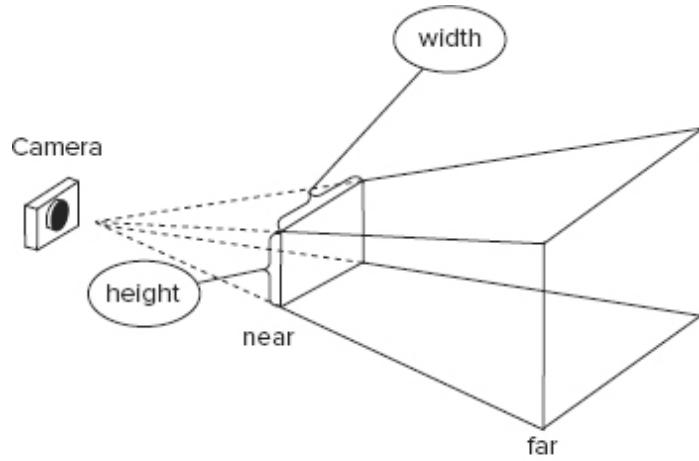
The first function, `mat4.perspective()`, is as follows:

```
mat4.perspective(fovy, aspect, near, far, projectionMatrix)
```

Here the `fovy` refers to the vertical field of view and the `aspect` is the aspect ratio (which is viewport width/viewport height). The aspect ratio is used to decide the horizontal field of view. The argument `near` is the distance to the near plane of the frustum and `far` is the distance to the far plane of the frustum. The result of the method is a projection matrix that is written to the last argument, `projectionMatrix`.

[Figure 4-6](#) shows the view frustum for a perspective projection. The bottom of [Figure 4-6](#) represents a view frustum where only one side is shown. Here you can see how the vertical field of view is defined.

FIGURE 4-6: The view volume for a perspective projection



You can use the second function, `mat4.frustum()`, to create a perspective projection matrix as follows:

```
mat4.frustum(left,      right,      bottom,      top,      near,      far,  
projectionMatrix)
```

The arguments have the following meanings:

- `left` sets the left bounds of the near plane.
- `right` sets the right bounds of the near plane.
- `bottom` sets the bottom bounds of the near plane.
- `top` sets the top bounds of the near plane.
- `near` sets the location of the near plane.
- `far` sets the location of the far plane.

- `projectionMatrix` sets the destination — i.e., the created projection matrix.

The function `mat4.frustum()` is more generic than `mat4.perspective()`, and in `glMatrix` (and several other libraries that have corresponding methods), `mat4.perspective()` is implemented with `mat4.frustum()`.



Many of the functions in the JavaScript libraries `glMatrix` and `WebGL-mjs` were not invented by the creators of these libraries. Instead, both of these libraries have borrowed from what is available in, for example, `OpenGL` and `GLU` (`OpenGL` utility library). `GLU` is a library with functions that were designed to complement desktop `OpenGL` by providing additional functionality such as building viewing and projection matrices.

It is, of course, a good thing that `glMatrix` and `WebGL-mjs` have borrowed from `OpenGL` and `GLU`. There are many books as well as documentation that describe `OpenGL` and `GLU`, and most developers in forums about 3D graphics are familiar with the functions from these libraries, so it is easy to find help if you need it. Some examples of functions from `glMatrix` and their corresponding functions in `OpenGL` and `GLU` are as follows:

- `mat4.rotate()` in `glMatrix` corresponds to `glRotate()` in `OpenGL`.
- `mat4.frustum()` in `glMatrix` corresponds to `glFrustum()` in `OpenGL`.
- `mat4.perspective()` in `glMatrix` corresponds to `gluPerspective()` in `GLU`.
- `mat4.lookAt()` in `glMatrix` corresponds to `gluLookAt()` in `GLU`.

Perspective Division

When the vertex shader writes the coordinates to the variable `gl_Position`, it is done in clip coordinates that are still on homogeneous notation with four components (x_c, y_c, z_c, w_c). During the primitive assembly, the vertices go through the **perspective division**, which divides all coordinates with w_c to yield **normalized device coordinates** (NDC). The relationship between clip coordinates (x_c, y_c, z_c, w_c) and normalized device coordinates (x_d, y_d, z_d) is as follows:

$$\begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix} = \begin{bmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{bmatrix}$$

As you can see, this step is not implemented as a matrix multiplication. In addition, it is different from the previous steps since it is normally not a step you can explicitly affect yourself. The fourth clip coordinate w_c is normally 1 and it is usually nothing that you change in your code.

Viewport Transformation

Like the perspective division, the viewport transformation is also not implemented as a matrix multiplication. The viewport transformation is performed for you as part of the primitive assembly stage, but you can affect how it is done by calling the following methods:

```
gl.viewport(x, y, w, h);  
gl.depthRange(n, f);
```

You have seen the method `gl.viewport()` before. It specifies the viewport by giving the lower-left coordinate of the viewport with `(x, y)` and the width and height of the viewport with `w` and `h`.

The method `gl.depthRange()` is used to specify the desired depth range. The argument `n` is the near plane and `f` is the far plane. Both `n` and `f` are clamped to lie within the range `[0.0, 1.0]`. You are not allowed to set `n` to a greater value than `f`, because then the WebGL implementation will generate an `gl.INVALID_OPERATION` error.

Also note that the default if you don't call this method is that `n` will have the value `0.0` and `f` will have the value `1.0`.

The actual transformation from normalized device coordinates (x_d, y_d, z_d) to window coordinates (x_w, y_w, z_w) is given by the following formula:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} (w/2) x_d + o_x \\ (h/2) y_d + o_y \\ ((f-n)/2) z_d + (n+f)/2 \end{bmatrix}$$

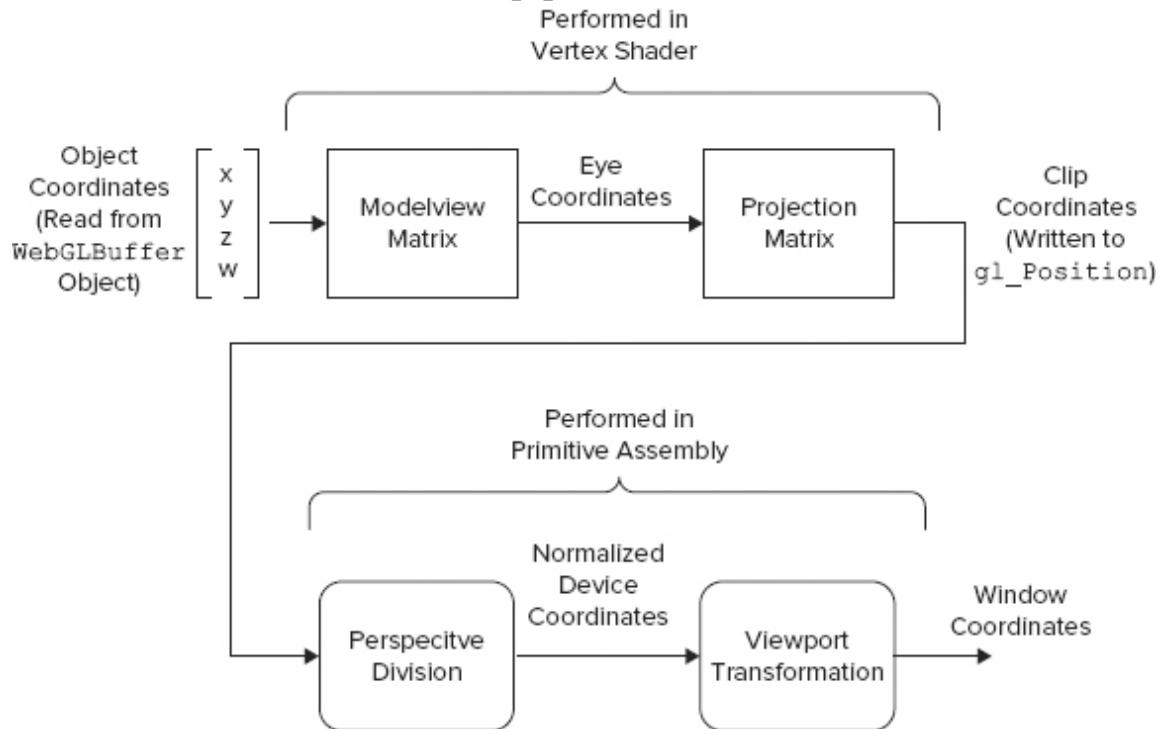
Here `w` and `h` are the width and height set in pixels with the `gl.viewport()` method. The values o_x and o_y represent the viewport's center and are defined as $o_x = (x + w)/2$ and $o_y = (y + h)/2$.

UNDERSTANDING THE COMPLETE TRANSFORMATION PIPELINE

Now that you have a basic understanding of important concepts such as model transform, view transform, modelview transform, and projection, it is time to

look at what the complete transformation pipeline looks like in WebGL. [Figure 4-7](#) shows an overview of the complete transformation pipeline.

FIGURE 4-7: The transformation pipeline



Note that as explained before, WebGL is fully shader based, which means that you, in theory, could work with whatever transformation matrices you would like in the vertex shader. However, the split in one modelview matrix and one projection matrix is commonly used, as shown in [Figure 4-7](#).

Even if you could make out the majority of this pipeline from the text in the previous sections, this section will give you a complete overview to make sure you understand it.

At the left of [Figure 4-7](#), your vertices are stored in a `WebGLBuffer` object in object coordinates. They are read by the vertex shader and the vertex shader then multiplies the vertices with the modelview matrix. After the modelview matrix, the vertices are in eye coordinates. If you perform lighting calculations in your vertex shader, these are normally done in eye coordinates.

The vertex shader then multiplies the vertices with the projection matrix to get clip coordinates. The vertex shader writes the vertices to the variable `gl_Position`, which expects the vertices to be in clip coordinates.

After the vertex shader has written the vertices to the `gl_Position`, they go through the perspective division where the four clip coordinates (x_c, y_c, z_c, w_c) are divided by w_c to yield normalized device coordinates. Finally, the normalized device coordinates are mapped to the actual screen coordinates by the viewport transformation.

GETTING PRACTICAL WITH TRANSFORMATIONS

Hopefully you now understand some of the basics of different transformations and how they are used in WebGL. You have seen some of the functions in the JavaScript library `glMatrix` that can be used to set up the transformation matrices, and you have also learned what the complete transformation pipeline looks like. Now it is time to have a closer look at some more code to see what the complete chain looks like from a more practical point of view. This is how it works on a high level:

1. You set up your `WebGLBuffer` objects that contain object coordinates for the objects in your scene.
2. Before you call any drawing method, you create the modelview matrix and the projection matrix, typically by calling functions provided by a JavaScript library.
3. The transformation matrices created in your JavaScript code need to be uploaded to the vertex shader in the GPU. You typically do this by setting a uniform in the vertex shader with the method `gl.uniformMatrix4fv()`.
4. You draw your scene by calling `gl.drawArrays()` or `gl.drawElements()`.
5. Now the vertex shader is executed for each vertex in the scene. The vertex shader gets the input from the `WebGLBuffer` objects, which contain object coordinates. It uses the uniforms where the transformation matrices were uploaded and performs the transformations by doing matrix multiplications.

Setting Up Buffers with Object Coordinates

Before you set up your buffers with the vertex data, you need to do the usual setup and initialization of WebGL, such as creating the WebGL context, loading, compiling, and linking your shaders. You are probably familiar with these steps by now. When you then come to the step of setting up your `WebGLBuffer` objects, the only news is that you don't have to give your objects vertex positions according to how you want them to be positioned in your final scene.

Previously when you did not do any transformations, your object coordinates needed to be equal to your clip coordinates. This meant that the coordinates in the `WebGLBuffer` objects had to be within the cube, ranging from $(-1, -1, -1)$ to $(1, 1, 1)$, to not be clipped. In addition, if you, for example, wanted one object to be positioned to the left of another object, then this had to be taken care of already when assigning the vertex positions that were loaded into the `WebGLBuffer` object.

Now, when you are taking advantage of transformations, you have more flexibility and your object coordinates for the models would typically be centered around the origin, regardless of where they will be positioned in the final scene. The following code snippet is just the usual setup of buffers and should be familiar to you:

```
function setupBuffers() {
    cubeVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);

    var cubeVertexPosition = [
        // Front face
        1.0, 1.0, 1.0, //v0
        -1.0, 1.0, 1.0, //v1
        -1.0, -1.0, 1.0, //v2
        1.0, -1.0, 1.0, //v3

        ...
        -1.0, -1.0, -1.0, //v22
        -1.0, -1.0, 1.0, //v23
    ];

    gl.bufferData(gl.ARRAY_BUFFER,
        new
        Float32Array(cubeVertexPosition),
```



```

mat4.identity(modelViewMatrix);
mat4.lookAt([8, 5, 10], [0, 0, 0], [0, 1, 0],
modelViewMatrix);

mat4.translate(modelViewMatrix, [0.0, 3.0, 0.0],
modelViewMatrix);

```

First, you use the method `mat4.create()` to create one 4×4 matrix that is used as a modelview matrix and one 4×4 matrix that is used as a projection matrix. Then you use `mat4.perspective()` to set up the projection matrix with a vertical field of view of 60 degrees, an aspect ratio that is given by the viewport width divided by the viewport height, a near plane 0.1 units in front of the viewer, and a far plane 100.0 units from the viewer.

Then you use `mat4.identity()` to load the identity matrix into the modelview matrix. You do this to reset the modelview matrix before the modelview transformations for the scene are applied on it. If you don't reset the modelview matrix to the identity matrix, it could hold values from the previous frame, which is probably not what you want.

Normally, you set up the view transform on the modelview matrix before the model transform. In this snippet, the view transform has been set up with the method `mat4.lookAt()`. The first argument to `mat4.lookAt()` specifies that the viewer is located at the position (8, 5, 10). The second argument specifies that the view direction is towards the origin. The third argument specifies that the up direction is the positive y -axis. The last call is to `mat4.translate()` and adds a translation of 3.0 units in the positive y -direction to the modelview matrix.

Uploading the Transformation Matrices to the Vertex Shader in the GPU

When you have created your transformation matrices in JavaScript, you must upload them to the GPU before they can be used to do any transformations in the vertex shader. You do this by loading the 16 float values into a uniform with the method `gl.uniformMatrix4fv()`. The following code shows the two helper functions that you can use to upload the modelview matrix and the projection matrix to the vertex shader:

```

function uploadModelViewMatrixToShader() {
    gl.uniformMatrix4fv(shaderProgram.uniformMVMMatrix, false,

```

```

        modelViewMatrix);
    }

function uploadProjectionMatrixToShader() {
    gl.uniformMatrix4fv(shaderProgram.uniformProjMatrix,
                        false, projectionMatrix);
}

```

The first argument to `gl.uniformMatrix4fv()` specifies which uniform variable you want to load data into. The second argument specifies whether you want to transpose the columns that are uploaded. In WebGL this argument must be set to `false`. If not, a `gl.INVALID_VALUE` is generated.



In WebGL (and in OpenGL ES 2.0), the transpose argument of the `gl.uniformMatrix` methods (like, for example, the `gl.uniformMatrix4fv()` used in this section) must be set to `false`. The reason that the argument is still included in the method definition is to keep the same arguments as for the desktop OpenGL specification. Remember, WebGL is based on OpenGL ES 2.0, which, in turn, is based on desktop OpenGL 2.0.*

Since `glMatrix` represents a matrix as `Float32Array` with 16 values in column-major order and the `mat4` type in OpenGL ES Shading Language also represents matrices in column-major order, the two representations of matrices fit well together, and you should not be tempted to set the transpose argument to anything other than `false`. The last argument is the actual data for the matrix. When you have used `glMatrix` to create and manipulate the matrix, this consists of a `Float32Array` with the 16 matrix elements.

Calling Your Drawing Methods

When you have your transformation matrices set up correctly and they have been uploaded to the vertex shader in the GPU, you can call `gl.drawArrays()` or `gl.drawElements()` to do the actual drawing. This step should be familiar to you and there is not much new here.

Performing Matrix Multiplications in the Vertex Shader

When you call `gl.drawArrays()` or `gl.drawElements()`, the vertex shader is executed for the vertices in the bound `WebGLBuffer` objects. In the

following snippet, you see a vertex shader where the vertex position in object coordinates is located in the attribute `aVertexPosition`. The modelview matrix is located in the uniform `uMVMMatrix` of the type `mat4` and the projection matrix is located in the uniform `uPMatrix`, also of the type `mat4`. You can see how the vertices are transformed from object coordinates (in `aVertexPosition`) to clip coordinates that are written to the variable `gl_Position` simply by multiplying first with the modelview matrix and then with the projection matrix:

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec4 aVertexColor;

    uniform mat4 uMVMMatrix;
    uniform mat4 uPMatrix;

    varying vec4 vColor;

    void main() {
        gl_Position = uPMatrix * uMVMMatrix *
        vec4(aVertexPosition, 1.0);

        vColor = aVertexColor;
    }

</script>
```

UNDERSTANDING THE IMPORTANCE OF TRANSFORMATION ORDER

When you learned about matrix multiplications in Chapter 1, you saw that the order in which the matrices are multiplied is important for the result. In general, if you have two matrices **M** and **N**, you get a different result depending on the order in which you multiply them. That is, in general:

$$MN \neq NM$$

You also know that the transformations that you perform in the vertex shader are actually matrix multiplications. In addition to the matrix multiplications in

the vertex shader, you might also perform several matrix multiplications in JavaScript before you upload your transformation matrices to the shader. For example, this is the case if you do several transformations (e.g., a transformation followed by a rotation) to set up your modelview matrix in JavaScript.

This means that the order in which you do your transformations will be important. If you think about translations and rotations that are important for your modelview transformation, you will generally get a different result if you do the translation before the rotation as opposed to the other way around. When performing a series of transformations, there are two ways to think about the transformations:

- Using a grand, fixed coordinate system
- Using a moving, local coordinate system

These are just two different ways to think about the transformations. No matter which way you use, the end result is the same. In the next section, you will learn how you think about transformations when you use the grand, fixed coordinate system. In the section after that you will learn how you think when you use a moving, local coordinate system.

Using a Grand, Fixed Coordinate System

A grand, fixed coordinate system refers to the fact that the coordinate system is fixed and does not move when you do the different transformations. This is probably what you would normally expect when you think about a coordinate system. To understand the thinking behind the fixed, grand coordinate system, you can look at the details of how the matrix multiplications are done when you do a series of transformations.

As an example, assume that you are using the JavaScript library `glMatrix` and using the function `mat4.rotate()` to perform a rotation and `mat4.translate()` to do a translation with the following code snippet:

```
mat4.identity(modelViewMatrix);
mat4.rotate(modelViewMatrix,           Math.PI/4,          [0,0,1],
modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
```

Think about what the modelview matrix (here, named `modelViewMatrix`) contains after each of these function calls. After the call to

`mat4.identity()`, the modelview matrix consists of the identity matrix **I**. After the call to `mat4.rotate()`, the modelview matrix consists of the identity matrix post-multiplied with the rotation matrix **R**, which means that the modelview matrix contains **IR**. After the call to `mat4.translate()`, the modelview matrix has now also been post-multiplied with the translation matrix **T**, which means that it contains **IRT**.

After the modelview matrix has been constructed with these calls to `glMatrix`, the modelview matrix will be uploaded to the vertex shader in the GPU with a call to `gl.uniformMatrix4fv()`. In the vertex shader, the vertices will be multiplied with this modelview matrix, as you have seen before.

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec4 aVertexColor;

    uniform mat4 uMVMMatrix;
    uniform mat4 uPMatrix;

    varying vec4 vColor;

    void main() {
        gl_Position = uPMatrix * uMVMMatrix *
vec4(aVertexPosition, 1.0);

        vColor = aVertexColor;
    }

</script>
```

This means that if you assume that you have the vertex position in a vector **v**, you do the following multiplication to transform it from object coordinates to eye coordinates:

$$\text{"Vertex in Eye Coordinates"} = \mathbf{IRTv}$$

As you can see, the vertex is transformed as if it were first multiplied with matrix **T**, then matrix **R**, and finally matrix **I**. Compare this to the order in which you called the functions in the `glMatrix` JavaScript library shown again here:

```
mat4.identity(modelViewMatrix);
mat4.rotate(modelViewMatrix,           Math.PI/4,
            [0,0,1],
```

```

modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);

```

As you can see, the vertex is transformed in the opposite order compared to how you called the functions in the code. [Figure 4-8](#) shows how this looks if you have a cube in the origin and then call `mat4.rotate()`, followed by `mat4.translate()` as previously discussed. To the left, the cube is located in the origin. In the middle of the figure, you can see what it looks like when the cube has been translated along the positive x -axis. Finally to the right, the cube has been rotated PI/4 radians around the z -axis. The end result is that the cube ends up in the plane where $z = 0$ in between the positive x -axis and the positive y -axis.

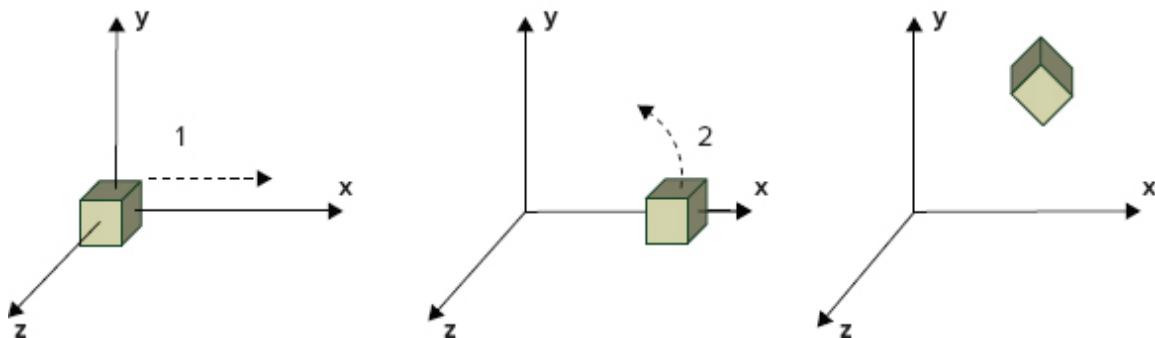
FIGURE 4-8: With a grand, fixed coordinate system, the translation happens before the rotation even though the call to `mat4.rotate()` happens before the call to `mat4.translate()` in the source code

Order of Calls in the Code

```

mat4.rotate()
mat4.translate()

```



If you change the order of the calls to `mat4.rotate()` and `mat4.translate()`, you will end up with the following code snippet:

```

mat4.identity(modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
mat4.rotate(modelViewMatrix, Math.PI/4, [0,0,1],
modelViewMatrix);

```

This means that your vertices will be multiplied with the modelview matrix **ITR** (where **I** is the identity matrix, **T** is the translation matrix, and **R** is the rotation matrix). This time, you get the vertex in eye coordinates with the following formula:

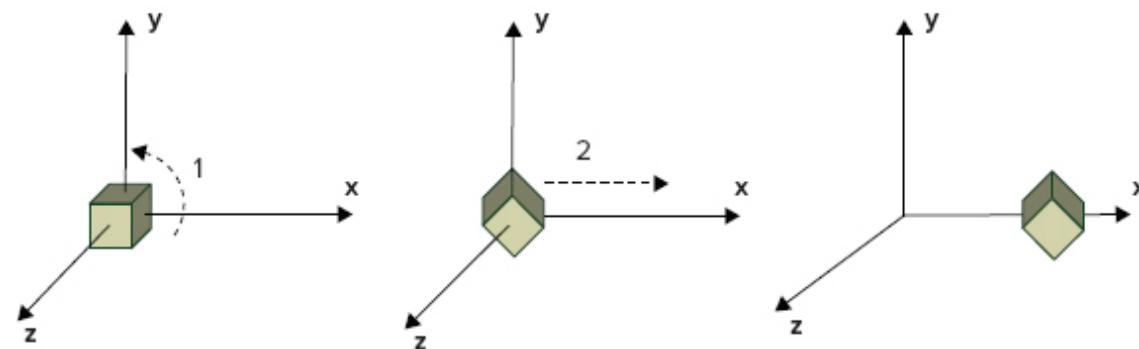
$$\text{"Vertex in Eye Coordinates"} = \mathbf{ITRv}$$

As you can see, this means that the rotation happens first, followed by the translation. [Figure 4-9](#) shows what this will look like for a cube that again is originally positioned in the origin. To the left, you can see the cube located in the origin. In the middle, the figure shows what it looks like when the cube has been rotated PI/4 radians around the z-axis. To the right, the cube has been translated along the positive x-axis. The end result is that the cube ends up on the positive x-axis rotated PI/4 radians.

FIGURE 4-9: In a grand, fixed coordinate system, the rotation happens before the translation even though the call to `mat4.translate()` happens before the call to `mat4.rotate()` in the source code

Order of Calls in the Code

```
mat4.translate()  
mat4.rotate()
```



When you use a grand, fixed coordinate system to think about your transformations a series of transformations are done in the opposite order of how they are written in the code.

Using a Moving, Local Coordinate System

Instead of thinking of a grand, fixed coordinate system when you do a series of transformations, an alternative way is to think of a moving, local coordinate system that is tied to the object that you transform. When you do a transformation, the object's local coordinate system follows the object. All subsequent transformations are done relative to this local coordinate system. If you use this way of thinking, the transformations happen in the same order

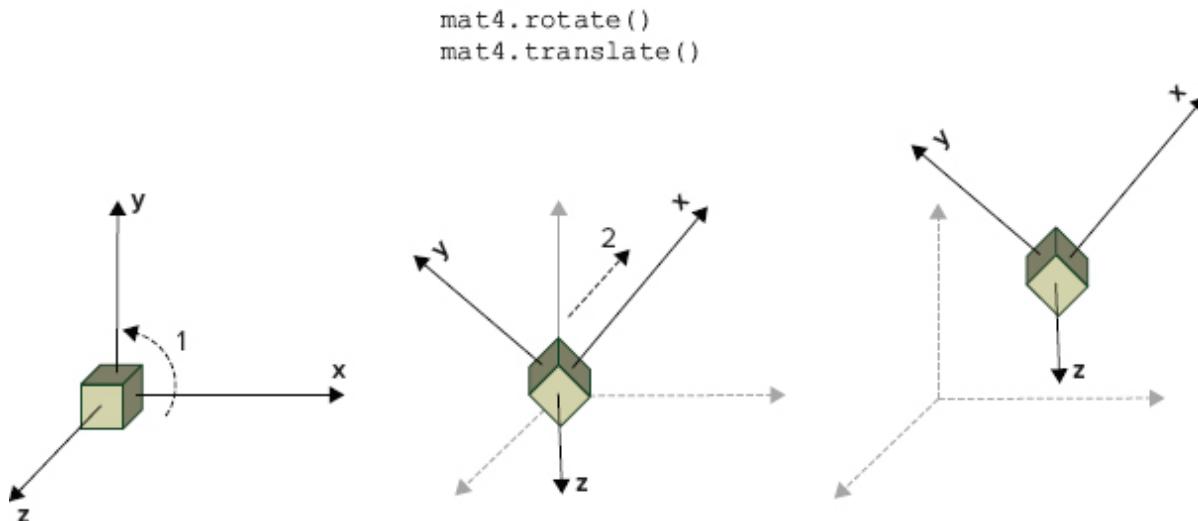
as they are written in the code. If you again take the case when you have the following code snippet,

```
mat4.identity(modelViewMatrix);
mat4.rotate(modelViewMatrix,           Math.PI/4,           [0,0,1],
modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
```

then you first imagine that you do the rotation of the object and its local coordinate system PI/4 radians around the z -axis. After this, you perform the translation along the x -axis (which was also rotated and now points PI/4 radians in between the positive x -axis and the y -axis). [Figure 4-10](#) shows how you would think of these transformations using the moving, local coordinate system.

FIGURE 4-10: In a moving, local coordinate system, the rotation happens before the translation, which matches that the call to `mat4.rotate()` happens before the call to `mat4.translate()` in the source code

Order of Calls in the Code



As previously explained, it is important to note that whether you use the moving, local coordinate system or the grand, fixed coordinate system, you get the same result and the source code is the same. These are only different ways to think of the problem.

To be consistent, you can see here how you should think when you have the `mat4.translate()` before the `mat4.rotate()` and use a local coordinate system. So assume you have the following code snippet:

```

mat4.identity(modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
mat4.rotate(modelViewMatrix, Math.PI/4, [0,0,1],
modelViewMatrix);

```

Next, imagine that you do the translation of the object and its local coordinate system 5 units along the x -axis. After this, you perform the rotation around the origin (which was also translated and is still at the center of the object). [Figure 4-11](#) shows how you would think of these transformations using the moving, local coordinate system.

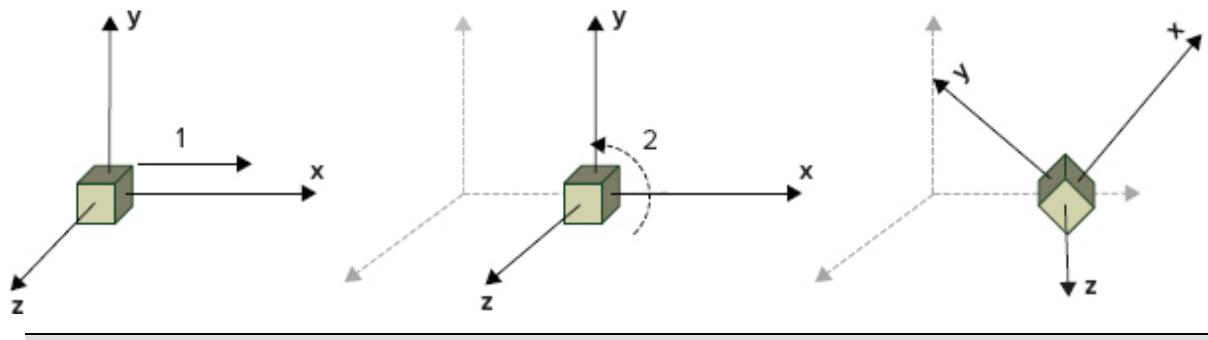
FIGURE 4-11: In a moving, local coordinate system, the translation happens before the rotation, which matches that the call to `mat4.translate()` happens before the call to `mat4.rotate()` in the source code

Order of Calls in the Code

```

mat4.translate()
mat4.rotate()

```



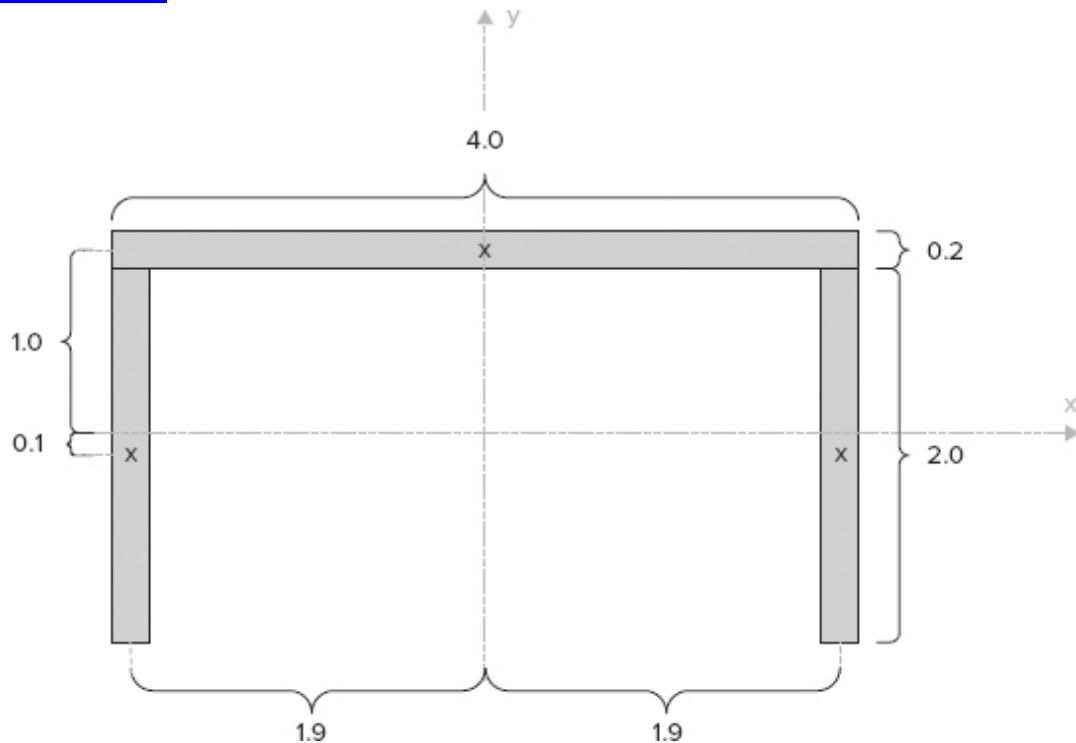
When you use a local coordinate system to think about your transformations, a series of transformations are done in the same order as they are written in the code. The local coordinate system follows the object that is transformed.

Pushing and Popping Transformation Matrices

Assume that you want to draw a table by using a cube that you scale to five cuboids (one cuboid for the table top and one cuboid for each of the four table legs). You could draw this table by performing a translation from the table's origin to where you want to have the table top positioned, then scale the cube

to a cuboid that looks like a table top, and finally draw the table top. Then you would translate to the position where you want to draw the first table leg, scale the cube to a cuboid that looks like a table leg, and draw it. Then you would translate to the position for the second table leg and draw it, and so on. [Figure 4-12](#) shows a 2D sketch of a table that is drawn in this way.

FIGURE 4-12: The sketch of a table from the side



From a performance point of view, you want to have as few calls to `gl.drawArrays()` and `gl.drawElements()` as possible. This means that drawing a table by using five cuboids might not be the most efficient approach. However, this technique is used in this chapter to demonstrate translations and pushing and popping from the modelview matrix stack.

It would be possible to draw a table like this, but it would also be very complicated. After you scale the cube for the table top, the translation to the first table leg must take into account this scaling. Instead, when you have compound objects like this, you often want hierarchical transformations and the possibility to save a transformation state anywhere in the hierarchy.

If you continue with the example with the table and use this second strategy, you start in the origin of the table. Before you translate to the position for the table top, you save the current modelview transformation matrix. Then you

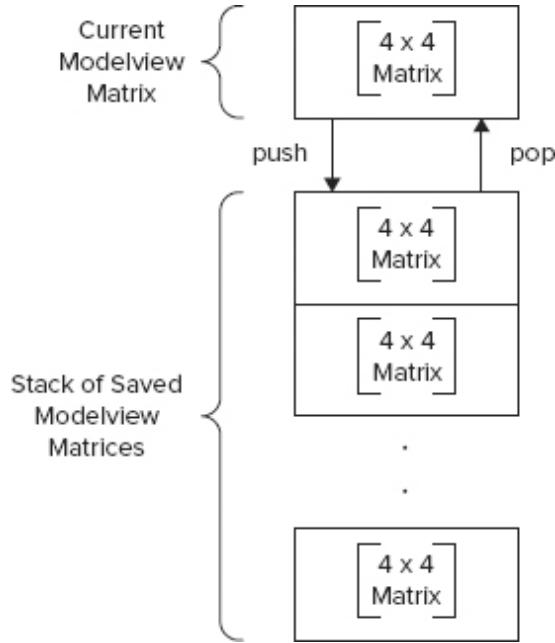
translate to the position where you want the table top, scale the cube to a cuboid that looks like a table top, and draw it.

When you are done with the table top, instead of trying to translate directly from the position of the table top to the position for the first leg, you first restore the modelview transformation matrix that you saved before the translation and scaling for the table top. This means that you are back in the origin of the table and you have no scaling in the modelview matrix.

From here, you again save the current modelview matrix, translate to the position of the first table leg, scale the cube to a cuboid that looks like a table leg, and draw it. When you are finished with the first table leg, you restore the saved transformation matrix. Then you continue with the second, third, and fourth table legs in the same way.

A convenient way to store a transformation matrix when you have a hierarchy of transformations like this is by using a stack of transformation matrices. When you want to store a transformation matrix, you *push* a copy of it to the top of the stack, and when you want to restore it, you *pop* it from the stack. [Figure 4-13](#) shows the concept of a stack of modelview matrices. The idea of a stack of transformation matrices is most useful for the modelview matrix, but it could in some cases be useful to have a stack of projection matrices as well. An example is if you have a WebGL application that uses both perspective and orthographic projection. This may occur when you want to be able to toggle between viewing the scene with perspective projection and orthographic projection.

[FIGURE 4-13](#): A stack of modelview matrices



The array data type in JavaScript contains the methods `push()` and `pop()`, which can be used to handle an array as if it were a stack. Based on a JavaScript array, it is easy to implement a stack of modelview matrices as in the following code snippet:

```

modelViewMatrix = mat4.create();
modelViewMatrixStack = [];

function pushModelViewMatrix() {
    var copyToPush = mat4.create(modelViewMatrix);
    modelViewMatrixStack.push(copyToPush);
}

function popModelViewMatrix() {
    if (modelViewMatrixStack.length == 0) {
        throw "Error popModelViewMatrix() - Stack was empty ";
    }
    modelViewMatrix = modelViewMatrixStack.pop();
}

```

In your code, you can now call `pushModelViewMatrix()` each time you want to save the modelview matrix, and then call `popModelViewMatrix()` when you want to restore it.

Now assume that you have a function with which you can draw a cube with endpoints at $(-1, -1, -1)$ and $(1, 1, 1)$ when it is untransformed:

```
drawCube(r, g, b, a)
```

The function draws a cube with the color (r, g, b, a) and with the side equal to 2 units when it is untransformed. However, it will take the current modelview transformation matrix into account. To draw the table shown in [Figure 4-12](#) based on the function `drawCube` and using the modelview matrix stack as previously described, you could use the following code snippet:

```
function drawTable() {
    // draw table top
    pushModelViewMatrix();
    mat4.translate(modelViewMatrix, [0.0, 1.0, 0.0], modelViewMatrix);
    mat4.scale(modelViewMatrix, [2.0, 0.1, 2.0], modelViewMatrix);
    uploadModelViewMatrixToShader();
    // draw the actual cube (now scaled to a cuboid) in brown color
    drawCube(0.72, 0.53, 0.04, 1.0); // argument sets brown color
    popModelViewMatrix();

    // draw table legs
    for (var i=-1; i<=1; i+=2) {
        for (var j= -1; j<=1; j+=2) {
            pushModelViewMatrix();
            mat4.translate(modelViewMatrix, [i*1.9, -0.1, j*1.9], modelViewMatrix);
            mat4.scale(modelViewMatrix, [0.1, 1.0, 0.1], modelViewMatrix);
            uploadModelViewMatrixToShader();
            drawCube(0.72, 0.53, 0.04, 1.0); // argument sets brown color
            popModelViewMatrix();
        }
    }
}
```

Immediately after you enter the function `drawTable()`, you call `pushModelViewMatrix()` to save the current modelview matrix. Then you translate 1.0 unit in the y-direction to end up in the position where you want the table top. You call `mat4.scale()` to scale the table by 2 in the x- and z-directions and 0.1 in the y-direction. Then you upload the modelview matrix to the vertex shader by calling `uploadModelViewMatrixToShader()`, and draw the table top by calling the function `drawCube()`. Finally, you call

`popModelViewMatrix()` to restore the modelview matrix to the value it had before the calls to `mat4.translate()` and `mat4.scale()`.

When the table top is finished, the function draws the four table legs by looping through the four positions, and scales to a cuboid that is 0.2 units in the x - and z -directions and 2 units in the y -direction. After each table leg is drawn, the modelview matrix is restored by popping the modelview matrix stack.



If you are rendering a scene where you want each object to have its own transform, you can use the modelview matrix stack to do this. You wrap each transform and rendering of the object with calls to pushing and popping the modelview matrix stack.

```
pushModelViewMatrix();
mat4.translate(modelViewMatrix, [xPos, yPos, zPos],
    modelViewMatrix);
drawObject();
popModelViewMatrix();
```

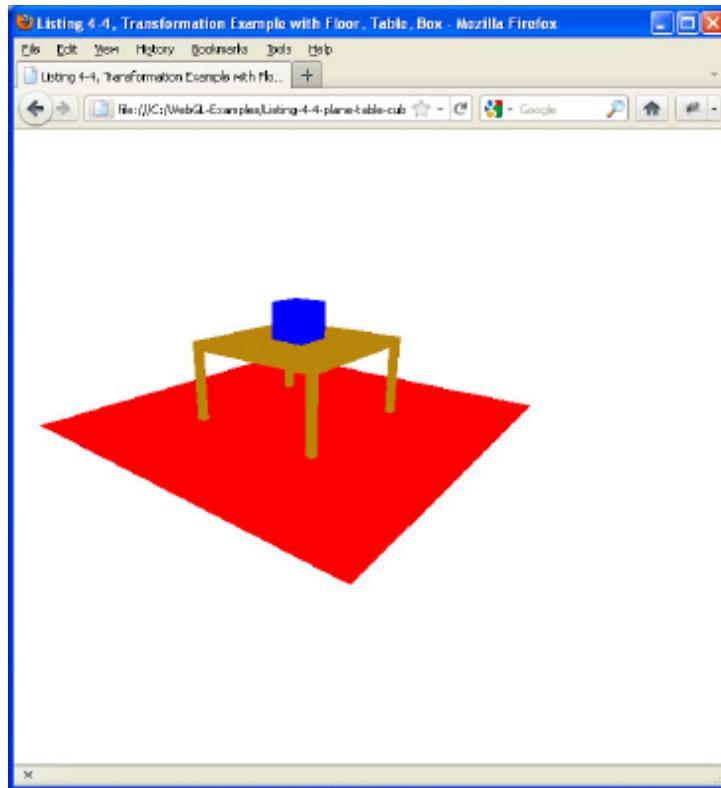
A COMPLETE EXAMPLE: DRAWING SEVERAL TRANSFORMED OBJECTS



Available for
download on

Wrox.com Now it is time to look at a complete example that uses a lot of the knowledge that you have gained in this chapter. Download Listing 4-4 from the book's website on [Wrox.com](#) and examine the code. (The listing is not shown in the chapter due to its length.) Listing 4-4 draws a scene where a table is placed on a floor. A box is placed on top of the table. If you load the code from Listing 4-4 into your browser, you should see something similar to what is shown in [Figure 4-14](#).

FIGURE 4-14: The scene that corresponds to Listing 4-4



Most of the code in Listing 4-4 should be familiar to you. You can start by looking at the top of the JavaScript code in Listing 4-4, which is shown here:

```
var floorVertexPositionBuffer;
var floorVertexIndexBuffer;
var cubeVertexPositionBuffer;
var cubeVertexIndexBuffer;

var modelViewMatrix;
var projectionMatrix;
var modelViewMatrixStack;
```

Here you can see the variables that represent the vertex positions and indices for the floor and the cube. You also have the variables for the modelview matrix, the projection matrix, and the modelview matrix stack.

The next chunk of code to consider is at the end of the function `setupShaders()`. Here the modelview matrix, the projection matrix, and the modelview matrix stack are created and assigned to their global variables:

```
modelViewMatrix = mat4.create();
projectionMatrix = mat4.create();
modelViewMatrixStack = [];
}
```

Directly after this code, you have the two helper functions to push and pop the modelview matrix stack:

```
function pushModelViewMatrix() {
    var copyToPush = mat4.create(modelViewMatrix);
    modelViewMatrixStack.push(copyToPush);
}

function popModelViewMatrix() {
    if (modelViewMatrixStack.length == 0) {
        throw "Error popModelViewMatrix() - Stack was empty ";
    }
    modelViewMatrix = modelViewMatrixStack.pop();
}
```

In this example, the setup of the buffers has been split into one function called `setupFloorBuffers()` that sets up the buffers for the floor, and one function called `setupCubeBuffers()` that sets up the buffers for the cube that is used to draw the table and the box on top of the table. The function `setupFloorBuffers()` is very simple and just sets up vertex positions that represent a plane in $y = 0$ ranging from -5.0 to 5.0 in the x - and z -directions, as shown in the following code snippet:

```
function setupFloorBuffers() {
    floorVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, floorVertexPositionBuffer);

    var floorVertexPosition = [
        // Plane in y=0
        5.0, 0.0, 5.0, //v0
        5.0, 0.0, -5.0, //v1
        -5.0, 0.0, -5.0, //v2
        -5.0, 0.0, 5.0]; //v3

        gl.bufferData(gl.ARRAY_BUFFER,
new
Float32Array(floorVertexPosition),
gl.STATIC_DRAW);
```

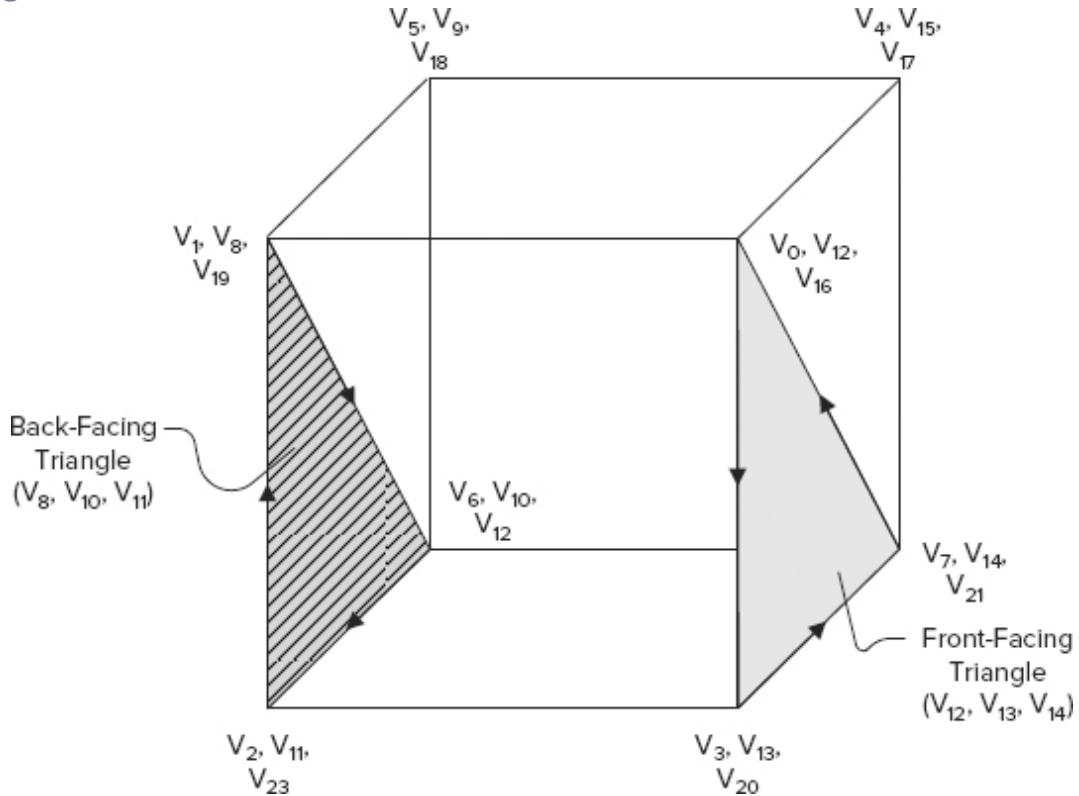
After this comes the function `setupCubeBuffers()`, which sets up the vertices for a cube. Some more details of how to set up the buffers with vertices for a cube are described in the next section.

Creating a Cube with WebGL

Creating a cube with WebGL is not difficult. You set up your vertex buffer as usual. The first time you do this, it can be a bit tricky to get the winding of all the triangles correct. Triangle winding order was discussed in Chapter 3, but at that time you only used it for objects in 2D. When drawing objects in 3D, you should normally use front-facing triangles to draw the outside surface of any solid objects.

In the following snippet, you can see the function `setupCubeBuffers()` again. The vertices in this snippet correspond to the vertices for the cube shown in [Figure 4-15](#). The origin in object coordinates is located at the center of the cube, and as you can see in the following code snippet, each side of the cube is 2 units long.

FIGURE 4-15: The cube with one back-facing triangle and one front-facing triangle shown



```
function setupCubeBuffers() {
    cubeVertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexBuffer);

    var cubeVertexPosition = [
        // Front face
        1.0,  1.0,  1.0, //v0
        -1.0,  1.0,  1.0, //v1
        -1.0, -1.0,  1.0, //v2
        1.0, -1.0,  1.0, //v3
        1.0,  1.0, -1.0, //v4
        -1.0,  1.0, -1.0, //v5
        -1.0, -1.0, -1.0, //v6
        1.0, -1.0, -1.0, //v7
    ];
}
```

```

        -1.0,  1.0,  1.0, //v1
        -1.0, -1.0,  1.0, //v2
         1.0, -1.0,  1.0, //v3

        // Back face
         1.0,  1.0, -1.0, //v4
        -1.0,  1.0, -1.0, //v5
        -1.0, -1.0, -1.0, //v6
         1.0, -1.0, -1.0, //v7

        // Left face
        -1.0,  1.0,  1.0, //v8
        -1.0,  1.0, -1.0, //v9
        -1.0, -1.0, -1.0, //v10
        -1.0, -1.0,  1.0, //v11

        // Right face
         1.0,  1.0,  1.0, //12
         1.0, -1.0,  1.0, //13
         1.0, -1.0, -1.0, //14
         1.0,  1.0, -1.0, //15

        // Top face
         1.0,  1.0,  1.0, //v16
         1.0,  1.0, -1.0, //v17
        -1.0,  1.0, -1.0, //v18
        -1.0,  1.0,  1.0, //v19

        // Bottom face
         1.0, -1.0,  1.0, //v20
         1.0, -1.0, -1.0, //v21
        -1.0, -1.0, -1.0, //v22
        -1.0, -1.0,  1.0, //v23
    ];

    gl.bufferData(gl.ARRAY_BUFFER,           new
Float32Array(cubeVertexPosition),
            gl.STATIC_DRAW);
    cubeVertexPositionBuffer.itemSize = 3;
    cubeVertexPositionBuffer.numberOfItems = 24;

    cubeVertexIndexBuffer = gl.createBuffer();
            gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
cubeVertexIndexBuffer);
    var cubeVertexIndices = [

```

```

        0, 1, 2,          0, 2, 3,      // Front face
        4, 6, 5,          4, 7, 6,      // Back face
        8, 9, 10,         8, 10, 11,    // Left face
        12, 13, 14,       12, 14, 15,    // Right face
        16, 17, 18,       16, 18, 19,    // Top face
        20, 22, 21,       20, 23, 22    // Bottom face
    ];
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,           new
Uint16Array(cubeVertexIndices),
    gl.STATIC_DRAW);
cubeVertexIndexBuffer.itemSize = 1;
cubeVertexIndexBuffer.numberOfItems = 36;
}

```

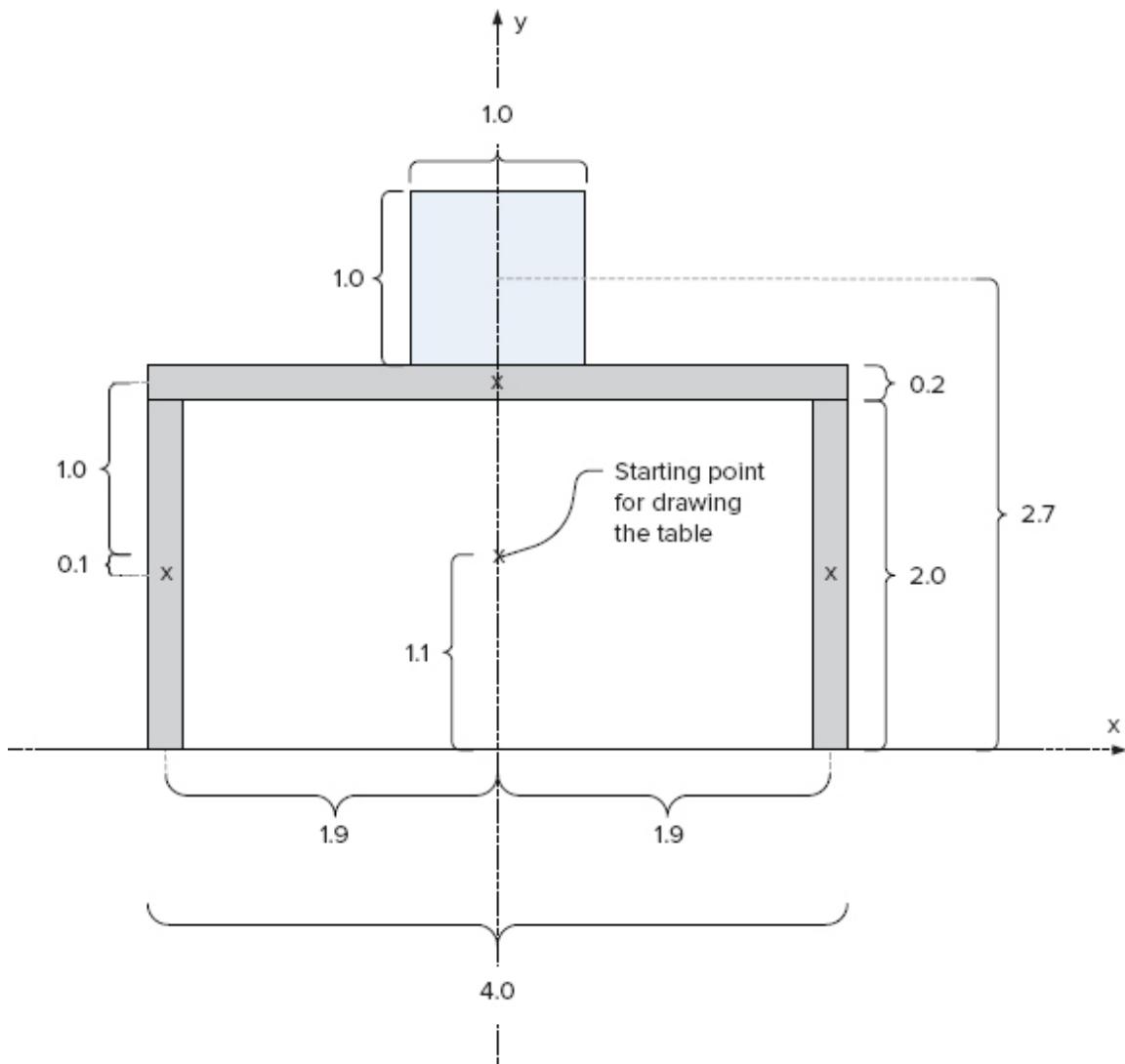
As shown in [Figure 4-15](#), each corner in the cube has three vertices written next to it. All three vertices correspond to the same position. A cube actually only has eight unique vertex positions and in this example, the whole cube only has one single color and no normals are needed, so eight vertices would have been enough. However, in a more generic case when you want to have a different color for each side of the cube or you need normals, you would need three vertices for each corner, since each corner shares three different faces of the cube. This example has 24 vertices, so you can see how it would look in the generic case and be easier for you to modify the code if, for example, you want a unique color for each face of the cube.

[Figure 4-15](#) also shows how one of the triangles that builds up the left face of the cube has clockwise winding and is back-facing. The shaded triangle that builds up the right face of the cube has counterclockwise winding and is front-facing.

Organization of the View Transformation and the Model Transformation

The rest of the code in Listing 4-4 should be fairly easy to follow since it is very close to the snippets that you already looked at in this chapter. [Figure 4-16](#) shows you the dimensions and positions of the plane, the table, and the box.

FIGURE 4-16: The dimensions of the floor, table, and box in Listing 4-4



What is worth commenting on again in the code is the order in which the view transformation and the model transformation are set up. First you set up the view transformation with `mat4.lookAt()`. Then you add the model transformation for the different objects to the `modelview` matrix. In this way, the vertices of your objects can be transformed as if they were multiplied with the model transformation first and then the view transformation. This is according to what you learned about the WebGL transformation pipeline earlier in this chapter.



One way that might help you to really master transformations is to be a bit old fashioned and use a pen and paper. Imagine an object, for example, a small cube, that you want to position and orient so you can look at it from different directions. Write down the transformations that you need to do. Then try it out in code by, for example, editing Listing 4-4. Perform the exercise over and over again until you feel comfortable with the transformations.

SUMMARY

In this chapter, you learned about three compact and useful JavaScript libraries that can help you to handle the matrix and vector mathematics that you need to do in JavaScript. These libraries are:

- Sylvester
- WebGL-mjs
- glMatrix

You saw that WebGL-mjs and glMatrix were quite similar, and both were written with WebGL in mind.

Further, you learned a lot about the different transformations:

- Model transformation
- View transformation
- Modelview transformation
- Projection transformation
- Perspective division
- Viewport transformation

You learned that the modelview transformation is a combination of the model transform and the view transform. You also learned that the order of the transformations is important, and how you could think of a series of transformations either in a grand, fixed coordinate system or with a moving, local coordinate system that follows the transformed object.

Chapter 5

Texturing

WHAT'S IN THIS CHAPTER?

- Learn about texturing and why it is useful in 3D graphics
- How to make your application more robust by handling lost context
- Learn the difference between a 2D texture and a cube map texture
- How to load a texture in WebGL
- Learn about magnification and minification
- Learn about filtering and mipmapping
- The factors you should consider when creating your own textures
- Why same-origin policy is important for your textures in WebGL
- Understand what Cross-Origin Resource Sharing is and how it can be useful for WebGL

Applying texture to a surface is an important operation in WebGL. The idea is to use a color image to represent details of an object that are not available in the geometry of the object. A simple way to describe texturing is that it is the process of mapping images onto geometrical objects.

For example, if you think about a brick wall, it could be modeled in a very simple way with two orange triangles. As you can imagine, this would not look very realistic, but since only a few vertices would be needed to represent the wall, the rendering would be very fast. The opposite would be to make a very realistic model of the wall and use a lot of vertices to be able to represent changes in the geometry or color of the wall. With this approach you could get a very realistic model, but it would be difficult to model it and the rendering would probably take more time.

With texturing, you can have a coarse polygon representing the geometry of the wall. You then apply a color image of the wall to the coarse polygon. As long as the viewer is not too close to the wall, it is possible to get a realistic wall even with very limited geometric details.

The majority of this chapter is about textures, but you will first learn about another important topic that has been ignored so far: the concept of lost context. Lost context affects the stability and robustness of your application.

UNDERSTANDING LOST CONTEXT

The GPU is a shared resource that can be used by several other clients than your WebGL application. WebGL has a concept of lost context that means that if the GPU is taken away from your application, it loses its context.

There are several reasons why your WebGL application could lose its context; one example is if a draw call takes too long to execute, and so the user's system becomes unresponsive or hangs. This is a potential problem even for desktop OpenGL, but since WebGL is running non-trusted code (which via the vertex shader and the fragment shader has low-level access to the GPU), there is a higher risk for problems with WebGL than with desktop OpenGL.

One solution for some operating systems and device drivers is that the device driver detects that the GPU becomes unresponsive. In this case, it resets the GPU, and then the WebGL application can be informed that it has lost its context by sending the `webglcontextlost` event to it.

The examples presented so far in the book have not been designed to take care of the situation where the application loses its context. They have not been set up to listen for the `webglcontextlost` event, and they also contain some constructs that are not recommended to use if you want to have code that is robust in the event your application loses its context.

When a context is lost, it has the following implications:

- The method `gl.isContextLost()` returns true.
- WebGL methods that are declared as void return immediately.
- WebGL methods that can return `null` return `null`.
- The method `gl.getAttributeLocation()` returns `-1` instead of a valid location of the attribute in the vertex shader.
- The method `gl.getError()` returns `gl.CONTEXT_LOST_WEBGL` the first time it is called while the context is lost. After this, it returns `gl.NO_ERROR` until the context has been restored.

- The method `gl.checkFramebufferStatus()` returns `gl.FRAMEBUFFER_UNSUPPORTED`.
- All WebGL methods that start with the “`is`” prefix (such as `isEnabled()`, `isProgram()`, and so on) return `false`.

All of this might sound complicated when you first read it; there certainly are some new concepts to think of if you want to properly handle lost context.

Understanding the Setup Required to Handle Lost Context

When a WebGL application loses its context, the default behavior is that it never gets it back. To continue using the WebGL application, the user must reload the application manually in the browser and hope that the context can now be used again.

You can override this default behavior by registering two event listeners: one in which your application is informed when its context is lost, and one in which your application is informed when it is restored.

To register an event listener that listens for the `webglcontextlost` event, use this code:

```
canvas = document.getElementById("myGLCanvas");
canvas.addEventListener('webglcontextlost',
                      handleContextLost, false);
```

The first argument to the method `addEventListener()` is the name of the event that you want to register the event listener for. The second argument is a listener that should be called when the event occurs. The third argument is a Boolean that specifies whether the event handler should capture events during what is called the capturing phase of the event propagation. In this case, you don’t need to capture any events during the capturing phase, so you should use `false` for this argument. You will learn more about event handlers and event propagation in Chapter 6 and then this last argument will be much clearer.

To register an event listener that can be invoked when your context is restored again, use this code:

```
canvas.addEventListener('webglcontextrestored',
                      handleContextRestored, false);
```

When your event listener for the `webglcontextlost` event is called, you need to prevent the default action, which is that the context will never be restored again. You do this by calling `preventDefault()` on the event, as follows:

```

function handleContextLost(event) {
    event.preventDefault();

    cancelRequestAnimFrame(pwgl.requestId);

    ...

}

```

In addition, you typically want to stop your rendering loop. This might sound odd to you since so far, all the code examples in the book only executed the drawing method once. In general, however, you typically have a scene with objects that can move around. In this case, your drawing routine, which has typically been named `draw()` in the examples so far, must be called repeatedly. You will learn more about this later in the chapter, but for now it is enough to understand that the call to `cancelRequestAnimFrame()` stops the calls to the `draw()` function.

When your context can be restored, your registered event listener is called. When this happens, the state that you have set up for WebGL is reset to the default state. In addition, all resources that you have allocated through WebGL are invalid.

This means that you have to set up your needed state again and also create the textures, buffers, shaders, programs, and other resources that you need again. What this usually means is that you need to call your functions to do the same setup and initialization as you did when your application was started the first time. The last thing that you typically do when your context is restored is to start the rendering loop again. You do this by calling `requestAnimFrame()`. You can see an example of the complete function that handles the `webglcontextrestored` event here:

```

function handleContextRestored(event) {
    setupShaders();
    setupBuffers();
    setupTextures();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    pwgl.requestId = requestAnimFrame(draw, canvas);
}

```

Simulating Context Lost Events

When you have implemented code in your WebGL application to handle the events `webglcontextlost` and `webglcontextrestored`, you should test the

code to make sure it really works. At this stage, an invaluable resource is the debug library `webgl-debug.js` that you looked at already in Chapter 2, where it was introduced to print WebGL errors to the JavaScript console.

When you are working with lost context, the library `webgl-debug.js` can be used to simulate the events `webglcontextlost` and `webglcontextrestored`. To use the library, you first have to include the file `webgl-debug.js` from your WebGL application as you would do with any external JavaScript:

```
<script src="webgl-debug.js"></script>
```

Then a simple way to use the library is to add code similar to this at your setup and initialization code:

```
canvas = document.getElementById("myGLCanvas");
canvas =
WebGLDebugUtils.makeLostContextSimulatingCanvas(canvas);

canvas.addEventListener('webglcontextlost',
                      handleContextLost, false);

canvas.addEventListener('webglcontextrestored',
                      handleContextRestored, false);

...
window.addEventListener('mousedown', function() {
  canvas.loseContext();
});
```

The `call` to the `method` `WebGLDebugUtils.makeLostContextSimulatingCanvas()` creates a wrapper around your original canvas that lets you simulate the `webglcontextlost` and `webglcontextrestored` events. Then you can simply add the event listeners for the two events and, finally, add an event listener to the `mousedown` event. In the event handler for the `mousedown` event, you call the method `canvas.loseContext()` (that is part of the wrapper canvas that you created), which will perform the simulation of the `webglcontextlost` event and trigger so your registered event handlers are called.

As mentioned in Chapter 2, you can download the library `webgl-debug.js` from the Khronos repository:

<https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/debug/webgl-debug.js>

The file `webgl-debug.js` contains some comments that further describe how you can use the different features that are part of the library.

Factors to Consider When Handling Lost Context

The previous examples in the book are not structured to be able to take care of a lost context. It is, of course, recommended that you handle this in a real-world application. Even if this doesn't happen often, you should structure your code to take care of it in case it does happen.

This section covers factors that you should consider to be more robust when a context is lost.



The Khronos WebGL wiki (www.khronos.org/webgl/wiki/HandlingContextLost) contains some really useful information and explanations about lost context. This information was initially collected and explained by Gregg Tavares, who is one of many people who have contributed a lot to WebGL as a technology.

Do Not Add New Properties to WebGL Resource Objects

The initial examples in this book have often added new properties to the different WebGL resource objects that are created from the WebGL API. For example, the `itemSize` and `numberOfItems` are often added to the created `WebGLBuffer` object (as shown in the following code snippet):

```
floorVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, floorVertexPositionBuffer);

var floorVertexPosition = [
    // Plane in y=0
    5.0, 0.0, 5.0, //v0
    5.0, 0.0, -5.0, //v1
    -5.0, 0.0, -5.0, //v2
    -5.0, 0.0, 5.0]; //v3

    gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(floorVertexPosition),
    gl.STATIC_DRAW);
```

```
// THE TWO LINES BELOW ARE BAD IF THE CONTEXT IS LOST!
floorVertexPositionBuffer.itemSize = 3;
floorVertexPositionBuffer.numberOfItems = 4;
```

While this might be a convenient way to organize your code and you probably will see similar code in many real-world WebGL applications, it is not a good way to have your code structured if your context is lost. If your context is lost, the method `gl.createBuffer()` returns `null` and then you get an exception when you try to add the properties `itemSize` and `numberOfItems` to the `floorVertexPositionBuffer` that is `null`. In the same way, you should not add properties to any other WebGL objects, such as texture objects, shader objects, program objects, and so on.

There are many other ways to organize your code. One way is to add the properties to another global object that is not created by WebGL. In this snippet, an object called `pwgl` (an acronym for Professional WebGL) is used to store the corresponding information.

```
pwgl.floorVertexPositionBuffer = gl.createBuffer();
                                gl.bindBuffer(gl.ARRAY_BUFFER,
pwgl.floorVertexPositionBuffer);

var floorVertexPosition = [
    // Plane in y=0
    5.0, 0.0, 5.0, //v0
    5.0, 0.0, -5.0, //v1
    -5.0, 0.0, -5.0, //v2
    -5.0, 0.0, 5.0]; //v3

                                gl.bufferData(gl.ARRAY_BUFFER,           new
Float32Array(floorVertexPosition),
                                gl.STATIC_DRAW);

pwgl.FLOOR_VERTEX_POS_BUF_ITEM_SIZE = 3;
pwgl.FLOOR_VERTEX_POS_BUF_NUM_ITEMS = 4;
```

Check Lost Context When You Check Shader Compilation

When you check that your shaders compiled successfully by calling `gl.getShaderParameter()`, you should also check that you have not lost the context by calling `gl.isContextLost()`. The reason for this is to prevent showing a compilation failed alert when the context is lost.

The code that does not check for lost context when the compilation status is checked looks like this:

```
gl.shaderSource(shader, shaderSource);
gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
}
```

You can make a simple change that results in a more robust behavior when your context is lost by adding a call to `gl.isContextLost()`. This is shown in the following code snippet:

```
gl.shaderSource(shader, shaderSource);
gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS) &&
    !gl.isContextLost()) { // To avoid alert() when context
is lost
    alert(gl.getShaderInfoLog(shader));
    return null;
}
return shader;
```

Check Lost Context When You Check Shader Linking

In the same way that you should check for lost context when you check the shader compilation, you should also check for lost context when you check for linking of the program. The original code that does not check the lost context when you check the linking typically looks like this:

```
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS))
{
    alert("Failed to setup shaders");
}
```

The updated code that checks the lost context is as follows:

```
var shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
```

```

gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)
&&
!gl.isContextLost()) {
alert("Failed to setup shaders");
}

```

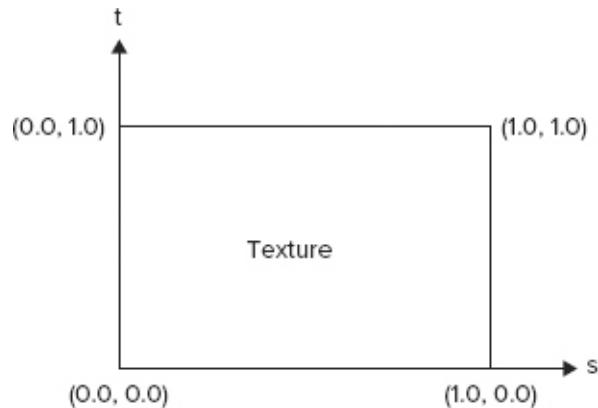
INTRODUCING 2D TEXTURES AND CUBEMAP TEXTURES

The most common form of texture in WebGL is referred to as a 2D texture. The most basic form of a 2D texture uses a single image as a texture, as in the previous example with the brick wall. Even if an image is most commonly used, WebGL can also use a canvas element, a video element, an `ImageData` object, or raw data in a typed array as input to a 2D texture.

Regardless of what the input to the 2D texture is, a coordinate system for the 2D texture is needed so that it is possible to specify where **texels** should be *sampled* from the texture. A *texel* is simply a pixel of the texture, but it is usually called a texel to differentiate it from the pixels on the screen. To sample from the texture basically means to fetch a texel from the texture.

[Figure 5-1](#) shows how the texture coordinates are defined for a 2D texture. Texture coordinates for a 2D texture are usually called (s, t) or (u, v) . The lower-left corner of the texture is defined as the origin and has the coordinates $(0.0, 0.0)$. The upper-right corner of the texture always has the coordinates $(1.0, 1.0)$, regardless of whether or not the texture is a square. The s -axis is horizontal and is pointing to the right and the t -axis is vertical and is pointing upwards.

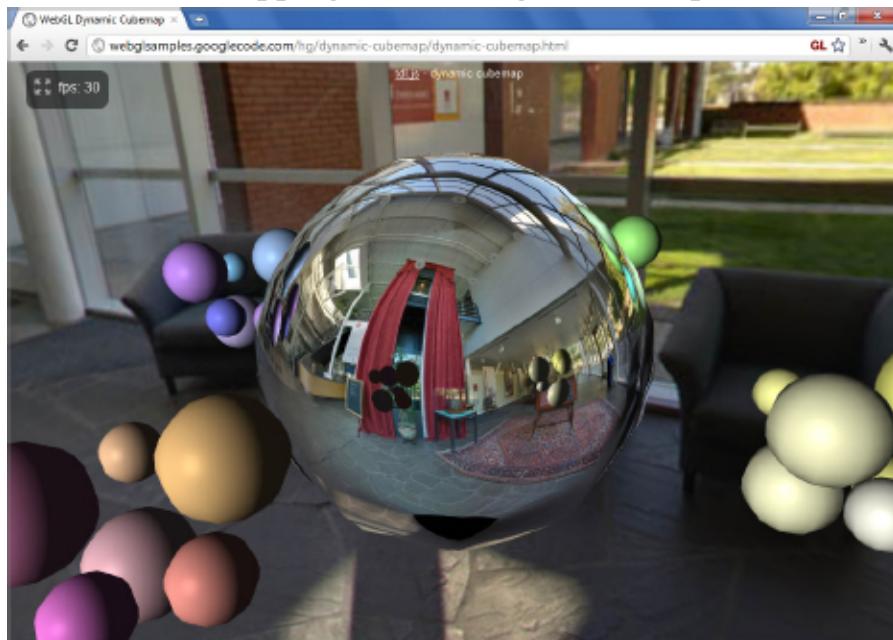
[**FIGURE 5-1:**](#) Texture coordinates for a 2D texture



In addition to the usual 2D textures, WebGL also supports a feature called *cube map* textures. A cube map texture is handled as a single texture object but is composed of six square textures, one for each face of a cube. Cube map textures are useful for *environment mapping*, which is a technique that produces reflections of the scene (or environment) on very shiny objects.

[Figure 5-2](#) shows an example of environment mapping that is implemented using a cube map. This is the same example you used in Chapter 2, when you learned about the WebGL Inspector. To really see what this demo looks like you should use your web browser and look at the online version. The demo, called “Dynamic Cubemap,” is developed by Gregg Tavares and is available at <http://webglsamples.googlecode.com>.

FIGURE 5-2: The demo “Dynamic Cubemap” by Gregg Tavares is a great example of environment mapping that is using a cube map



Cube maps can also be used to create a *skybox*, which is a technique that is common in games. The idea is that faraway objects such as the sky, distant mountains, or distant buildings are part of a texture that you render on the faces of a cube. If the cube is big enough, you can create the illusion of distant 3D surroundings.

LOADING YOUR TEXTURES

To be able to apply a texture to your geometry, the texture first needs to be loaded. The following sections show you how to do this when your texture is a regular image in, for example, PNG, JPEG, or GIF format, and you want to use this image as input for a 2D texture.

When you understand the basics, it is not difficult to change the input for the texture to a canvas element or a video element, and you will be offered some hints of how to change the input when going through this example. Using a cube map is quite similar to using a 2D texture, so also in this case, the best approach is to first learn how a 2D texture works and then learn how a cube map is different.

Creating a WebGLTexture Object

One of the first steps that you normally perform when you want to use textures in WebGL is to create a `WebGLTexture` object for each of the textures you need. You do this by calling the code as follows:

```
var texture = gl.createTexture();
```

The `WebGLTexture` object is a container object that is used as a reference to the texture and the settings that are related to the handling of this texture. If you have used OpenGL or OpenGL ES, the method `gl.createTexture()` in WebGL corresponds to the function `glGenTextures()` in OpenGL and OpenGL ES.

There is also a method to explicitly delete a `WebGLTexture` object. For example, if you want to delete a `WebGLTexture` object named `texture`, you would call the code as follows:

```
gl.deleteTexture(texture);
```

Note that you don't have to call `gl.deleteTexture()` when you have finished using a texture. The texture object will be deleted when the `WebGLTexture` object is destroyed by the JavaScript garbage collection. The

method only gives you a greater control over when the texture object is destroyed.

Binding Your Texture

Before you can do anything with the texture object you have created, you need to bind it as the current texture object. For example, to bind a texture object named `texture` as a 2D texture, you would call the following code:

```
gl.bindTexture(gl.TEXTURE_2D, texture);
```

The call to `gl.bindTexture()` tells WebGL that from now on this is the texture object that you want to work with. The method has a similar meaning to your `WebGLTexture` object as the method `gl.bindBuffer()` has to your `WebGLBuffer` object.

Loading the Image Data

When you have bound your texture object, you can load the texture data into it, which means that you upload the texture data to the GPU (or a memory that the GPU can access). The method that uploads the texture data to the GPU is called `gl.texImage2D()` and is described in the next section. The method can take the input texture data in different formats, but the version that you typically want to use if you have the image on a regular file (such as a PNG, GIF, or JPEG) takes the texture data as an HTML DOM `Image` object. So before you call `gl.texImage2D()`, you need to have the data in an `Image` object.

An `Image` object is implicitly created for each `` tag in an HTML document. But you can also create an `Image` object explicitly by calling this code:

```
image = new Image();
```

This creates an `Image` object, but without any image data loaded into it. To load the image data into the `Image` object, you can set the `src` property of the `Image` object to the URL of the image you want to load. As soon as you assign the URL to the `src` property, the image is loaded asynchronously.

To know when the image has finished loading, you use the `onload` event that occurs immediately after the image has finished loading. The following code snippet creates an `Image` object and loads the image data:

```
function loadImageForTexture(url, texture) {  
    var image = new Image();  
    image.onload = function() {
```

```

pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(i
mage), 1);
    textureFinishedLoading(image, texture);
}
pwgl.ongoingImageLoads.push(image);
image.src = url;
}

```

When the image is loaded and the `onload` event triggers the anonymous function, the call to `textureFinishedLoading()` takes you to the next step where the image data is uploaded to the GPU.

The pattern with assigning the `onload` event handler to the `Image` object is very common when loading data into an `Image` object with JavaScript. It is used in this case when you want to use the `Image` object as input to a texture in WebGL, but similar code is also common when you want to draw the `Image` object on an HTML5 2D canvas.

The function `loadImageForTexture()` in this example also has two lines of code that are specific to the handling of the WebGL lost context. This is to keep track of which image requests are sent in case you get a lost context before the image has finished loading and you get the `onload` event. Before you assign the URL to the `src` attribute of the `Image` object, you remember the image by adding it to an array by calling:

```
pwgl.ongoingImageLoads.push(image);
```

Once the `onload` event is triggered, you remove the image from the array by calling the following code:

```
pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(i
mage), 1);
```

If you get a lost context, you can go through the array of ongoing image loads and ignore them by removing their `onload` handler as follows:

```

function handleContextLost(event) {
    event.preventDefault();
    cancelRequestAnimationFrame(pwgl.requestId);

    // Ignore all ongoing image loads by removing
    // their onload handler
    for (var i = 0; i < pwgl.ongoingImageLoads.length; i++) {
        pwgl.ongoingImageLoads[i].onload = undefined;
    }
    pwgl.ongoingImageLoads = [];
}

```

The Power of Two Texture Sizes

As you learn how to load your image data, it's important to understand the acceptable sizes for the images. Very often, developers choose to have textures with a width and a height that is a power of two (1, 2, 4, 8, 16, 32, 64, 128, and so on). Another way to express it is that the texture has a size of $2^m \times 2^n$, where m and n are non-negative integers.

One reason for this is that older GPUs only supported textures where the width and height was a power of two. Desktop OpenGL 2.0 and later actually has support for non-power of two (NPOT) textures. In OpenGL ES 2.0 and WebGL, it is possible to use NPOT textures, but these are associated with the following restrictions:

- You cannot use mipmapping if you use an NPOT texture.
- The only repeat mode that is allowed is `gl.CLAMP_TO_EDGE`.

You will understand what these restrictions mean when you have read about mipmapping and texture coordinate wrapping later in this chapter.

Uploading the Texture to the GPU

To upload the texture to the GPU, you use the method `gl.texImage2D()`. This method exists in several different versions that have different arguments, depending on what type of data you want to use for your texture. The formal prototypes for three of the methods are as follows:

```
void texImage2D(GLenum target, GLint level, GLenum
internalformat,
GLenum format, GLenum type, HTMLImageElement image) raises
(DOMException)

void texImage2D(GLenum target, GLint level, GLenum
internalformat,
GLenum format, GLenum type, HTMLCanvasElement canvas) raises
(DOMException)

void texImage2D(GLenum target, GLint level, GLenum
internalformat,
GLenum format, GLenum type, HTMLVideoElement video) raises
(DOMException)
```

For the first version, the texture data is an HTML DOM `Image` object. The second method takes an HTML5 canvas element as input data for the texture. The last method takes a video element as input.

The version that takes an HTML `Image` object as input is probably the version that you will use most often. To upload the `Image` object to the GPU, you can call the method `gl.texImage2D()` as shown here:

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,  
             gl.UNSIGNED_BYTE, image);
```

The first argument states that the target is a 2D texture. The second argument specifies something called the mipmap level. Mipmapping is discussed later in this chapter. For now, just accept that you should set this argument to 0 (zero) in this case.

The third argument is called internal format and the fourth argument is called format. These two arguments must always have the same value in WebGL. In this case, `gl.RGBA` is used to specify that the texture should be stored with a red, green, blue, and alpha channel for each texel in the texture.

The fifth argument specifies the type in which you want to store the data for the texels. Using `gl.UNSIGNED_BYTE` specifies that you want to use one byte for each channel of the red, green, blue, and alpha information. This means that each texel occupies four bytes of memory.

The last argument takes the HTML DOM `Image` object that you have loaded with image data. As you saw on the formal prototypes, this argument can also consist of an HTML5 canvas element or a video element.

In the following example, the format `gl.RGBA` is used together with the type `gl.UNSIGNED_BYTE` as arguments to `gl.texImage2D()`:

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,  
             gl.UNSIGNED_BYTE, image);
```

[Table 5-1](#) shows the other valid combinations that you can use for arguments 3, 4, and 5.

TABLE 5-1: Valid Combinations of Format and Type

FORMAT	TYPE	BYTES PER TEXEL
<code>gl.RGBA</code>	<code>gl.UNSIGNED_BYTE</code>	4
<code>gl.RGB</code>	<code>gl.UNSIGNED_BYTE</code>	3
<code>gl.RGBA</code>	<code>gl.UNSIGNED_SHORT_4_4_4_4</code>	2
<code>gl.RGBA</code>	<code>gl.UNSIGNED_SHORT_5_5_5_1</code>	2
<code>gl.RGB</code>	<code>gl.UNSIGNED_SHORT_5_6_5</code>	2
<code>gl.LUMINANCE_ALPHA</code>	<code>gl.UNSIGNED_BYTE</code>	2
<code>gl.LUMINANCE</code>	<code>gl.UNSIGNED_BYTE</code>	1
<code>gl.ALPHA</code>	<code>gl.UNSIGNED_BYTE</code>	1

The first column specifies the format or internal format that can be used as arguments 3 and 4 in the method `gl.texImage2D()`. As already explained, `gl.RGBA` means that each texel has a red, green, blue, and alpha channel. The format `gl.RGB` has a red, green, and blue channel but no alpha channel. `gl.LUMINANCE_ALPHA` has luminance (brightness) and an alpha channel. The format `gl.LUMINANCE` has only the luminance channel, and `gl.ALPHA` has only an alpha channel.

The second column includes the type that can be used as argument 5 to `gl.texImage2D()`. When the type is set to `gl.UNSIGNED_BYTE`, it means that each channel of the specified format occupies one byte. You can see the total number of bytes occupied for each texel in the third column of the table.

The other types in the table are a way to get a more compact representation of the data for one texel. The type `gl.UNSIGNED_SHORT_4_4_4_4` means that four bits are used for each channel in the `RGBA` format. For `gl.UNSIGNED_SHORT_5_5_5_1`, five bits are used for red, five bits for green, five bits for blue, and one bit for the alpha channel. Finally, for the type `gl.UNSIGNED_SHORT_5_6_5`, which is used together with the format `gl.RGB`, five bits are used for red, six bits for green, and five bits for blue.



When uploading the texture to the GPU, the texture is stored in a memory that the GPU uses for textures. This can be a special video memory that the GPU can access faster than the usual system memory, but for some systems it can be a part of the regular system memory that is dedicated to the GPU. During the texture upload, a process called swizzling is often performed. Swizzling is a technique that rearranges the bytes of the texture in memory in order to improve cache performance.

Specifying Texture Parameters

There are several parameters that you can set that affect how the textures are used during rendering. These parameters are set with the method `gl.texParameteri()`. The method and its different arguments are presented in this section, but you will gradually learn what the arguments mean in the rest of the chapter.

The prototype for the method is as follows:

```
void texParameteri(GLenum target, GLenum pname, GLint param)
```

The arguments can have the following values:

- target can be either `gl.TEXTURE_2D` or `gl.TEXTURE_CUBE_MAP`.
- pname specifies the parameter that you want to set. The possible parameters are as follows:
 - `gl.TEXTURE_MAG_FILTER`
 - `gl.TEXTURE_MIN_FILTER`
 - `gl.TEXTURE_WRAP_S`
 - `gl.TEXTURE_WRAP_T`
- param contains the value for the parameter you want to set. The allowed value depends on which `pname` you specify as the second argument.

If `pname` is `gl.TEXTURE_MAG_FILTER`, the `param` can be one of these values:

- `gl.NEAREST`
- `gl.LINEAR`

If `pname` is `gl.TEXTURE_MIN_FILTER`, the `param` can be one of these values:

- `gl.NEAREST`
- `gl.LINEAR`
- `gl.NEAREST_MIPMAP_NEAREST`
- `gl.LINEAR_MIPMAP_NEAREST`
- `gl.NEAREST_MIPMAP_LINEAR`
- `gl.LINEAR_MIPMAP_LINEAR`

If `pname` is `gl.TEXTURE_WRAP_S` or `gl.TEXTURE_WRAP_T` the `param` can have one of these values:

- `gl.REPEAT`
- `gl.CLAMP_TO_EDGE`
- `gl.MIRRORED_REPEAT`

As previously mentioned, the details of all the arguments are not described here. Instead, the method will be revisited when the relevant topic in this chapter is described.

Understanding the Complete Procedure of Loading a Texture

Now it is time to look at the complete snippet of source code that creates a texture object, creates an `Image` object, and then loads the image data into the

`Image` object. The following code snippet contains a few lines of code that have not been explained yet; they will be explained shortly.

```
function textureFinishedLoading(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

        gl.texImage2D(gl.TEXTURE_2D,      0,      gl.RGBA,      gl.RGBA,
gl.UNSIGNED_BYTE,
                    image);

    gl.texParameteri(gl.TEXTURE_2D,      gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D,      gl.TEXTURE_MIN_FILTER,
gl.NEAREST);

    gl.bindTexture(gl.TEXTURE_2D, null);
}

function loadImageForTexture(url, texture) {
    var image = new Image();
    image.onload = function() {

pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(i
mage), 1);
    textureFinishedLoading(image, texture);
}
    pwgl.ongoingImageLoads.push(image);
    image.src = url;
}

function setupTextures() {
    // Texture for the table
    pwgl.woodTexture = gl.createTexture();
    loadImageForTexture("wood_128x128.jpg", pwgl.woodTexture);
}
```

When the code is executed, the function `setupTextures()` is the first of the three functions that is called. You will typically call this function as part of the setup and initialization of your WebGL application. The function creates a `texture` object by calling `gl.createTexture()`. It then calls the function `loadImageForTexture()` and sends in the relative URL to the image you want to use for your texture as the first argument and the `texture` object as the second argument.

The function `loadImageForTexture()` creates an `Image` object that will hold the data for the texture. It sets up the `onload` event handler and keeps track of

which image loads are ongoing by using the array `ongoingImageLoads` to be able to handle a lost context event.

When the image has finished loading, the `onload` event is triggered on the `Image` object and the function `textureFinishedLoading()` is called with the `Image` object as the first argument and the texture object as the second argument. The first thing that this function does is to bind the texture object as the current texture by calling `gl.bindTexture()`. Then, before the texture data is loaded into the texture, there is the following call to a method that was not explained previously:

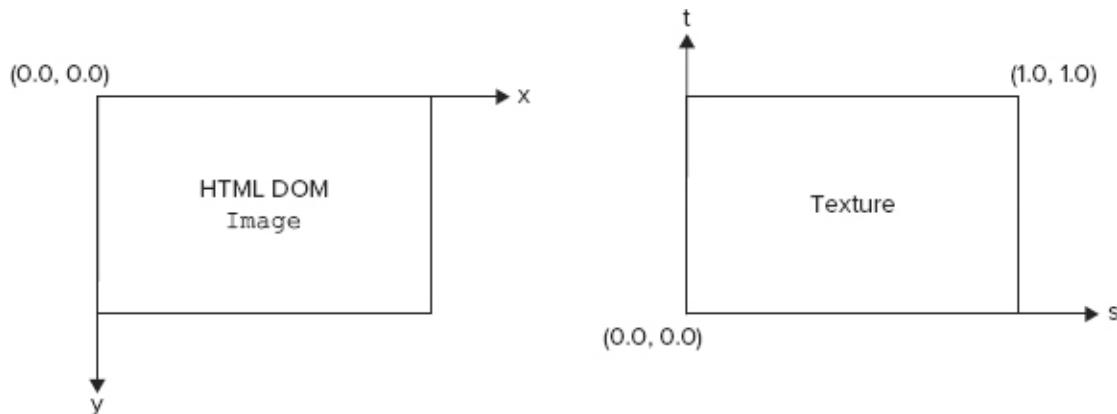
```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

The method `gl.pixelStorei()` affects how the subsequent call to `gl.texImage2D()` behaves. When `gl.pixelStorei()` is called with the first argument set to `gl.UNPACK_FLIP_Y_WEBGL` and the second argument set to `true`, the image is flipped around the horizontal axis when it is read with `gl.texImage2D()`.

The reason to do this is because the coordinate system that is used for textures in WebGL (and all versions of OpenGL as well) is different from the coordinate system used for the `Image` object. A texture in WebGL has the origin $(0.0, 0.0)$ in the bottom-left corner of the texture with the horizontal axis pointing to the right and the vertical axis pointing upwards.

For an `Image` object, the origin $(0.0, 0.0)$ is located in the top-left corner of the image. The horizontal axis is pointing to the right, but the vertical axis points downwards. [Figure 5-3](#) shows the coordinate system for the HTML DOM `Image` object to the left and the coordinate system for a WebGL texture to the right.

FIGURE 5-3: Coordinates for an HTML DOM Image object and a WebGL texture



After you have specified that the texture data should be flipped, the method `gl.texImage2D()` is called to upload the texture data to the GPU. Finally, there are two calls to the method `gl.texParameteri()` to set the parameters `gl.TEXTURE_MAG_FILTER` and `gl.TEXTURE_MIN_FILTER` like this:

```
gl.texParameteri(gl.TEXTURE_2D,           gl.TEXTURE_MAG_FILTER,  
gl.NEAREST);  
gl.texParameteri(gl.TEXTURE_2D,           gl.TEXTURE_MIN_FILTER,  
gl.NEAREST);
```

These calls specify how the texture mapping should be done when there is not a one-to-one mapping between the texels in the texture and the pixels on the screen. The first call specifies how the mapping should be done when the texture is magnified (or stretched). The second call specifies how the mapping should be done when the texture is minified (or shrunken).

You will learn more about minification and magnification later in this chapter.

Creating a Texture Object and Loading the Data

The following steps summarize how to create a texture object and load the data:

1. Create a `WebGLTexture` object using `gl.createTexture()`.
2. Create an HTML DOM `Image` object by calling `new Image()`.
3. Set up the `onload` handler of the `Image` object so you can call a function that can bind your texture object and upload the texture data to the GPU.
4. Set the `src` property of the `Image` object to the URL of the image you want to load into the texture.
5. When the image data has finished loading into your `Image` object, the `onload` event is triggered and you can bind the texture with `gl.bindTexture()`.
6. If you don't want the image upside down when it is used as a texture, call the following code:

```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

7. When the texture object is bound, you can upload the data to the GPU by calling `gl.texImage2D()`.
8. Use the method `gl.texParameteri()` to specify any texture parameter that you want to set. Specifically, you must make sure to set the minification filter (`gl.TEXTURE_MIN_FILTER`) to `gl.NEAREST` or `gl.LINEAR` if you have not loaded a complete mipmap chain. You can set the minification filter to `gl.NEAREST` with the following code:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,  
gl.NEAREST);
```

DEFINING YOUR TEXTURE COORDINATES

Earlier in this chapter, you learned that texture mapping could be seen as a way to map an image onto your geometric objects. In the previous sections, you learned how the WebGL API was used to upload this image to the GPU so it could be used as a texture. In this section, you will learn how to decide where on your geometry the different parts of the texture should be mapped. You do this by specifying texture coordinates for your vertices.

In Chapter 3, you learned how a different color could be specified for each vertex in a triangle. In the following snippet, you can once again see the code snippet that is used to set up the vertex color in a `WebGLBuffer` object. (This code snippet has been updated to handle the lost context compared to the corresponding code in Chapter 3.)

```
// This code snippet is listed here so you can  
// compare with setting up colors.  
  
pwgl.triangleVertexColorBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER,  
pwgl.triangleVertexColorBuffer);  
var colors = [  
    1.0, 0.0, 0.0, 1.0, //v0  
    0.0, 1.0, 0.0, 1.0, //v1  
    0.0, 0.0, 1.0, 1.0 //v2  
];  
  
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),  
gl.STATIC_DRAW);  
pwgl.TRIANGLE_VERTEX_COLOR_BUF_ITEM_SIZE = 4;  
pwgl.TRIANGLE_VERTEX_COLOR_BUF_NUM_ITEMS = 3;
```

In addition to setting up the `WebGLBuffer` object with the colors, you have to bind the buffer with `gl.bindBuffer()` and call `gl.vertexAttribPointer()` to connect the buffer to the correct attribute in the vertex shader before you make your call to `gl.drawArrays()` or `gl.drawElements()`. This snippet was already shown in Chapter 3, but it's provided here again for your convenience:

```

// This code snippet is listed here so you can
// compare with setting up colors.

gl.bindBuffer(gl.ARRAY_BUFFER,
pwgl.triangleVertexColorBuffer);
gl.vertexAttribPointer(pwgl.vertexColorAttributeLoc,
pwgl.TRIANGLE_VERTEX_COLOR_BUF_ITEM_SIZE,
gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0,
pwgl.TRIANGLE_VERTEX_COLOR_BUF_NUM_ITEMS);

```

In a similar way to how the colors can be specified for the vertices when you set up your buffers, you can instead specify a texture coordinate for each vertex. Since you understand how colors are specified for the vertices, it is straightforward to learn how you specify your texture coordinates.

Here is the code to set up the texture coordinates:

```

// Setup buffer with texture coordinates
pwgl.triangleVertexTextureCoordinateBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER,
pwgl.triangleVertexTextureCoordinateBuffer);
var textureCoordinates = [
  0.0, 0.0, //v0
  1.0, 0.0, //v1
  0.5, 1.0, //v2
];

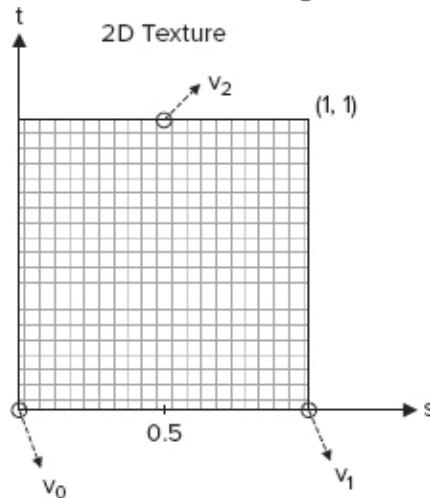
gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(textureCoordinates),
gl.STATIC_DRAW);
pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 3;

```

As you can see, this code is almost identical to the code you used to set up the colors. WebGL does not know if the buffers contain colors, texture coordinates, or something else. It only sees a buffer containing numbers. You can see that texture coordinates are stored in the array named `textureCoordinates` and then uploaded to the `WebGLBuffer` object.

The buffer with vertex positions is not shown in this snippet, but you can imagine that you have three vertex positions for the triangle and that the texture coordinates match these positions, as shown in [Figure 5-4](#).

FIGURE 5-4: Texture coordinates for a triangle



One minor difference between the code that sets up the buffers with the texture coordinates and the code that sets up the buffers with color information is that different names on the variables were chosen for the code to be easier to understand. The other minor difference is that the color in this case is specified with four floats per vertex while the texture coordinates are specified with two floats per vertex.

Before you can do any drawing based on the texture coordinates in the buffers, you need to bind the buffer with the texture coordinates and call `gl.vertexAttribPointer()` to specify how your data with texture coordinates is organized and to which attribute in the vertex shader the buffer should be connected. In addition, you need to make sure you have the correct texture bound by calling `gl.bindTexture()`. This code, which typically would be part of one of your drawing functions, is as follows:

```

gl.bindBuffer(gl.ARRAY_BUFFER,
pwgl.tringleVertexTextureCoordinateBuffer);
gl.vertexAttribPointer(pwgl.vertexTextureAttributeLoc,
pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_ITEM_SIZE,
gl.FLOAT, false, 0, 0);

gl.bindTexture(gl.TEXTURE_2D, pwgl.woodTexture);

gl.drawArrays(gl.TRIANGLES, 0,
pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_NUM_ITEMS);

```

In this code, the variable `pwgl.tringleVertexTextureCoordinateBuffer` is the `WebGLBuffer` object that is loaded with the texture coordinates for each

vertex (as shown earlier in this section). The first argument to the method `gl.vertexAttribPointer()`, which is `pwgl.vertexPositionAttributeLoc`, contains the attribute location in the shader program. This location has typically been queried with the method `gl.getAttribLocation()` as you have seen many times before. However, this call is not shown in this code snippet.

USING YOUR TEXTURES IN SHADERS

You have now learned how to create your texture object and upload the texture data to the GPU. In addition, you have learned how to specify the texture coordinates in a `WebGLBuffer` object.

In this section, you will learn how to use the textures in the shaders. As a repetition, the vertex shader and fragment shader that only handles regular colors are first shown here:

```
<script id="shader-vs" type="x-shader/x-vertex">
    // vertex shader that uses colors, NOT textures
    attribute vec3 aVertexPosition;
    attribute vec4 aVertexColor;
    varying vec4 vColor;

    void main() {
        vColor = aVertexColor;
        gl_Position = vec4(aVertexPosition, 1.0);
    }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
    // Fragment shader that uses colors, NOT textures
    precision mediump float;

    varying vec4 vColor;
    void main() {
        gl_FragColor = vColor;
    }
</script>
```

The vertex shader gets the color value in the attribute named `aVertexColor` from the `WebGLBuffer` object that you load with the colors from your JavaScript. The vertex shader normally just assigns the attribute to a varying

variable that is then linearly interpolated by the WebGL pipeline and then read in the fragment shader. In the fragment shader, the varying `vColor` is assigned to the special variable `gl_FragColor` that contains the color of the fragment when the fragment shader is finished with it.

Now that you have refreshed your previous knowledge, the sample vertex shader and fragment shader that use textures are as follows:

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec2 aTextureCoordinates;

    varying vec2 vTextureCoordinates;

    void main() {
        gl_Position = vec4(aVertexPosition, 1.0);
        vTextureCoordinate = aTextureCoordinates;
    }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
    precision mediump float;

    varying vec2 vTextureCoordinates;
    uniform sampler2D uSampler;
    void main() {
        gl_FragColor = texture2D(uSampler, vTextureCoordinates);
    }
</script>
```

As shown here, the code in the vertex shader is very similar to the vertex shader where colors were used instead of textures. The difference is that the attribute that was used for color information has now been replaced with an attribute called `aTextureCoordinates` of the type `vec2` that gets the texture coordinates as input. There is also a varying variable called `vTextureCoordinates` that sends the texture coordinates to the fragment shader.

In the fragment shader, the varying variable `vTextureCoordinates` is read and used to fetch texels from the texture. The special uniform variable `uSampler` of type `sampler2D` is declared. The sampler is used to indicate which texture image unit should be accessed. (You will learn more about texture image units in the next section.)

The value for the sampler uniform is set from JavaScript with the method `gl.uniform1i()` and the value that is set should match the texture image unit to which your texture is bound. For example, if you have the texture that you want

to use bound to the texture image unit `gl.TEXTURE0`, you use the method `gl.uniform1i()` to set the `uSampler` to have the value 0 (zero). Here you see some code fragments that should make it clearer how the sampler and the texture image unit are related:

```
// Get the location of the uniform uSampler
pwgl.uniformSamplerLoc = gl.getUniformLocation(shaderProgram,
                                              "uSampler");
...

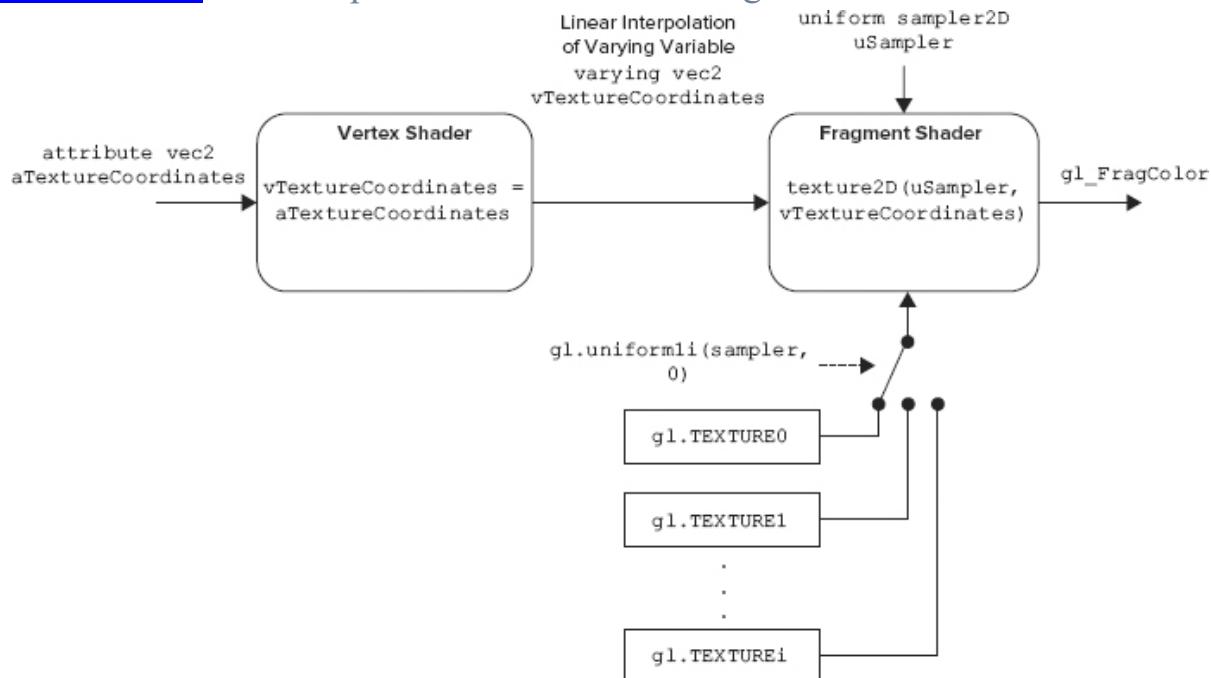
// Bind the texture to texture unit gl.TEXTURE0
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture);

...

// Set uSampler in fragment shader to have the value 0
// so it matches the texture unit gl.TEXTURE0
gl.uniform1i(pwgl.uniformSamplerLoc, 0);
```

[Figure 5-5](#) shows a conceptual diagram of how the texturing works with the different texture units. You can see how setting the sampler uniform with `gl.uniform1i()` selects which texture image unit should be used by setting the value of the sampler uniform variable `uSampler`.

FIGURE 5-5: A conceptual overview of texturing



Working with Texture Image Units

WebGL can have several texture image units (which are sometimes just called texture units). These are named `gl.TEXTURE0`, `gl.TEXTURE1`, `gl.TEXTURE2`, and so on.

The number of supported texture image units depends on the implementation but can be queried by using the method `gl.getParameter()` with the argument `gl.MAX_TEXTURE_IMAGE_UNITS`, as shown here:

```
var nbrOfTextureImageUnits =  
    gl.getParameter(gl.MAX_TEXTURE_IMAGE_UNITS);
```

When you work with textures, you always work with one of these texture image units. You can set which one you want to work with by calling the method `gl.activeTexture()`. For example, if you want to work with `gl.TEXTURE1`, you call the code:

```
gl.activeTexture(gl.TEXTURE1);
```

If you don't call the method `gl.activeTexture()` to set the active texture image unit, it will, by default, be `gl.TEXTURE0`.

You can check which texture image unit is the active one by calling:

```
var activeTextureUnit = gl.getParameter(gl.ACTIVE_TEXTURE);
```

Here you should note that the returned value is not zero for `gl.TEXTURE0`, one for `gl.TEXTURE1`, and so on. Instead, these values are defined by the WebGL specification in the following way:

```
/* TextureUnit */  
const GLenum TEXTURE0 = 0x84C0;  
const GLenum TEXTURE1 = 0x84C1;  
const GLenum TEXTURE2 = 0x84C2;  
const GLenum TEXTURE3 = 0x84C3;  
const GLenum TEXTURE4 = 0x84C4;  
const GLenum TEXTURE5 = 0x84C5;  
const GLenum TEXTURE6 = 0x84C6;  
const GLenum TEXTURE7 = 0x84C7;  
const GLenum TEXTURE8 = 0x84C8;  
const GLenum TEXTURE9 = 0x84C9;  
const GLenum TEXTURE10 = 0x84CA;  
const GLenum TEXTURE11 = 0x84CB;  
  
...  
  
const GLenum TEXTURE29 = 0x84DD;  
const GLenum TEXTURE30 = 0x84DE;  
const GLenum TEXTURE31 = 0x84DF;
```

The exact values of the different units are not that important to you, but the point is that if you want to ask for which texture image unit is active and then execute code dependent on this, then you should, of course, test against the symbolic values (`gl.TEXTURE0`, `gl.TEXTURE1`, and so on) in a way similar to this:

```
if (activeTextureUnit == gl.TEXTURE0) {  
  
    // Do something  
  
}  
else if (activeTextureUnit == gl.TEXTURE1) {  
  
    // Do something else  
  
}
```

WORKING WITH TEXTURE FILTERING

Assume that you have a texture that has the size 512×512 texels. Also assume that this texture is applied to a square that, when projected on the screen, has approximately the same size as the texture. In this case, the textured square will look similar to the original image.

In a general 3D scene, you might have objects that are sometimes closer and sometimes farther away from the camera. With perspective projection, this also means that when the objects are closer to the camera, their projection on the screen will be bigger, and when the objects are farther away, their projection on the screen will be smaller.

If the projected object on the screen is covering an area containing many more pixels than the original image that is used for the texture, the texture needs to be stretched. This is called *magnification*. The opposite is when the projected object on the screen covers an area that contains fewer pixels than the original image. In this case, the texture needs to be reduced in size. This is referred to as *minification*.

The general process of calculating the fragment color when a texture has been magnified or minified is called *texture filtering*. It is possible to affect how the texture filtering should be done by calling the method `gl.texParameteri()`.

You have already seen the prototype for this method, but for your convenience, it is shown here again:

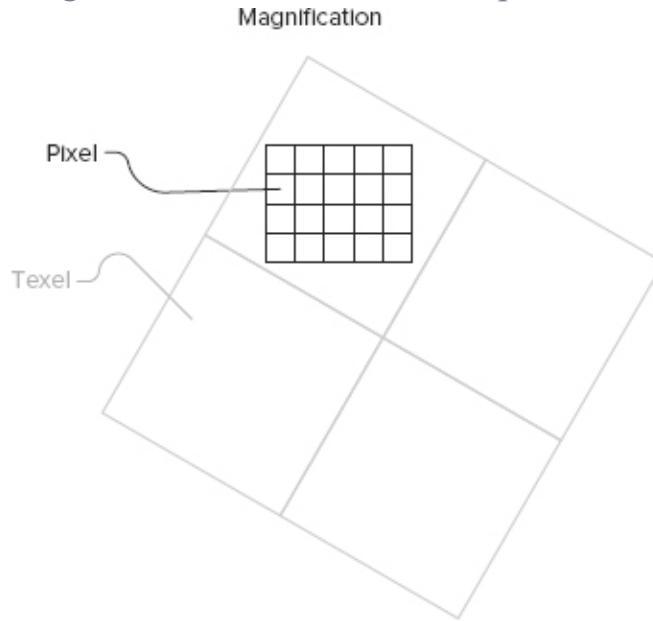
```
void texParameteri(GLenum target, GLenum pname, GLint param)
```

To specify the magnification filter, you set the second argument to `gl.TEXTURE_MAG_FILTER`, and to specify the minification filter, you set the second argument to `gl.TEXTURE_MIN_FILTER`. You will learn more about magnification and minification and the different filters that you can use in the following sections.

Understanding Magnification

[Figure 5-6](#) shows an example of how the pixels and the texels relate in the case where you have magnification. The texture has been magnified or stretched so one texel corresponds to several on-screen pixels.

FIGURE 5-6: For magnification, each texel corresponds to many pixels



The texture filtering has to decide which color a pixel (or actually a fragment) should have. For magnification, you can choose between two filters. The most basic filter is called *nearest neighbor* and is specified with the name `gl.NEAREST`. This means that you set the magnification filter to nearest neighbor by calling the following code:

```
gl.texParameteri(gl.TEXTURE_2D,  
gl.NEAREST);
```

```
gl.TEXTURE_MAG_FILTER,
```

The nearest neighbor filter works by taking the color value of the nearest texel to each texture coordinate. This filtering is fast since only a single texel is used as input for each texture coordinate.

If the texture is stretched a lot, this filtering results in a blocky appearance. This is sometimes referred to as *pixelation*. If you look at [Figure 5-6](#) again, you can see that since many of the pixels in this figure fall within one texel, many pixels will get the same color.

The other filter alternative that you have for magnification is called *linear filtering* or *bilinear filtering* and is specified with the name `gl.LINEAR`. Linear filtering does not only use one texel as input to the color for each texture coordinate. Instead, it takes a weighted average of the four texels surrounding the texture coordinate. To set the magnification filter to linear filtering, you call `gl.texParameteri()` like this:

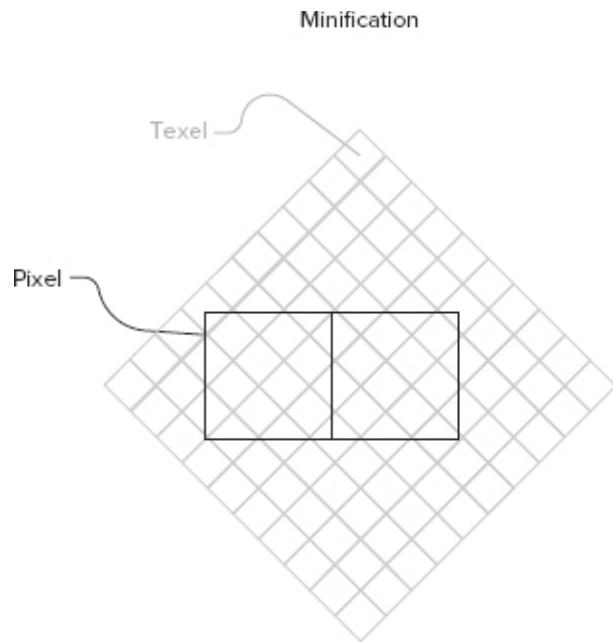
```
gl.texParameteri(gl.TEXTURE_2D,           gl.TEXTURE_MAG_FILTER,  
    gl.LINEAR);
```

Linear filtering does not result in the same blocky appearance as nearest neighbor filtering. Instead, linear filtering is characterized by a blurrier appearance when the texture is stretched.

Understanding Minification

[Figure 5-7](#) shows an example of a texture that is minified. Several texels correspond to the one pixel on the screen. This means that there are several texels that should affect the color of the pixel on the screen. In practice, however, it is difficult to determine the exact influence of all texels that correspond to a particular pixel.

FIGURE 5-7: For minification, each pixel corresponds to many texels



For minification, you can choose between the same two basic filters that you can use for magnification: nearest neighbor and linear filtering. (As you will see in the next section, you can also use a few other filters that are based on mipmapping.)

You set the nearest neighbor filtering for minification with the following call:

```
gl.texParameteri(gl.TEXTURE_2D,           gl.TEXTURE_MIN_FILTER,  
gl.NEAREST);
```

The nearest neighbor filter works in the same way as it does for magnification. A single color is fetched from the texel that is closest to the texture coordinate. This filter will result in aliasing problems when each pixel corresponds to an area in the texture that covers many texels.

The other basic filter that can be used for minification is the linear filter. You set the linear filter for minification with the following code:

```
gl.texParameteri(gl.TEXTURE_2D,           gl.TEXTURE_MIN_FILTER,  
gl.LINEAR);
```

The linear filter also works in the same way as it does for magnification. The four closest texels to the texture coordinate are used to create a weighted average that is used as the color for a given texture coordinate. The linear filter can give a slightly better result than the nearest neighbor for minification, but when substantially more than four texels correspond to one pixel, even this filter will result in aliasing.

Understanding Mipmapping

As you learned in the previous section, minification with both nearest neighbor and linear filtering can result in aliasing when a lot of texels correspond to each pixel. One solution would be to have a texture with a smaller size so that not so many texels correspond to each pixel. But for objects in your scene that are close to the viewer, you might want the textures with higher resolution to avoid a magnification that results in the textured object looking blocky or blurry.

Mipmapping is a solution to this problem. In addition to the base texture (called level zero), mipmapping uses a series of smaller textures as well. Each new texture is half the dimensional size of the previous texture in the series. This series of textures forms a chain of textures that is called a *mipmap chain*. A complete mipmap chain occupies approximately one-third more memory than if you only use the base texture without mipmapping.

The textures do not have to be square, but the chain continues until the last texture has the size 1×1 . As an example, a mipmap chain could contain the textures of sizes 256×256 , 128×128 , 64×64 , 32×32 , 16×16 , 8×8 , 4×4 , 2×2 , and 1×1 . [Figure 5-8](#) shows the first four images of a mipmap chain.

FIGURE 5-8: The first four textures in a mipmap chain



The easiest way to using mipmapping in your WebGL application is to let WebGL generate the mipmap chain for you. You only have to load the base texture as level zero, and then you can call the function `gl.generateMipmap()` and WebGL automatically generates the complete mipmap chain for you. This is what the code typically looks like for a 2D texture:

```
gl.texImage2D(gl.TEXTURE_2D,          0,           gl.RGBA,           gl.RGBA,  
gl.UNSIGNED_BYTE,  
           image);  
gl.generateMipmap(gl.TEXTURE_2D);
```

The other alternative is to generate the images for the mipmap chain offline with an image editor or a texture tool that is specifically for generating images for mipmapping. Then, when you have all the images for your mipmap chain generated, you load each image with `gl.texImage2D()` and specify the mipmap level that the loaded image corresponds to as the second argument to the method. The code to load mipmap levels zero to four could typically look something like this:

```
gl.texImage2D(gl.TEXTURE_2D,      0,      gl.RGBA,      gl.RGBA,  
gl.UNSIGNED_BYTE,  
imageLevel_0);  
  
gl.texImage2D(gl.TEXTURE_2D,      1,      gl.RGBA,      gl.RGBA,  
gl.UNSIGNED_BYTE,  
imageLevel_1);  
  
gl.texImage2D(gl.TEXTURE_2D,      2,      gl.RGBA,      gl.RGBA,  
gl.UNSIGNED_BYTE,  
imageLevel_2);  
  
gl.texImage2D(gl.TEXTURE_2D,      3,      gl.RGBA,      gl.RGBA,  
gl.UNSIGNED_BYTE,  
imageLevel_3);  
  
gl.texImage2D(gl.TEXTURE_2D,      4,      gl.RGBA,      gl.RGBA,  
gl.UNSIGNED_BYTE,  
imageLevel_4);
```

As you can see, this second approach requires some more work. However, it is good to know that this option exists if you have very specific requirements on how the lower-resolution images in the mipmap chain are generated.



To use mipmapping, both dimensions of the base texture must be a power of two (1, 2, 4, 8, 16, ..., 512, 1024, and so on).

Selecting Texture Filters

When you have a complete mipmap chain for your texture, you can select four new filters for the minification in addition to the non-mipmapped filters `gl.NEAREST` and `gl.LINEAR`. The following list provides an overview of all the available filters:

- `gl.NEAREST` — Nearest neighbor filtering is used on the base texture. This means that the color for the texel nearest to the texture coordinate is used.
- `gl.LINEAR` — Linear filtering (also called bilinear filtering) is used on the base texture. This means that the color is calculated as the weighted average of the four texels surrounding the texture coordinate.
- `gl.NEAREST_MIPMAP_NEAREST` — The nearest mipmap level is selected and then nearest neighbor filtering is used within this mipmap level.
- `gl.NEAREST_MIPMAP_LINEAR` — The two closest mipmap levels are selected. Within these two levels, the nearest neighbor filtering is used to get an intermediate result within each mipmap level. Linear interpolation is used between these intermediate results to get the final result.
- `gl.LINEAR_MIPMAP_NEAREST` — The nearest mipmap level is selected and then linear filtering is used within this mipmap level.
- `gl.LINEAR_MIPMAP_LINEAR` — The two closest mipmap levels are selected. Within these two levels, linear filtering is used to get one intermediate result within each mipmap level. Linear interpolation is then used between these intermediate results to get the final result. This filter typically produces the best result of all the filters. It is sometimes referred to as *trilinear filtering*.

As you can see, the four filters that use mipmapping all have names of the form `gl.A_MIPMAP_B`, where `A` and `B` can be either `NEAREST` or `LINEAR`. Here `A` specifies how the texels are used within one mipmap level. If `A` is `NEAREST`, it means that the nearest neighbor is used within a mipmap level. If `A` is `LINEAR`, it means that linear interpolation is used within a mipmap level.

`B` specifies if a single mipmap level is used or if the two closest mipmap levels are used. If `B` is `NEAREST` it means that the closest mipmap level is used, and if `B` is `LINEAR` it means that the two closest mipmap levels are used and within these levels the method specified by `A` is used. The result from the two mipmap levels is then linearly interpolated to get the final result.

Realizing Better Performance with Mipmapping

Although the visual appearance of mipmapping is the main reason to use it, mipmapping can also give you better performance. The reason is that you typically get better cache utilization when you use mipmapping, as texel fetches are more likely to happen closer to each other in the memory when you have minification and use coarser textures.

When you don't use mipmapping and have minification, the texels that are fetched can be far away from each other in the memory. This reduces the probability that the fetched texel is already available in a cache closer to the GPU.

If you use trilinear filtering by specifying `gl.LINEAR_MIPMAP_LINEAR`, the GPU usually needs to work more, and more texels need to be fetched from the texture memory, than if you use a filter such as `gl.LINEAR_MIPMAP_NEAREST`. It is a good idea to test a few different filtering modes on different devices and choose a mode that gives you the quality and performance that you need.

UNDERSTANDING TEXTURE COORDINATE WRAPPING

You have learned that texture coordinates are defined so that the lower-left corner of your texture has the coordinates $(0, 0)$ and the upper-right corner has the coordinates $(1, 1)$ regardless of whether or not the texture is a square. But you are actually allowed to specify texture coordinates for your geometry that lie outside this range.

If you specify coordinates outside the range, WebGL handles this according to the texture wrap mode. You can specify the texture wrap mode independently for the s -direction and the t -direction by calling the method `gl.texParameteri()` and specifying `gl.TEXTURE_WRAP_S` or `gl.TEXTURE_WRAP_T` as the second argument. As the third argument to `gl.texParameteri()`, you can specify these wrap modes:

- `gl.REPEAT`
- `gl.MIRRORED_REPEAT`
- `gl.CLAMP_TO_EDGE`

Each wrap mode is discussed in the following sections.

Using the `gl.REPEAT` Wrap Mode

The wrap mode `gl.REPEAT` is the default behavior that WebGL uses if you don't specify any wrap mode explicitly. In this mode, WebGL repeats the texture for each integer boundary of the texture coordinate. Before WebGL samples from the texture, it removes the integer part from the texture coordinate. This can be

useful if the geometry that you are applying your texture to is many times larger than the image you are using as input for the texture, so the image would look bad if you stretched it over the complete geometry. The behavior in this mode is that a tiled pattern is applied to your geometry.

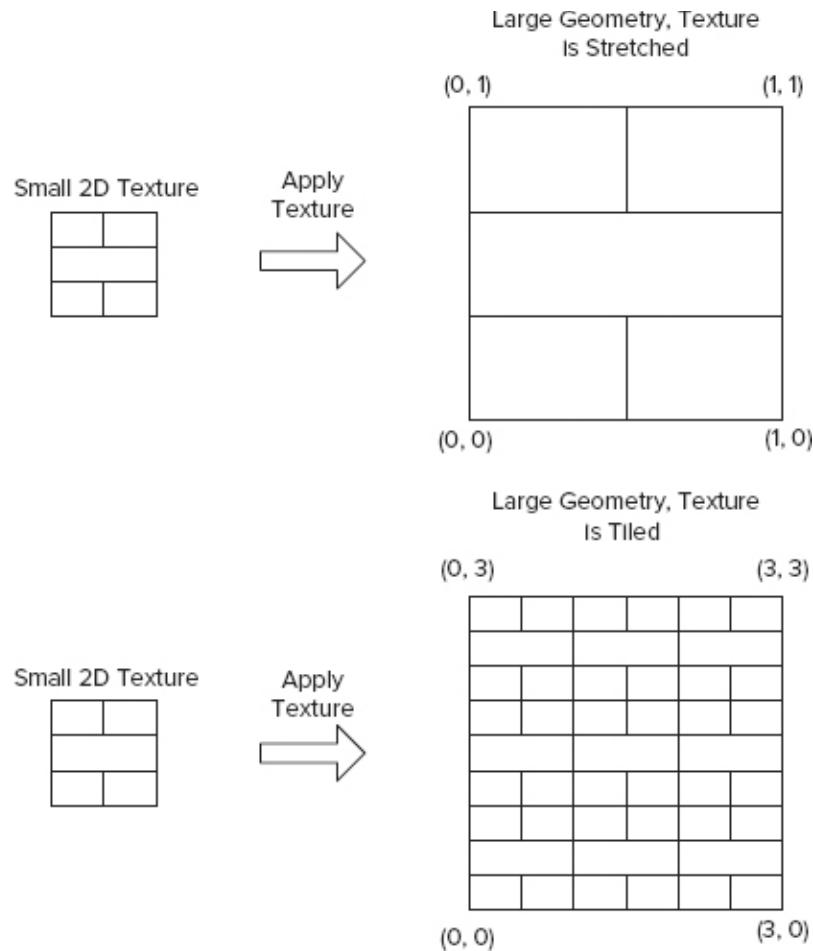


Tiling is general term in computer graphics that often means that you have a small texture that you apply repeatedly to a larger object. An example could be a large field with grass. The size of your texture only has to cover a small part of the field. Then the texture is repeatedly applied so the whole field is covered with grass.

To make this look good, you typically want the edges of the texture to blend well together so the seams between the different textures are as invisible as possible.

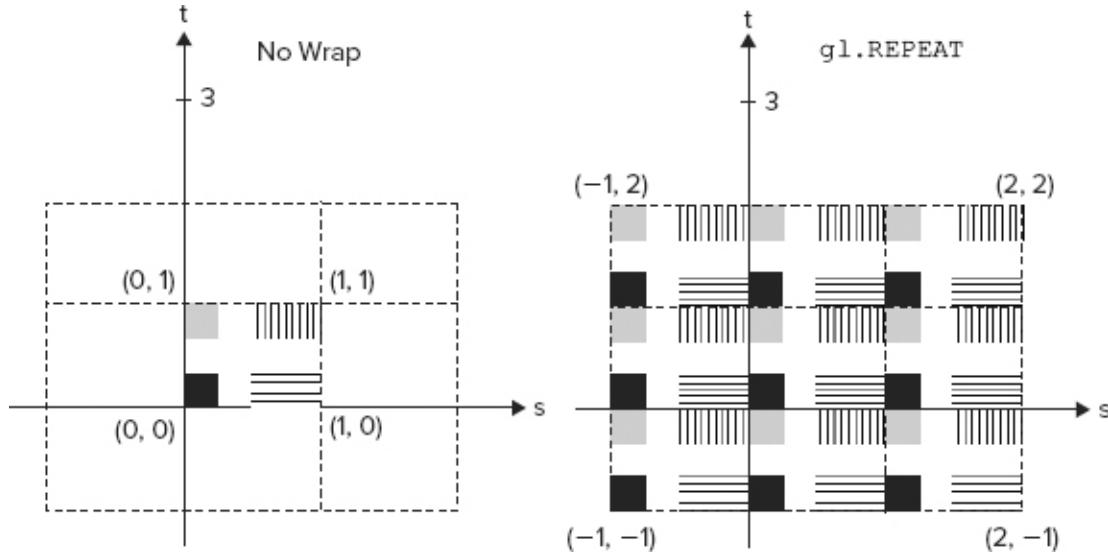
[Figure 5-9](#) shows a small 2D texture applied to a larger geometry. At the top of the figure, the vertices for the geometry are given the texture coordinates (0, 0), (1, 0), (1, 1), and (0, 1). This results in a magnification, and so the small texture is stretched and the textured geometry looks quite different from the original texture. At the bottom of the figure, the vertices are given the texture coordinates (0, 0), (3, 0), (3, 3), and (0, 3). In this case, the texture is not stretched over the geometry.

[**FIGURE 5-9:**](#) The effect of specifying texture coordinates outside the range [0, 1]



[Figure 5-10](#) illustrates in more detail how the `gl.REPEAT` mode works. To the left, you see the texture in a coordinate system when no wrapping occurs. To the right, you see the effect of `gl.REPEAT`.

FIGURE 5-10: To the left, a texture when no wrapping occurs. To the right, the same texture is shown when wrapping occurs and the wrap mode `gl.REPEAT` is used



To explicitly set the wrap mode for the currently bound texture object to `gl.REPEAT` in both the *s*-direction and the *t*-direction, call these two lines of code:

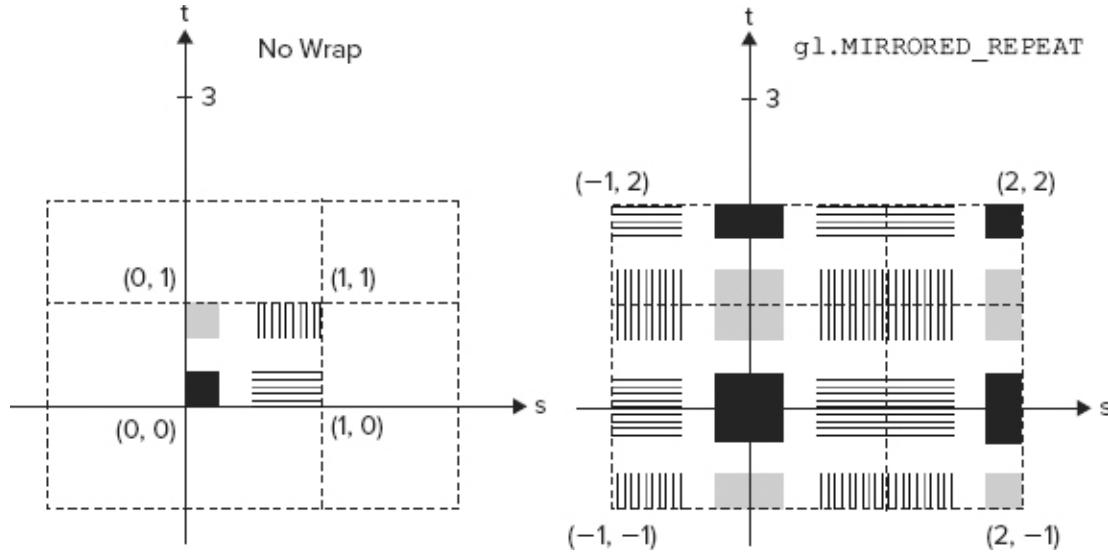
```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.REPEAT);
```

Using the `gl.MIRRORED_REPEAT` Wrap Mode

The wrap mode `gl.MIRRORED_REPEAT` is similar to `gl.REPEAT`, but it mirrors the original image when the integer portion of the texture coordinate is even. This mode can be good if you want to create a seamless tiling on a surface with a texture that is not prepared for seamless tiling. When a texture is not prepared for seamless tiling, this just means that the right edge of the texture does not blend well with the left edge, so when the two textures are placed next to each other, the seam between them is very obvious.

[Figure 5-11](#) shows how the mode `gl.MIRRORED_REPEAT` works. To the left, you see the texture in a coordinate system when no wrapping occurs. To the right, you see the effect of `gl.MIRRORED_REPEAT`.

FIGURE 5-11: To the left, a texture when no wrapping occurs. To the right, the same texture is shown when wrapping occurs and the wrap mode `gl.MIRRORED_REPEAT` is used



To set the wrap mode for the currently bound texture object to `gl.MIRRORED_REPEAT` in both the *s*-direction and the *t*-direction, you call these two lines of code:

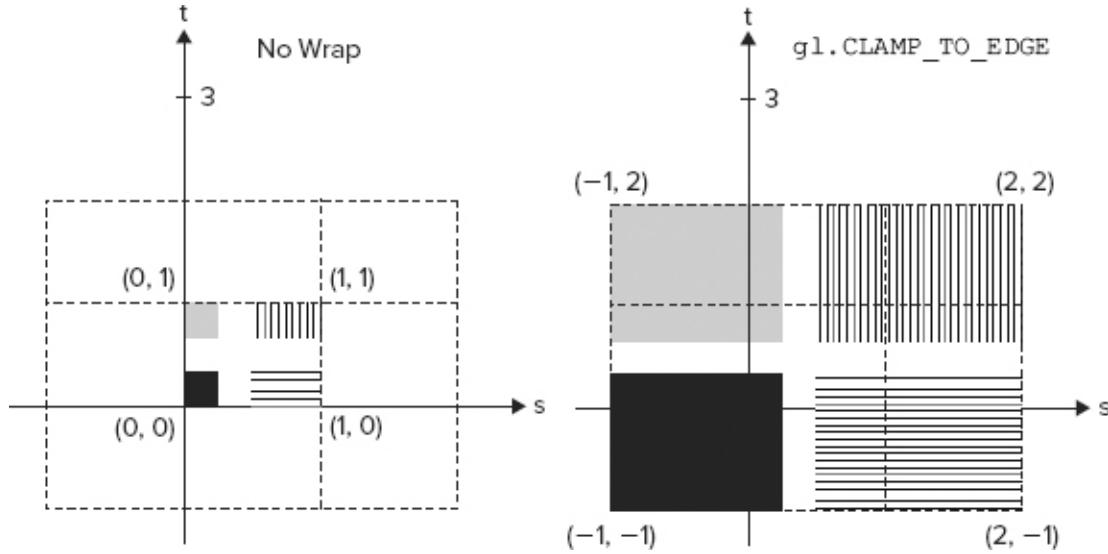
```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
    gl.MIRRORED_REPEAT);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
    gl.MIRRORED_REPEAT);
```

Using the `gl.CLAMP_TO_EDGE` Wrap Mode

When you use the mode `gl.CLAMP_TO_EDGE`, all texture coordinates are clamped to the range [0, 1]. Texture coordinates that are outside this range will sample from the closest edge of the texture.

[Figure 5-12](#) shows how the mode `gl.CLAMP_TO_EDGE` works. To the left, you see the texture in a coordinate system when no wrapping occurs. To the right, you see the effect of `gl.CLAMP_TO_EDGE`.

FIGURE 5-12: To the left, a texture when no wrapping occurs. To the right, the same texture is shown when wrapping occurs and the wrap mode `gl.CLAMP_TO_EDGE` is used



To set the wrap mode for the currently bound texture object to `gl.CLAMP_TO_EDGE` in both the *s*-direction and the *t*-direction, call these two lines of code:

```
gl.texParameteri(gl.TEXTURE_2D,
    gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D,
    gl.CLAMP_TO_EDGE);
```

A COMPLETE TEXTURE EXAMPLE

Now that you've learned about textures in WebGL, you can use some of this knowledge to create a complete example that uses textures. The code in Listing 5-1 is based on Listing 4-4 from Chapter 4, but it uses textures instead of basic colors.

In addition, it is slightly restructured to be able to handle a lost context. This means that, for example, no properties are added to WebGL objects. Instead, the needed properties are added to a global object that is named `pwgl`. You should also look at how the event listeners for the `webglcontextlost` event and the `webglcontextrestored` event are registered. This is done in the `startup()` function close to the end of Listing 5-1. In the `startup()` function you can also see how functionality from the JavaScript library `webgl-debug.js` is used, so it is possible to simulate the `webglcontextlost` event and the `webglcontextrestored` event as you learned in the beginning of this chapter.



By default, most browsers have strong restrictions on what JavaScript can do with local files. If you are running a WebGL application by opening it locally in a browser, keep in mind that most browsers do not let you upload the images you load from the file system as a texture. This is a problem if you develop WebGL code with textures that you test locally. It is also a problem if you select “Save As” on a WebGL application that you find online and then try to run this application by opening it from the file system.

If you run into this problem, one solution is to change the default setting of how local files are treated. In Firefox, you can type `about:config` in the address bar. Then you go to `security.fileuri.strict_origin_policy` and set the value for it to false. [Figure 5-13](#) shows what this looks like in Firefox.

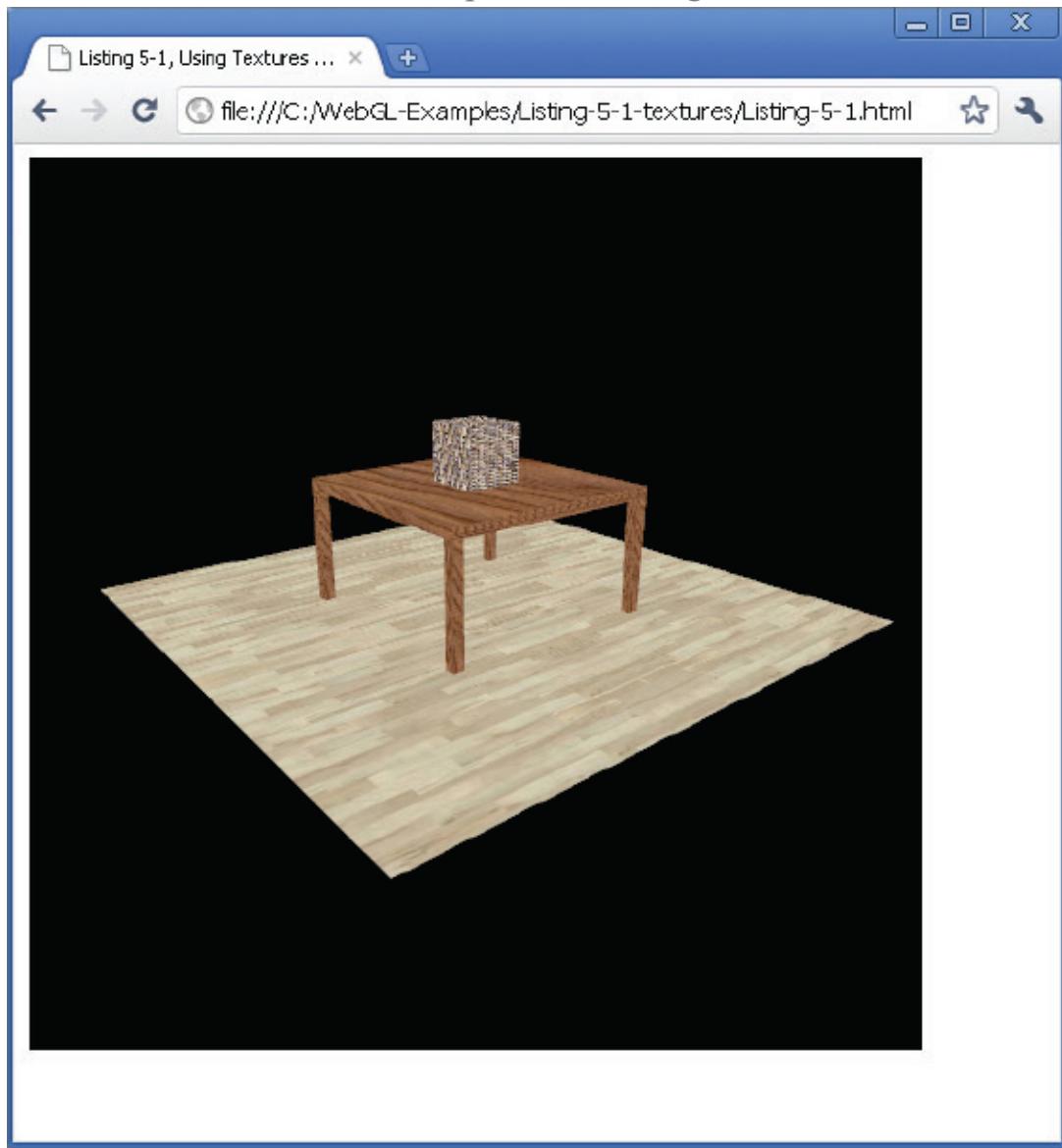
FIGURE 5-13: If you type `about:config` in Firefox’s address bar, you can change a setting that will let you use images from your file system as textures

Preference Name	Status	Type	Value
browser.link.open_newwindow.restriction	default	integer	2
browser.urlbar.restrict.bookmark	default	string	*
browser.urlbar.restrict.history	default	string	^
browser.urlbar.restrict.openpage	default	string	%
browser.urlbar.restrict.tog	default	string	+
browser.urlbar.restrict.typed	default	string	~
javascript.options.strict	default	boolean	false
javascript.options.strict.debug	default	boolean	false
security.fileuri.strict_origin_policy	user set	boolean	false
security.ssl.allow_unrestricted_renego_everywhere_temporarily_available_pref	default	boolean	false
security.ssl.renego_unrestricted_hosts	default	string	
security.xpconnect.plugin.unrestricted	default	boolean	true

In Chrome, you can start Chrome with the command-line switch `--allow-file-access-from-files`.

[Figure 5-14](#) shows what it looks like when the code in Listing 5-1 is loaded into a browser. If you compare [Figure 5-14](#) to Figure 4-14, you can see that the texturing results in a more realistic scene than the one produced by Listing 4-4.

FIGURE 5-14: A scene that corresponds to Listing 5-1



Available for
download on
[Wrox.com](#)

Listing 5-1 now uses a small JavaScript utility library that has not been used before. Download Listing 5-1 from [Wrox.com](#) and take a look at the code. (The listing is not shown in the chapter due to its length.) This utility library is named `webgl-utils.js` and is used to handle the rendering loop. The library provides the functions `requestAnimationFrame()` and `cancelRequestAnimationFrame()`.

The rendering loop and this library are described more in Chapter 6 where you will learn about animations. It is introduced in this chapter even though there

are no moving objects in the scene because you are loading images for textures here.

Because the image data is loaded asynchronously into the `Image` object when a URL is assigned to the `src` attribute, there is a race condition if you render your scene once. If the scene is rendered (which means that `draw()` is called) before the images for the textures have been loaded, the textures do not appear as they should.



A race condition means that the outcome of a process (in this case, rendering the scene) is dependent on the sequence or timing of two (or more) events. In this case, the two events are that the image data has finished loading and that the `draw()` function is called.

Another thing that is interesting to point out is how the texture coordinates are specified for the floor. By using texture coordinates outside the range [0.0, 1.0], texture wrapping is used. This means that the texture for the floor does not have to be magnified so much. If you change the texture coordinates in the buffer `floorVertexTextureCoordinates` that have the value 2.0 to the value 1.0 and view the result in your web browser, you can see that the texture for the floor needs to be magnified more.

```
pwgl.floorVertexTextureCoordinateBuffer = gl.createBuffer();
                                         gl.bindBuffer(gl.ARRAY_BUFFER,
pwgl.floorVertexTextureCoordinateBuffer);
var floorVertexTextureCoordinates = [
  2.0, 0.0,
  2.0, 2.0,
  0.0, 2.0,
  0.0, 0.0
];
                                         gl.bufferData(gl.ARRAY_BUFFER,           new
Float32Array(floorVertexTextureCoordinates),
                                         gl.STATIC_DRAW);

pwgl.FLOOR_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
pwgl.FLOOR_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 4;
```

The rest of the code in Listing 5-1 should be easy enough for you to understand.

USING IMAGES FOR YOUR TEXTURES

There are several ways to get the images that you use for your textures in WebGL:

- You can search the web for “free textures.” There are many sites that have large collections of textures that you can use for free. However, read the license carefully so you understand how you are allowed to use the textures.
- You can use your camera to take your own photos and then use these photos as textures.
- You can draw your own images with an image editor such as GIMP, Photoshop, or Paint.NET.
- You can buy textures from companies that specialize in selling them.

Downloading Free Textures

There are many sites where you can download textures for free. These sites often provide good texture images that you can download and use for your projects. However, read the license carefully to make sure you understand how you are allowed to use the textures.

The license often permits you to use the texture in 2D or 3D models without paying any royalties. However, you are usually not allowed to sell or distribute the texture as a standalone file that competes with the original site.

If you are looking for free textures, two sites that might be worth looking at are:

- www.cgtextures.com
- www.textureking.com

Otherwise, the best way to find the free textures that you need for your project is probably to search on the web.



The textures used in this book are from www.cgtextures.com, which has several great textures with much higher resolution than the ones used in the examples in this book.

Basing Textures on Your Own Photos

Even if you can download a large variety of different textures, you might have a WebGL project that requires very specific textures. Maybe you want something specific that nobody thought of before. Or maybe you just want to take your own photos to be able to tell your friends that you created the textures yourself.

Regardless of the reason, this section contains a few useful tips for when you grab your digital camera to take photos for your textures:

- Point your camera directly towards the surface you are photographing. You generally don't want perspective in the image you are going to use as a texture.
- Do not zoom in too much. If you do, the area you are able to photograph will be smaller. A smaller area makes it more difficult to tile the photo properly.
- Avoid taking your photos when it is dark. If you don't have enough light, your images will be blurry and therefore difficult to use as textures.
- Avoid photographing metal objects in sunlight. Metal is highly reflective, so the sun creates highlights that do not look good if you have to tile your texture.

Drawing Images

If you have experience using, or want to learn how to use, an image editor such as GIMP, Photoshop, or Paint.NET, it could be worthwhile to draw your textures in one of these programs. Either you can just experiment and use your imagination to draw something, or you can search for a tutorial on the web. There are many step-by-step tutorials to help you create a texture.

Buying Textures

Instead of trying to find free textures to download, you can buy them. There are also some sites that offer a combination: You either download a few textures, or you become a member and can download a huge package of textures at once.

One question you might ask yourself is why someone would buy textures when they can download them for free on many sites. Well, sometimes the textures that you buy are of higher resolution and better quality. You have to decide whether it is worth it for your WebGL project.

UNDERSTANDING SAME-ORIGIN POLICY AND CROSS-ORIGIN RESOURCE SHARING

The *same-origin policy* is an important security concept for web applications. Briefly, it means that a script running in a browser is not allowed to obtain data that comes from another origin. Two resources (for example, scripts, documents, images, and so on) come from the same origin if the document that includes them fulfills the following conditions:

- Both originate from the same domain name (for example, both come from example.com).
- Both are accessed with the same scheme (such as `http`).
- Both are accessed on the same port number (port 80 is the default for HTTP).

All three of these conditions must be fulfilled for the resources to be considered to come from the same origin. [Table 5-2](#) shows a few examples to make this clearer. The first column lists a number of URLs, and the second column states which of these URLs have the same origin as the URL in the table header (www.example.com/page-1.html).

TABLE 5.2: Examples of Same-Origin Policy. (Compare them against www.example.com/page-1.html.)

URL TO DOCUMENT CONTAINING THE SECOND RESOURCE	SAME ORIGIN
http://www.example.com/page-2.html	Yes
http://www.example.com/dir/page-2.html	Yes
http://www.domain2-example.com/page-1.html	No. The domain is different.
https://www.example.com/page-1.html	No. The scheme is different. Note that https is different from http.
http://www.example.com:8080/page-2.html	No. The port is different.

Understanding Same-Origin Policy for Images in General

The same-origin policy is relevant for several different kinds of resources. This section describes how it affects images, as these are the resources that are

relevant for textures. But first, you will learn how images are affected in general, and more specifically how they are handled when they are drawn on an HTML5 canvas.

An HTML page from one origin can display an image from another origin with HTML code like this:

```

```

What is not allowed is for a JavaScript to read the pixels from an image that comes from another origin than the JavaScript. For example, assume that you have a JavaScript that creates an `Image` object, reads an image from a different domain than the script comes from into the `Image` object, and finally draws the `Image` object onto an HTML5 canvas.

This is allowed according to the same-origin policy, but after the image is drawn on the canvas with `context2D.drawImage()`, the canvas is marked so the browser will not allow the script to serialize the data from the canvas with a call to `canvas.toDataURL()`. The following code snippet illustrates this idea:

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Example of Same Origin Policy</title>
<script type="text/javascript">
// Assume that this document is loaded from http://example-
domain.com
function draw() {
    var canvas = document.getElementById("canvas");

    if (canvas.getContext) {
        var context2D = canvas.getContext("2d");

        // Create an Image element
        var img = new Image();
        img.onload = function(){
            context2D.drawImage(img, 0, 0);

            // The call to canvas.toDataURL() is not allowed and
            // will generate a security error if the domain that the
            // image.jpg (see below) originates from is different
            // from
            // the domain that this document originates from.
            var pixelData = canvas.toDataURL();
        }
    }
}
```

```

        img.src = 'http://other-domain.com/image.jpg';
    }

</script>
</head>
<body onload="draw();">
    <canvas id="canvas" width="300" height="300">

        Your browser does not support the HTML5 canvas element.

    </canvas>
</body>
</html>

```



The method `canvas.toDataURL()` is a way to serialize the data in a canvas to an image. If the method is called without any arguments, then by default, the data in the canvas is serialized to the PNG format. The method returns a URL on the `data: URL` scheme, where the PNG image is encoded inline as a base64 encoded string. (The `data: URL` scheme is specified in RFC 2397.) Base64 encoding is a way to encode binary data as an ASCII-formatted string. This means that after a successful call to this:

```
pixelData = canvas.toDataURL();
```

the `pixelData` contains a string that looks something like this:

```
"data:image/png;base64,iVBORw0KGgoAAA...ErkJggg=="
```

The base64 encoded data starts with "iVBORw". This string can be assigned to the `src` attribute of an `` tag to be displayed in the browser, but because the data is available inside the string, the JavaScript that makes a successful call to `canvas.toDataURL()` also has full access to the pixel data in the image.

Understanding Same-Origin Policy for Textures

As you saw in the previous section, it is possible to read a cross-domain image into an `Image` object and draw it onto a canvas. If the image has a different origin than the canvas, then the browser marks the canvas as tainted so the pixels cannot be read from the canvas after this.

If you compare this with WebGL, the corresponding strategy would be that it would be allowed to read cross-domain images into a texture and then draw the texture on the WebGL canvas. However, after the cross-domain image is drawn on the canvas, it is not allowed to read the pixels from the canvas. This was also

what the initial WebGL specification proposed and how it was initially implemented in the browsers.

However, during the development of the specification, security expert Steve Baker pointed out that it might be possible to figure out the content of the texture by writing a special fragment shader that is then uploaded to the GPU when the user visits a malicious site. The fragment shader could measure the time it takes to test the color of a texel, and from this time it could calculate the brightness of the texel in the texture. There has also been a proof of concept implementation that demonstrated that this attack is actually possible in reality.

The result is that WebGL does not generally allow cross-domain images or videos to be uploaded as textures with `gl.texImage2D()` or `gl.texSubImage2D()`. The following code snippet shows an example of an image that is loaded from the domain www.otherdomain.com. If you assume that this domain is different from the domain where you download the canvas that you get the `WebGLRenderingContext` from, the call to `gl.texImage2D()` generates a security error.

```
function textureFinishedLoading(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

    // The call to texImage2D() will generate a security error
    // if the image is loaded from another domain.

    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
    gl.UNSIGNED_BYTE,
        image);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
    gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
    gl.LINEAR);

    gl.bindTexture(gl.TEXTURE_2D, null);
}

function loadImageForTexture(url, texture) {
    var image = new Image();
    image.onload = function() {

pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(i
mage), 1);
        textureFinishedLoading(image, texture);
    }
}
```

```

        }
        pwgl.onGoingImageLoads.push(image);
        image.src = url;
    }

function setupTextures() {
    pwgl.woodTexture = gl.createTexture();

    // Load an image to be used as a texture from another domain
    loadImageForTexture("www.otherdomain.com/wood_texture.jpg",
pwgl.woodTexture);
}

```

If you need to use images or videos that are hosted at another domain, it is possible to use Cross-Origin Resource Sharing (CORS) to cooperate with the server that hosts the images and videos that you want to use for your textures. You will learn more about CORS in the next section.

Understanding Cross-Origin Resource Sharing

If you want to use images or videos that are hosted at another domain as textures in your WebGL application, it is possible to do so. If both your browser and the server hosting the media support it, you can use Cross-Origin Resource Sharing (CORS) to cooperate with the server.

CORS is a general way for resources from different origins to communicate with each other in a controlled manner. The specification is not specific to WebGL, but was seen as a good solution to solve the problem with cross-origin textures. It is based on the fact that the browser sends a special HTTP header (named `Origin`) in the request to the server if it wants to use a resource from the server in a script coming from another origin. If the server allows this, then it includes another HTTP header (named `Access-Control-Allow-Origin`) in the response and specifies that this is allowed.

As a WebGL developer, you will not find it difficult to use CORS to request images or videos from a server. You can basically do it by adding one line to your previous code that loads an image or a video. The following code snippet shows how you load an image from a different origin by setting the `crossOrigin` property of the `Image` to `anonymous` before you assign the URL to the `src` property of the `Image`. The additional line of code needed for CORS is highlighted.

```

function textureFinishedLoading(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    // The call to texImage2D() will now be allowed
    // provided that the server permitted the CORS request
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
    gl.UNSIGNED_BYTE,
                image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
    gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
    gl.LINEAR);

    gl.bindTexture(gl.TEXTURE_2D, null);
}

function loadImageForTexture(url, texture) {
    var image = new Image();
    image.onload = function() {

pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(i
mage), 1);
        textureFinishedLoading(image, texture);
    }
    pwgl.ongoingImageLoads.push(image);
    image.crossOrigin = "anonymous";
    image.src = url;
}

function setupTextures() {
    pwgl.woodTexture = gl.createTexture();

    // Load an image to be used as a texture from another domain
    loadImageForTexture("www.otherdomain.com/wood_texture.jpg",
                        pwgl.woodTexture);
}

```

You should note that when you use CORS to request the image data, the WebGL canvas that you draw the texture to is never tainted. This means that in addition to being able to upload the `Image` as a texture to the GPU, you will also be allowed to use `canvas.toDataURL()` to read the pixels from the canvas after you have drawn the texture to it.

Even though it is less related to WebGL, it can be good to know that the CORS support can also help you to read pixels from a 2D canvas with

`canvas.toDataURL()`. The following code snippet shows how data can be read from a canvas where an image from another origin has been drawn:

```
var canvas = document.getElementById("canvas");

if (canvas.getContext) {
    var context2D = canvas.getContext("2d");

    // Create an Image element
    var img = new Image();
    img.onload = function(){
        context2D.drawImage(img, 0, 0);

        // The call to canvas.toDataURL() is now allowed
        provided that
        // the server permitted the CORS request
        var pixelData = canvas.toDataURL();
    }
}


```

SUMMARY

In this chapter, you learned what texturing is and how you use texturing in WebGL. You also learned how to make your WebGL application more robust by handling the situation of a lost context.

You learned the different steps that are needed to use a texture when you draw with WebGL.

You learned about texture filtering and what magnification and minification mean. You also learned how mipmapping can improve both the visual quality of the scene and the rendering performance when you have minification. You now understand how the three texture wrapping modes,

- `gl.REPEAT`
- `gl.MIRRORED_REPEAT`
- `gl.CLAMP_TO_EDGE`

affect how the texture is applied when you have coordinates that are outside the range [0.0, 1.0].

Furthermore, you learned why the same-origin policy is important to understand when you are working with textures in WebGL and what restrictions

it sets on your usage of textures. You also learned what CORS is and how it can be used with cross-origin textures in your WebGL application.

Both texturing and the handling of lost context are very important topics for WebGL. Now you are prepared to continue with the next chapter, where you will learn how to use animations to create motion in your scene.

Chapter 6

Animations and User Input

WHAT'S IN THIS CHAPTER?

- How to create animations in WebGL
- Learn the difference between `requestAnimationFrame()`, `setInterval()`, and `setTimeout()`
- How to measure the frame rate in your WebGL application
- Learn the details of event handling
- How to handle key input
- How to handle mouse events

In the examples you have worked with so far in this book, all the objects in your scene have remained in the same position and location. In this chapter you will learn how to use animation with WebGL to create motion in your 3D scene.

You will also learn how to let the user affect your WebGL scene by handling user input such as key presses and mouse events. Since user input in a web application is based on event handling in JavaScript, you will first be taken through the details of event handling to make sure you fully understand it.

ANIMATING THE SCENE

Animation simulates movement by displaying a series of images (or frames) in rapid sequence. In general, there are two major types of animations on the web:

- *Declarative animations* such as Cascading Style Sheets (CSS) animations or the SVG `<animate>` tag. These animations are handled without any scripts. You specify how the element should animate but you don't have to generate each new frame yourself.
- *Script-based animations* where each new frame is updated by a JavaScript as the result of a callback. This group contains animations that are created by updating the style object of a DOM element, drawing on an HTML5 2D canvas, and drawing with WebGL.

In addition to these two types of animations, there are others, such as animated GIFs and Flash animations. However, these other types are different, since the separate images that build up the animation are part of the file containing the GIF images or the Flash animation.



If you have some web development background, you are probably familiar with CSS and can skip this note. If you don't know what CSS is, this note will give you a brief introduction.

CSS is a standard used to describe the presentation of a document. It is most commonly used together with an HTML document (but it can also be used with XML documents). The idea is that the HTML document should describe the content and structure, and CSS should describe the presentation, such as the look and the formatting of the document.

A CSS stylesheet consists of a set of rules that specifies how the document should be presented. Each rule has the following format:

```
selector {  
    property1: value1;  
    property2: value2;  
    ...  
}
```

The selector specifies which element or elements the rule applies to in the HTML document. After the selector, there is a list of property-value pairs that describe how the element is presented.

A very simple example of a rule in a CSS stylesheet is:

```
h1 {  
    color: red;  
    text-align: center;  
}
```

This example specifies that the text within the `<h1>` headings should be colored red and the headers should be centered. In this way, it is possible to specify how different DOM elements on the page should look and where they should be positioned.

It is also possible to use JavaScript to update the style of these DOM elements by using the `style` object that is part of all DOM elements. As an example, you can update the position of a DOM element with the following lines of JavaScript code:

```
e = document.getElementById('id');  
e.style.left = parseInt(e.style.left)+10+'px';
```

Making changes to the style object of an element is the basic idea behind creating DOM style-based animations. By repeatedly executing the second line above, you can make the element move to the right. Normally, updates to the style object like the one above change the position of the DOM object instantly. However, with CSS transitions and CSS animations, you can specify how an object should make a smooth transition from an initial position to a final position without using any JavaScript.

CSS is a huge topic and there are many books that focus only on CSS, so if you want to really learn how to use it (which is highly recommended), you should read one of these books or consult one of the many free online resources that exist on the topic. Two good online resources you could start with are:

- www.w3.org/Style/CSS/
- www.w3schools.com/css/

To use animation to make your models move around in your WebGL scene, you need to draw your scene repeatedly, and between each time you draw the scene, you need to update the position of the models you want to animate. This is exactly the same paradigm that is used for animations on an HTML5 2D canvas, but for 2D animation, you typically would only update the *x* and *y*-positions of your shapes before you draw each frame.

Traditionally, script-based animations like these have been based on the JavaScript methods `setTimeout()` or `setInterval()`. These methods are used to specify a callback that you want to be called a specified number of milliseconds in the future. In the callback, you can update your animation and then draw it. By doing this repeatedly, you can create an animation. The use of `setTimeout()` and `setInterval()` to create an animation is described in the next section.

Using `setInterval()` and `setTimeout()`

As mentioned in the previous section, the traditional way to implement JavaScript-based animations is to use one of these two JavaScript methods:

- `setTimeout(codeToCall, timeoutInMilliseconds)`
- `setInterval(codeToCall, timeoutInMilliseconds)`

Both of these methods are part of the `window` object in JavaScript, which means that they are global methods. The method `setTimeout()` calls the code or function in the first argument after the number of milliseconds specified by the second argument. The method returns an opaque value that can be passed as argument to the method `clearTimeout()` to cancel the call to the scheduled function or code.

The method `setInterval()` is similar to `setTimeout()`, but it calls the code or function in the first argument repeatedly with an interval specified in the second argument. Also, `setInterval()` returns an opaque value that can be sent as argument to `clearInterval()` to cancel future invocations of the scheduled function or code.

As an example, a very basic rendering loop based on `setInterval()` could be structured like this:

```
function draw() {  
  
    // 1. Update the positions of the objects in your scene  
  
    // 2. Draw the current frame of your scene  
  
}  
  
function startup() {  
  
    // Do your usual setup and initialization  
  
    setInterval(draw, 16.7);  
  
}
```

The `startup()` function is typically called after the document has been loaded (or whenever you want to start your animation). You do the usual setup and initialization of your application, and finally you call `setInterval()` and specify that the `draw()` function should be called with an interval of, for example, 16.7 milliseconds. The 16.7 milliseconds was selected in this example because an LCD does not typically update faster than 60Hz, which means that there is usually no point trying to update your animation faster than 60 times per second. This gives you a frame time of approximately $1/60\text{Hz} = 0.0167\text{s} = 16.7\text{ms}$.

Using `requestAnimationFrame()`

Even though web developers have successfully created animations with `setInterval()` and `setTimeout()` for a long time, there is a newer, recommended way of implementing these kinds of script-based animations. This is to use the method `requestAnimationFrame()`, which is specified to be part of the HTML DOM `window` object in the same way as `setInterval()` and `setTimeout()`.



You are strongly recommended to use `requestAnimationFrame()` instead of `setInterval()` and `setTimeout()` for animating your WebGL scene.

Understanding the Advantages of `requestAnimationFrame()`

The method `requestAnimationFrame()` is specified explicitly for the purpose of doing script-based animations. By having a specific method for animations, the browser vendors can optimize this method to be even better suited for animations than the traditional `setInterval()` and `setTimeout()` methods.

When you use `setInterval()` or `setTimeout()` to perform your animation, you try to choose a frequency for the updates of your animation that you think produces the best results. But the optimal frequency is actually very difficult for the author of the animation to know.

It is easier for the browser to know the optimal frame rate, especially since there might be many animations running in the browser at the same time, which might affect the frame rate. In this case, the browser can slow down the frame rate for all the animations so they are running smoothly but at a slightly slower rate.

The browser can also throttle or pause the updates to an animation that is running in a tab that is not visible at the moment. In this way, using this method can not only improve the performance of your animations but also use less power, which is important, especially for mobile devices where the battery lifetime is often a factor.



If your WebGL application includes network communication or any other logic that is timing-based but not related to the rendering, you might want to base this around another timing mechanism rather than `requestAnimationFrame()`. The reason is that if the tab is not visible and the browser throttles or pauses the calls to the callback, you might still want to continue your network activity or whatever other periodic logic you have in your application.

Using `requestAnimationFrame()`

The method `requestAnimationFrame()` is quite easy to use. You call it to request the browser to call the callback function you pass as an argument to the method. If your callback function is called `draw()`, it is as simple as this:

```
requestAnimationFrame(draw) ;
```

When your callback function is called, the current time is sent as argument to the function. The current time is often useful in an animation loop because it is commonly used to calculate how much time has passed since the last time you drew a frame, and then compensate the movement of your objects for any fluctuations in the frame rate. You will learn more about this later in this chapter.

This code snippet shows, on a high level, how you can use the method `requestAnimationFrame()` to structure your animation loop:

```
function draw(currentTime) {  
    // 1. Request a new call to draw the next frame before
```

```

//      you actually start drawing the current frame.
requestAnimationFrame(draw);

// 2. Update the positions of the moving objects in your scene

// 3. Draw your scene

}

function startup() {
    // Do your usual setup and initialization
    canvas = document.getElementById("myGLCanvas");
    gl = createGLContext(canvas);
    setupShaders();
    setupBuffers();
    setupTextures();

    draw();
}

```

Note how the `draw()` function is defined to take `currentTime` as argument, but when the `draw()` function is called from within the `startup()` function, no arguments are specified. In many other programming languages this would not be allowed, but in JavaScript a function can be invoked with any number of arguments, regardless of the number of arguments that are specified in the function definition.



The World Wide Web Consortium (W3C) has published a Working Draft of the specification that describes the method `requestAnimationFrame()`. The official name of the Working Draft is “Timing control for script-based animations.” If you are interested in the details, you can find the specification here:

www.w3.org/TR/animation-timing/

A Working Draft is the first level of maturity of a W3C standard. A Working Draft means that the standard is published so it can be reviewed by the “community.” This means that not everything is completely frozen in this standard and some things will probably change.

When it comes to implementation of the method `requestAnimationFrame()`, a problem arises because the different browser vendors have used slightly different names for the method. A practical solution to this is to use the small shim (wrapper) available in the utility JavaScript library, `webgl-utils.js`. This small library was originally written by Google developers, and you can find it online, for example, here:

<https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/demos/common/webgl-utils.js>

The shim for cross-platform support of `requestAnimationFrame()` in `webgl-utils.js` is written as follows:

```
/**  
 * Provides requestAnimationFrame in a cross browser way.  
 */  
window.requestAnimFrame = (function() {  
    return window.requestAnimationFrame ||  
        window.webkitRequestAnimationFrame ||  
        window.mozRequestAnimationFrame ||  
        window.oRequestAnimationFrame ||  
        window.msRequestAnimationFrame ||  
        function(/* function FrameRequestCallback */ callback,  
               /*  
                * DOMElement Element */ element) {  
            return window.setTimeout(callback, 1000/60);  
        };  
})();
```

As you can see, `setTimeout()` is the fallback if no other method is found. Note that when this fallback is used, the current time is not sent as an argument to the function you have registered as a callback function. Also note that this library names the method `requestAnimFrame()` and not `requestAnimationFrame()`. The reason is that the standard is in a W3C Working Draft and is still expected to change.

Compensating Movement for Different Frame Rates

Regardless of whether you use `setTimeout()`, `setInterval()`, or `requestAnimationFrame()`, the simplest way of doing an animation in WebGL is to update the position of the models that you want to animate with a fixed value for each new frame. This works well in many cases. However, there can be some disadvantages with this approach. For example:

- The speed at which you move your objects depends on the frame rate. This means that for slow frame rates, the objects move slower than for faster frame rates. If you’re developing a racing game, you probably don’t want the cars to drive faster just because someone plays the game on a more powerful device, and slower when someone plays the game on a less powerful system such as low-end mobile device.
- If your frame rate fluctuates due to, for example, different loads on the CPU, the GPU, or the memory bus, the distance that the objects move during each frame also fluctuates. This gives you an erratic animation.

The following code snippet shows how the animation can be made smoother by taking into account how often a new frame is drawn:

```

function draw(currentTime) {
    // 1. Request a new call to draw the next frame before
    //     you actually start drawing the current frame.
    requestAnimationFrame(draw);

    // 2. Calculate how to compensate for varying frame rate
    //     and then update the position of the moving objects.
    if (currentTime === undefined) {
        currentTime = Date.now();
    }
    if (pwgl.animationStartTime === undefined) {
        pwgl.animationStartTime = currentTime;
    }
    if (pwgl.y < 5) {
        // Move your object. In this specific example the movement is just
        // moving a box vertically from the position where y = 2.7 to y = 5
        // The movement should take 3 seconds.
        pwgl.y = 2.7 + (currentTime - pwgl.animationStartTime)/3000 * (5.0-
2.7);
    }

    // 3. Draw your scene
}

function startup() {
    // Do your usual setup and initialization

    canvas = document.getElementById("myGLCanvas");
    gl = createGLContext(canvas);
    setupShaders();
    setupBuffers();
    setupTextures();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);

    pwgl.x = 0.0;
    pwgl.y = 2.7;
    pwgl.z = 0.0;
    pwgl.animationStartTime = undefined;

    draw();
}

```

Creating an FPS Counter to Measure the Smoothness of Your Animation

When you have an animated scene in WebGL, you usually want your motion to look as smooth as possible. A common way to measure the performance or smoothness in animations is by specifying how many frames per second (FPS) are rendered. A higher FPS value means better performance and generally a smoother animation. Remember that you will normally not reach a faster FPS rate than 60 frames per second since most LCD screens update at 60Hz.



A frequency of 60Hz is basically the same thing as 60 FPS. The unit hertz (Hz) is generally used to define frequency for hardware (such as displays), while FPS is used when you talk about an application.

In theory the FPS value is simple to calculate. You measure how long time it takes to render one frame, for example, by checking the time at the beginning of the frame and saving this time. Then, during the next frame when you come to the same place in the code, you check the time again. The difference between the second time and the first time is the frame time. The FPS value is the inverse of the frame time. This means that the basic formula is:

$$\text{FPS} = \frac{1}{\text{Frame Time}}$$

When you do this calculation in your application and want to show the result on the display, you have two options:

- Show an instantaneous FPS value by measuring the time it takes to render each frame, and calculate and display the FPS value directly as the inverse of the frame time.
- Show some kind of average FPS value. This can be done in several different ways.

A disadvantage with the instantaneous FPS value that you update for each frame is that it will probably change very frequently. If you are running your application at approximately 30 FPS, it means that the value could change approximately 30 times each second. If you update the FPS on the display for each frame, this just looks like a flickering to the user and it is difficult to see what the FPS value really is.

It is usually better to calculate and display some kind of average. You could, for example, use a moving average. An even simpler approach is to count how many frames are rendered during one second, and when that second has elapsed you show the number of rendered frames as the FPS value. Then you restart the counting of frames, continue for another second, show the new FPS value, and so on.

The following code snippet shows how you can do this by including some code at the beginning and the end of the function that draws the scene:

```
function draw(currentTime) {
    pwgl.requestId = requestAnimFrame(draw);
    if (currentTime === undefined) {
        currentTime = Date.now();
    }

    // Update FPS if a second or more has passed since last FPS update
    if(currentTime - pwgl.previousFrameTimeStamp >= 1000) {
        pwgl.fpsCounter.innerHTML = pwgl.nbrOfFramesForFPS;
        pwgl.nbrOfFramesForFPS = 0;
        pwgl.previousFrameTimeStamp = currentTime;
    }

    // Draw the scene

    ...

    // When a frame is completed, update the number of drawn
    // frames to be able to calculate the FPS value
```

```

        pwgl.nbrOfFramesForFPS++;
    }
}

```

When one second has elapsed, use the property `innerHTML` on the `pwgl.fpsCounter` to set the FPS value to the number of frames that have been rendered during the elapsed second.

The following code snippet shows how the `startup()` function is used to initialize some of the variables that are needed to calculate the FPS. There is also a `` element with an `id` added to the HTML code. This is used to display the FPS value by first getting a reference to the `` by calling `document.getElementById()` and then using the `innerHTML` property to set the FPS value as already shown.

```

<script type="text/javascript">

    ...

function startup() {
    ...

    pwgl.nbrOfFramesForFPS = 0;
    pwgl.previousFrameTimeStamp = Date.now();
    pwgl.fpsCounter = document.getElementById("fps");
    draw();
}

</script>

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
    <div id="fps-counter">
        FPS: <span id="fps">--</span>
    </div>
</body>

</html>

```

Understanding the Disadvantages of Using FPS as a Measurement

Even though FPS is probably the most common way to measure graphics performance, there is a disadvantage with using this measurement if you are not careful. The disadvantage is that the FPS is not linear against the time it takes to render a frame.

Assume that your WebGL application is running at 60 FPS and you modify the code (for example, by adding more objects to the scene) and the FPS value drops to 50 FPS. How bad is that? You have obviously lost 10 FPS, but how much longer does it take to render the scene with the extra objects? You get the answer by doing the following calculation:

$$\frac{1}{50} - \frac{1}{60} \approx 0.020 - 0.0167 = 0.0033\text{s} = 3.3\text{ms}$$

So in this example, the change to your code results in it taking an extra 3.3ms to render a frame.

Now instead, assume that your WebGL application is running at 30 FPS from the beginning. You modify the code and the FPS value drops to 20 FPS. How much longer does it take to render the scene in this case?

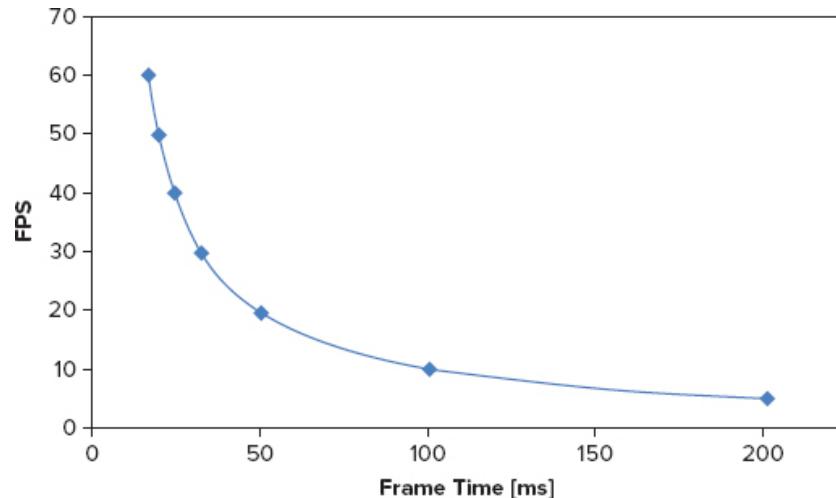
$$\frac{1}{20} - \frac{1}{30} \approx 0.050 - 0.033 = 0.017\text{s} = 17\text{ms}$$

In the second example, the modification to the code results in it taking an extra 17ms to render a frame.

In both the first and the second example, the frame rate dropped by 10 FPS. But the drop from 60 to 50 FPS was due to a small change in the frame time. It only took an additional 3.3ms to render a frame in this case. On the other hand, when the frame rate drops from 30 to 20 FPS something during the rendering takes 17ms longer than before.

When you have a higher frame rate from the beginning, a small change in the frame time causes a big change in the FPS value (see [Figure 6-1](#)). In this diagram, the FPS value is plotted against different frame times.

FIGURE 6-1: The relation between FPS and frame time



[Table 6-1](#) shows examples of the actual numbers.

TABLE 6.1: FPS versus Frame Time

FRAME TIME	FPS = 1/(FRAME TIME)
17 ms	60
20 ms	50
25 ms	40
33 ms	30
50 ms	20
100 ms	10
200 ms	5

If you have a high FPS value, it means that you are far to the left on the graph shown in [Figure 6-1](#), and a small change in the frame time obviously causes a big change in the FPS value.

Conversely, if you have a low FPS value, it means that you are far to the right on the graph, and so the frame time can obviously change more with only a smaller change in the FPS value because the graph is less steep here.

EVENT HANDLING FOR USER INTERACTION

To take care of user input in your WebGL applications, you need to understand the basics of event handling in JavaScript. This kind of event handling is not specific to WebGL, so if you have a web development background, you have probably seen some of this before.

But since event handling is important to being able to make your WebGL application interactive — for example, responding to key presses or mouse events — this section will go over the basics to make sure you have the knowledge you need.

The basic idea behind event handling is that when something interesting happens in a browser, an event is created and sent by the browser. Your application can use event handlers to listen to these events.

There are many different events that can be created. You have actually seen examples of some events in previous chapters even though not much time was spent on discussing or explaining the event handling at that time. For example, in Chapter 5, you used the following events even though they are not related to user input:

- `load` — A resource has finished loading. You used this to know when the image resource for your texture was loaded.
- `webglcontextlost` — The WebGL context is lost.
- `webglcontextrestored` — The WebGL context is restored.

There are also many events that are related to user input. Some examples that are useful are:

- `keydown`
- `keypress`
- `keyup`
- `mousedown`
- `mouseup`
- `mousemove`

There are two major alternatives to handling events in the web browser. They are:

- Basic Event Handling with DOM Level 0
- More Advanced Event Handling with DOM Level 2

In addition, older versions of Internet Explorer use their own way, which is a mix of the above two alternatives. This alternative will not be further described in the book. The other two alternatives are described in the following sections.

Basic Event Handling with DOM Level 0

DOM Level 0 event handling refers to the legacy event handling that is supported in any browser that supports JavaScript. In this event handling model, the event handler is specified using the attributes of the HTML elements or by assigning the corresponding property a value from JavaScript. The event handling names for DOM Level 0 start with the prefix “`on`”, such as `onload`, `onmousedown`, and `onmouseup`.

The Event Handler as an Attribute of an HTML Element

To see what it looks like when the event handler is specified using the attribute of the HTML element, here is a code snippet that you should recognize from several of the previous examples in the book:

```
<!DOCTYPE HTML>
<html lang="en">
<head>

    ...

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>
</html>
```

Here the `onload` property of the body element is assigned the event handler that is executed when the document is loaded. Note that the event handler in this case can be an arbitrary string of JavaScript code. You could actually have several JavaScript statements directly in the code that are assigned as an event handler if these statements are separated by semicolons. But if you need several statements, it is usually better from a structural point of view to call a function that in turn contains these statements.

The Event Handler as a Property of a JavaScript Object

All HTML elements in a document have a corresponding DOM element that can be accessed from JavaScript as a JavaScript object. The properties of this JavaScript object correspond to the attributes of the HTML element.

This means that an alternative way to assign an event handler to an HTML element is to assign the function that you want to use as your event handler directly to a property of a JavaScript object. One difference that is worth noting is that while the value of the HTML event handler attribute is a string of JavaScript, the value that is assigned to the property of a JavaScript object must be a function.

You used the technique of assigning a function to the JavaScript property in Chapter 5 when you assigned an anonymous function to the `onload` event handler of an image to know when the image had finished loading its data. The following code snippet shows this:

```
var image = new Image();
image.onload = function() { // Assign event handler as anonymous
    function
    // handle the loaded image
    ...
}

image.src = url;
```

You can also achieve the same thing by using a regular function instead of an anonymous function. In this case, the code to assign a function to the event handler looks like this:

```
function imageLoadHandler() {
    // handle the loaded image
```

```

    ...
}

var image = new Image();
image.onload = imageLoadHandler; // Assign the event handler
image.src = url;

```

Note that there are no parentheses after the name of the function you assign as the event handler in this case.

In some cases it is actually an advantage to use JavaScript properties to assign values to the event handlers as shown in this section, instead of specifying strings of JavaScript code to the HTML attribute. Two advantages that are often mentioned by web developers who prefer the approach with JavaScript properties are as follows:

- The code is cleaner when you separate the HTML code from the JavaScript code. Possibly the code could be more modularized and easier to maintain this way.
- If you need to, you can let the event handler functions be dynamic and assign them to a JavaScript property when you need an event handler, and then remove the event handler when you don't want it. If you have a complex application, this can be useful.

Advanced Event Handling with DOM Level 2

The DOM Level 2 event handling model is quite different from the DOM Level 0 event model that was described in the previous sections. On the surface, the two models are different in the way an event handler is registered. For the DOM Level 0 model, either you set a JavaScript string to an attribute of an HTML element or you assign a function to the JavaScript property.

For DOM Level 2, you use the method `addEventListener()` to register an event listener on an element. You saw how this method was used in Chapter 5 to register an event handler for the `webglcontextlost` and the `webglcontextrestored` events.

```

canvas.addEventListener('webglcontextlost', handleContextLost, false);
canvas.addEventListener('webglcontextrestored', handleContextRestored,
false);

```

In addition, the event handling names for DOM Level 2 do not have the “`on`” prefix like the event handlers for DOM Level 0 have. For example, `onload` for DOM Level 0 corresponds to just `load` for DOM Level 2 event handling.

However, it is not only how event handlers are registered and the naming of the event handlers that are different between the DOM Level 0 model and the DOM Level 2 model. A bigger difference lies in what is called the event propagation.

Event Propagation

For DOM Level 0, the events are dispatched to the document elements on which they occur. If this element has a registered event handler, that event handler is executed. This is a simple approach that works well in many cases.

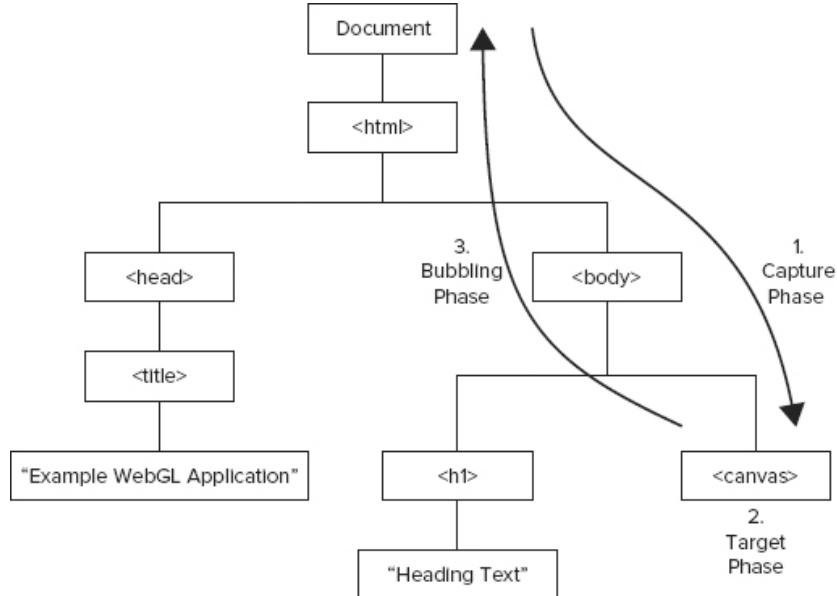
The DOM Level 2 event model is much more advanced. It has an event propagation that happens in three phases:

1. Event capturing phase
2. Handlers at the actual target node are executed

3. Event bubbling phase

These three phases are illustrated in [Figure 6-2](#).

FIGURE 6-2: Event propagation in the DOM Level 2 event model



During the first phase, which is the *event capturing phase*, the events start at the top of the DOM tree, which is generally at the document object, and then propagate down towards the target node. If any of the ancestors to the target node has registered an event handler that has capturing enabled, these handlers are executed during the event capturing phase. You register an event handler with capturing enabled by calling `addEventListener()` with the last argument set to true.

The second phase is that the event handler *at the actual target node* is executed. This corresponds to the behavior of the DOM Level 0 event model.

The third and last phase is the *event bubbling phase*. In a way, this is the opposite of the capturing phase. During the bubbling phase, the event bubbles up towards the top of the DOM tree again until it reaches the document object. It should be mentioned that not all events bubble up again. However, in general, lower-level events such as `mousedown`, `mouseup`, and `mousemove` all bubble up again. During the bubbling phase, any registered event handlers on the ancestor nodes to the target are triggered.

Now you know more about the event handling in a web application. You can also fully understand the details about the third argument of the method `addEventListener()`. When you were previously told to call the method with the last argument set to `false` as shown below, it was because you were not interested in the capture phase.

```
canvas.addEventListener('webglcontextlost', handleContextLost, false);
```

Key Input

The event handling for key input is fragmented between different browser vendors and different operating systems. However, there are some basic things that work in the same way in most

browsers. In general, these three keyboard events are generated when an alphanumeric key is pressed:

- `keydown`
- `keypress`
- `keyup`

When an alphanumeric key is pressed, a `keydown` event is first generated. This is followed immediately by a `keypress` event. Then when the key is released, a `keyup` event is generated.

The `keydown` and `keyup` events are actually different from the `keypress` event. It is easier to understand the difference between these events if you distinguish between a “key” and a “character.” You can think of a key as a physical key on the device you are using, while a character is a symbol (A, a, b, @, \$, &, and so on) that is typed when you press a physical key.

In general, the `keydown` and `keyup` events represent physical keys that are pressed down or released. The `keypress` event, on the other hand, represents which character is typed. There are two properties on a key event that are interesting to read when you want to know which key is pressed or which character the pressed key corresponds to:

- `keyCode` contains a virtual keycode that gives you information about which key the user pressed. The virtual keycode corresponds to the ASCII value for the uppercase version of the key. For example, pressing the key labeled “A” results in the virtual keycode 65, regardless of whether or not Caps lock is enabled.
- `charCode` gives you the ASCII value for the resulting character.

An example should make this easier to understand. Assume that you add your event handlers for `keydown`, `keyup`, and `keypress` with this code:

```
document.addEventListener('keydown', handleKeyDown, false);
document.addEventListener('keyup', handleKeyUp, false);
document.addEventListener('keypress', handleKeyPress, false);
```

Further assume that your three event handlers use `console.log()` to write a message to the console with the `charCode` and the `keyCode` like this:

```
function handleKeyDown(event) {
    console.log("keydown - keyCode=%d, charCode=%d",
        event.keyCode, event.charCode);
}

function handleKeyUp(event) {
    console.log("keyup - keyCode=%d, charCode=%d",
        event.keyCode, event.charCode);
}

function handleKeyPress(event) {
    console.log("keypress - keyCode=%d, charCode=%d",
        event.keyCode, event.charCode);
}
```

If you press the key labeled “A” (without the Caps lock enabled) with the event handlers set up as shown here, you get the following result in the console of Google Chrome:

```
keydown - keyCode=65, charCode=0
keypress - keyCode=97, charCode=97
keyup - keyCode=65, charCode=0
```

In Firefox, pressing the key labeled “A” (without the Caps lock enabled) generates the following result in the console:

```
keydown - keyCode=65, charCode=0  
keypress - keyCode=0, charCode=97  
keyup - keyCode=65, charCode=0
```

As you can see, all three events — `keydown`, `keypress`, and `keyup` — are generated in both the browsers for this alphanumeric key. The virtual keycode given by `event.keyCode` is the same for the `keydown` and `keyup` event in both the browsers and has the value 65. This is the ASCII value for the character “A”.

However, for the `keypress` event the `keyCode` in Google Chrome is 97, which corresponds to the ASCII value for the character “a”. In Firefox, the `keyCode` for the `keypress` event is instead 0 (zero), and you need to read the `charCode` for the `keypress` to get the corresponding information.

For nonprinting function keys — such as Esc, the arrow keys, or F1 through F12 — a `keydown` and `keyup` event is generated in both Chrome and Firefox. However, Firefox also generates a `keypress` event while Chrome does not. As an example, if you press the Esc key in Chrome (again with the event handlers printing `keyCode` and `charCode`), you get this result:

```
keydown - keyCode=27, charCode=0  
keyup - keyCode=27, charCode=0
```

If you press the Esc key in Firefox, you get this result:

```
keydown - keyCode=27, charCode=0  
keypress - keyCode=27, charCode=0  
keyup - keyCode=27, charCode=0
```



The key event handling is an area that behaves differently in different browsers. This means that when your WebGL application uses key events, you should be prepared to test your application very carefully in the different browsers you want to support.

As shown in these basic examples, you quickly run into the problem that different browsers behave in slightly different ways when it comes to key input events. However, when using key input to affect something in your WebGL scene, it is often more important to know which key was pressed rather than which character the key corresponds to. In a racing game, for example, you typically want to know if the user pressed the gas pedal or the break. In a flight simulator, you might want to know if the user pressed the key that fires a missile.

If you were observant when going through the examples in this section, you probably recognized that the `keyCode` property for the `keydown` and `keyup` events behaved in the same way for all the examples. It turns out that using the `keyCode` property for the `keydown` and `keyup` events can often be a good strategy for your WebGL application. You can see the `keyCodes` for some useful keys in [Table 6-2](#).

TABLE 6-2: Some JavaScript keyCodes

KEY	CODE	KEY	CODE
Escape	27	G	71
left arrow	37	H	72
up arrow	38	I	73
right arrow	39	J	74
down arrow	40	K	75
0	48	L	76
1	49	M	77
2	50	N	78
3	51	O	79
4	52	P	80
5	53	Q	81
6	54	R	82
7	55	S	83
8	56	T	84
9	57	U	85
A	65	V	86
B	66	W	87
C	67	X	88
D	68	Y	89
E	69	Z	90
F	70		



If you search the web for “JavaScript keyCodes” or “JavaScript keyCode test,” you will find several web pages that have a text field and a short snippet of JavaScript code that prints the corresponding keyCode when you press a key on your keyboard.

Handling a Single-Key Press or Multiple Simultaneously Pressed Keys

From what you have learned about key press handling so far, you know how to take an action based on a single keydown, keyup, or keypress event. This is good enough in many cases. However, sometimes you want to be able to simultaneously use two or more keys for some functionality in an application. This is especially common in games.

Assume that you are writing a racing game with WebGL and that you want to have one key that corresponds to the gas pedal of the car and two other keys to steer the car. Further assume that this is a nasty racing game that allows you to fire missiles at your competitors, so you want to dedicate yet another key to fire the missiles. You probably want the player to be able to press the gas pedal at the same time he is steering the car and also be able to fire missiles at the same time.

In the following code snippet, the event handler `handleKeyDown()` is registered for the `keydown` event and `handleKeyUp()` is registered for the `keyup` event.

Pressing the key labeled “M” has an immediate action, which is to fire a missile. In the function `handleKeyDown()`, the static method `String.fromCharCode()` is used to convert the ASCII value of the `keyCode` to a character. It would have worked fine to compare the `keyCode` directly with the ASCII code for “M”, which is 77, but using the construct with `String.fromCharCode()`, you don’t need to know which ASCII code “M” corresponds to in order to write or read this code.

To be able to handle the user pressing the gas pedal and making a turn at the same time, you need to remember which keys are currently pressed down. This is tracked in the variable `pwgl.listOfPressedKeys`. This list is then used in the function `handlePressedDownKeys()`, which is called once per frame, typically before the scene is drawn. In this function, there are checks to see which keys are currently pressed down, and based on this, different actions are taken. Note that in this simple example, the handling of the gas pedal in the function `handlePressedDownKeys()` leads to an acceleration that is actually dependent on the frame rate. To avoid this, you can typically use the technique you learned in the section “Compensating Movement for Different Frame Rates” previously in this chapter.

```
function handleKeyDown(event) {
    // When you get a keydown you first handle any immediate actions
    // that are relevant.
    if (String.fromCharCode(event.keyCode) == "M") {

        // Fire missile since M key was pressed down
        fireMissile();
    }

    // Store information about which key has been pressed
    // so we can check this in the function handlePressedDownKeys().
    // This strategy let you have several keys pressed down simultaneously
    pwgl.listOfPressedKeys[event.keyCode] = true;
}

function handleKeyUp(event) {
    // Update list of keys that are pressed down,
    // by setting the released key to false;
    pwgl.listOfPressedKeys[event.keyCode] = false;
}

...

function handlePressedDownKeys() {
    if (pwgl.listOfPressedKeys[38]) {
        // Arrow up, the user pressed the gas pedal.
        speed += 0.5;
    }
    if (pwgl.listOfPressedKeys[40]) {
        // Arrow down, the user pressed the brake.
        speed -= 0.5;
    }
    if (pwgl.listOfPressedKeys[37]) {
        // Arrow left, the user wants to turn left.
    }
}
```

```

        turnLeft();
    }
    if (pwgl.listOfPressedKeys[39]) {
        // Arrow right, the user wants to turn right.
        turnRight();
    }
}

```

Mouse Input

Now that you know how to handle key events in your WebGL application, it is time to have a quick look at mouse events, which are another important way to get user input to your WebGL application. Several different mouse events exist, but these three basic ones can be especially useful in your WebGL application:

- `mousemove`
- `mousedown`
- `mouseup`

The `mousemove` event is generated when the mouse is moved over an element; the `mousedown` event is generated when the mouse button is pressed down on an element; and the `mouseup` event is generated when the mouse button is released over an element. Each of the three events has a number of properties that contain useful information. All three events contain the properties `clientX` and `clientY`, which contain the distance from the mouse pointer to the upper-left corner of the browser's viewport.

The `mousedown` and `mouseup` events also contain a property called `button` that specifies which mouse button is pressed or released. If `button` equals 0 (zero), it means that the event is related to the left mouse button; if `button` equals 1 (one), the event is related to the middle mouse button; and if `button` equals 2 (two), the event is related to the right mouse button.

You can register an event handler for each of these three mouse events with the following lines of code:

```

document.addEventListener('mousemove', handleMouseMove, false);
document.addEventListener('mousedown', handleMouseDown, false);
document.addEventListener('mouseup', handleMouseUp, false);

```

The following code contains a simple event handler function for each of the three events. The only thing that the event handler functions do in this case is to use `console.log()` to write a message to the browser console with information about which event is generated and the value for the properties `clientX` and `clientY`. For the `mousedown` and `mouseup` events, it also writes the value for the `button` property to the browser console.

```

function handleMouseMove(event) {
    console.log("mousemove - clientX=%d, clientY=%d",
               event.clientX, event.clientY);
}

function handleMouseDown(event) {
    console.log("mousedown - clientX=%d, clientY=%d, button=%d",
               event.clientX, event.clientY, event.button);
}

```

```

function handleMouseUp(event) {
    console.log("mouseup - clientX=%d, clientY=%d, button=%d",
                event.clientX, event.clientY, event.button);
}

```

With these three event handlers installed, you get the following messages in the console if you move the mouse pointer from the top-left corner downwards to the window coordinate where x=5 and y=5.

```

mousemove - clientX=0, clientY=0
mousemove - clientX=1, clientY=0
mousemove - clientX=2, clientY=0
mousemove - clientX=2, clientY=1
mousemove - clientX=2, clientY=2
mousemove - clientX=2, clientY=3
mousemove - clientX=3, clientY=3
mousemove - clientX=3, clientY=4
mousemove - clientX=4, clientY=4
mousemove - clientX=5, clientY=5

```

In a similar way, you get these messages in the console if you press and release the left mouse button at the window coordinate where x=5 and y=5:

```

mousedown - clientX=5, clientY=5, button=0
mouseup - clientX=5, clientY=5, button=0

```

If you press and release the right mouse button on the same location, you get these messages in the console:

```

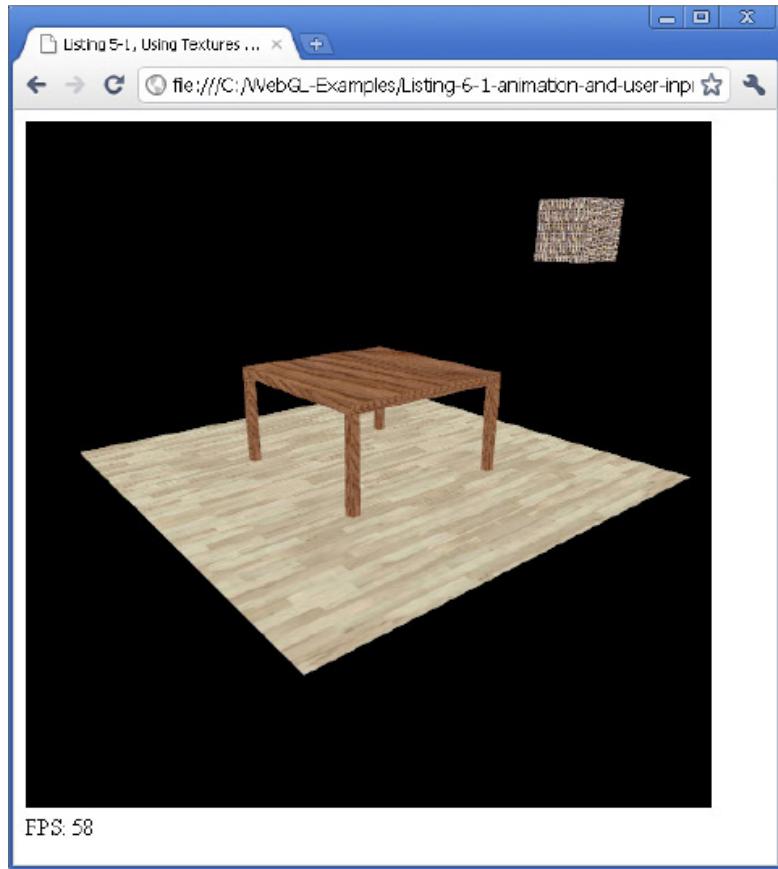
mousedown - clientX=5, clientY=5, button=2
mouseup - clientX=5, clientY=5, button=2

```

APPLYING YOUR NEW KNOWLEDGE

It is now time to apply what you have learned in this chapter to a larger example. This example is based on the same scene that you worked with in Chapter 5, but in this case, the scene is animated to let the box that was previously static on top of the table move around in a circle above the table. In addition, you can use the up and down arrow keys to change the radius of the circle that the box is moving in. [Figure 6-3](#) shows what the scene looks like.

FIGURE 6-3: Animated scene that corresponds to [Listing 6-1](#)



The source code that corresponds to this scene is available in [Listing 6-1](#). You should be able to understand most of the code without any further explanations, since it is mainly based on what you have learned up to this point in the book.

The complete source code for [Listing 6-1](#) is available for you to download from the book's website on [Wrox.com](#), but since you are probably starting to get very familiar with the code that is needed to create the WebGL context, load the shaders, compile the shaders, link the shaders, set up the vertex buffers and so on, not all of this code is shown here. You have seen this code many times in the previous chapters. The following text provides snippets from [Listing 6-1](#) and provides explanation focusing on the things that are a bit newer to you.

To get started however, the beginning of the HTML file is shown here. After some initial HTML code, this snippet includes a vertex shader and a fragment shader that should be familiar to you by now.



Available for
download on
[Wrox.com](#)

[LISTING 6-1: An animated WebGL scene](#)

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 6-1, Animated WebGL Scene</title>
<script src="webgl-debug.js"></script>
<script type="text/javascript" src="glMatrix.js"></script>
<script src="webgl-utils.js"></script>
```

```

<meta charset="utf-8">

<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec2 aTextureCoordinates;

    uniform mat4 uMVMMatrix;
    uniform mat4 uPMatrix;

    varying vec2 vTextureCoordinates;

    void main() {
        gl_Position = uPMatrix * uMVMMatrix * vec4(aVertexPosition, 1.0);
        vTextureCoordinates = aTextureCoordinates;
    }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
    precision mediump float;

    varying vec2 vTextureCoordinates;
    uniform sampler2D uSampler;
    void main() {
        gl_FragColor = texture2D(uSampler, vTextureCoordinates);
    }
</script>

```

The next step is to look at some code from the end of the HTML file, where you have the `<body>` tag that includes the `<canvas>` tag and the `<div>` tag for the FPS counter. It also includes the `startup()` function that is called when the `onload` event is triggered.

The `startup()` function first retrieves a reference to the canvas by using the method `document.getElementById()` (as you have seen many times before). Then the method `WebGLDebugUtils.makeLostContextSimulatingContext()` from the library `webgl-debug.js` is called to create a wrapper canvas around your original canvas so you can simulate the events `webglcontextlost` and `webglcontextrestored`. Also note that there are three lines related to the simulation of lost context that are commented out in this function. These lines should be “uncommented” to be able to simulate the lost context with a mouse press. Event listeners are added to the canvas, and then the `WebGLRenderingContext` is created by calling the function `createGLContext()`.

Then there is a call to the function `init()`. This function contains the setup and initialization code that you need the first time your application is started and also in case you get a lost context followed by a restored context. This way the `init()` function can be called both from the `startup()` function when the application is started and also from the function `handleContextRestored()` that is called when a previously lost context is restored again. Finally, when everything is set up, the `startup()` function calls the `draw()` function to start drawing the scene.

```

function startup() {
    canvas = document.getElementById("myGLCanvas");
    canvas = WebGLDebugUtils.makeLostContextSimulatingCanvas(canvas);

```

```

        canvas.addEventListener('webglcontextlost', handleContextLost, false);
        canvas.addEventListener('webglcontextrestored',
            handleContextRestored, false);
    document.addEventListener('keydown', handleKeyDown, false);
    document.addEventListener('keyup', handleKeyUp, false);
    document.addEventListener('keypress', handleKeyPress, false);
    document.addEventListener('mousemove', handleMouseMove, false);
    document.addEventListener('mouseup', handleMouseUp, false);

    gl = createGLContext(canvas);
    init();

    pwgl.fpsCounter = document.getElementById("fps");

    // Uncomment the three lines of code below to be able to test lost
    context
    // window.addEventListener('mousedown', function() {
    //     canvas.loseContext();
    // });

    // Draw the complete scene
    draw();
}
</script>

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
    <div id="fps-counter">
        FPS: <span id="fps">--</span>
    </div>
</body>

</html>

```

Now you should look at the code of the `init()` function to get a feeling of what is included here. As you can see it makes sure that the shaders, vertex buffers, and textures are set up. In addition it contains some other initialization.

```

function init() {
    // Initialization that is performed during first startup and when the
    // event webglcontextrestored is received is included in this function.
    setupShaders();
    setupBuffers();
    setupTextures();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);

    // Initialize some variables for the moving box
    pwgl.x = 0.0;
    pwgl.y = 2.7;
    pwgl.z = 0.0;
    pwgl.circleRadius = 4.0;
    pwgl.angle = 0;
    // Initialize some variables related to the animation
    pwgl.animationStartTime = undefined;

```

```

    pwgl.nbrOfFramesForFPS = 0;
    pwgl.previousFrameTimeStamp = Date.now();
}

```

Now that you have seen the `init()` function, it can be appropriate to look at the event handlers for the context lost and context restored. The function `handleContextLost()` is shown first. It calls the method `event.preventDefault()` to prevent the default behavior when you get a lost context. The reason is that the default behavior is such that the context is never restored, and this is not what you want this time. It also cancels the animation and loop through all ongoing image loads and makes sure that they are ignored by setting their `onload` handler to `undefined`.

```

function handleContextLost(event) {
    event.preventDefault();
    cancelRequestAnimFrame(pwgl.requestId);

    // Ignore all ongoing image loads by removing
    // their onload handler
    for (var i = 0; i < pwgl.ongoingImageLoads.length; i++) {
        pwgl.ongoingImageLoads[i].onload = undefined;
    }
    pwgl.ongoingImageLoads = [];
}

```

When a context is restored the function `handleContextRestored()` is called, which calls `init()` to initialize your states again, and then the animation is started:

```

function handleContextRestored(event) {
    init();
    pwgl.requestId = requestAnimFrame(draw, canvas);
}

```

Next the event handlers for key presses are shown. The functions `handleKeyDown()` and `handleKeyUp()` are used to update a list of the currently pressed down keys. This list is then read once per frame by the function `handlePressedDownKeys()`. If the arrow up key is pressed, the radius of the circle that the box in the scene is moving in is increased by 0.1; if the arrow down key is pressed, the radius of the circle is decreased by 0.1. The function `handleKeyPress()` only contains commented out code that can write to `console.log()`.

```

function handleKeyDown(event) {
    pwgl.listOfPressedKeys[event.keyCode] = true;

    // If you want to have a log for keydown you can
    // uncomment the two lines below.
    // console.log("keydown - keyCode=%d, charCode=%d",
    //             event.keyCode, event.charCode);
}

function handleKeyUp(event) {
    pwgl.listOfPressedKeys[event.keyCode] = false;

    // If you want to have a log for keyup you can
    // uncomment the two lines below.
    // console.log("keyup - keyCode=%d, charCode=%d",
    //             event.keyCode, event.charCode);
}

```

```

function handleKeyPress(event) {
    // If you want to have a log for keypress you can
    // uncomment the two lines below.
    // console.log("keypress - keyCode=%d, charCode=%d",
    //             event.keyCode, event.charCode);
}

function handlePressedDownKeys() {
    if (pwgl.listOfPressedKeys[38]) {
        // Arrow up, increase radius of circle
        pwgl.circleRadius += 0.1;
    }
    if (pwgl.listOfPressedKeys[40]) {
        // Arrow down, decrease radius of circle
        pwgl.circleRadius -= 0.1;
        if (pwgl.circleRadius < 0) {
            pwgl.circleRadius = 0;
        }
    }
}

```

Finally it is time to look at the `draw()` function, which is responsible for drawing the complete scene. The `draw()` function starts with a call to `requestAnimationFrame()` to request the next drawing call for the animation. Then it calls the function `handlePressedDownKeys()`, which is the function you just looked at. When the keys are handled, the `draw()` function sets up the transformations and draws the scene with help of other helper functions.

```

function draw(currentTime) {
    pwgl.requestId = requestAnimationFrame(draw);
    if (currentTime === undefined) {
        currentTime = Date.now();
    }

    handlePressedDownKeys();

    // Update FPS if a second or more has passed since last FPS update
    if(currentTime - pwgl.previousFrameTimeStamp >= 1000) {
        pwgl.fpsCounter.innerHTML = pwgl.nbrOfFramesForFPS;
        pwgl.nbrOfFramesForFPS = 0;
        pwgl.previousFrameTimeStamp = currentTime;
    }

    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
                     1, 100.0, pwgl.projectionMatrix);
    mat4.identity(pwgl.modelViewMatrix);
    mat4.lookAt([8, 5, 10],[0, 0, 0], [0, 1,0], pwgl.modelViewMatrix);

    uploadModelViewMatrixToShader();
    uploadProjectionMatrixToShader();
    gl.uniform1i(pwgl.uniformSamplerLoc, 0);

    drawFloor();

    // Draw table
}

```

```

pushModelViewMatrix();
    mat4.translate(pwgl.modelViewMatrix, [0.0, 1.1, 0.0],
pwgl.modelViewMatrix);
uploadModelViewMatrixToShader();
drawTable();
popModelViewMatrix();

// Calculate the position for the box that is initially
// on top of the table but will then be moved during animation
pushModelViewMatrix();
if (currentTime === undefined) {
    currentTime = Date.now();
}
if (pwgl.animationStartTime === undefined) {
    pwgl.animationStartTime = currentTime;
}
// Update the position of the box
if (pwgl.y < 5) {
    // First move the box vertically from its original position on top of
    // the table (where y = 2.7) to 5 units above the floor (y = 5).
    // Let this movement take 3 seconds
    pwgl.y = 2.7 + (currentTime - pwgl.animationStartTime)/3000 * (5.0-
2.7);
}
else {
    // Then move the box in a circle where one revolution takes 2 seconds
    pwgl.angle = (currentTime - pwgl.animationStartTime)/
        2000*2*Math.PI % (2*Math.PI);

    pwgl.x = Math.cos(pwgl.angle) * pwgl.circleRadius;
    pwgl.z = Math.sin(pwgl.angle) * pwgl.circleRadius;
}

mat4.translate(pwgl.modelViewMatrix,
    [pwgl.x, pwgl.y, pwgl.z], pwgl.modelViewMatrix);
mat4.scale(pwgl.modelViewMatrix, [0.5, 0.5, 0.5], pwgl.modelViewMatrix);
uploadModelViewMatrixToShader();
drawCube(pwgl.boxTexture);
popModelViewMatrix();

// Update number of drawn frames to be able to count fps
pwgl.nbrOfFramesForFPS++;
}

```

SUMMARY

In this chapter you learned how to animate your WebGL scene to create the illusion of motion. You looked at the legacy way to build your render loop using `setInterval()` or `setTimeout()`, but you also learned the advantages of using the newer `requestAnimationFrame()` method. You learned how to measure the FPS of your animation and how to display it on the screen. You also now understand the consequences of using the nonlinear FPS as a measurement of the performance of your WebGL application.

You learned about event handling and, specifically, how you can use it to handle user input. You now also know the difference between DOM Level 0 event handling and DOM Level 2 event handling. Finally, you saw that the key input handling was an area of event handling that was complicated since different browsers have different behaviors.

In the next chapter, you will learn about the exciting topic of how you can simulate lights in WebGL. This will be an important ingredient when you want to create more realistic WebGL scenes.

Chapter 7

Lighting

WHAT'S IN THIS CHAPTER?

- The difference between a local lighting model and a global lighting model
- How to use the Phong reflection model for lighting and how to implement it in OpenGL ES Shading Language
- The difference between ambient reflection, diffuse reflection, and specular reflection
- The difference between an illumination model and the interpolation technique
- The differences between flat shading, Gouraud shading, and Phong shading
- How directional lights, point lights, and spot lights can be implemented in OpenGL ES Shading Language
- How to simulate the attenuation of light

Lighting is an important ingredient when you want to create a realistic 3D scene. In the real world, light greatly affects how you see the environment around you. For example, think about what the sea looks like on a sunny day compared to what it looks like on a really cloudy day. In this chapter, you will learn how to handle lighting in WebGL.

UNDERSTANDING LIGHT

From your physics class at school, you might remember that light is electromagnetic radiation. The human eye can perceive light with a wavelength between approximately 400 and 700nm. Light with a wavelength around 400nm is perceived as violet, and light with a wavelength around 700nm is perceived as red.

In 3D graphics you typically want to determine how different light sources (like the sun or a light bulb) affect the different surfaces you have in your scene. To do this in an exact way is very difficult since the interaction between light and different objects and materials is quite complicated.

Even if you have only one light source, it can still emit light rays in different directions. The light travels along a straight line until it is reflected or absorbed by an object. The light can then be reflected several times, and the reflected light can illuminate other objects.

The light can also be refracted — for example, when hitting a water surface — and if the light hits an object that scatters it, it can continue in several different directions. To simulate real-world light in 3D graphics, you obviously need to use some sort of model that can be simplified compared to how real-world light works.

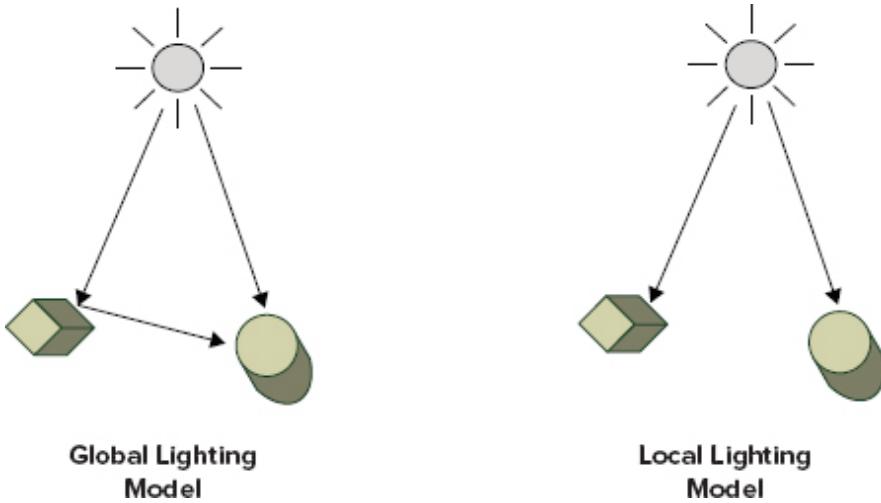
WORKING WITH A LOCAL LIGHTING MODEL

When simulating light in 3D graphics, you can use two different types of lighting models:

- Global lighting model
- Local lighting model

[Figure 7-1](#) illustrates the difference between a global lighting model and a local lighting model.

FIGURE 7-1: The difference between a global lighting model and a local lighting model



A global lighting model uses information from other objects than the ones that the light directly illuminates. For example, the model can take into account that light that originates from a light source is reflected on one object, and that the reflected light then illuminates a second object.

An example of a global lighting model is *ray tracing*. This technique tries to mimic the complicated behavior of light. It can produce a highly realistic scene, but because it requires a lot of computer resources, it is most commonly used when you render the scene well in advance of when you need to show it, such as for still images or film. Another example of a global lighting model is *radiosity*, but even radiosity is an uncommon algorithm for real-time 3D graphics.

A local lighting model only accounts for the light that comes directly from designated light sources. As a result, objects are not illuminated by light that is reflected from other objects. Another important property of a local lighting model is that objects will not block light that hits them. This means that shadows are not automatically created in a local lighting model.

Since WebGL is generally used for real-time 3D graphics, some kind of local lighting model is often used in WebGL. If you want to have shadows in a local lighting model, you can create them by using a technique such as shadow mapping. A common local lighting model that can be used for WebGL, and that you will learn about in this chapter, is the Phong reflection model.

UNDERSTANDING THE PHONG REFLECTION MODEL

The *Phong reflection model* is named after a computer graphics researcher named Bui Tuong Phong, who developed this model and published it in 1973. To understand the model, you should note that the color of a real-world object is indicated by the color of the light that leaves its surface. For example, an object that looks red reflects (or possibly transmits) mostly red light.



Note that the Phong reflection model is different from Phong shading, even though Phong shading was also developed and published by the same Bui Tuong Phong. You will learn about Phong shading later in this chapter.

In the Phong reflection model, the resulting color of a point (a vertex or a fragment) is the sum of three different reflection components:

- Ambient
- Diffuse
- Specular

This means that you can describe the total color at a vertex or fragment as the total reflection at the vertex or fragment:

$$\text{Total Reflection} = \text{Ambient Reflection} + \text{Diffuse Reflection} + \text{Specular Reflection}$$

To calculate each of the three different reflection components, the model also specifies that all light sources in the scene have three different light components: ambient light, diffuse light, and specular light.

Furthermore, each material in the scene has three corresponding material properties that describe the ambient reflectivity, the diffuse reflectivity, and the specular reflectivity of the material. In addition to these three material properties, there is also a fourth material property, which is the shininess that is used to calculate the specular reflection component.

The “Ambient Reflection,” the “Diffuse Reflection,” and the “Specular Reflection” in the previous equation are the result of the interaction between the light and the material property for the corresponding component. You will learn more about this interaction in the following sections.

One point that should be clarified now is that ambient light is light that exhibits ambient reflection, diffuse light is light that exhibits diffuse reflection, and specular light is light that exhibits specular reflection. This means that in some contexts when the Phong reflection model is discussed, people (myself included) tend to intermingle the wording “ambient light” with “ambient reflection,” “diffuse light” with “diffuse reflection,” and “specular light” with “specular reflection.”



*The Phong reflection model is sometimes referred to as the **ADS light model** after the first letter in each of the three reflection components, Ambient, Diffuse, and Specular.*

Ambient Reflection

Ambient light is light that has bounced around the scene so many times that it does not seem to come from any particular direction. As a result, all sides of an object are evenly lit by the ambient light. If you have an ambient light component called \mathbf{I}_a that contains an RGB value, and an ambient material property called \mathbf{k}_a that has one material property for the red light, one for the green light and one for the blue light, , then you get the ambient reflection \mathbf{I} that is reflected from the surface with the following equation:

$$\mathbf{I} = \mathbf{k}_a \mathbf{I}_a$$

So what you do is a component-based multiplication of \mathbf{k}_a and \mathbf{I}_a . You multiply the red component of the material property with the red component of the light, the green component of the material property with the green component of the light, and the blue component of the material property with the blue component of the light. With OpenGL ES Shading language, it looks something like this:

```
uniform vec3 uAmbientMaterial;  
uniform vec3 uAmbientLight;  
  
...  
  
vec3 ambientReflection = uAmbientMaterial * uAmbientLight;
```

You can see from the snippet of shader code and also from the equation ($\mathbf{I} = \mathbf{k}_a \mathbf{I}_a$) that the ambient light does not take the position or direction of the light

into account. It also does not take the viewing direction into account.



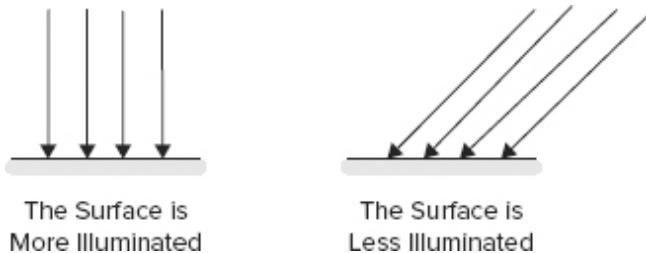
The Phong reflection model is a local lighting model, which means that it does not directly take into account that reflected light can illuminate other objects. The ambient light compensates for this, since the ambient light can be seen as light that has been reflected many times.

Without the ambient light, the sides of an object that are not directly illuminated by a light source would be totally black. For example, when you are indoors, you normally have walls, a ceiling, and other objects that reflect some light on surfaces that are not directly illuminated. Therefore, in an indoor scene, surfaces that are totally black when no light rays hit them directly do not look realistic.

Diffuse Reflection

Diffuse light takes the direction of the incoming light into account when the amount of reflected light is calculated. Light rays that hit a surface at a perpendicular angle reflect more light and therefore illuminate a surface more than light rays that hit a surface with an angle between the surface normal and the incoming light rays. [Figure 7-2](#) shows a simple illustration of this.

FIGURE 7-2: An example of diffuse reflection. The light rays to the left are perpendicular to the surface, thereby illuminating the surface more than the light rays to the right (which strike the surface at an angle)



For diffuse reflection, the light is reflected uniformly in all different directions. This means that the viewing direction doesn't matter. Matte and dull materials such as chalk and clay typically exhibit diffuse reflection.



When you read other texts about 3D graphics, keep in mind that diffuse reflection is sometimes referred to as Lambertian reflection.

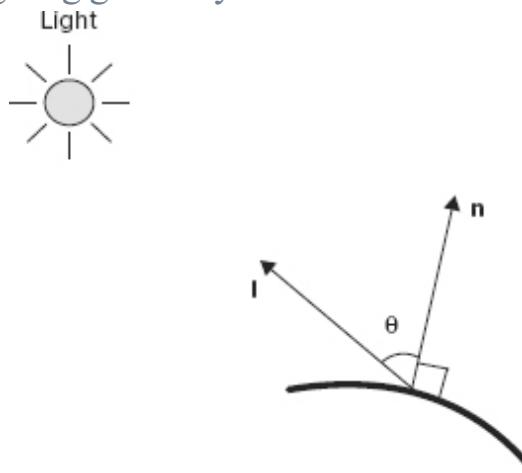
The equation for calculating diffuse reflectance is similar to the equation for ambient reflectance, but in addition to the diffuse material property k_d and the diffuse light component I_d , the diffuse reflectance also contains the factor $\cos \theta$. It is this factor that is used to take the direction of the incoming light into account. You can write the equation in two ways. This is the first way:

$$I = k_d I_d \max(\cos \theta, 0)$$

The angle θ is defined as the smallest angle between the surface normal n and the vector pointing in the light direction I . The $\max()$ function in the equation specifies that negative values for $\cos \theta$ should be clamped to zero (i.e. negative values will be set to zero). You will understand the reason for this as you continue to read.

[Figure 7-3](#) shows the lighting geometry for the diffuse reflectance. As you can see from the equation for the diffuse reflectance, the reflected light is greatest when the angle θ between the light and the normal is 0° , because $\cos 0^\circ$ is 1. When θ is 90° , it means that the light is coming in from the side and just touches the surface. Since $\cos 90^\circ$ is zero, the whole expression becomes zero, which means that no light is reflected.

[FIGURE 7-3](#): The lighting geometry for diffuse reflectance



Note that when the cosine term is negative, this means that the angle θ is larger than 90° , which means that the light rays come from behind the surface. These rays do not illuminate the surface, and therefore only non-negative values for cosine are considered. This is the reason for the $\max()$ function that clamps negative values of cosine to zero.

In Chapter 1, which discussed linear algebra for 3D graphics, you learned that the dot product or scalar product could be used in lighting calculations for

WebGL. Now it's time to show you how. First, to refresh your memory, one of the definitions of the dot product for the two vectors \mathbf{u} and \mathbf{v} is:

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}||\mathbf{v}| \cos \theta$$

Here the angle θ is the smallest angle between the vectors \mathbf{u} and \mathbf{v} . If you use this definition of the dot product and combine it with the equation for the diffuse reflectance, you can see that if both the normal \mathbf{n} and the light direction \mathbf{l} have unit length, there is a second way you can write the diffuse reflectance equation:

$$I = k_d I_d \max(n \cdot l, 0)$$

Again, note that negative values for the dot product in the formula are clamped to 0. Since the lighting calculations for WebGL are generally done in the vertex shader or the fragment shader and OpenGL ES Shading Language has the built-in function `dot()` to calculate the dot product, this second definition for the diffuse light is usually the most convenient to use in your WebGL application. The snippet of shader source code that calculates the diffuse reflectance looks something like this:

```

    0.0);

    vec3 diffuseReflectance = uDiffuseMaterial * uDiffuseLight
    *
    diffuseLightWeightning;

    ...
}

}

```

Note that this snippet starts with some code to calculate the input (such as the normal vector and the light vector in eye coordinates) that is needed to calculate the diffuse reflectance. This code should not be that difficult to understand with your current knowledge, but it will not be discussed until later in this chapter when you look at the complete shader code for the Phong reflection model. For now, you can concentrate on the highlighted code, which corresponds to the equation for the diffuse reflectance.



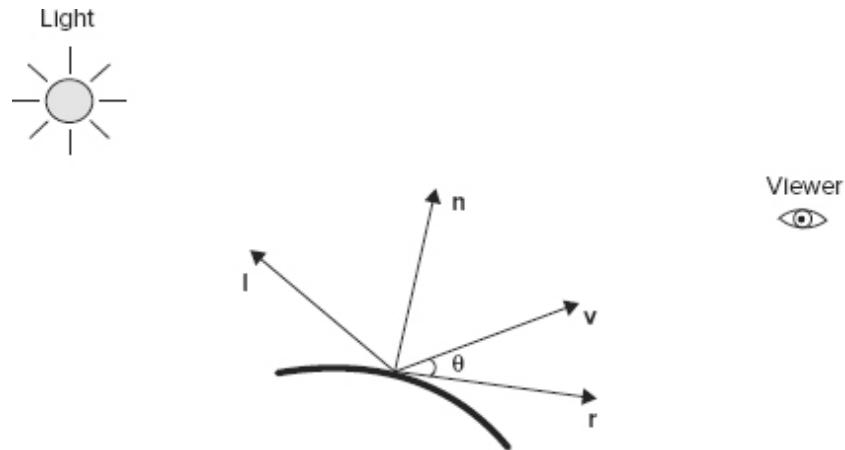
If you want to simplify lighting calculations in WebGL, it is possible to only use the diffuse reflectance and the ambient reflectance instead of using the complete Phong reflection model, and still get an adequate result.

Specular Reflection

If you have a very shiny object — for example, something made of polished metal — then the light that falls on the object can create a bright spot or a highlight on it. The specular reflection is used to model this behavior in Phong's reflection model.

You have seen that ambient reflection and diffuse reflection do not take the viewing direction into account. However, specular light is reflected similarly to how light is reflected in a mirror. Most of the light is reflected in a specific direction, and therefore the viewing direction is very important. The geometry for specular reflectance is shown in [Figure 7-4](#).

FIGURE 7-4: The geometry used for specular reflectance



In this figure, the light rays come from a specific direction given by the vector \mathbf{l} . Note that the vector is pointing towards the light source. The normal to the surface is given by \mathbf{n} . For a very shiny object, all light is generally reflected in the direction \mathbf{r} . For less shiny objects, the reflected light is spread around the vector \mathbf{r} .

The vector \mathbf{v} is pointing towards the viewer. Most light is reflected towards the viewer when the angle θ between \mathbf{v} and \mathbf{r} is zero. The larger the angle θ between the reflection vector \mathbf{r} and the viewer \mathbf{v} , the less light is reflected towards the viewer.

Usually this drop-off is approximated with $\cos \theta$ that is raised to the power α , where α represents the shininess of the material. The equation for the specular reflected light \mathbf{I} can be written like this:

$$\mathbf{I} = k_s \mathbf{I}_s \max(\cos \theta, 0)^\alpha$$

Here k_s and \mathbf{I}_s are similar to the corresponding factors for the ambient and diffuse reflectance. This means that the specular material property is given by k_s and the specular light component of the incoming light is given by \mathbf{I}_s . As already described, the angle θ is the angle between \mathbf{v} and \mathbf{r} , and α represents the shininess of the material. (You will learn more about shininess in the next section.)

In a manner similar to diffuse reflectance, it is possible to use the definition of the dot product to rewrite the expression for the specular reflectance. If the vectors \mathbf{r} , \mathbf{v} , \mathbf{n} and \mathbf{l} all have unit length, the specular reflection can be written on this second form:

$$\mathbf{I} = k_s \mathbf{I}_s \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha$$

The mathematical formula to calculate the reflectance vector \mathbf{r} is:

$$\mathbf{r} = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}-\mathbf{l}$$

However, you do not need to calculate the reflectance vector manually. The OpenGL ES Shading Language contains the useful built-in function `reflect()`, which you can use to calculate the reflectance vector \mathbf{r} based on the vector \mathbf{l} and the normal \mathbf{n} ; however, you have to be careful. The function `reflect()` assumes that the vector specifying the light direction points from the light source towards the surface, which is the opposite of how \mathbf{l} is defined (as shown in [Figure 7-4](#)). In the following snippet of OpenGL ES Shading Language, you can see what the code could look like if you were to calculate the specular reflectance in the vertex shader.

```
uniform vec3 uSpecularMaterial;
uniform vec3 uSpecularLight;
uniform vec3 uLightPosition;

uniform mat4 uMVMMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;

const float shininess = 32.0;

main() {
    // Get vertex position in eye coordinates
    vec4 vertexPositionEye4 = uMVMMatrix *
vec4(aVertexPosition, 1.0);
    vec3 vertexPositionEye3 = vertexPositionEye4.xyz /
vertexPositionEye4.w;

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition -
vertexPositionEye3);

    // Transform the normal (n) to eye coordinates
    vec3 normalEye = normalize(uNMatrix * aVertexNormal);

    // Calculate the reflection vector (r)
    vec3 reflectionVector = normalize(reflect(-
vectorToLightSource, normalEye));

    // The camera in eye coordinates is located in the origin
    // and pointing along the negative z-axis.
    // Calculate viewVectorEye (v) in eye coordinates as
```

```

// (0.0, 0.0, 0.0) - vertexPositionEye3
vec3 viewVectorEye = -normalize(vertexPositionEye3);

float rdotv = max(dot(reflectionVector, viewVectorEye),
0.0);
float specularLightWeight = pow(rdotv, shininess);

vec3 specularReflection = uSpecularMaterial *
uSpecularLight *
specularLightWeight;

...
}

}

```

In the same way as for the example shader code for the diffuse reflectance, this snippet for the specular reflectance contains some code that is used to calculate the input that is needed for the specular reflectance and that will be discussed later. The highlighted code corresponds to the equation for the specular reflectance.

Understanding Shininess

As already described, the shininess for the specular reflectance is given by α in the specular reflectance equation. A large value for α represents high shininess and means that the drop-off in intensity will happen faster when the angle θ between \mathbf{r} and \mathbf{v} increases. A small value for α represents lower shininess and means that the drop-off will happen more gradually when the angle θ between \mathbf{r} and \mathbf{v} increases.

To give you a better understanding of how the shininess α affects the reflected light, [Figure 7-5](#) shows a diagram of how the factor $(\cos \theta^\circ)^\alpha$ (which affects the intensity of the reflected specular light) varies for different values of the angle θ and the shininess α . The numeric values that correspond to the diagram are shown in [Table 7-1](#).

FIGURE 7-5: A diagram showing $(\cos \theta^\circ)^\alpha$, which affects the intensity of the specular highlight. The result is shown for different values of the shininess (α) and the angle (θ) between the viewing direction (\mathbf{v}) and reflection direction (\mathbf{r})

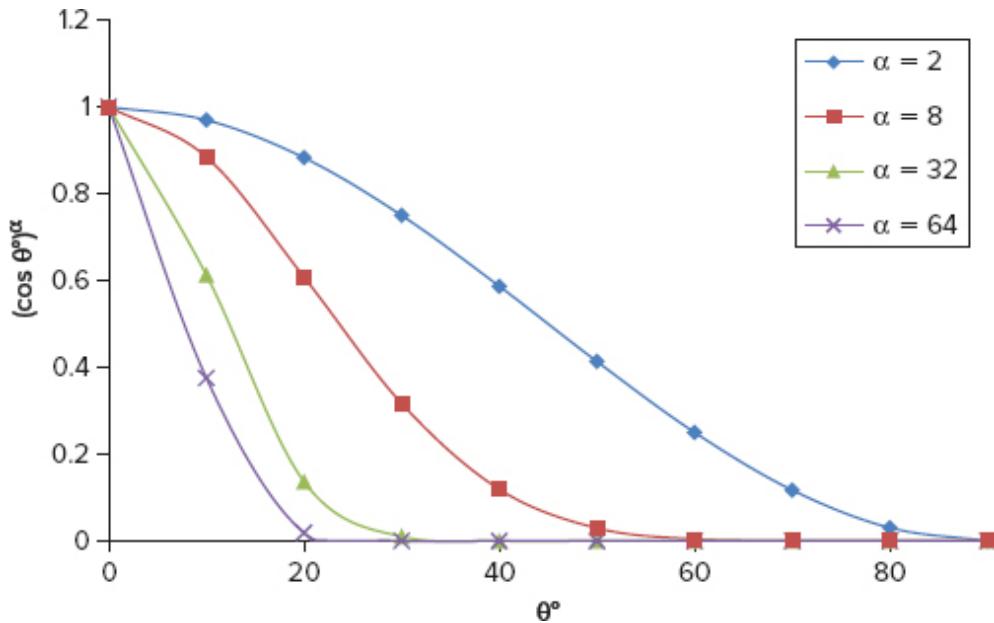


TABLE 7-1: Intensity factor $(\cos \theta^\circ)^\alpha$ for different angles θ and shininess α

ANGLE θ°	$\alpha = 2$	$\alpha = 8$	$\alpha = 32$	$\alpha = 64$
0°	1	1	1	1
10°	0.97	0.88	0.61	0.38
20°	0.88	0.61	0.14	0.02
30°	0.75	0.32	0.01	0.00
40°	0.59	0.12	0.00	0.00
50°	0.41	0.03	0.00	0.00
60°	0.25	0.00	0.00	0.00
70°	0.12	0.00	0.00	0.00
80°	0.03	0.00	0.00	0.00
90°	0.00	0.00	0.00	0.00

Counting from the top, the first curve in Figure 7-5 shows how the factor $(\cos \theta^\circ)^\alpha$ varies for different angles θ when the shininess $\alpha = 2$, which is regarded as a low value for shininess. You can see that for this value, the intensity drops off quite slowly when the angle θ increases. For example, when the angle between the reflectance vector and the viewer is 30° , the intensity factor has “only” dropped off to 0.75.

The second curve from the top shows the intensity factor when $\alpha = 8$, the third curve when $\alpha = 32$, and the bottom curve when $\alpha = 64$. When the

shininess α is as large as 64, you can see that even for the relatively small angle of 10° , the intensity factor has dropped off to 0.38.

One more thing that is important to understand about the shininess α is that it only affects how fast the intensity declines when the angle between \mathbf{r} and \mathbf{v} increases. This means that it affects the size of the specular highlight but not the intensity (how bright it is). When the view vector (\mathbf{v}) and the reflection vector (\mathbf{r}) coincide (that is $\mathbf{v} = \mathbf{r}$), the factor $(\cos \theta^\circ)^\alpha$ is always 1, regardless of the value of shininess α .

Understanding the Complete Equation and Shaders for the Phong Reflection Model

Now that you have looked at the different reflection components of the Phong reflection model, it is time to put it all together. The complete equation for the Phong reflection model can be written like this:

$$\mathbf{I}_{\text{Phong}} = \mathbf{k}_a \mathbf{I}_a + \mathbf{k}_d \mathbf{I}_d \max(\mathbf{n} \cdot \mathbf{l}, 0) + \mathbf{k}_s \mathbf{I}_s \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha$$

where the mathematical formula for calculating the reflectance vector \mathbf{r} is given by:

$$\mathbf{r} = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}-\mathbf{l}$$

The meanings of the different variables have already been discussed in detail in the previous sections, but for your convenience, they are briefly listed here:

- $\mathbf{I}_{\text{Phong}}$ — The resulting lighting intensity
- \mathbf{k}_a — The ambient material property. A high value means that more light is reflected
- \mathbf{I}_a — The ambient light component
- \mathbf{k}_d — The diffuse material property. A high value means that more light is reflected
- \mathbf{I}_d — The diffuse light component
- \mathbf{n} — The surface normal in unit length
- \mathbf{l} — The light vector in unit length and pointing towards the light source
- \mathbf{k}_s — The specular material property. A high value means that more light is reflected
- \mathbf{I}_s — The specular light component

- \mathbf{r} — The reflectance vector in unit length
- \mathbf{v} — The view vector in unit length and pointing towards the viewer
- α — The shininess of the material

As you can see, there are many parameters to take into account when you want to calculate the resulting lighting intensity.



Note that if you have several light sources, their individual contributions should be added to the total light intensity. Sometimes the equation for the Phong reflection model is written with a Σ to show that all the individual components from all light sources should be added.

[Listing 7-1](#) shows a complete vertex shader that uses the Phong reflection model to calculate the lighting. This code contains a minor simplification compared to the equation for the Phong reflection model and the snippets of code that were presented in each of the previous sections describing the ambient, diffuse, and specular reflection.

This simplification is sometimes applied when the Phong reflection model is actually implemented in shader source code. The idea is that the light component and the material component have been multiplied together in advance. This means that the shader uniform variable `uAmbientLightColor` corresponds to $\mathbf{k}_a \mathbf{I}_a$, the uniform `uDifuseLightColor` corresponds to $\mathbf{k}_d \mathbf{I}_d$, and the uniform `uSpecularLightColor` corresponds to $\mathbf{k}_s \mathbf{I}_s$.

If you do not intend to have different material properties for different vertices, this optimization often makes sense.



For lighting, a lot of the important code is located in the vertex shader and the fragment shader. Therefore, several of the code listings in this chapter show only the shader source code. The interesting parts of the JavaScript code will also be described, but to save space and not bore you too much, they will not be included for every code listing.

However, since you need the complete source code (including the JavaScript) to execute the shaders, all of the source code is available in the code listings that you can download from www.wrox.com.

Note that one WebGL application that you can download from www.wrox.com often corresponds to two code listings in this chapter of the book since it contains both a vertex shader and a fragment shader. For example, when you download the source code, the WebGL application located in the folder entitled “Listing-7-1-and-7-2-per-vertex-lighting-Phong-reflection-model-no-textures” corresponds to both [Listing 7-1](#) (the vertex shader) and [Listing 7-2](#) (the fragment shader) in the book. In addition to the vertex shader and the fragment shader, the downloaded code also contains the JavaScript code so that you can actually try the shaders by loading the complete WebGL application into your web browser.



Available for
download on
Wrox.com

[LISTING 7-1: The vertex shader for the Phong reflection model implemented with per-vertex shading](#)

```
<script id="shader-vs" type="x-shader/x-vertex">
    // Vertex shader implemented to perform lighting according
    to the
    // Phong reflection model.
    attribute vec3 aVertexPosition;
    attribute vec3 aVertexNormal;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    uniform mat3 uNMatrix;
    uniform vec3 uLightPosition;
    uniform vec3 uAmbientLightColor;
    uniform vec3 uDiffuseLightColor;
    uniform vec3 uSpecularLightColor;

    varying vec3 vLightWeighting;

    const float shininess = 32.0;

    void main() {
        // Get the vertex position in eye coordinates
```

```

        vec4    vertexPositionEye4      =      uMVMMatrix     *
vec4(aVertexPosition, 1.0);
        vec3    vertexPositionEye3   =  vertexPositionEye4.xyz / 
vertexPositionEye4.w;

        // Calculate the vector (l) to the light source
        vec3    vectorToLightSource = normalize(uLightPosition -
vertexPositionEye3);

        // Transform the normal (n) to eye coordinates
        vec3    normalEye = normalize(uNMatrix * aVertexNormal);

        // Calculate n dot l for diffuse lighting
        float diffuseLightWeightning = max(dot(normalEye,
vectorToLightSource),
0.0);

        // Calculate the reflection vector (r) that is needed
for specular light
        vec3    reflectionVector     =      normalize(reflect(-
vectorToLightSource,
normalEye));

        // The camera in eye coordinates is located in the
origin and is pointing
        // along negative z-axis. Calculate viewVectorEye (v)
        // in eye coordinates as:
        // (0.0, 0.0, 0.0) - vertexPositionEye3
        vec3    viewVectorEye = -normalize(vertexPositionEye3);

        float rdotv = max(dot(reflectionVector, viewVectorEye),
0.0);

        float specularLightWeightning = pow(rdotv, shininess);

        // Sum up all three reflection components and send to
the fragment shader
        vLightWeightting = uAmbientLightColor +
                                uDiffuseLightColor   *
diffuseLightWeightning +
                                uSpecularLightColor  *
specularLightWeightning;

        // Finally transform the geometry
        gl_Position     =      uPMatrix      *      uMVMMatrix     *
vec4(aVertexPosition, 1.0);

```

```
}
```

```
</script>
```

The lighting calculations performed in the code in [Listing 7-1](#) should be easy to understand since you have now learned about the different parts that comprise the Phong reflection model.

What is important to remember is that all lighting calculations have to be done in the same coordinate system, usually the eye coordinate system. This means that the vertex shader needs to first transform both the vertex position and the normal to eye coordinates.

Transforming the vertex is typically done by multiplying the vertex position with the modelview matrix like this:

```
vec4 vertexPositionEye4 = uMVMMatrix * vec4(aVertexPosition,  
1.0);
```

The normal cannot generally be transformed with the modelview matrix. If you transform a vertex belonging to a surface with a matrix \mathbf{M} , you need to transform the normal to the surface with the inverse transpose of \mathbf{M} , which is written as \mathbf{M}^{-T} . Alternatively, you can use the inverse transpose of the upper 3×3 matrix of \mathbf{M} . In this code snippet, the second alternative is used. This might sound confusing now, but you will learn more about how normals need to be transformed later in this chapter.

The uniform `uNMatrix` in the shader contains the inverse transpose of the upper 3×3 matrix of `uMVMMatrix`. The matrix to transform the normals is calculated in JavaScript (which is not shown here, but later in this chapter) and sent in to the shader as a uniform. The normals can then be transformed with this code in the shader:

```
vec3 normalEye = normalize(uNMatrix * aVertexNormal);
```

When the lighting calculations are done in the vertex shader and no textures are used, the fragment shader is really trivial. In [Listing 7-2](#), you can see the fragment shader that works together with the vertex shader in [Listing 7-1](#). All it does is to expand the varying variable `vLightWeighting` to homogeneous coordinates and assign it to the built-in variable `gl_FragColor`.



Available for
download on
[Wrox.com](#)

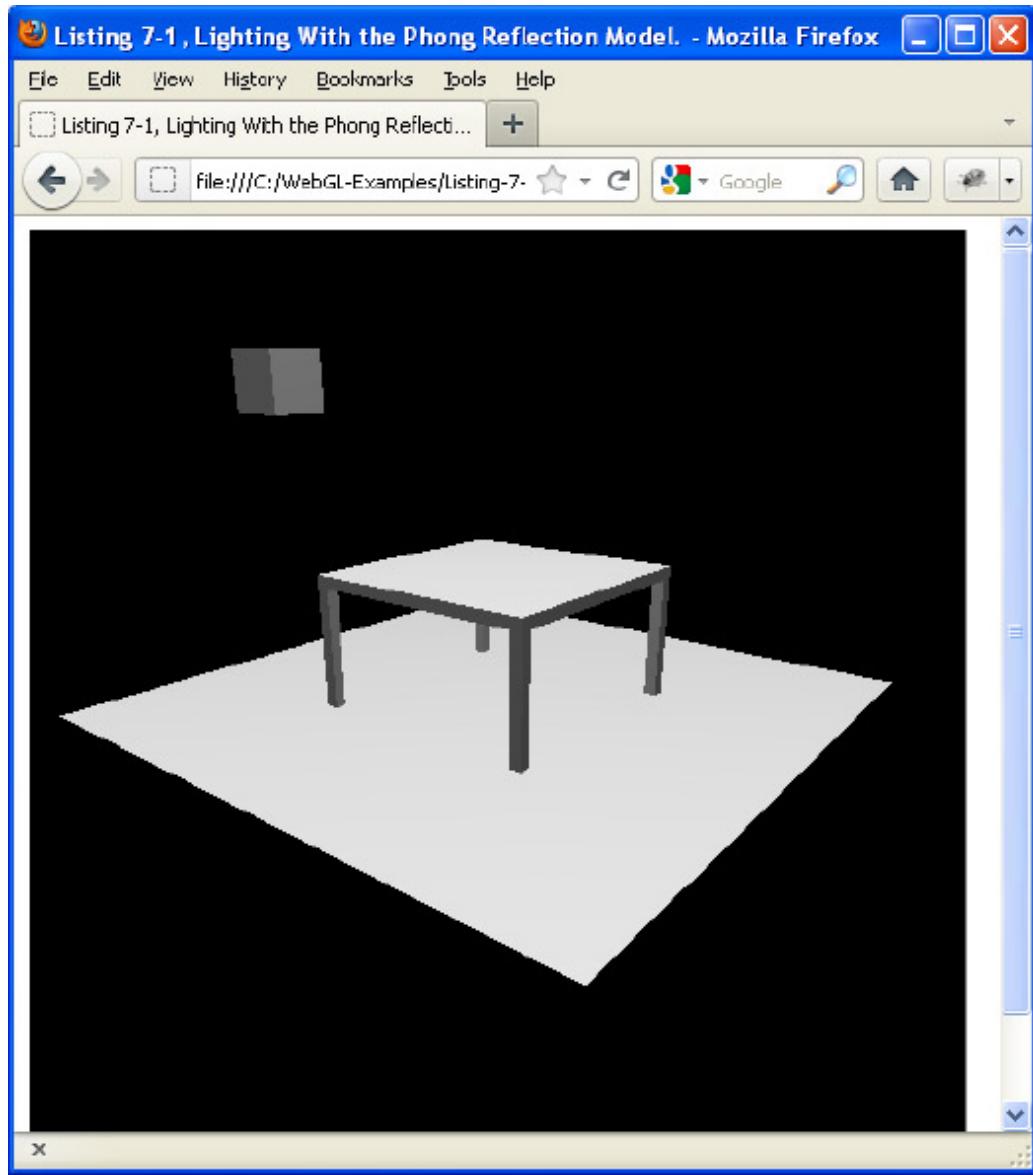
LISTING 7-2: The fragment shader used when the Phong reflection model is implemented in the vertex shader

```
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
varying vec3 vLightWeighting;

void main() {
    gl_FragColor = vec4(vLightWeighting.rgb, 1.0);
}
</script>
```

[Figure 7-6](#) shows what it looks like when the vertex shader from [Listing 7-1](#) and the fragment shader from [Listing 7-2](#) are used in the basic WebGL application that draws a floor, a table, and a box without any textures. The tabletop and the floor are almost completely white since the light is coming from above. The table legs have a dark-gray color since not much light is reaching them. In the next section, you will learn how easy it is to adapt these shaders so you can use lighting for your textured objects as well.

FIGURE 7-6: A WebGL scene with lighting but without textures



Using Lighting with Texturing

Since you learned about texturing in Chapter 5 and have seen that your objects generally look much more realistic when you apply textures to them, you of course want to be able to use lighting with texturing.

When you know how to perform texturing as well as lighting calculations, it is very easy to combine these two techniques. You perform your lighting calculations as usual, and then you use the fragment shader to combine the color from the sampled texels with the color of the lighting calculations.

The most common way to do this is to perform a component-wise multiplication of the values you get in your light calculations with the color value that is sampled from the texture. This component-wise multiplication is sometimes called *modulation* and looks like this:

```
gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb,  
texelColor.a);
```

You will have the opportunity to look at the complete fragment shader soon, but first you should look at the vertex shader. The only thing that the vertex shader does related to texturing is to take the texture coordinates as input in an attribute variable and write this attribute to a varying variable. Therefore, the vertex shader shown in [Listing 7-3](#) is almost identical to the vertex shader shown in [Listing 7-1](#). The few differences (which are all related to the texture coordinates) are highlighted in the code. Since the code includes several comments and the majority of it was also explained in detail in [Listing 7-1](#), the vertex shader in [Listing 7-3](#) will not be further discussed here.



Available for
download on
[Wrox.com](#)

[LISTING 7-3: The vertex shader for the Phong reflection model](#)

```
<script id="shader-vs" type="x-shader/x-vertex">  
    // Vertex shader implemented to perform lighting according  
    to the  
    // Phong reflection model. Forwards texture coordinates to  
    fragment  
    // shader.  
    attribute vec3 aVertexPosition;  
    attribute vec3 aVertexNormal;  
    attribute vec2 aTextureCoordinates;  
  
    uniform mat4 uMVMatrix;  
    uniform mat4 uPMatrix;  
    uniform mat3 uNMatrix;  
  
    uniform vec3 uLightPosition;  
    uniform vec3 uAmbientLightColor;  
    uniform vec3 uDiffuseLightColor;  
    uniform vec3 uSpecularLightColor;  
  
    varying vec2 vTextureCoordinates;  
    varying vec3 vLightWeighting;
```

```

const float shininess = 32.0;

void main() {
    // Get the vertex position in eye coordinates
    vec4 vertexPositionEye4 = uMVMMatrix * vec4(aVertexPosition, 1.0);
    vec3 vertexPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition - vertexPositionEye3);

    // Transform the normal (n) to eye coordinates
    vec3 normalEye = normalize(uNMatrix * aVertexNormal);

    // Calculate n dot l for diffuse lighting
    float diffuseLightWeightning = max(dot(normalEye,
                                             vectorToLightSource),
                                         0.0);

    // Calculate the reflection vector (r) that is needed
    // for specular light
    vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                                normalEye));

    // The camera in eye coordinates is located in the
    // origin and is pointing
    // along the negative z-axis. Calculate viewVectorEye
    (v)
    // in eye coordinates as:
    // (0.0, 0.0, 0.0) - vertexPositionEye3
    vec3 viewVectorEye = -normalize(vertexPositionEye3);

    float rdotv = max(dot(reflectionVector, viewVectorEye),
                      0.0);

    float specularLightWeightning = pow(rdotv, shininess);

    // Sum up all three reflection components and send to
    // the fragment shader
    vLightWeighting = uAmbientLightColor +
                      uDiffuseLightColor * diffuseLightWeightning +
                      uSpecularLightColor *

```

```

specularLightWeightning;

    // Finally transform the geometry
    gl_Position = uPMatrix * uMVMMatrix *
vec4(aVertexPosition, 1.0);
    vTextureCoordinates = aTextureCoordinates;
}
</script>

```

[Listing 7-4](#) shows the fragment shader that combines texturing with lighting. As you can see from the code, the texture is first sampled and then the sampled value is multiplied with the result from the lighting calculations that was sent in as the varying variable `vLightWeightning`. The actual lighting calculations are done in the corresponding vertex shader (shown in [Listing 7-3](#)).



Available for
download on
[Wrox.com](#)

[LISTING 7-4: The fragment shader that combines texturing with lighting](#)

```

<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;

varying vec2 vTextureCoordinates;
varying vec3 vLightWeightning;
uniform sampler2D uSampler;

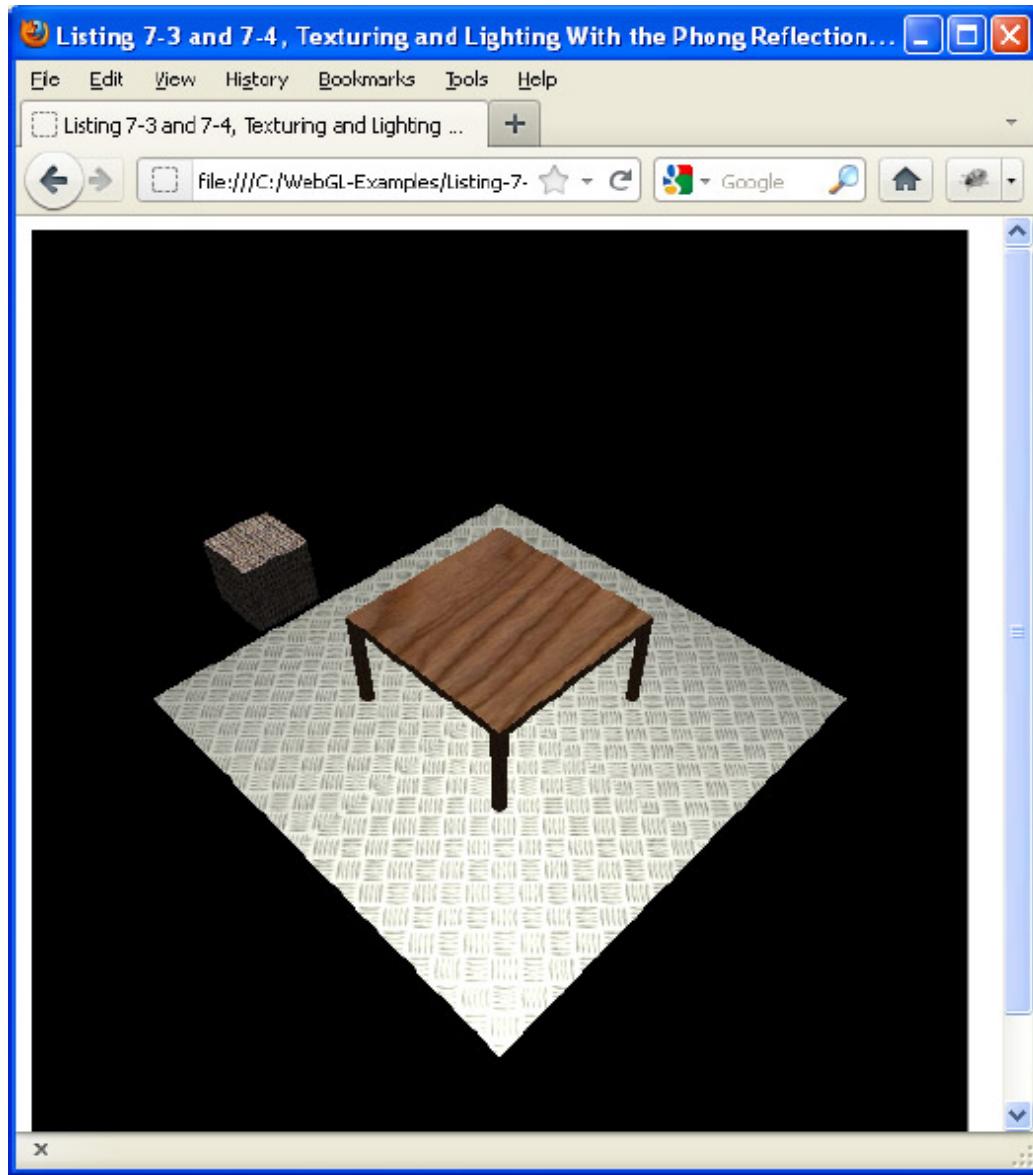
void main() {
    vec4 texelColor = texture2D(uSampler,
vTextureCoordinates);

    gl_FragColor = vec4(vLightWeightning.rgb *
texelColor.rgb, texelColor.a);
}
</script>

```

[Figure 7-7](#) shows the result when the vertex shader in [Listing 7-3](#) is used together with the fragment shader in [Listing 7-4](#) in a basic WebGL application that draws a floor, a table, and a box with textures applied to all three objects.

[FIGURE 7-7:](#) An example of lighting in a scene with textured objects



Modulating with a Late Add to Create Specular Highlighting

A variant of the basic modulation that is used in [Listing 7-4](#) is to keep the calculated specular light separate from the ambient and the diffuse light and only modulate the sampled texel color with the combination of the ambient and diffuse light. The specular light is then added after the modulation like this:

```
void main() {  
    vec4      texelColor      =      texture2D(uSampler,  
    vTextureCoordinates);
```

```

    gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb
+
                                         vSpecularLightWeighting.rgb,
texelColor.a);
}

```

In this code snippet, it is assumed that the variable `vLightWeighting` contains the calculated ambient and diffuse light, and the variable `vSpecularLightWeighting` contains the calculated specular light. As you can see, the specular light is added after the modulation is done. This is what the name “modulation with a late add” refers to.

The reason for using this approach is that a valid color value in OpenGL ES Shading Language and WebGL is normally between 0.0 and 1.0. This means that in theory, the maximum sum of the ambient, diffuse, and specular light is white, which is represented with (1.0, 1.0, 1.0, 1.0). If you multiply a sampled texel color with this value, you get the original sampled texel color. This means that there is no valid color representation that you can multiply the sampled texel color with to get a brighter color at a specular highlight. This is why this second approach is sometimes used. By adding the specular term after the modulation, you can add a brighter specular highlight.

In reality, however, the sum of the ambient, diffuse, and specular light can add up to a value that is larger than 1.0 for each color channel. This means that you don’t really have to keep the specular term separate and add it after the modulation. But this can be useful to know if you see shader code that is written in this way.

UNDERSTANDING THE JAVASCRIPT CODE NEEDED FOR WEBGL LIGHTING

When dealing with lighting in WebGL, most of the work is done in the shaders on the GPU. As you learned in the previous sections of this chapter, the Phong reflection model is implemented in the shaders.

The work that is handled in JavaScript on the CPU is usually limited to feeding the GPU some input that it needs for the lighting calculations. For

example, you usually need some JavaScript code to send in the following data to the shader:

- Light positions or light directions
- Light colors (or possibly separate material and color components)
- Vertex normals
- Normal matrix

The light positions or light directions and the light colors could potentially be hardcoded in the shader if you intend to set them only once. However, the vertex normals and the normal matrix are usually sent in from the JavaScript that executes on the CPU.

One reason to send in the vertex normals from JavaScript is that to calculate the vertex normals for a general mesh, you typically need to use neighboring vertices as well to determine how the surface is oriented. But the vertex shader does not generally have any information about neighboring vertices, so the normal vectors are usually sent in from JavaScript. In theory, the normal matrix could be calculated in the shader, but it is often calculated by JavaScript on the CPU and sent in as a uniform as well.

Follow these steps to make your lighting work in JavaScript:

1. Set up buffers that contain the vertex normals.
2. Calculate the normal matrix from the modelview matrix and send it as a uniform to the vertex shader.
3. Send light information (light position, light direction, light color) to the vertex shader.

You'll learn more about these steps in the following sections.

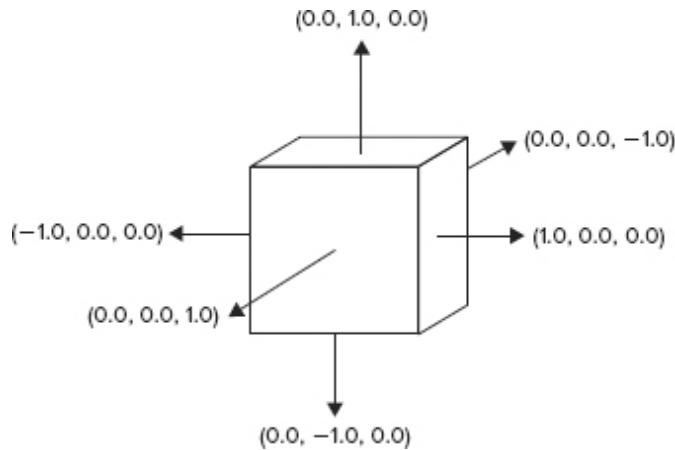
Setting Up Buffers with Vertex Normals

In a simple case when your 3D object is a cube, it is very easy to write down the normals for the different faces of the cube without any calculations:

- Front face has the normal $(0.0, 0.0, 1.0)$.
- Back face has the normal $(0.0, 0.0, -1.0)$.
- Left face has the normal $(-1.0, 0.0, 0.0)$.
- Right face has the normal $(1.0, 0.0, 0.0)$.
- Top face has the normal $(0.0, 1.0, 0.0)$.
- Bottom face has the normal $(0.0, -1.0, 0.0)$.

[Figure 7-8](#) shows a cube with surface normals for each of its six faces according to what was just described. Note that when the normals are specified for a cube, or any other object in WebGL for that matter, you generally have one normal for each vertex, and not what is shown in [Figure 7-8](#) where there is one normal for each face. The example in [Figure 7-8](#) is only to show you the direction for the different normals of a cube.

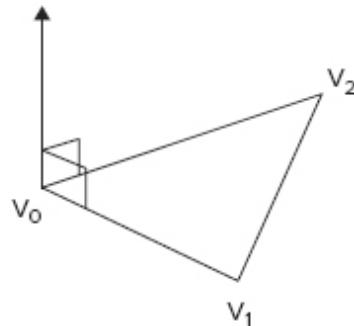
FIGURE 7-8: A cube with its surface normals



If your object is not a simple cube where the faces point in the same direction as the axis of the coordinate system, it can be more difficult to write down the normals without calculation. But since you learned quite a bit about linear algebra for 3D graphics in Chapter 1, you know that you can use the cross product to calculate the theoretical normal to a triangle in the more general case. [Figure 7-9](#) shows how the normal is calculated for a triangle with the three vertices V_0 , V_1 , and V_2 .

FIGURE 7-9: Using the cross product to calculate the normal for a triangle

$$\mathbf{n} = (\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)$$





For more information about the cross product, see Chapter 1.

If you want to calculate normals in JavaScript, one way is to use the function that calculates the cross product in the JavaScript library glMatrix.

```
vec3.cross(vector1, vector2, normal);
```

Here you should note that you would also need to normalize the resulting normal before you can correctly use it in the lighting calculations you have seen earlier in this chapter.

If you have objects that you have created in a 3D modeling program such as Blender or Maya, the program typically creates the normals for you automatically. When you read such a model into your WebGL application, you need to parse the vertex data (such as vertex positions and vertex normals) and then load it into a WebGL buffer.

The code to create and bind the buffer for the normals and then load the buffer with the data is very similar to the code that is loading other vertex data (such as vertex positions or texture coordinates) into a buffer. You can see how the vertex normals for a cube are loaded in this snippet of JavaScript code.

```
pwgl.cubeVertexNormalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexNormalBuffer);

var cubeVertexNormals = [
    // Front face
    0.0, 0.0, 1.0, //v0
    0.0, 0.0, 1.0, //v1
    0.0, 0.0, 1.0, //v2
    0.0, 0.0, 1.0, //v3

    // Back face
    0.0, 0.0, -1.0, //v4
    0.0, 0.0, -1.0, //v5
    0.0, 0.0, -1.0, //v6
    0.0, 0.0, -1.0, //v7

    // Left face
    -1.0, 0.0, 0.0, //v8
    -1.0, 0.0, 0.0, //v9
    -1.0, 0.0, 0.0, //v10
    -1.0, 0.0, 0.0, //v11
```

```

// Right face
1.0, 0.0, 0.0, //12
1.0, 0.0, 0.0, //13
1.0, 0.0, 0.0, //14
1.0, 0.0, 0.0, //15

// Top face
0.0, 1.0, 0.0, //v16
0.0, 1.0, 0.0, //v17
0.0, 1.0, 0.0, //v18
0.0, 1.0, 0.0, //v19

// Bottom face
0.0, -1.0, 0.0, //v20
0.0, -1.0, 0.0, //v21
0.0, -1.0, 0.0, //v22
0.0, -1.0, 0.0, //v23
];

gl.bufferData(gl.ARRAY_BUFFER,
new
Float32Array(cubeVertexNormals),
gl.STATIC_DRAW);

pwgl.CUBE_VERTEX_NORMAL_BUF_ITEM_SIZE = 3;
pwgl.CUBE_VERTEX_NORMAL_BUF_NUM_ITEMS = 24;

```

Calculating and Uploading the Normal Matrix to the Shader

The transformation of normals can be confusing. One reason for this is that transforming normals differs from how vertices or other vectors are transformed. But the basic rule of how to transform normals is easy to learn.

What probably adds to the confusion is that when you have learned the basic rule, you might read code examples or get advice from other people who do not follow the basic rule. However, it is not that difficult, as you will see here. There are three main options for how the normals can be transformed:

1. In general you cannot use the modelview matrix that you use to transform your vertices for an object to transform the surface normals for the object. Instead, if you transform a vertex belonging to a surface with a matrix \mathbf{M} , then you need to transform the normal to the surface with the

inverse transpose of \mathbf{M} , which is written as $\mathbf{M}^{-\top}$. This is the basic rule that works most of the time.

2. A first optimization of the basic rule can be used since the normal is a vector. If you are transforming the vertices with \mathbf{M} , then you can use the inverse transpose of the upper 3×3 matrix of \mathbf{M} to transform the normals.

3. In some cases, the transformation can be further optimized. If you know that your matrix \mathbf{M} is only a composition of the following transforms (which is not uncommon), then you can use the original matrix directly:

- Translations
- Rotations
- Uniform scaling (this means no stretching or squashing)

It might be confusing if you have learned to use the inverse transpose of a matrix when you transform normals and then you see code that does something else. However, when you understand these rules, it should not be too difficult to read or write code that transforms normals.

The second option is the one used in this book: to use the inverse transpose of the upper 3×3 matrix of \mathbf{M} to transform the normals. This option is also well supported in the JavaScript library `glMatrix` that is used in most examples in the book. The `glMatrix` library has a method named `mat4.toInverseMat3()` that transforms the upper 3×3 matrix of an original 4×4 matrix. After this, the 3×3 matrix can be transposed by calling `mat3.transpose()`. The code snippet that calculates a normal matrix from a modelview matrix and then uploads the normal matrix to a uniform in the shader is shown here:

```
var normalMatrix = mat3.create();
mat4.toInverseMat3(pwgl.modelViewMatrix, normalMatrix);
mat3.transpose(normalMatrix);
gl.uniformMatrix3fv(pwgl.uniformNormalMatrixLoc, false,
normalMatrix);
```

Uploading the Light Information to the Shader

If you don't hardcode the information about the lights in the shader, you need to upload this data to the shader from the JavaScript code. Assuming that you use uniforms to upload the data about the lights, it is not different from

uploading any other data to uniforms. First, you need to find the location of the relevant uniforms in the active program object. You do this with the method `gl.getUniformLocation()`, as shown here:

```
pwgl.uniformLightPositionLoc =
    gl.getUniformLocation(shaderProgram, "uLightPosition");

pwgl.uniformAmbientLightColorLoc =
    gl.getUniformLocation(shaderProgram,
"uAmbientLightColor");

pwgl.uniformDiffuseLightColorLoc =
    gl.getUniformLocation(shaderProgram,
"uDifffuseLightColor");

pwgl.uniformSpecularLightColorLoc =
    gl.getUniformLocation(shaderProgram,
"uSpecularLightColor");
```

Then the values for the uniforms must be uploaded to the GPU. In this case, all the uniforms consist of vectors of three components, so the method `gl.uniform3fv()` is used. The code is shown here:

```
function setupLights() {
    gl.uniform3fv(pwgl.uniformLightPositionLoc, [0.0, 15.0,
5.0]);
    gl.uniform3fv(pwgl.uniformAmbientLightColorLoc, [0.2, 0.2,
0.2]);
    gl.uniform3fv(pwgl.uniformDiffuseLightColorLoc, [0.7, 0.7,
0.7]);
    gl.uniform3fv(pwgl.uniformSpecularLightColorLoc, [1.0,
1.0, 1.0]);
}
```

USING DIFFERENT INTERPOLATION TECHNIQUES FOR SHADING

When the vertex shader and the fragment shader were initially introduced to you in Chapter 1, it was briefly explained what shading actually means in 3D graphics. Now you will learn the concept of shading in more detail.

Shading is the process of determining the color at a point, a primitive (such as a triangle), or an object based on, for example:

- Position of light sources
- Distance to light sources
- Color of the light
- Normal vectors (orientation of the shaded position)
- Material properties

The most common case is, of course, that you have something more than a single point in your 3D scene. Even in the first example in this book, you at least drew a triangle, and in a real-world WebGL application you will normally have more advanced models consisting of many triangles. When shading is referred to as the process of determining the color of a triangle or an object consisting of several triangles, there are normally two different parts of the shading that can be distinguished:

- The illumination model
- The interpolation technique

The illumination model defines how you use, for example, the light color, the material, and the different angles to calculate the color at a specific point. Put in another way, the illumination model defines the equation that you use to calculate the color at a specific point. For example, you have learned how to calculate the color at a specific point with the Phong reflection model.

The interpolation technique has to do with the positions on your object on which you should use the illumination model to calculate the color. In the previous examples in this chapter, the color was calculated for each vertex. When the color for a vertex is known, the color for the fragments are determined by linear interpolation of the colors at the vertices. But to calculate the color at the vertices and then use linear interpolation to determine the color for the fragments is not the only solution.

Traditionally there have been three ways to perform the interpolation for the shading:

- Flat shading
- Gouraud shading
- Phong shading

Sometimes these methods are just referred to as different ways to perform the shading. The general rule is that flat shading is simplest and gives the

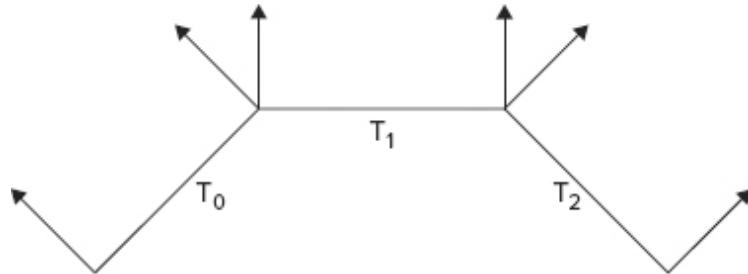
poorest result, Gouraud shading involves more calculation but also gives a better result, and Phong shading is the most advanced of the three and gives the best result. Each method is described in the following sections.

Flat Shading

Flat shading involves calculating the shading based on one single normal for each triangle. The shading calculation is performed once based on this normal, and the complete triangle is given the resulting color. Now it should be mentioned that flat shading is usually not that interesting for WebGL (or OpenGL ES 2.0 either, for that matter), since you normally define one normal for each vertex in WebGL (and OpenGL ES 2.0). However, if you use the primitive `gl.TRIANGLES` in WebGL and you provide the same normal for all three vertices of each triangle, the result looks like flat shading.

This concept is shown in [Figure 7-10](#), which represents a surface where you see the edge of three triangles (T_0 , T_1 , and T_2) that build up the surface. (Since only the edge of each triangle can be seen, the triangle looks like a line.) You can also see two normals for each triangle. Since the normals for triangle T_0 point in one direction, the normals for T_1 in another, and the normals for T_2 in a third direction, the calculated shading for all the triangles has a different color. However, if the light source and the viewer are far away from the surface, the color is constant over each triangle. This creates the effect of flat shading.

FIGURE 7-10: A surface consisting of three triangles. The normals are oriented so the result looks like flat shading.



To understand the limitations of flat shading, you must consider the assumptions that are made for flat shading. These are the cases when flat shading can look acceptable:

- When the light source is infinitely far away, so that the angle of the incident rays is the same over the complete triangle. (A light source infinitely far away is usually called directional light. You will learn more about that later in this chapter.)
- When the viewer is infinitely far away, so that the angle towards the viewer is the same on all positions of the triangle.
- When the triangle represents a flat surface (for example, one of the faces of a cube) and is not an approximation of a curved surface.

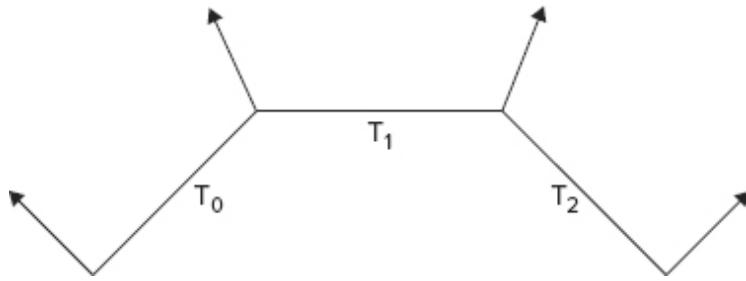
Usually one or several of these conditions are not true, and so flat shading gives a less successful result. This is especially true if the triangle is part of an approximation for a curved surface, such as a sphere, where the result of the flat shading normally looks faceted rather than having the desired smooth surface. The larger the triangles are, the worse the result of the flat shading for a curved surface. Luckily, there is never a reason to use flat shading to approximate a curved surface in WebGL.

Gouraud Shading

Gouraud shading is named after the French computer scientist Henri Gouraud (pronounced *guh-row*), who invented the method and published his findings in 1971. It is sometimes called *per-vertex shading* (or per-vertex lighting), since the shading is calculated for each vertex and then the resulting color at each vertex is linearly interpolated over the fragments. The code for Gouraud shading is used in [Listing 7-1](#) and [Listing 7-3](#), where the Phong reflection model was discussed.

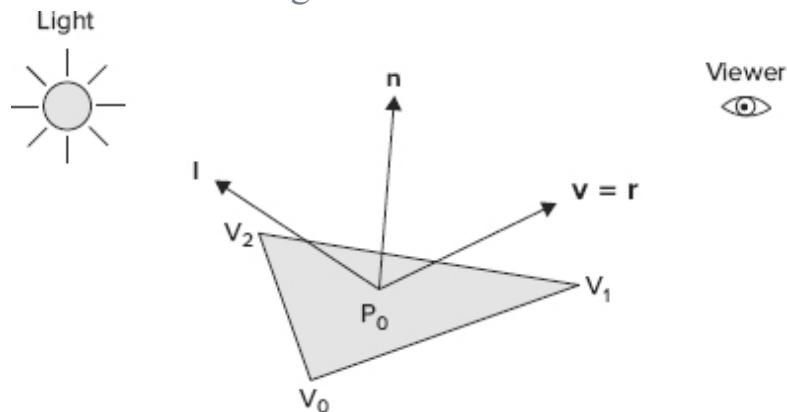
If a triangle is an approximation of a curved surface such as a sphere, the normal at each vertex should not be the mathematical correct normal for one of the triangles. Instead, you should use a normal that takes into account the orientation of the underlying curved surface that you try to model (see [Figure 7-11](#)). The average normal gives a smoother transition of the color from one triangle to adjacent triangles, so the seams between triangles are less obvious than they are for flat shading.

FIGURE 7-11: A surface consisting of three triangles. The normals are an average of the mathematical correct normals for the adjacent triangles



Because of this, Gouraud shading produces much better results than flat shading for curved surfaces. In general, Gouraud shading produces a decent result for most matte surfaces. For shiny surfaces, however, the Gouraud shading can produce artifacts. Since the shading equation is only calculated at the vertices, a specular highlight that falls between vertices can be missed when Gouraud shading is used. This concept is shown in [Figure 7-12](#).

FIGURE 7-12: A specular highlight that falls in the middle of a triangle can be missed when Gouraud shading is used



[Figure 7-12](#) shows a triangle with three vertices (V_0 , V_1 , and V_2). In the point P_0 , the reflection vector r and the view vector v coincide so that a specular highlight should occur at P_0 . However, since the shading equation is only calculated at the three vertices (V_0 , V_1 , and V_2), the specular highlight can be missed.

Phong Shading

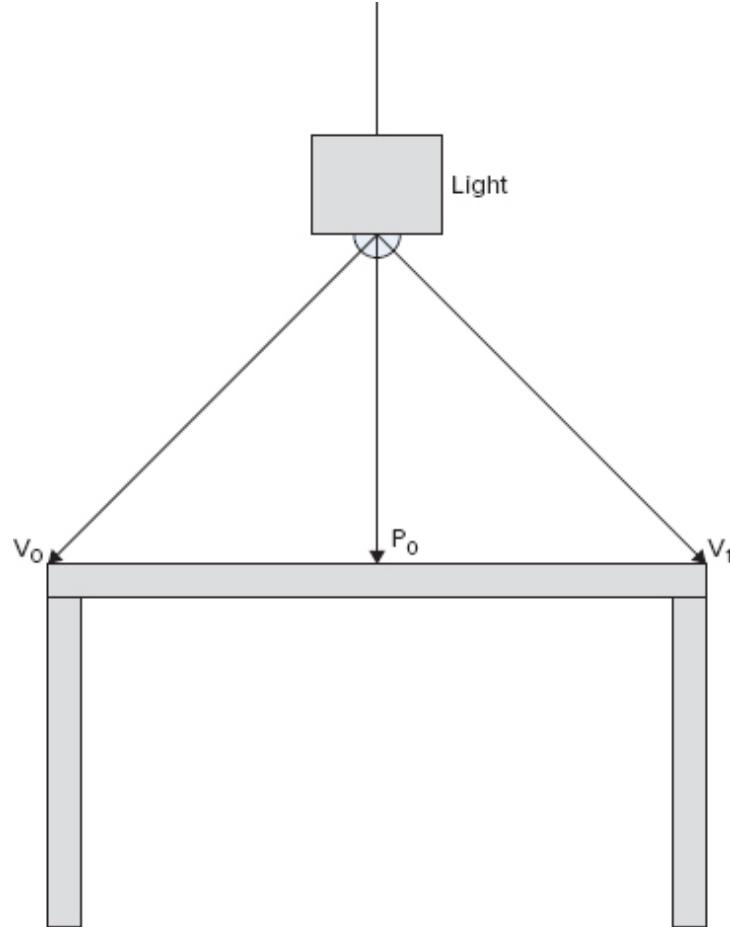
Phong shading, which is named after Bui Tuong Phong (the same person who developed the Phong reflection model), calculates the shading equation for every fragment of a triangle. Sometimes the name *per-fragment shading* (or

per-fragment lighting) is used for this method, which is actually a better description of how the method works.

Since the color is calculated for every fragment, the result is better than the previously described methods, especially for shiny surfaces. With Phong shading, you do not have the same artifacts that Gouraud shading can create for shiny objects. You also do not have the situation where a specular highlight that falls within a triangle can be missed.

Even without considering the specular highlight, the Phong shading usually produces a better result than Gouraud shading. [Figure 7-13](#) represents a sketch of the table you worked with in some previous examples in the book. This time, however, assume that you have a lamp hanging over the table quite close to the tabletop. Three rays of light are shown. At both the vertices V_0 and V_1 , the angle between the light and the surface normal is around 45° . In the middle of the table at the point P_0 , the angle between the light and the normal is 0° .

FIGURE 7-13: An example of when Phong shading gives a better result than Gouraud shading (even for diffuse light)



You learned earlier in the chapter that the diffuse reflection component is given by:

$$I = k_d I_d \max(\cos \theta, 0)$$

The important part in this discussion is that the diffuse reflection is proportional to $\cos \theta$, where θ is the angle between the normal and the direction to the light. This means that the reflection is largest when the angle between the direction to the light and the surface normal is 0° . When the angle is 45° , the intensity of the reflection has declined to $\cos 45^\circ = 0.71$. So the middle of the table should be considerably brighter than the edges at the vertices V_0 and V_1 .

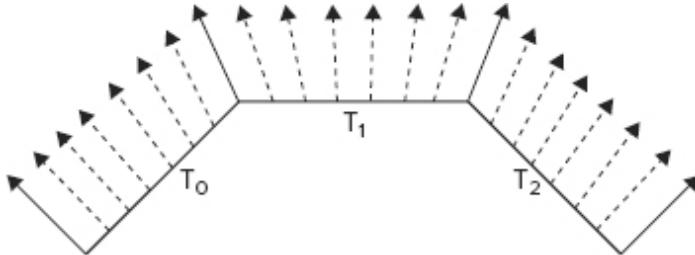
But if Gouraud shading is used, the middle of the table at the point P_0 is not brighter. The shading equation is only calculated at the vertices V_0 and V_1 , and so the table is dim at these vertices. Then the result is linearly interpolated over the fragments, which means that the middle of the table at point P_0 is also dim.

If Phong shading is used, the normals that are sent into the vertex shader are passed on as varying variables to the fragment shader. As you know, all varying variables are linearly interpolated and the linearly interpolated normals are used to calculate the lighting in the fragment shader at every fragment. In the table example, this means that the lighting calculation is also performed in the middle of the table top at the point P_0 , and this part is now brighter than the table edges at V_0 and V_1 .

In this example, it is assumed that the lamp is a point light, which is a light source that emits light in all directions. When you have finished reading this chapter, you will understand that the difference between Phong shading and Gouraud shading in this example is even bigger if the lamp is modeled as a spot light and also if attenuation for the light is taken into account. But don't worry about that now; you will understand this when you have finished the chapter.

For completeness, [Figure 7-14](#) shows the same representation of a surface with the normals that was shown for the flat shading and the Gouraud shading. The four solid normals represent the normals at the vertices that are sent in to the vertex shader as attributes. The dotted normals are the linearly interpolated normals in the fragment shader.

FIGURE 7-14: A surface consisting of three triangles. The dotted normals represent the linear interpolation of the normals at the vertices



[Listing 7-5](#) shows an example of the Phong reflection model using Phong shading as an interpolation method. You can see that when Phong shading is used as an interpolation method, not much work needs to be done in the vertex shader, since the actual lighting calculations have now been moved to the fragment shader.

The work that remains in the vertex shader is to transform the vertex position and the normal to eye coordinates and pass these on as varying variables to the fragment shader. As usual, the vertex shader transforms the geometry to clip coordinates and also passes on the texture coordinates to the fragment shader.



Available for
download on
Wrox.com

[LISTING 7-5: An example vertex shader for the Phong reflection model using Phong shading as an interpolation method](#)

```
<script id="shader-vs" type="x-shader/x-vertex">
    // Vertex shader implemented to perform lighting according
    to the
    // Phong reflection model. The interpolation method that
    is used is
    // Phong shading (per-fragment shading) and therefore the
    actual
    // lighting calculations are performed in the fragment
    shader.
    attribute vec3 aVertexPosition;
    attribute vec3 aVertexNormal;
    attribute vec2 aTextureCoordinates;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    uniform mat3 uNMatrix;
```

```

varying vec2 vTextureCoordinates;
varying vec3 vNormalEye;
varying vec3 vPositionEye3;

void main() {
    // Get vertex position in eye coordinates and send to
    // the fragment shader
    vec4 vertexPositionEye4 = uMVMMatrix *
    vec4(aVertexPosition, 1.0);
    vPositionEye3 = vertexPositionEye4.xyz /
    vertexPositionEye4.w;

    // Transform the normal to eye coordinates and send to
    // fragment shader
    vNormalEye = normalize(uNMatrix * aVertexNormal);

    // Transform the geometry
    gl_Position = uPMatrix * uMVMMatrix *
    vec4(aVertexPosition, 1.0);
    vTextureCoordinates = aTextureCoordinates;
}
</script>

```

For Phong shading (or per-fragment shading), most of the actual lighting calculations happen in the fragment shader (see [Listing 7-6](#)). The actual source code should be easy to understand since you are basically performing the same calculations for the Phong reflection model that you have done in the vertex shader before.

Note that since the lighting calculations are done in the fragment shader, this is also where the uniform variables that are needed for the lighting calculations are defined. When the lighting calculations are finished, the result from these calculations is multiplied with the sampled texel color to combine the texture with the lighting.



Available for
download on
Wrox.com

[LISTING 7-6: The fragment shader when the Phong reflection model is implemented with Phong shading \(per-fragment shading\)](#)

```

<script id="shader-fs" type="x-shader/x-fragment">
    // Fragment shader implemented to perform lighting
    // according to the
    // Phong reflection model. The interpolation method that
    // is used is

```

```

// Phong shading (per-fragment shading) and therefore the
actual
// lighting calculations are implemented here in the
fragment shader.
precision mediump float;

varying vec2 vTextureCoordinates;
varying vec3 vNormalEye;
varying vec3 vPositionEye3;

uniform vec3 uLightPosition;
uniform vec3 uAmbientLightColor;
uniform vec3 uDiffuseLightColor;
uniform vec3 uSpecularLightColor;

uniform sampler2D uSampler;

const float shininess = 32.0;

void main() {

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition -
vPositionEye3);

    // Calculate n dot l for diffuse lighting
    float diffuseLightWeighting = max(dot(vNormalEye,
vectorToLightSource), 0.0);

    // Calculate the reflection vector (r) that is needed
    // for specular light
    vec3 reflectionVector = normalize(reflect(-
vectorToLightSource,
vNormalEye));

    // Camera in eye space is in origin pointing along the
negative z-axis.
    // Calculate viewVector (v) in eye coordinates as
    // (0.0, 0.0, 0.0) - vPositionEye3
    vec3 viewVectorEye = -normalize(vPositionEye3);

    float rdotv = max(dot(reflectionVector, viewVectorEye),
0.0);

    float specularLightWeighting = pow(rdotv, shininess);
}

```

```

// Sum up all three reflection components
vec3 lightWeighting = uAmbientLightColor +
                      uDiffuseLightColor * 
diffuseLightWeighting +
                      uSpecularLightColor * 
specularLightWeighting;
// Sample the texture
    vec4 texelColor = texture2D(uSampler,
vTextureCoordinates);
// modulate texel color with lightweigthing and write as
final color
    gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb,
texelColor.a);
}
</script>

```

UNDERSTANDING THE VECTORS THAT MUST BE NORMALIZED

You have learned that the dot product can be used to calculate the cosine of the angle between two vectors if both vectors have unit length. This technique is often used in shader code. The requirement that the vectors must be of unit length is the reason why you often need to call the built-in OpenGL ES Shading Language function `normalize()` in the vertex shader and the fragment shader. In the following sections, you will learn more about when you need to normalize a vector and when you can skip it.

Normalization in the Vertex Shader

A vector that is sent in to the vertex shader as an attribute is often transformed in the vertex shader, and after this transformation the vector is normalized. Take a normal vector as an example:

```

// Transform the normal to eye coordinates and send to
fragment shader
vNormalEye = normalize(uNMatrix * aVertexNormal);

```

In general, the call to `normalize()` here is needed for two reasons:

- The normal that is sent in as an attribute from the JavaScript might not be of unit length.
- Even if the normal that is sent in as an attribute is of unit length, the transformation can change the length of the normal.

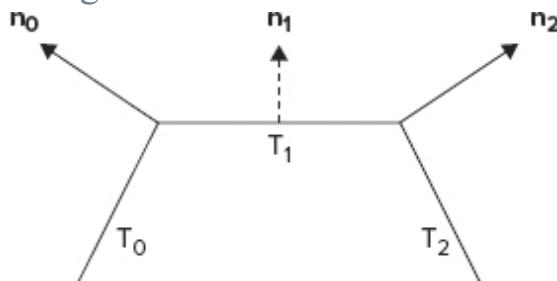
The normalization used in the previous example can be skipped if the vector is of unit length when it is sent in from JavaScript and if the transformation matrix does not change the length of the vector. A transformation matrix that consists of only rotations and translations does not change the length of the vector, and if the vector is of unit length when it is sent in to the vertex shader, it is of unit length after the transformation as well.

If you use the method `mat4.lookAt()` to set up your modelview matrix and don't make any changes to it other than rotations and translations, then your modelview matrix only consists of a combination of rotations and translations. If you also make sure that the vector you transform is of unit length when it is sent in from JavaScript, then you can actually skip the normalization in the vertex shader in this case.

Normalization in the Fragment Shader

In general, vectors that are sent as varying variables from the vertex shader to the fragment shader actually need to be normalized again in the fragment shader even if they were normalized in the vertex shader. This is because in general, the linear interpolation can change the length of the vector (see [Figure 7-15](#)).

FIGURE 7-15: Linear interpolation of the normal vectors n_0 and n_2 gives n_1 that is shorter than unit length



[Figure 7-15](#) shows how two normal vectors n_0 and n_2 of unit length are used for linear interpolation of the normal vector n_1 . In this example, n_1 is

shorter than unit length and needs to be normalized in the fragment shader to have unit length.

You should also note that if the normals at the vertices do not have unit length, it is not only the length of the normals in the fragment shader that is wrong. In general, even the direction can be wrong. This means that to have normalized normals with correct direction, you generally need to normalize the normals in the vertex shader even if they are only used for calculations in the fragment shader. In the fragment shader, you then need to normalize them again to get the correct length.

There is an exception for which you don't need to normalize the vectors in the fragment shader. This is when all vectors are of unit length when they are sent as varying variables from the vertex shader and the vectors at all the vertices have the same direction. When these vectors are interpolated, the result is vectors that are also of unit length and have the same direction as the vectors at the vertices.

An example of this is when you have directional light on a surface. Directional light comes from a light source infinitely far away and therefore has the same direction for all vertices. (You will learn more about directional light in the next section.)

USING DIFFERENT TYPES OF LIGHTS

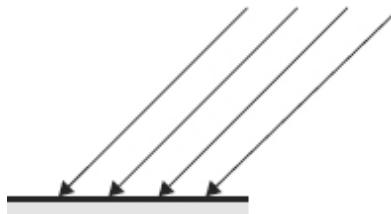
You have now learned a lot about the different reflections (ambient, diffuse, and specular) and the light components of the Phong reflection model. You have also learned about the different interpolation techniques that can be used for the shading.

What you have not yet learned, however, are the different types of light sources and how they emit light. In the previous examples, you have seen lights that have been positioned in a specific location and that have emitted light in all directions. This type of light is called *point light*. In the following sections, you will look at the differences between point lights and the two other common types of light sources: *directional lights* and *spot lights*.

Directional Lights

Directional light is a model for a light source that is infinitely far away from the surface it shines on. Since it is infinitely far away, the rays of light that hit the surface are parallel to each other and have the same direction for all vertices or fragments on the surface (see [Figure 7-16](#)). This means that the light source is defined by a direction (a vector) instead of a location.

[FIGURE 7-16](#): Directional light has parallel rays of light



Since the light direction is the same for all vertices or fragments of a surface, you do not need to calculate the vector to the light source for each vertex or fragment as you have done for the point lights. This makes the directional light even simpler to model than point lights.

To calculate a diffuse reflection for a point light, you need to first calculate the vector towards the light source, and then you can use the dot product to calculate the diffuse reflection, as follows:

```
// Calculate the vector (l) to the light source. This is NOT
needed for
// for a directional light since you typically already have
// vectorToLightSource.
vec3  vectorToLightSource  =  normalize(uLightPosition  -
vPositionEye3);

// Calculate n dot l for diffuse lighting
float diffuseLightWeighting = max(dot(vNormalEye,
                                         vectorToLightSource),
                                     0.0);
```

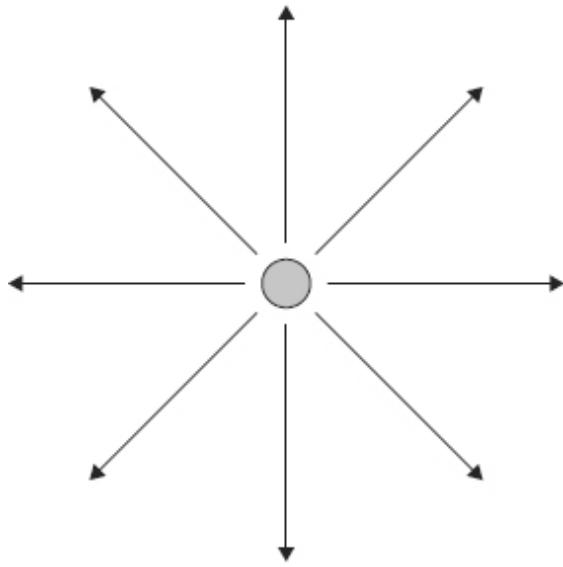
For a directional light, the first calculation is not needed and you can directly calculate the dot product between the normal and the vector pointing towards the light.

An example of light that can typically be modeled with directional light is sunlight shining on the earth.

Point Lights

A point light is located at a specific position and emits light in every direction (see [Figure 7-17](#)). An example of a real-world light that could be modeled with a point light is a light bulb without a lampshade.

[FIGURE 7-17](#): An example of a point light

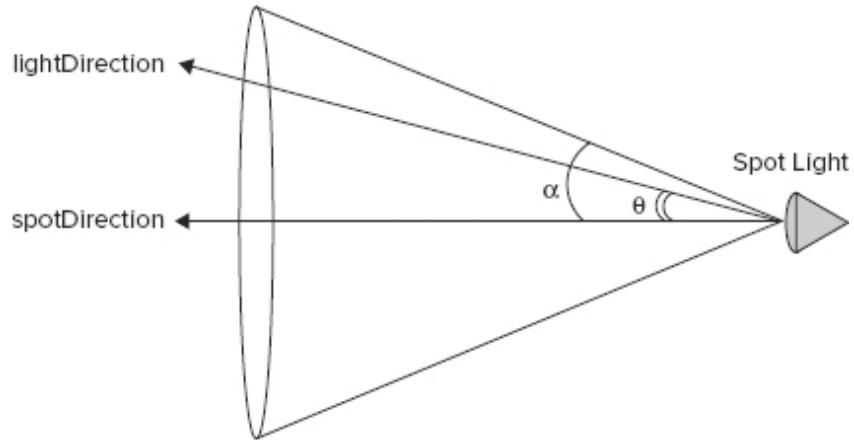


You have already seen several code examples that use point lights, so no special code snippet is shown for point lights in this section. One thing to keep in mind, though, is that sometimes point lights are modeled with attenuation. This has not been addressed so far. You will learn about attenuation for light later in the chapter.

Spot Lights

A spot light emits light in a cone with a certain direction. An example of a real-world light source that can be modeled as a spot light is a car headlight. Spot lights can be modeled in a few different ways; the geometry that you will use to understand spot lights in this book is shown in [Figure 7-18](#).

[FIGURE 7-18](#): The geometry for a spot light



In [Figure 7-18](#), the `spotDirection` is the direction in which the spotlight is directed. It follows the axis of the light cone. The `lightDirection` is the direction from the spot light to the point (fragment or possibly vertex) for which you perform your lighting calculations.

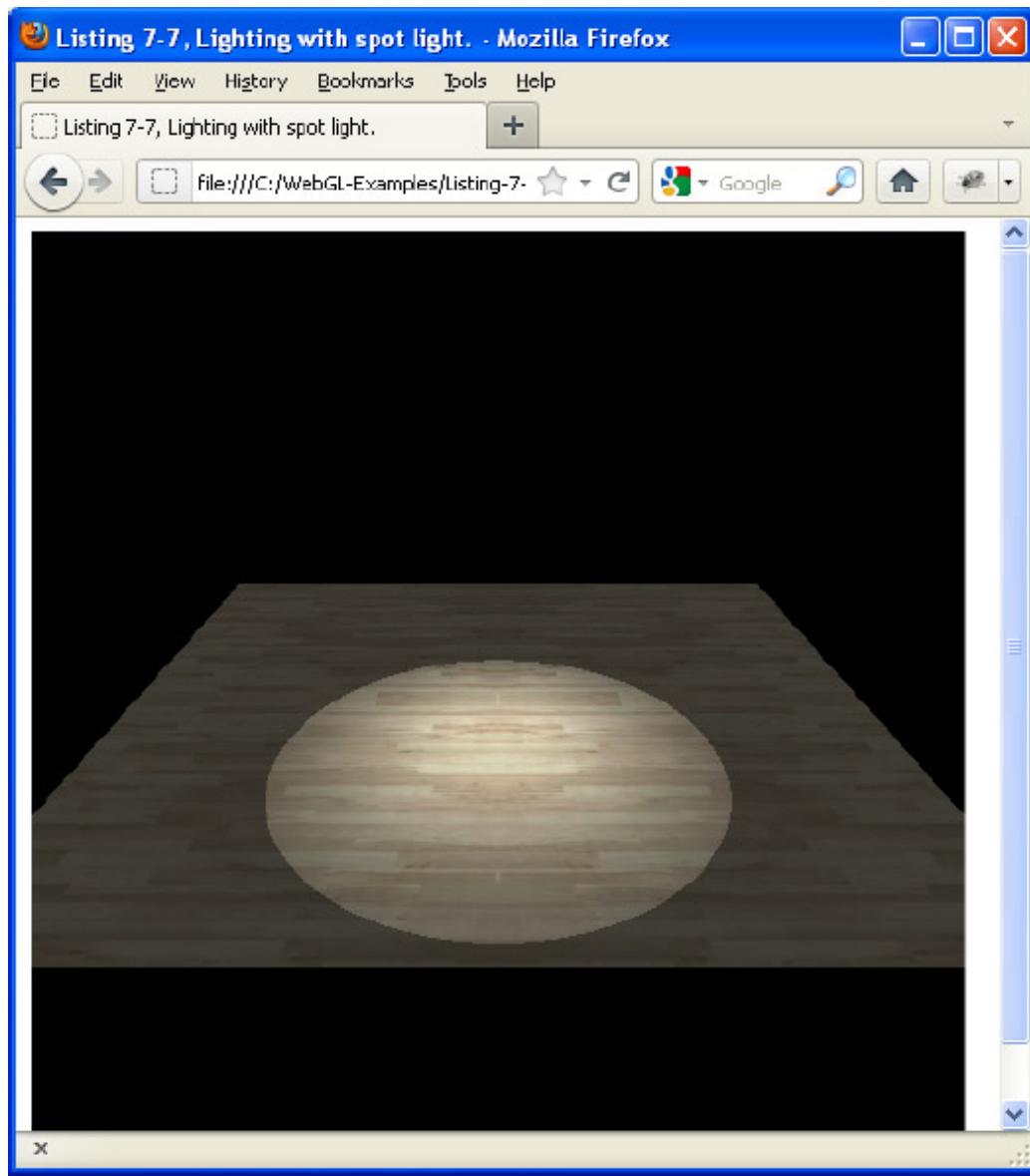
[Figure 7-18](#) also shows the two angles (θ and α). The angle between the `spotDirection` and the `lightDirection` is represented by θ . The spot cutoff angle is represented by α , which means that when θ is larger than α , the point you perform your lighting calculations for is outside the light cone and is not lit by the spot light.

On the other hand, if the point is inside the light cone, the light intensity is highest in the direction given by the `spotDirection` and then falls off gradually when θ increases. A common technique, which you will also use here, is to let the falloff follow $(\cos \theta)^{\text{spotExponent}}$. A large value for the `spotExponent` makes the light intensity fall off faster when the angle θ between the `spotDirection` and the `lightDirection` increases. You might recognize this from the specular reflection, which is modeled in a very similar way. Given that the falloff in intensity within the light cone is called `spotEffect`, you have the following formula:

$$\text{spotEffect} = (\cos \theta)^{\text{spotExponent}} = (\text{spotDirection} \cdot \text{lightDirection})^{\text{spotExponent}}$$

If you implement a spot light and use Phong shading (per-fragment shading), there is no difference in the vertex shader compared with a point light. The changes will be in the fragment shader, and these are actually minor. You can see the shader code for a fragment shader that implements a spot light in [Listing 7-7](#). [Figure 7-19](#) shows the result when this shader is used to light up a textured floor.

FIGURE 7-19: A spot light on a textured floor



Available for
download on
[Wrox.com](#)

LISTING 7-7: A FRAGMENT SHADER THAT IMPLEMENTS A SPOT LIGHT

```
<script id="shader-fs" type="x-shader/x-fragment">
    // Fragment shader implemented to perform lighting
    according to the
    // Phong reflection model with a spot light. The
    interpolation method
    // that is used is Phong shading (per-fragment shading)
    and therefore the
```

```

// actual lighting calculations are implemented here in
the fragment shader.

precision mediump float;

varying vec2 vTextureCoordinates;
varying vec3 vNormalEye;
varying vec3 vPositionEye3;

uniform vec3 uAmbientLightColor;
uniform vec3 uDiffuseLightColor;
uniform vec3 uSpecularLightColor;
uniform vec3 uLightPosition;
uniform vec3 uSpotDirection;
uniform sampler2D uSampler;

const float shininess = 32.0;
const float spotExponent = 40.0;
const float spotCosCutoff = 0.97;      // corresponds to 14
degrees

vec3 lightWeighting = vec3(0.0, 0.0, 0.0);

void main() {

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition -
vPositionEye3);

    // Calculate n dot l for diffuse lighting
    float diffuseLightWeighting = max(dot(vNormalEye,
                                           vectorToLightSource),
0.0);

    // We only do spot and specular light calculations if we
    // have diffuse light term.
    if (diffuseLightWeighting > 0.0) {
        float spotEffect = dot(normalize(uSpotDirection),
                               normalize(-
vectorToLightSource));

        // Check that we are inside the spot light cone
        if (spotEffect > spotCosCutoff) {
            spotEffect = pow(spotEffect, spotExponent);

            // Calculate the reflection vector (r) needed for
specular light

```

```

        vec3 reflectionVector = normalize(reflect(
vectorToLightSource,
vNormalEye));

        // Camera in eye space is in origin pointing along
the negative z-axis.
        // Calculate viewVector (v) in eye coordinates as
        // (0.0, 0.0, 0.0) - vPositionEye3
        vec3 viewVectorEye = -normalize(vPositionEye3);

        float rdotv = max(dot(reflectionVector,
viewVectorEye), 0.0);

        float specularLightWeighting = pow(rdotv,
shininess);

        lightWeighting =
            spotEffect * uDiffuseLightColor *
diffuseLightWeighting +
            spotEffect * uSpecularLightColor *
specularLightWeighting;
    }
}

// Always add the ambient light
lightWeighting += uAmbientLightColor;

        vec4 texelColor = texture2D(uSampler,
vTextureCoordinates);
        // modulate texel color with lightweigthing and write as
final color
        gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb,
texelColor.a);
    }
</script>
```

UNDERSTANDING THE ATTENUATION OF LIGHT

The intensity of light is usually attenuated when the light travels from its source through a medium (such as air). In the real world, the intensity of light usually falls off proportionally to $1/r^2$, where r is the distance from the light

source. The function used to describe the falloff is sometimes referred to as the *distance falloff function*.

In computer graphics, it is common to use a distance falloff function that is different than $1/r^2$. One reason for this is that the function $1/r^2$ is typically too great a change in the illumination for the simulated light that is used in computer graphics. The following falloff function has been more commonly used in the fixed-function pipelines of desktop OpenGL and DirectX and can also be used for WebGL:

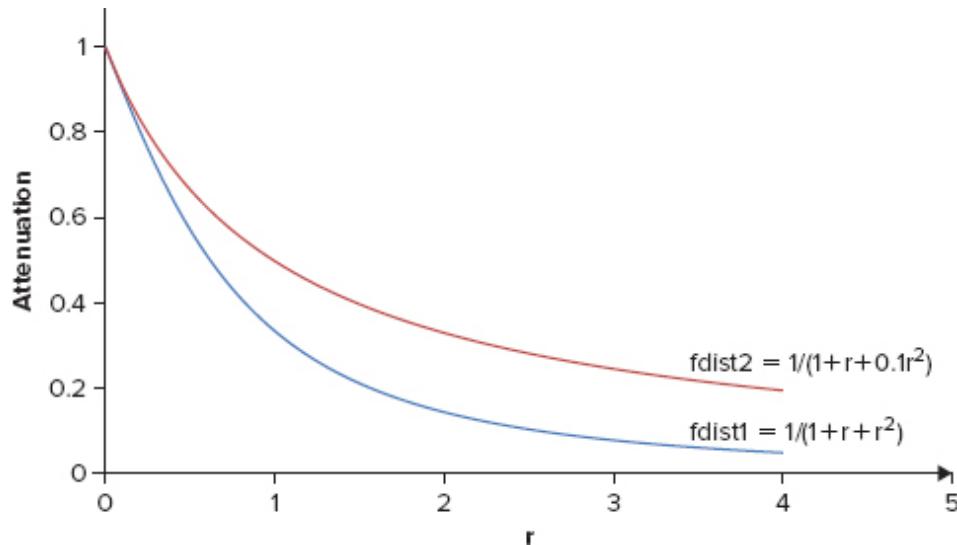
$$\frac{1}{constantAtt + linearAtt \times r + quadraticAtt \times r^2}$$

The variables in this function are as follows:

- *constantAtt* is the constant attenuation
- *linearAtt* is the linear attenuation
- *quadraticAtt* is the quadratic attenuation
- *r* is the distance from the light source

[Figure 7-20](#) shows the curve for this falloff function. The *x*-axis shows *r*, which is the distance from the light source. The *y*-axis shows the attenuation. The bottom curve (named fdist1) shows the falloff function when all constants are equal to 1.

FIGURE 7-20: An example of a falloff function with two sets of constants. The *x*-axis shows the distance from the light source and the *y*-axis shows the attenuation



This means that you have the following falloff function:

$$\frac{1}{1 + r + r^2}$$

The upper curve (named fdist2) shows the falloff function when the quadratic attenuation constant equals 0.1 and the other two constants equal 1. This means that you have this falloff function:

$$\frac{1}{1 + r + 0.1 r^2}$$

You can see that by decreasing the quadratic constant from 1 to 0.1, you “lift” the curve so that the attenuation is happening at a slightly slower rate when the distance is increased.

Very little new shader code is needed to implement the attenuation based on this falloff function. [Listing 7-8](#) shows a complete vertex shader that implements a point light with attenuation. The code that calculates attenuation is highlighted.



Available for
download on
[Wrox.com](#)

[**LISTING 7-8:** A fragment shader for a point light with attenuation](#)

```
<script id="shader-fs" type="x-shader/x-fragment">
    // Fragment shader implemented to perform lighting
    according to the
    // Phong reflection model with a point light and
    attenuation.
    precision mediump float;

    varying vec2 vTextureCoordinates;
    varying vec3 vNormalEye;
    varying vec3 vPositionEye3;
    varying vec3 vLightPositionEye3;

    uniform vec3 uAmbientLightColor;
    uniform vec3 uDiffuseLightColor;
    uniform vec3 uSpecularLightColor;
    uniform vec3 uLightPosition;
    uniform sampler2D uSampler;

    const float shininess = 32.0;
    // Attenuation constants
    const float constantAtt = 1.0;
    const float linearAtt = 0.1;
    const float quadraticAtt = 0.05;
```

```

vec3 lightWeighting = vec3(0.0, 0.0, 0.0);

void main() {
    float att = 0.0; // Attenuation

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(vLightPositionEye3 -
- vPositionEye3);

    // Calculate n dot l for diffuse lighting
    float diffuseLightWeighting = max(dot(vNormalEye,
                                           vectorToLightSource),
0.0);
    // Only do spot calculations if diffuse term is
available
    if (diffuseLightWeighting > 0.0) {
        // Calculate attenuation
        float distance = length(vec3(vLightPositionEye3 -
vPositionEye3));

        att = 1.0/(constantAtt+linearAtt * distance +
quadraticAtt * distance * distance);

        // Calculate the reflection vector (r) needed for
specular light
        vec3 reflectionVector = normalize(reflect(-
vectorToLightSource,
                                           vNormalEye));
    }

    // Camera in eye space is in origin pointing along the
negative z-axis.
    // Calculate viewVector (v) in eye coordinates as
    // (0.0, 0.0, 0.0) - vPositionEye3
    vec3 viewVectorEye = -normalize(vPositionEye3);

    float rdotv = max(dot(reflectionVector,
viewVectorEye), 0.0);

    float specularLightWeighting = pow(rdotv, shininess);

    lightWeighting =
        att * uDiffuseLightColor * diffuseLightWeighting +
        att * uSpecularLightColor * specularLightWeighting;
}

```

```

        }

        // Always add the ambient light
        lightWeighting += uAmbientLightColor;

        vec4      texelColor      =      texture2D(uSampler,
vTextureCoordinates);
        // modulate texel color with lightweigthing and write as
final color
        gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb,
texelColor.a);
    }
</script>

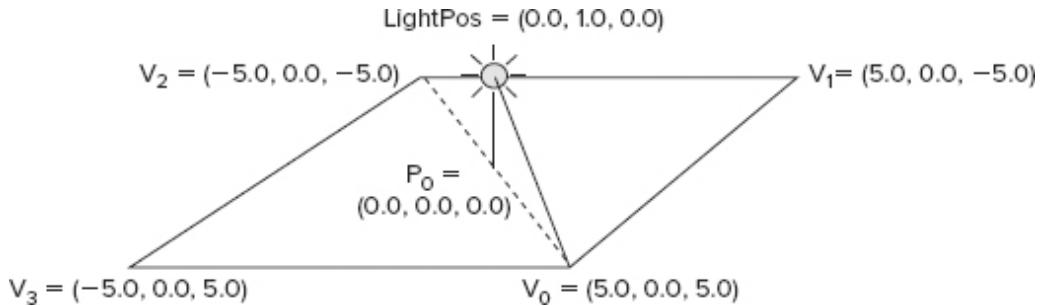
```

Even though [Listing 7-8](#) shows only a small amount of new code, it's important to consider how the calculation of the attenuation is performed in this example. First, since the attenuation does not fall off linearly with the distance, it is not really correct to calculate the attenuation in the vertex shader and use the linearly interpolated value in the fragment shader. Normally you could calculate the distance in the vertex shader and send the value to the fragment shader where you would then calculate the attenuation.

But if you have a coarse model where the vertices are far away from each other (such as the floor and the table that you have seen several times in this book), this might produce the wrong result. The reason is that you calculate the distance at the vertices and this distance is then linearly interpolated over the fragments. With a coarse model, and especially when the light is close to the model, the distance at the vertices can be considerably longer than the distance at the closest fragments for the corresponding triangle.

This should be easier to understand by looking at [Figure 7-21](#). The sketch represents the textured floor that is modeled with two triangles with the following vertex positions:

FIGURE 7-21: An example of how a model that is too coarse can present problems when the distance that is needed for the attenuation is calculated in the vertex shader



```
var floorVertexPosition = [
    // Plane in y=0
    5.0, 0.0, 5.0, //v0
    5.0, 0.0, -5.0, //v1
    -5.0, 0.0, -5.0, //v2
    -5.0, 0.0, 5.0]; //v3
```

A point light has the coordinates $0.0, 1.0, 0.0$, which are at a distance of 1.0 above the center of the floor (which is named P_0 and has the coordinates $0.0, 0.0, 0.0$). The distance from the point light to the vertex V_0 is given by:

$$\|(5.0, 0.0, 5.0) - (0.0, 1.0, 0.0)\| \sqrt{5.0^2 + (-1.0)^2 + 5.0^2} = \sqrt{51} = 7.1$$

In the shader, you don't have to calculate the length manually like this. OpenGL ES Shading Language contains the built-in function `length()`, which is useful for calculating the length of a vector.

Since all four vertices have the same distance to the light source, it means that when the distance is linearly interpolated, the fragment at P_0 also gets the same distance. This is obviously wrong because the distance from P_0 to the light source should be 1.0, but the calculation with `length()` gives you a length of 7.1. Since the distance is too large, the fragments around P_0 are much too dark.

This is the reason for putting the calculation of the distance in the fragment shader in [Listing 7-8](#). Another approach is, of course, to have slightly less coarse models with more vertices and instead perform the distance calculation in the vertex shader.



Note that attenuation is not generally used for a directional light. Since the light source for the directional light is infinitely far away, it does not make sense to attenuate the light based on the distance from the light source.

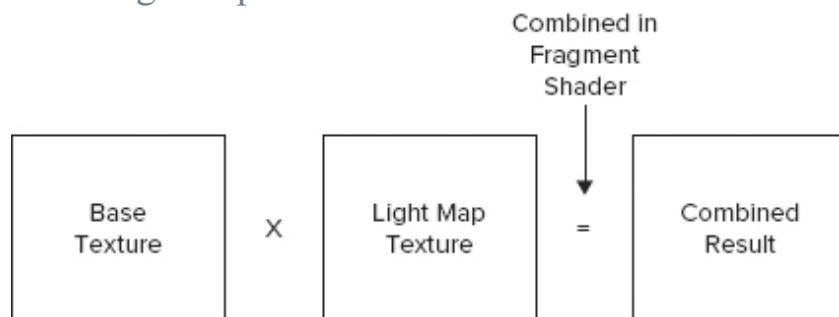
UNDERSTANDING LIGHT MAPPING

You should now understand how you can simulate lighting in real-time with, for example, the Phong reflection model and Gouraud shading or Phong shading. It's also important to understand *light mapping*, which is a slightly different technique to simulate lights and is especially common in 3D graphics games. The first 3D graphics game that used the technique was Quake.

The idea with light mapping is that if you have static objects in your scene, you can simulate lighting on these objects by creating a texture that contains precomputed lighting information. By doing this, you can have high-quality lighting that is calculated with a global lighting model, such as radiosity, but you don't need to do a lot of calculations in real time. The calculations can be done offline in advance by using a 3D modeling program such as Blender or Maya, and then the result can be stored in a texture that can be used in your WebGL application.

In theory, you can use the same texture to store the base texture image (such as a brick wall) and the lighting information. However, often a better strategy is to use a separate texture called a *light map* for your lighting information. (The word light map typically refers to a texture that contains only the lighting information.) The light map is then combined with a base texture image in the fragment shader. [Figure 7-22](#) shows the basic idea.

FIGURE 7-22: An example of light mapping using a base texture that is combined with the light map



By keeping the light map separate from the base texture, the lighting can be reused on several different base textures. You can also more easily change the

light on the base texture, which might be desirable, for example, if the light map simulates a flashlight that is switched off.

In addition, a base texture such as a brick wall is often tiled in some way to reduce the memory usage. Normally it does not make sense to have the lighting tiled in the same way.

Since you have learned how to use regular textures in WebGL, it is not difficult to use light maps once you have created them. To give you an idea of how it can be done, the following code snippet shows a fragment shader that has two different samplers. The sampler `uSamplerBase` is used to sample from the base map, while the sampler `uSamplerLight` is used to sample from the light map.

```
precision mediump float;
varying vec2 vTextureCoordinates;
uniform sampler2D uSamplerBase;
uniform sampler2D uSamplerLight;

void main() {
    vec4 baseColor = texture2D(uSamplerBase, vTextureCoordinates);
    vec4 lightValue = texture2D(uSamplerLight,
vLightCoordinates);
    gl_FragColor = baseColor * lightValue;
}
```

For this fragment shader to work, you would typically have the following code snippet in JavaScript:

```
// Get the location of the uniform uSamplerBase for the base
map
pwgl.uniformSamplerBaseLoc =
gl.getUniformLocation(shaderProgram,
"uSamplerBase");

// Get the location of the uniform uSamplerLight for the
light map
pwgl.uniformSamplerLightLoc =
gl.getUniformLocation(shaderProgram,
"uSamplerLight");
...

// Bind the base texture to texture unit gl.TEXTURE0
gl.activeTexture(gl.TEXTURE0);
```

```

gl.bindTexture(gl.TEXTURE_2D, textureBase);

// Bind the light texture to texture unit gl.TEXTURE1
gl.activeTexture(gl.TEXTURE1);
gl.bindTexture(gl.TEXTURE_2D, textureLight);

...

// Set base map sampler to the value 0
// so it matches the texture unit gl.TEXTURE0
gl.uniform1i(pwgl.uniformSamplerBaseLoc, 0);

// Set light map sampler to the value 1
// so it matches the texture unit gl.TEXTURE1
gl.uniform1i(pwgl.uniformSamplerLightLoc, 1);

```

SUMMARY

In this chapter, you have learned a lot about lighting in WebGL. You should now understand the difference between a local lighting model and a global lighting model, and you should understand the Phong reflection model, which is a popular local lighting model. You should now know the three different reflection components that are used in the Phong reflection model:

- Ambient reflection
- Diffuse reflection
- Specular reflection

You should also be familiar with the three major interpolation techniques that are used for shading in 3D graphics,

- Flat shading
- Gouraud shading (per-vertex shading)
- Phong shading (per-fragment shading)

as well as the three different types of lights:

- Directional lights
- Point lights
- Spot lights

You should now understand the difference between these types of lights and be able to write shader code in OpenGL ES Shading Language that models

lighting based on these lights. You should also know how attenuation can be used to make a point light or a spot light more realistic. Finally, you should now understand how light mapping can be used to simulate light in WebGL.

Chapter 8

WebGL Performance Optimizations

WHAT'S IN THIS CHAPTER?

- Real-world hardware that powers WebGL
- Key software components that power WebGL
- How to locate a performance bottleneck in the WebGL graphics pipeline
- How to optimize different WebGL bottlenecks
- General performance tips for WebGL
- How blending works in WebGL and how it can be used to implement transparency

In some cases, you might develop a WebGL application where you only show relatively static graphics, and so the performance might not be so important. However, in most cases, the performance is very important. In the previous chapters you learned what you should consider from a performance point of view. But since performance is such an important topic, the majority of this chapter is devoted to WebGL performance optimizations.

However, before you jump into the performance-related topics in this chapter, you will get a short overview of what WebGL looks like under the hood — both the hardware and the software components. Familiarizing yourself with these components will help you understand and analyze performance-related problems better.

At the end of the chapter, you will learn more about blending and how this technique can be used to implement transparency. Finally, you will be pointed to some resources that will help you to continue learning WebGL, even after you have finished reading this book.

WEBGL UNDER THE HOOD

In this section you will see some examples of real-world hardware and software that power WebGL-enabled devices. Since the usage of mobile devices such as smartphones and tablets is increasing very quickly, many users will likely use a mobile device to run the WebGL applications that you create. Because of this (and also because I have a lot of experience in the mobile industry), the examples discussed in this section are taken from the mobile industry. However, you should note that the hardware and software components used in mobile devices today are in many ways similar to the components used in the PC industry, so most of the discussion is generally relevant.

Hardware That Powers WebGL

Today, smartphones and tablets are built using two different types of technical solutions:

- Integrating the application processor and the cellular modem into a single piece of silicon. The advantage with this solution is that it keeps down both cost and power usage.
- Keeping the modem and the application processor separate, which can give device manufacturers more flexibility when combining the application processor and modem in a certain product.

ST-Ericsson has three product families that device manufacturers can use to build smartphones and tablets:

- *NovaThor* is a family of integrated, complete mobile platforms that combine the application processor and the cellular modem into one piece of silicon.
- The *Nova* family of standalone application processors do not contain an integrated cellular modem.
- *Thor* cellular modems give mobile broadband access to the device in which they are integrated.

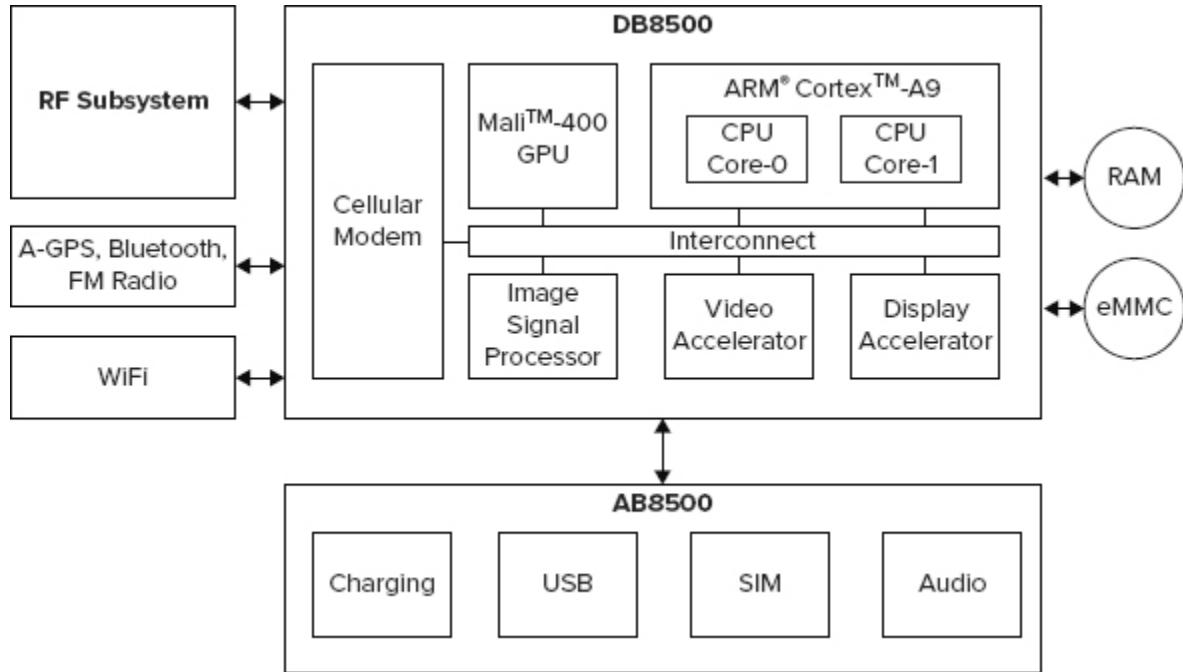
The following sections describe NovaThor U8500 and NovaThor L9540.

NovaThor U8500

NovaThor U8500 is based on a dual-core ARM Cortex-A9 processor running at a maximum frequency of 1GHz. [Figure 8-1](#) shows a simplified

block diagram of the hardware for this mobile platform. All of the blocks in this figure are not described here; the focus is on the hardware blocks that are most interesting from a WebGL point of view.

FIGURE 8-1: A simplified block diagram for NovaThor U8500



The dual-core ARM Cortex-A9 processor is the CPU where most of the software is executed. For example, the Linux kernel and Android are executed on the Cortex-A9. This is also where the web browser and its support libraries, such as WebKit and the V8 JavaScript engine, run. Since NovaThor U8500 uses a dual-core CPU, different threads and processes can be scheduled at the two different cores simultaneously.

The Interconnect is a high-performance, on-chip bus that connects the Cortex-A9 processor with the other on-chip blocks, such as the Mali-400 GPU from ARM, the image signal processor, the video accelerator, the display accelerator, and the cellular modem.

From a WebGL perspective, the Mali-400 GPU is, of course, a very important piece of hardware. This GPU is based on what is called *tile-based deferred rendering* (sometimes abbreviated as TBDR). This technology is especially common for GPUs in mobile devices, and it is based on the concept that the framebuffer is divided into smaller tiles. For Mali-400, the tiles are 16×16 pixels in size, and since such a small tile does not require

much memory, the tile that the GPU is currently working with can be kept in very fast on-chip memory. This is the main advantage of this architecture. When a tile has been rasterized and all per-fragment processing has finished for the tile, the content (color and depth) can be written to the complete framebuffer that is located in external memory.

Even though most performance discussions related to WebGL usually focus on the GPU, the CPU or possibly the bus between the GPU and the CPU, as well as the network connection also greatly affect the overall performance that the user will experience. With a fast network connection your WebGL application can load faster, which can be very important, especially if you need to download large 3D models for your application.

There are two ways in which a WebGL application can be downloaded to a NovaThor U8500-based device: Either the WLAN is used or the cellular modem is used together with the radio frequency (RF) subsystem. In the second case, the cellular modem modulates a carrier signal with the digital information that can then be transmitted with the RF subsystem.

NovaThor L9540

NovaThor L9540 consists of an application processor that is based on a dual-core ARM Cortex-A9 processor combined with the mobile broadband modem Thor M7400 from ST-Ericsson. The application processor contains a PowerVR Series5 GPU from Imagination Technologies, which is much more powerful than the Mali-400 GPU that is used in NovaThor U8500. In addition the frequency and the size of the caches of the Cortex-A9 processor are increased in NovaThor L9540 compared to NovaThor U8500.

Even though WebGL works well on the NovaThor U8500 platform, the efficient broadband modem M7400, the new PowerVR Series5 GPU and the improved Cortex-A9 processor in NovaThor L9540 make it an even more powerful platform to run WebGL on than NovaThor U8500.

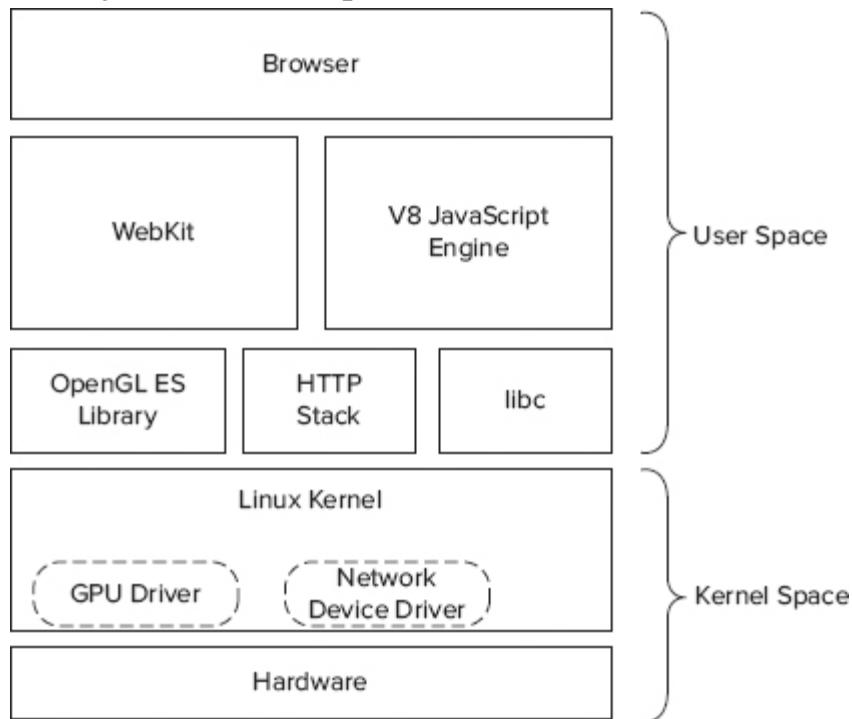


If you are the kind of person who likes to prototype new embedded designs and you want to control the complete software for the device you work with, then you should have a look at the IGLOO open source community (www.igloocommunity.org). This open source community is based around Snowball, which is a low-cost, single-board computer. Snowball is in turn based on the ST-Ericsson Nova A9500 processor, which is one of the predecessors to NovaThor L9540.

Key Software Components

In a smartphone, tablet, or PC that supports WebGL, there are a lot of different software components. [Figure 8-2](#) shows a simplified figure where you can see some of the most important software components from a WebGL perspective.

[FIGURE 8-2:](#) Key software components in a WebGL-enabled device



On the top you have the web browser. The web browser typically controls the window that is used to display WebGL and other web content. The web browser is also responsible for the address bar, bookmarks, history, and all other menus that are available in the browser.

All major web browsers today are built on some kind of browser engine that performs the heavy and complicated work of parsing HTML, building the DOM tree, creating the layout of the web page, and so on. For example, Google Chrome and Apple Safari both use the open source browser engine WebKit, while Firefox uses Gecko, which is open source as well.

V8 JavaScript engine is a modern and efficient JavaScript engine that is used to execute the JavaScript. It is created primarily by Google developers, but since it is open source, other companies and individuals have contributed to it as well. V8 parses the JavaScript source code and builds what is called an Abstract syntax tree (AST). With help from the AST, it then compiles the JavaScript code to binary executable code for the specific target architecture (such as IA-32, x64, ARM, or MIPS). The CPU then executes the binary code.

The software drivers that are needed for the GPU are usually split into one user space library (which in [Figure 8-2](#) is called OpenGL ES Library) and one kernel space driver. In a Linux environment, one reason for this split is that the Linux kernel (including the kernel drivers) is released under GPLv2, which requires that the source code for the drivers be released.

However, most of the GPU vendors do not want to reveal all the source code for their drivers. They often try to keep their code secret by including most of the functionality in the user space library, which they keep proprietary and do not release publicly. The kernel space driver is GPLv2 licensed, but it is just a thin “shim” that the user space library is using to communicate with the hardware, so it does not reveal much about the real logic.



The user space library is sometimes called User Mode Driver (UMD) and corresponds to a shared library (.so file) in a Linux environment and a dynamic-link library (*.dll file) in a Microsoft Windows environment.*

[Figure 8-2](#) also contains the HTTP-stack that WebKit uses to request web content from the web server. The HTTP-stack uses sockets that export the functionality for the TCP/IP-stack in the Linux kernel. The TCP/IP-stack in the Linux kernel in turn uses a network device driver to send the IP-packets to, for example, the cellular modem or WLAN.

The `libc` represents standard library functionality that the browser, WebKit, and JavaScript engine depend on.



Sony Ericsson (now Sony Mobile Communications) was the first vendor to release WebGL support in the stock Android browser. The source code has been released as open source, and if you are interested in contributing to it or just experimenting with it, you can find it on the web: <https://github.com/sonyericssondev/WebGL>.

Example Use Case for WebGL

To help you understand how the different components are interacting, this section contains a scenario describing the interaction when a WebGL application is downloaded and displayed in a web browser. The following steps are simplified and kept on a high level.

1. The user enters the URL of a web page containing WebGL content in the address bar of the web browser. The URL is sent down to WebKit, which uses the HTTP-stack to create an HTTP request message for the requested web page.
2. The HTTP request message is sent on a socket down to the TCP/IP-stack, which uses the Transmission Control Protocol (TCP) and the Internet Protocol (IP) to encapsulate the HTTP request message. The IP packets are placed in a buffer and transmitted by the network device driver over, for example, the cellular modem or the WLAN.
3. When the HTTP request message reaches the web server, the web server responds with an HTTP response message and sends the requested content back to the user's device. To keep it simple, assume that all the content (HTML, JavaScript, and shader source code) is included in a single file.
4. The requested content arrives at the user's device and is sent up through the network device drivers, the TCP/IP-stack, and the HTTP-stack to WebKit, where it is parsed and an internal DOM tree is built. WebKit also creates something called a *render tree*, which is an internal description of how the different elements should be rendered. The JavaScript is sent to the V8 JavaScript engine, which parses the

JavaScript and compiles it into binary executable code that then is executed by the CPU.

5. When the JavaScript code contains calls to the WebGL API, these calls go back to WebKit, which calls the OpenGL ES 2.0 API. The source code for the vertex shader and the fragment shader (which is written in OpenGL ES Shading Language) is compiled to binary code by the OpenGL ES 2.0 library, and then the binary code is uploaded to the GPU via the kernel mode driver.
6. Finally, textures, vertex buffers, and uniforms are set up and uploaded to the GPU so the rendering can start.

WEBGL PERFORMANCE OPTIMIZATIONS

Most of the examples in this book have been relatively limited in order to illustrate some specific features of WebGL. This means that from a performance point of view, it does not really matter how the code for the applications has been structured. Most WebGL implementations have no problem running these limited examples. However, when you start to write large, real-world applications with a lot of objects and possibly also more complicated shaders, you will eventually run into performance problems. This part of the chapter will give you some tips on how you tackle these situations.

Avoiding a Typical Beginner's Mistake

When you try to measure the performance of different parts of an application that only runs on the CPU, a basic strategy is to add code that starts a clock, performs some work, and then stops the clock. You then know how long it took to perform the work.

If the same idea is applied to a WebGL application, you might be tempted to write code that looks something like the following pseudo-code:

```
// This code does NOT measure how long  
// it takes to do the actual drawing.
```

```
startClock();  
  
gl.drawElements(gl.TRIANGLE_STRIP, ...);  
  
stopClock();
```

This code will typically not work as you might expect if you are new to WebGL. The call to `gl.drawElements()` does not measure the time it actually takes to render the triangle strip.

This is because the graphics pipeline works in an asynchronous way. When you call `gl.drawElements()` or any other WebGL method, a lower-level function usually places a command in a command buffer somewhere and then returns. These commands are then executed by the GPU sometime later. This means that the previous code only measures how long it takes to place a command in a command buffer, which is not that interesting from a performance point of view.

The methods `gl.flush()` and `gl.finish()`

WebGL contains two methods which can be tempting to use to measure the rendering time with the procedure described in the previous section:

- `gl.flush()`
- `gl.finish()`

When the method `gl.flush()` is called, it indicates that all commands that have previously been sent to WebGL must complete in a finite amount of time. Obviously that will not help you much, even though the name of the method might sound promising.

The method `gl.finish()` is specified to force all previous WebGL commands to complete, and the method should not return until the framebuffer is fully updated. Although it sounds like the method `gl.finish()` could solve the problem, in reality, many GPU vendors try to gain additional performance by not making `glFinish()` (which is the OpenGL ES function that `gl.finish()` typically relies on) fully synchronous either.

This means that you cannot be completely sure that this code measures the time it takes to render the triangle strip:

```
// This code does NOT measure how long  
// it takes to do the actual drawing.
```

```
startClock();  
  
gl.drawElements(gl.TRIANGLE_STRIP, ...);  
  
gl.finish();  
  
stopClock();
```

Locating the Bottleneck

You learned in Chapter 1 that the rendering in WebGL is highly pipelined. This means that for a specific device at a specific time, your WebGL application has a performance bottleneck in one of the pipeline stages. When one of the stages is the performance bottleneck, this means that it is the slowest stage of the pipeline. The bottleneck stage sets the limit for the total rendering performance.

To improve the overall rendering performance, you need to optimize the bottleneck stage. However, you should note that during one frame, the bottleneck can move between different stages of the pipeline. So when a specific stage of the pipeline is said to be the bottleneck, it means that this stage is the bottleneck during the majority of the frame.

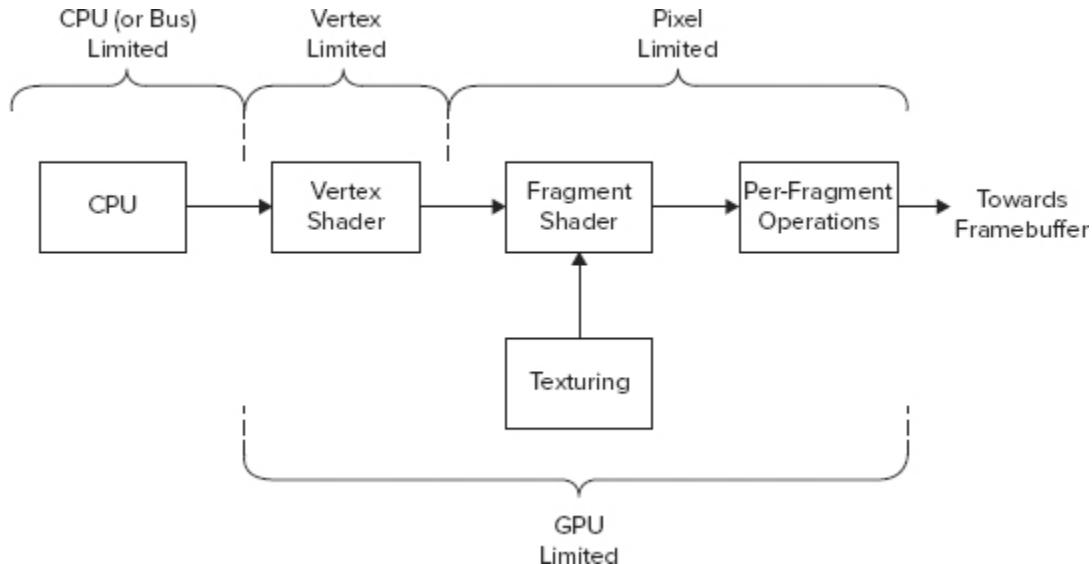


A major advantage with web applications is that they can be used on so many different devices, operating systems, and browsers. But this is also a real challenge when you try to locate the current bottleneck of your WebGL application. The bottleneck can be located in different stages for different devices.

Even though this might sound obvious, it is important to remember. To find out where your application has a bottleneck for a certain device, you have to test on that device. However, it is practically impossible to test your application on all WebGL-enabled devices. So you have to test a few selected devices and then hope that your application will also run well on other devices. In general, you can assume that if your application works well on a low-end mobile device, it will also work on a high-end desktop.

The WebGL performance bottlenecks fall into three broad categories (as shown in [Figure 8-3](#)):

FIGURE 8-3: A simplified WebGL pipeline showing performance bottlenecks



- CPU-limited (sometimes called application-limited)
- Vertex-limited (sometimes called geometry-limited)
- Pixel-limited

The pipeline in [Figure 8-3](#) is a simplified representation of both the real and the theoretical WebGL pipeline that was described in Chapter 1. It includes boxes to make it easier to understand where you have a performance bottleneck. Also note that there are many different ways to categorize a bottleneck in a WebGL graphics pipeline. For example, you can see from [Figure 8-3](#) that with the broad categories described by the three bullets in this section, a bottleneck that is related to textures is grouped as pixel-limited. Another alternative would be to have texture-limited as its own category.

When you want to improve the performance of your WebGL application, the first step is to locate the current bottleneck. To do this, you can perform a number of tests where you change the workload at a specific stage:

- Increase the work that a particular stage has to do. If the performance drops, you have found the current bottleneck.
- Decrease the work on a particular pipeline stage. If you decrease the work that a particular stage has to do and the performance increases, then you have also found your bottleneck.

A related technique that can also help you locate the bottleneck is to change the frequency of the CPU or the GPU. If you have an environment where this

is easy to do, you can quickly find out whether or not your application is CPU-limited.

The following sections describe the tests you can perform to determine where your bottleneck is located. It is not necessary to perform the tests in any specific order. Each test that is described will give you a clue about where the bottleneck is located. If you have a device where some tests are easier to perform than others, it often makes sense to start with the easy tests. Performing several different tests on the same stage will make you more confident in your conclusions about the bottleneck.



New GPUs sometimes support what is called Unified Shader Architecture. This means that all shader types use the same hardware to execute the instructions, and the driver can load-balance during run time and decide which hardware block in the GPU should be used to execute the vertex shader or the fragment shader.

For GPUs that are using a Unified Shader Architecture, it is difficult to use the different tests presented in this chapter to decide where in the GPU a bottleneck is located. In this case, it might only make sense to say that the application is CPU-limited or GPU-limited. If you need more information, most GPU vendors also provide different profiling tools that can help when you need to perform detailed performance analysis.

Testing for CPU-Limited WebGL Applications

CPU-limited means that a WebGL application is limited by the speed of the CPU. However, the expression is sometimes used in a broader sense, meaning that the application is limited by something other than the GPU. In this case, it means that the application could actually be limited by, for example, the bus bandwidth, the cache sizes of the CPU, or the amount of available RAM.

The work in your WebGL application that is typically done on the CPU depends on the code that is written in JavaScript, and can include the following:

- Handling of user input
- Physics simulations
- Matrix multiplications due to object transformations
- Collision detection
- Artificial intelligence

Before you start thinking about whether your WebGL application is CPU-limited, it is a good idea to first make sure that you don't include any code in your render loop that slows down the rendering. If you have JavaScript code that accesses the file-system API, local storage, or indexed database inside your rendering loop, it can ruin the performance completely. You also do not want to include network access and compilation of shader source code inside your rendering loop.

If you write a complete WebGL application yourself, then you probably have control over which APIs you access and when. But if you are reusing code and libraries that others have written, it can be good to keep this in mind. For complex code, it helps to use Chrome Developer Tools or Firebug to see exactly what is going on.

You can use the following tests to decide whether your application is CPU-limited:

- Look at the CPU usage with a program that is available for the platform you use. For example, on Linux you have *top*, on Windows the *Task Manager*, and on Mac OS the *Activity Monitor*. If your application is close to 100 percent in CPU usage, this can mean that your application is CPU-limited. However, you should be careful with your conclusions from this test. There could be cases where you are waiting for a GPU with a busy-wait construction; for example, with a spinlock, you might have a high CPU usage even though it is actually the GPU that is the bottleneck. You should also note that even if these tools show a low CPU usage, it does not necessarily mean that your bottleneck is in the GPU. The speed of buses, RAM, network activity, and so on can often be limiting factors in the performance.
- Clock down your CPU to see whether the performance changes (this is a more direct test). If you clock down the CPU with x percent and the performance drops x percent, then you are likely CPU-limited. However, if you clock down the CPU with x percent and the performance does not drop, then you are most likely not limited by the CPU frequency.
- Change the GPU frequency if that is possible on the device you test on. This is a complement to changing the CPU frequency. On a PC, the BIOS sometimes lets you change the clock frequency for the GPU, and for several GPUs there are also specific utility programs (such as *ATITool*,

CoolBits, and *PowerStrip*) that you can use to modify the clock frequency of the GPU. If you clock down the GPU and your performance drops, you are likely GPU-limited.

- If you can easily change your JavaScript code so you do less logic and other work on your CPU but it still renders the same scene, you can get an indication of whether you are CPU-limited. If less work on the CPU speeds up performance, you are likely CPU-limited.

If you suspect that you are CPU-limited, Chrome Developer Tools and Firebug can be invaluable in helping you understand the details of what happens in the JavaScript code. As already mentioned, the tools can also give you other valuable clues about network activity or file-system activity that limits the performance of your WebGL application.

If you are interested in how the complete system behaves on the CPU side, including lower-level software and hardware, you could use a tool such as OProfile or perf. These tools let you profile the native code so you can see in which native (C/C++) functions most time is spent. All modern CPUs also contain a Performance Monitor Unit (PMU) that counts and reports low-level events such as:

- Data and instruction cache misses
- TLB misses
- Branch prediction misses
- Elapsed cycles
- Executed number of instructions

Perf, which is included in the Linux kernel, is particularly useful when you need to listen to these low-level events.

If you are an application developer and interested only in optimizing your WebGL application, these native performance-analysis tools are probably not the first ones you will use. But if you want to contribute performance improvements to open source software components such as WebKit, Chromium, Gecko, or Firefox, these tools can be very useful.

Testing for Vertex-Limited WebGL Applications

Vertex-limited (sometimes referred to as geometry-limited) describes a WebGL application whose overall speed is limited by how fast vertices can

be fetched to the vertex shader or processed in the vertex shader. The actual processing work that is done in the vertex shader can include:

- Vertex transformations
- Per-vertex lighting

To test whether your application is vertex-limited, you can perform the following tests:

- If you are using 3D models with many vertices, you could try models that have fewer vertices.
- Decrease the number of per-vertex lights or use less complex shading calculations for the lights.
- See if you can remove some transformations or any other work that is done in the vertex shader.

If the performance increases for any of these tests, you are probably vertex-limited. You can, of course, also do the reverse; for example, you can add extra processing code in the vertex shader, and if the performance is decreased, it is the vertex shader that is your bottleneck. To investigate whether you have a bottleneck that is due to vertex fetching, you can add some extra dummy data as vertex data. If the performance decreases, you likely have a bottleneck in vertex fetching and should think of ways to reduce the amount of vertex data or possibly try to organize your vertex data to better use the pre-transform vertex cache and the post-transform vertex cache.

Testing for Pixel-Limited WebGL Applications

An application that is *pixel-limited* is limited by something that happens after the vertex shader. That basically means performing rasterization, processing fragments in the fragment shader, and performing texturing or the per-vertex operations.

Here are some ways that you can test whether you have a bottleneck at this stage:

- Decrease the `width` and `height` properties for the `<canvas>` tag that is used for rendering. For example, if you have code that looks something like `<canvas width="1024" height="1024">` and you change this to `<canvas width="512" height="512">` and the performance

increases, then you are likely pixel-limited. Note that the `width` and `height` attributes of the canvas are different from the `width` and `height` CSS properties that can also be applied to the canvas. You will learn more about this later in the chapter.

- Change the amount of processing that is done for each fragment in the fragment shader. For example, you can remove lights or use less complex lighting calculations.
- Disable texturing or use lower-resolution textures to see if you have a bottleneck in the texturing. You can also change the texture filtering.

General Performance Advice

Even though the recommended way to solve performance problems is to find out where your current bottleneck is, it is often useful to follow some general performance guidelines early on in your design and development. This reduces your chances of running into performance problems in the first place. This section describes some general performance advice that is useful to keep in mind.



In addition to reading the performance advice in this chapter, you can check out two different talks given by Google engineers:

- Gregg Tavares gave a talk called “*WebGL Techniques and Performance*” at Google I/O 2011.
- Ben Vanik and Kenneth Russell talked about “*Debugging and Optimizing WebGL Applications*” at the “*New Game Conference 2011*.”

*You can find both videos on YouTube. (Do a keyword search for “*WebGL Techniques and Performance*” and “*Debugging and Optimizing WebGL Applications*,” respectively.) Some of the advice in this chapter is based on these two talks.*

Reducing the Number of Draw Calls

All calls to the WebGL API are associated with some overhead. Each call requires extra processing on the CPU as well as the copying of data, which takes time and requires the CPU to do some extra work. In general, GPUs can also work more efficiently if they receive a large batch of data that they can process in parallel.

One of the most important pieces of advice regarding WebGL performance is to minimize the number of calls to `gl.drawArrays()` and `gl.drawElements()` during each frame. Often referred to as *batching*, this means that if, for example, you need to draw 1000 triangles to render a part of your scene, it is better to call `gl.drawElements()` or `gl.drawArrays()` twice with 500 triangles in each call than to call the method 100 times with 10 triangles in each call.

This was briefly discussed in Chapter 3 where you also learned how to use degenerate triangles to connect two different triangle strips that are not adjacent, so that you can draw them with a single call to `gl.drawElements()` or `gl.drawArrays()`. If you have a complicated WebGL application where `gl.drawElements()` or `gl.drawArrays()` are called in many different places in the code, it can be helpful to look in the WebGL Inspector, since you will easily see the number of draw calls that are done for each frame.

Avoiding Reading Back Data or States from the GPU While You Are Rendering

GPUs typically work best when you don't read back data from them. All *read* and *get* calls — such as `gl.readPixels()` and `gl.getError()` — can really slow down the performance because the pipeline in the GPU often needs to be flushed.

From a performance point of view, it is much better to read a value once during setup and then cache values that you need in your JavaScript code instead of calling the WebGL API to ask for the value when you are rendering your scene.

Removing Error-Checking and Debugging Libraries from Production Code

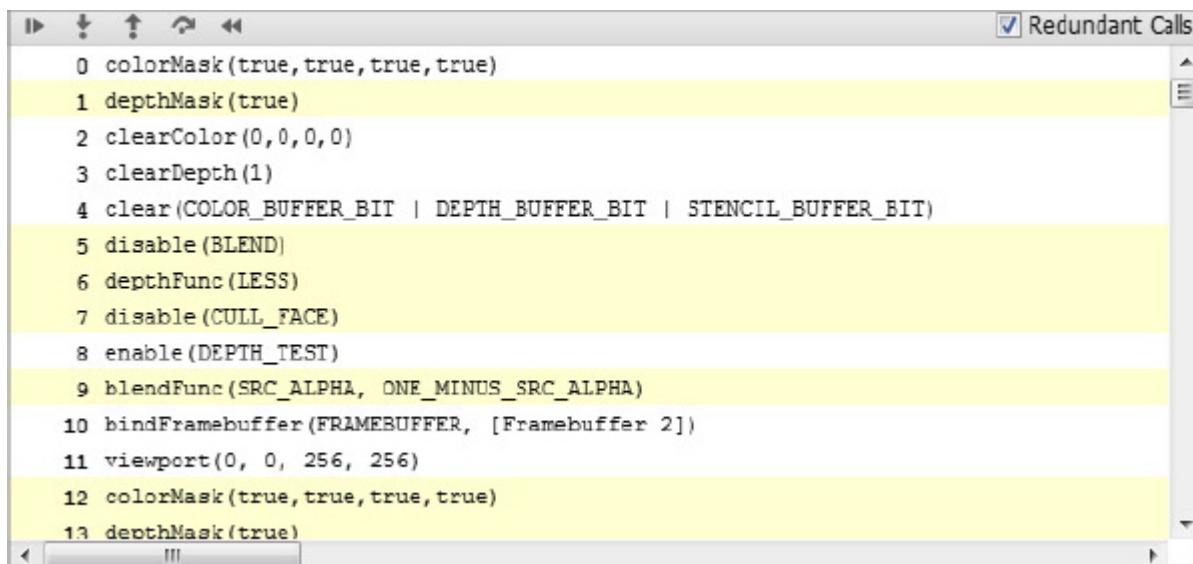
The different libraries that you can use for error checking and debugging during development can be very helpful. However, you should remove libraries such as `webgl-debug.js` from your production code after the development stage. This is because it is not a good idea to have extra JavaScript code that needs to be executed in production code. In addition, the

libraries often include calls to `gl.getError()` and, as you have learned, it is never a good idea to read back values from the GPU if you want good performance.

Using the WebGL Inspector to Find Redundant Calls

In Chapter 2, you learned how useful the WebGL Inspector could be. Since JavaScript calls are not free and especially because changing states for the WebGL pipeline is expensive, you should avoid making redundant calls to the WebGL API. As you probably remember from Chapter 2, the Trace panel in the WebGL Inspector highlights redundant calls, as shown in [Figure 8-4](#).

FIGURE 8-4: The Trace panel of the WebGL Inspector is useful for finding redundant calls to the WebGL API



The screenshot shows the Trace panel of the WebGL Inspector. At the top right is a checkbox labeled "Redundant Calls" which is checked. The main area displays a list of WebGL API calls, each preceded by a line number. The calls are:

```
0 colorMask(true,true,true,true)
1 depthMask(true)
2 clearColor(0,0,0,0)
3 clearDepth(1)
4 clear(COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT | STENCIL_BUFFER_BIT)
5 disable(BLEND)
6 depthFunc(LESS)
7 disable(CULL_FACE)
8 enable(DEPTH_TEST)
9 blendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
10 bindFramebuffer(FRAMEBUFFER, [Framebuffer 2])
11 viewport(0, 0, 256, 256)
12 colorMask(true,true,true,true)
13 depthMask(true)
```

The calls at indices 1, 4, 5, 6, 7, 8, 9, 10, 11, 12, and 13 are highlighted in yellow, indicating they are redundant. The first two calls (0 and 3) are not highlighted.

A good thing to keep in mind is that states in WebGL are persistent across frames. For example, if you want to have depth testing on all the time, you can call the following during setup of your code:

```
gl.enable(gl.DEPTH_TEST);
```

You don't need to call `gl.enable(gl.DEPTH_TEST)` at the beginning of every frame.

Sorting Your Models to Avoid State Changes

In general, you should avoid making unnecessary state changes to the GPU. When you perform state changes, the GPU often has to flush its pipeline, which reduces the performance. If you have a lot of instances of a few objects that you draw in your scene, it is often a good idea to keep your objects sorted so you can set up what is common to a group of objects (for example, states for blending or depth testing; textures, attributes, and uniforms; and the shader program that is used). You then render these objects, set up what is common for the next group of objects, render those objects, and so on.

When you draw, you typically want to avoid going through a list of thousands of objects where you need to make changes to the state for each new object that you draw. So instead of having code that looks like the following pseudo-code:

```
for (var i = 0; i < numberOfModels; i++) {  
    var currentModel = listOfModels[i];  
    gl.useProgram(currentModel.program);  
    currentModel.setupAttributes();  
    currentModel.setupUniforms();  
    gl.drawElements();  
}
```

it is better to keep your models sorted and keep track of whether the last model you rendered was the same as the current model you are going to render. If it is, you don't need to call the WebGL API to do a lot of new updates. This means that the pseudo-code instead looks like this:

```
var previousModelType = null;  
for (var i = 0; i < numberOfModels; i++) {  
    var currentModel = listOfModels[i];  
    if (currentModel.type != previousModelType) {  
        gl.useProgram(currentModel.program);  
        currentModel.setupAttributes();  
        currentModel.setupUniforms();  
        previousModel.type = currentModelType;  
    }  
    gl.drawElements();  
}
```

If you need to prioritize, you should, if possible, sort your objects on what is most expensive to change or update first. For example, it is generally cheaper to update uniforms with a call to `gl.uniform3fv()` than to change the complete shader program with a call to `gl.useProgram()`. From this, it

follows that it is more important to keep objects that are rendered with the same shader program together than to keep objects where you need to change a uniform together.



This is one of the techniques explained by Gregg Tavares in his Google I/O Talk 2011. The source code for this and other examples is available at <http://webglsamples.googlecode.com/hg/google-io/2011/index.html>

Performance Advice for CPU-Limited WebGL

If you are CPU-limited, an obvious way to increase the CPU performance is to decrease the work that the CPU has to do. You can do that by simplifying your application so it performs fewer physics calculations, less game logic, or whatever you have in your JavaScript. If you don't want to simplify your application, you can try the ideas in the following sections.

Moving Out What You Can from the Rendering Loop

This has been partly discussed already, but it's important enough to discuss here as well. You should move out what you can from your rendering loop, and set up what you can before you enter your rendering loop. Especially try to avoid compiling and linking shaders, loading textures, performing network access, and performing file-system access if possible. If you have tight loops within your rendering loop, it also helps to move out common content from them. For example, instead of code that looks like this:

```
for (var i=0; i++; i< 10000) {  
    setupCommon();  
    doSomeSpecificWork();  
}
```

simply use code like this:

```
setupCommon();  
for (var i=0; i++; i< 10000) {  
    doSomeSpecificWork();  
}
```

As you probably know, JavaScript is using garbage collection (GC) to clean up unused objects from the heap. Even though the JavaScript engines are constantly improving, performing GC takes time. If you want to render in 60 frames per second (fps), you have to complete a frame within approximately 16ms, so if you want run smoothly, you don't have much time to perform GC during rendering.

It is difficult to control when the JavaScript engine is performing its GC, but one way to reduce the necessity of a GC is to minimize allocation of new objects. This is not easy in JavaScript since objects are allocated for a lot of things that you do. However, if you have a really critical loop and you know you need a larger block of memory, then you can allocate that memory before you enter the loop. Smaller allocations that happen within the loop will hopefully only lead to GCs that can be performed relatively fast.

Moving Work from the CPU to the GPU

If your application is CPU-limited, you can try to move work from your CPU to your GPU. Even though the JavaScript engines are getting faster, you can still gain performance if, for example, you need to perform many matrix multiplications for each frame in the rendering loop.

If you only draw a few objects, it does not really matter; the JavaScript engines will be fast enough. But if you are drawing thousands of objects in each frame and you need to perform several matrix multiplications to transform each of the objects, this can quickly deteriorate your frame rate if you are CPU-limited.

On the other hand, you should be careful with just moving operations to the GPU and assuming that your performance will increase. In general, you have a relationship where one object in your scene consists of many vertices and even more fragments. So if you are not careful, moving operations from the CPU to the GPU might mean that instead of doing a specific calculation once per object on the CPU, you will do the corresponding calculation many times on the GPU. Doing the calculations on the GPU can result in better performance, but it all depends on where your bottleneck is.

Using Typed Arrays Instead of the JavaScript Array Object

If you suspect that your application is CPU-limited, it can be worthwhile to check if using typed arrays instead of usual arrays in your JavaScript gives you better performance.



For more information about typed arrays, see Chapter 3.

Since typed arrays were invented to serve performance-sensitive functionality and since the typed arrays have a fixed total size and a fixed element size, they should theoretically allow faster read and write access to elements. So instead of creating an array with code like this:

```
myArray = new Array(size);
```

you would create a typed array with the following code:

```
myArray = new Float32Array(size);
```

However, the potential speed increase from using typed arrays typically depends on how you use the arrays — for example, whether you read or write a lot, when you create the arrays, and so on.

This is also typically an optimization that might behave differently in different browsers and also in different versions of the same browser. If you consider using this optimization, then it's a good idea to search the web for the latest discussions in forums, on blogs, and so on, since JavaScript engines and web browsers evolve very quickly.



The JavaScript library `glMatrix`, which has been used in several examples in this book, checks whether the typed array `Float32Array` is supported in the browser; and if it is, it is used for all arrays. The JavaScript library `Sylvester` (which you also learned about in Chapter 4) uses only the typical JavaScript `Array` object to implement its functionality.

Performance Advice for Vertex-Limited WebGL

The following sections describe some optimizations that can improve performance if your application is vertex-limited. Some of these topics were discussed in Chapter 3, which covered the different ways to draw in WebGL. It may be useful to refer to Chapter 3 if you have forgotten some of the details.

Using Triangle Strips or Triangle Fans

If you use triangle strips (`gl.TRIANGLE_STRIP`) or triangle fans (`gl.TRIANGLE_FAN`) instead of separate triangles (`gl.TRIANGLES`), you have to specify fewer vertices for the same number of triangles. (For more information, see Chapter 3.)

If you manage to construct your models with fewer vertices, this will require that fewer vertices are fetched and processed in the vertex shader. This can improve your performance if you are vertex-limited. However, you must also remember the importance of having as few draw calls as possible. So even if you use triangle strips, you should try to have them as long as possible by using, for example, degenerate triangles.

Using Indexed Drawing with `gl.drawElements()`

Especially when you have large models with many shared vertices, you should try to use indexed drawing with `gl.drawElements()` instead of `gl.drawArrays()`. The indexed drawing lets you avoid duplicating the same vertex data. In addition, the post-transform vertex cache normally only works with indexed drawing.

Using Interleaved Vertex Data

If you interleave your vertex data in a single vertex array instead of having separate vertex arrays for different attributes, you typically get better performance due to better memory locality for the vertex data. For example, when the vertex position is fetched into the pre-transform vertex cache, it is likely that the vertex normal for the same vertex is also fetched into the pre-transform vertex cache and is available for the vertex shader when it is needed.

Using Level of Detail to Simplify Your Models

If you are vertex-limited but don't want to reduce the complexity of all your 3D models, you can use a technique called *level of detail* (LOD). This means that you reduce the complexity of your models when they move away from the viewer. In practice, you can typically have different versions of your models that you can change in run time. Of course, you should remember that implementing LOD logic in JavaScript typically increases the work that the CPU has to do, so there is a risk that you will move the bottleneck from the GPU to the CPU if you are not careful.

Avoiding Duplication of Constant Data in Vertex Arrays

In some cases, you may have models that include data that has the same value for all vertices. In theory you could, of course, duplicate the constant data in a vertex array where you let the value be the same for each vertex.

A better strategy is to use a constant vertex attribute or a uniform for this data. You use a constant vertex attribute by disabling the generic attribute index that corresponds to the attribute in the vertex shader with a call to the method `gl.disableVertexAttribArray()`. Then you can set the value of the constant vertex attribute by calling, for example, the following method:

```
gl.vertexAttrib4f(index, r, g, b, a);
```

This method lets you set the four values `r`, `g`, `b`, and `a` as float values. There are corresponding methods to set three, two, or one float values as well.

You have set the value for a uniform variable in many of the examples in this book. For example, if the uniform consists of four float values, you can use the following method to set it:

```
gl.uniform4f(location, r, g, b, a);
```

Performance Advice for Pixel-Limited WebGL

The following sections describe a few techniques that you can try if you are pixel-limited. Note that this book also includes texture-limited in this category.

Stretching Your Canvas

If you have a fragment shader that is doing a lot of heavy work and you suspect that it contains a bottleneck, you can let the fragment shader in the GPU work with a smaller area, then let the compositor in the browser scale to a larger area. This is possible since there are two different ways to specify width and height for the canvas and these two ways have different meanings.

First you can specify the `width` and `height` attributes of the `canvas` element as you have done in all of the examples in this book:

```
<canvas id="myGLCanvas" width="500" height="500"></canvas>
```

This specifies that the area that is used for rendering in the GPU is 500×500 pixels.

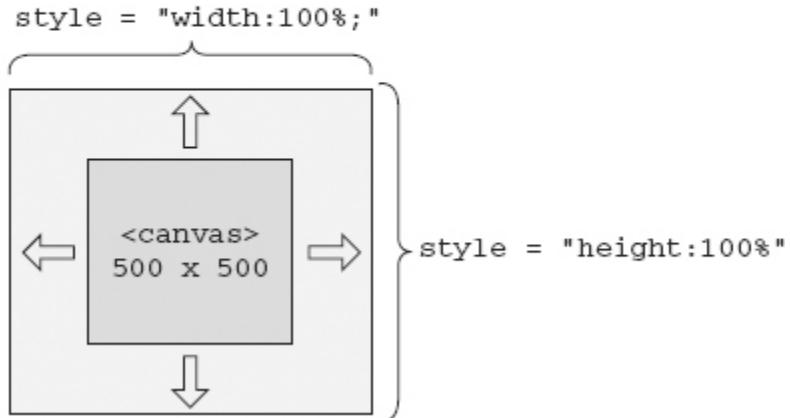
You can also specify the `width` and `height` as a `style` attribute or with an external CSS file. If you combine the `width` and `height` attributes for the `canvas` with setting the `width` and `height` as a `style` attribute, it can look like this:

```
<canvas id="myGLCanvas" width="500" height="500"  
style="width:100%;height:100%"></canvas>
```

In this code the size of the canvas that you render to is still set to 500×500 pixels, but the size that is displayed is set to 100 percent of the containing box. Assuming that the browser window is the containing box and that it is larger than 500×500 pixels, the canvas will be scaled to this larger size. [Figure 8-5](#) shows a simple illustration of this technique. The inner rectangle represents the size of the canvas and the outer box is the containing box that the compositor in the browser scales the canvas to. The more heavy work you are doing in your fragment shader, the more efficient this technique is.

FIGURE 8-5: Stretching the canvas for better performance when pixel-limited

```
<canvas id="myGLCanvas" width="500" height="500"  
style="width:100%; height:100%"></canvas>
```



Moving Calculations to the Vertex Shader

If you suspect that you have a bottleneck in the fragment shader, you could move some of the calculations that can be linearized to the vertex shader and then send the result to the fragment shader as a varying variable. For example, in Chapter 7 you learned how you could do the same lighting calculations in either the vertex shader or the fragment shader.

In general, you can increase performance when you perform calculations at an earlier stage of the WebGL pipeline. The reason for this is that doing a calculation early often means that you have to do the calculation fewer times. The number of vertices is generally less than the number of fragments, so this means that you have to do the calculation fewer times if you do it in the vertex shader. However, the improved performance is based on the fact that you are not vertex-limited.

Using Mipmapping for Your Texturing

It is usually a good idea to use mipmapping for your textures. When minification occurs and you use mipmapping, you generally get better cache utilization since texels are fetched closer in memory than if you don't use mipmapping. See Chapter 5 for a thorough discussion of mipmapping and texture filtering.

A CLOSER LOOK AT BLENDING

Chapter 1 gave you a brief overview of what happens in the WebGL graphics pipeline after the fragments leave the fragment shader. You also learned that a common name for these operations is “Per-Fragment Operations.” As a reminder, the per-fragment operations are as follows:

- Scissor test
- Multisample fragment operations
- Stencil test
- Depth buffer test
- Blending
- Dithering

In the following sections, you will learn more about blending, which together with depth testing is one of the most important and useful per-fragment operations.

Introducing Blending

Blending is a technique for combining two colors. Normally (that is, without blending), fragments that pass the depth test of the WebGL graphics pipeline are written directly to the drawing buffer and replace the corresponding pixel in the drawing buffer. (As you probably remember from Chapter 1, the drawing buffer can be seen as a WebGL framebuffer that is composited with the HTML page before being displayed.) With blending you can combine the color that is present in the drawing buffer at a certain fragment’s position with the color of the incoming fragment. You can enable blending in WebGL with the following call:

```
gl.enable(gl.BLEND);
```

The color of the incoming fragment is called the *source color* and the color of the fragment that is present in the drawing buffer is called the *destination color*.

The combination of the colors happens according to the blending equation, which is highly customizable. The blending equation can be written in a

generic form as follows:

$$\text{color}_{\text{final}} = \text{factor}_{\text{source}} \times \text{color}_{\text{source}} \text{ op factor}_{\text{dest}} \times \text{color}_{\text{dest}}$$

The different components of the equation have the following meanings:

- **color_{final}** is the final color that a fragment has after the blending operation is performed.
- **factor_{source}** is a scaling factor that is multiplied with the color of the incoming fragment.
- **color_{source}** is the color of the incoming fragment.
- **op** is the mathematical operator that is used to combine the color of the incoming fragment with the color of the destination fragment after they have been multiplied with the scaling factors. The default operator for WebGL is addition.
- **factor_{dest}** is a scaling factor that is multiplied with the color of the destination fragment.
- **color_{dest}** is the color of the destination fragment.

Setting the Blending Functions

The scaling factors in the generic blending equation, factor_{source} and factor_{dest}, are specified by calling one of the WebGL methods, `gl.blendFunc()` and `gl.blendFuncSeparate()`. Both methods are part of the `WebGLRenderingContext` and they have the following definitions:

```
void blendFunc(GLenum sfactor, GLenum dfactor);

void blendFuncSeparate(GLenum srcRGB, GLenum dstRGB,
                      GLenum srcAlpha, GLenum dstAlpha);
```

The arguments of these methods are used to set what is referred to as the *blending function*, and the possible values for the arguments are shown in the first column of [Table 8-1](#). The details of this table will be explained later in this chapter.

TABLE 8-1: Blending Functions

FUNCTION	RGB BLEND FACTORS	ALPHA BLEND FACTOR
<code>gl.ZERO</code>	(0, 0, 0)	0
<code>gl.ONE</code>	(1, 1, 1)	1
<code>gl.SRC_COLOR</code>	(R _s , G _s , B _s)	A _s
<code>gl.ONE_MINUS_SRC_COLOR</code>	(1, 1, 1) - (R _s , G _s , B _s)	1 - A _s

FUNCTION	RGB BLEND FACTORS	ALPHA BLEND FACTOR
gl.DST_COLOR	(R _d , G _d , B _d)	A _d
gl.ONE_MINUS_DST_COLOR	(1, 1, 1) - (R _d , G _d , B _d)	1 - A _d
gl.SRC_ALPHA	(A _s , A _s , A _s)	A _s
gl.ONE_MINUS_SRC_ALPHA	(1, 1, 1) - (A _s , A _s , A _s)	1 - A _s
gl.DST_ALPHA	(A _d , A _d , A _d)	A _d
gl.ONE_MINUS_DST_ALPHA	(1, 1, 1) - (A _d , A _d , A _d)	1 - A _d
gl.CONSTANT_COLOR	(R _c , G _c , B _c)	A _c
gl.ONE_MINUS_CONSTANT_COLOR	(1, 1, 1) - (R _c , G _c , B _c)	1 - A _c
gl.CONSTANT_ALPHA	(A _c , A _c , A _c)	A _c
gl.ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1) - (A _c , A _c , A _c)	1 - A _c
gl.SRC_ALPHA_SATURATE	(f, f, f)	1

The simplest alternative is to use the method `gl.blendFunc()`, which only takes two arguments. The argument `sfactor` specifies the source blending function for both the RGB and alpha values, and the argument `dfactor` specifies the destination blending function for both the RGB and alpha values. A common way to call `gl.blendFunc()` is as follows:

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

If you recall, the generic blending equation that was presented in the previous section is as follows:

$$\text{color}_{\text{final}} = \text{factor}_{\text{source}} \times \text{color}_{\text{source}} + \text{factor}_{\text{dest}} \times \text{color}_{\text{dest}}$$

Assume that the default operator (addition) is used in the generic blending equation. When `gl.blendFunc()` is called to set the `factorsource` to `gl.SRC_ALPHA` and the `factordest` to `gl.ONE_MINUS_SRC_ALPHA`, you can write the blending equation in a more specific way:

$$\text{color}_{\text{final}} = \alpha_{\text{source}} \times \text{color}_{\text{source}} + (1 - \alpha_{\text{source}}) \times \text{color}_{\text{dest}}$$

This is a common way to write the blending equation and it is often used to implement semi-transparency in WebGL as well as in many other types of computer graphics. The α_{source} is used to represent the transparency (or actually the opacity, since $\alpha = 0.0$ means fully transparent and $\alpha = 1.0$ means fully opaque) of the semi-transparent object. To get a better understanding of the equation, you can look at how the following source fragments affect the resulting $\text{color}_{\text{final}}$:

- An opaque fragment has $\alpha_{\text{source}} = 1.0$, which gives $\text{color}_{\text{final}} = \text{color}_{\text{source}}$ (the source fragment overwrites the destination fragment).

- A fully transparent fragment has $\alpha_{\text{source}} = 0.0$, which gives $\text{color}_{\text{final}} = \text{color}_{\text{dest}}$ (the source fragment does not contribute to the final color; the final color is equal to the color of the destination fragment).
- A 50-percent semi-transparent fragment has $\alpha_{\text{source}} = 0.5$, which gives $\text{color}_{\text{final}} = 0.5 \times \text{color}_{\text{source}} + 0.5 \times \text{color}_{\text{dest}}$ (the final color is a mix between the source and the destination color).



The default blending functions are `gl.ONE` for the source RGB and alpha functions, and `gl.ZERO` for the destination RGB and alpha functions. This means that to actually enable blending, it is not enough to just call:

```
gl.enable(gl.BLEND);
```

You must also call either `gl.blendFunc()` or `gl.blendFuncSeparate()` to set up the blending functions to something useful before you can see any effect of the blending.

Now you should have a look at the slightly more advanced method `gl.blendFuncSeparate()`, which is shown here again:

```
void blendFuncSeparate(GLenum srcRGB, GLenum dstRGB,
                      GLenum srcAlpha, GLenum dstAlpha);
```

In some cases it can be useful to specify a separate blending function for the RGB and alpha values of the source and destination. The method `gl.blendFuncSeparate()` has four arguments so you can set separate blending functions for the RGB and alpha values of the source and destination color. The arguments `srcRGB` and `dstRGB` specify the source and destination RGB blend functions, respectively. The arguments `srcAlpha` and `dstAlpha` specify the source and destination alpha blend functions. This is an example of how the method `gl.blendFuncSeparate()` can be called:

```
gl.blendFuncSeparate(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA,
                     gl.ZERO, gl.ONE);
```

Calling the `gl.blendFuncSeparate()` method with the arguments shown here sets up the blending so that the RGB values are blended in the same way as for the example with `gl.blendFunc()` that was discussed previously. However, the way that the blend functions are set up for the alpha values will result in the destination alpha not being affected by the blending. This can be useful if, for example, you use the destination alpha to store some information other than the transparency.

Now you should have a closer look at [Table 8-1](#) even though I can immediately reveal that several of the blending functions are not often used. As already mentioned, the first column contains the blending functions that you can use as arguments to `gl.blendFunc()` and `gl.blendFuncSeparate()`. The second and third columns specify which scaling factors (or blend factors) are used for a specific blending function. In the table (R_s , G_s , B_s , A_s) are the color components for the source (or incoming) fragments, while (R_d , G_d , B_d , A_d) are the color components for the destination fragments. In addition, you can see that the table contains (R_c , G_c , B_c , A_c), which represent constant color components that you can set by calling the method:

```
void blendColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

The arguments `red`, `green`, `blue`, and `alpha` specify the component values (R_c , G_c , B_c , A_c) for the constant blending color.



When discussing the use of constant color, it should be mentioned that WebGL imposes a limitation regarding how the constant blending functions can be combined as arguments to `gl.blendFunc()` and `gl.blendFuncSeparate()`. Constant color and constant alpha cannot be used together as source and destination factors in the blend function. For example, if you call `gl.blendFunc()` with one of the two factors set to `gl.CONSTANT_COLOR` or `gl.ONE_MINUS_CONSTANT_COLOR` and the other to `gl.CONSTANT_ALPHA` or `gl.ONE_MINUS_CONSTANT_ALPHA`, a `gl.INVALID_OPERATION` error is generated.

The last row in [Table 8-1](#) contains the blending function `gl.SRC_ALPHA_SATURATE`, which is special because it is only allowed as source RGB and alpha. The f in the second column of the last row represents the function:

$$f = \min(A_s, 1 - A_d)$$

Understanding Drawing Order and the Depth Buffer

As mentioned in the previous section, blending can be used to implement the effect of semi-transparent objects by setting your blending functions with the

following call:

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

However, for this to work, the semi-transparent objects need to be rendered in back-to-front order (that is, farthest to nearest). The reason is that the blending equation you set up with `gl.blendFunc()` is order-dependent. A special case is when you only have two semi-transparent surfaces that overlap and the alpha of both is 0.5. In this case, the drawing order does not matter.

Another important thing to remember when you use blending is how the depth buffer works. If you draw a semi-transparent surface and then draw another surface behind it with depth testing enabled, the second surface does not appear. This is because the fragments for the second surface are removed in the depth test. So, in general, you want to draw opaque objects before any semi-transparent objects in your scene. You will learn more about this in the next section.

Drawing a Scene with Both Opaque and Semi-Transparent Objects

If you have a scene with both opaque and semi-transparent objects, you typically want to have the depth testing enabled to remove objects that are behind the opaque objects. If you have semi-transparent objects that are closer to the viewpoint than the opaque objects, you want the semi-transparent objects to be blended with the opaque objects.

You have learned that for semi-transparent objects to blend correctly, you need to draw them in back-to-front order. However, it can sometimes be difficult to sort all semi-transparent objects in correct back-to-front order. This is especially true when you have objects that have interpenetrating surfaces. In this case, it is not guaranteed that you will get all surfaces correct, even if you sort the objects after, for example, the centroid of the object.

If you want to make sure that all surfaces are at least visible even when you cannot make sure that every surface in the scene is sorted correctly, it can help to make the depth buffer read-only, but still keep the depth test enabled. You can do this in WebGL with the following call:

```
gl.depthMask(false);
```

After this call, the depth testing is still performed, which means that fragments are discarded if the value in the depth buffer shows that they should be obscured by the pixel that is currently in the drawing buffer. The important difference is that since the depth buffer is read-only, it is not updated with new values. This way, the depth buffer can remember the values it has when you disable writing to it by calling `gl.depthMask(false)`.

If you draw your semi-transparent objects with depth testing enabled, but with the depth buffer set to read-only, you can at least be sure that all your semi-transparent objects are visible and none are discarded because another semi-transparent surface is in front of them.

To enable writing to the depth buffer again, you just call `gl.depthMask()` again with the argument set to `true`.

As a summary, the following code snippet with comments included gives you a general idea of how to render a scene with both opaque and semi-transparent objects:

```
// 1. Enable depth testing, make sure the depth buffer is  
//      writable  
//      and disable blending before you draw your opaque  
//      objects.  
gl.enable(gl.DEPTH_TEST);  
gl.depthMask(true);  
gl.disable(gl.BLEND);  
  
// 2. Draw your opaque objects in any order (preferably  
//      sorted on state)  
  
// 3. Keep depth testing enabled, but make depth buffer  
//      read-only  
//      and enable blending  
gl.depthMask(false);  
gl.enable(gl.BLEND);  
  
// 4. Draw your semi-transparent objects back-to-front  
  
// 5. If you have UI that you want to draw on top of your  
//      regular scene, you can finally disable depth testing  
gl.disable(gl.DEPTH_TEST);
```

```
// 6. Draw any UI you want to be on top of everything else.
```

Changing the Default Operator for the Blend Equation

Have another look at the generic blending equation:

$$\text{color}_{\text{final}} = \text{factor}_{\text{source}} \times \text{color}_{\text{source}} \text{ op } \text{factor}_{\text{dest}} \times \text{color}_{\text{dest}}$$

You have already learned how this equation can be configured by using the methods `gl.blendFunc()` and `gl.blendFuncSeparate()` to set $\text{factor}_{\text{source}}$ and $\text{factor}_{\text{dest}}$. You also learned that the default mathematical operator for combining the source and the destination is addition. If you want to change this, you can use the methods `gl.blendEquation()` and `gl.blendEquationSeparate()`, which have the following definitions:

```
void blendEquation(GLenum mode);  
  
void blendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);
```

You can use the following three values as arguments to these methods:

- `gl.FUNC_ADD` adds the source operand to the destination operand. This is the default value.
- `gl.FUNC_SUBTRACT` subtracts the destination operand from the source operand.
- `gl.FUNC_REVERSE_SUBTRACT` subtracts the source operand from the destination.

The generic blending equation looks like this in the three cases:

$$\text{color}_{\text{final}} = \text{factor}_{\text{source}} \times \text{color}_{\text{source}} + \text{factor}_{\text{dest}} \times \text{color}_{\text{dest}}$$

$$\text{color}_{\text{final}} = \text{factor}_{\text{source}} \times \text{color}_{\text{source}} - \text{factor}_{\text{dest}} \times \text{color}_{\text{dest}}$$

$$\text{color}_{\text{final}} = \text{factor}_{\text{dest}} \times \text{color}_{\text{dest}} - \text{factor}_{\text{source}} \times \text{color}_{\text{source}}$$

Using Premultiplied Alpha

To perform blending based on the alpha channel, you obviously need to make sure your color contains the alpha channel. If the color comes from a vertex

attribute, all four components of the color must be specified. If it comes from a texture, the texture format must contain an alpha channel. A PNG image is an example of a format that supports an alpha channel.

There are two ways that the alpha channel can be used in computer graphics:

- Premultiplied alpha (also called associated alpha)
- Non-premultiplied alpha (also called unassociated alpha)

Premultiplied alpha means that the RGB value is multiplied with the alpha channel before it is stored. This means that a half-opaque red color is represented by the RGBA value (0.5, 0.0, 0.0, 0.5).

Non-premultiplied alpha means that the RGB value is not multiplied with the alpha channel before it is stored. With this technique, a half-opaque red color is represented by the RGBA value (1.0, 0.0, 0.0, 0.5). PNG images normally use non-premultiplied alpha.

When working with WebGL and blending, it is important to plan ahead if you are going to use premultiplied alpha or non-premultiplied alpha. Even though PNG images use non-premultiplied alpha, you can specify that you want to multiply the RGB values with the alpha value when the image is loaded as a texture with `gl.texImage2D()` or `gl.texSubImage2D()`. You specify this by making the following call to the WebGL API:

```
gl.pixelStorei(gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL, true);
```

Note that if you have premultiplied alpha, the default blending equation that is commonly used for semi-transparency is no longer written as follows:

$$\text{color}_{\text{final}} = \alpha_{\text{source}} \times \text{color}_{\text{source}} + (1 - \alpha_{\text{source}}) \times \text{color}_{\text{dest}}$$

Since the alpha value has already been multiplied with the RGB value, the corresponding equation for premultiplied alpha values is as follows:

$$\text{color}_{\text{final}} = \text{colorPreMult}_{\text{source}} + (1 - \alpha_{\text{source}}) \times \text{color}_{\text{dest}}$$

where `colorPreMultsource` is the premultiplied source color. This equation is set up with the following call:

```
gl.blendFunc(gl.ONE, gl.ONE_MINUS_SRC_ALPHA);
```

TAKING WEBGL FURTHER

Although you have learned a lot about WebGL in this book, as with most interesting technologies, there is always much more to learn. The following section offers some suggestions of where you can find more information.

Using WebGL Frameworks

This book has focused on teaching you WebGL using only smaller JavaScript libraries such as `glMatrix`, to ensure that you concentrated on learning WebGL. The advantage to this approach is that you have very low-level control over the GPU as well as over how you want to perform operations. In addition, since WebGL is an open standard that is very similar to OpenGL and OpenGL ES, it will be easy for you to reuse ideas or code from these technologies in your WebGL applications. There are a lot of books, technical articles, and other documents that describe OpenGL, and you can use many of these to learn more about WebGL as well.

However, if you don't like the fact that WebGL is a low-level API and want to try something more high-level to create your applications, there are many higher-level libraries or frameworks that are built on top of WebGL, including the following:

- GLGE
- PhiloGL
- SceneJS
- SpiderGL
- Three.js

The aim with these frameworks is to speed up application development by abstracting away some of the details in the WebGL API.

The easiest way to find these frameworks is by doing a search on Google. You can also look at the Khronos WebGL wiki, which has a section with user contributions, where several different WebGL frameworks are described.

Publishing to the Google Chrome Web Store

The Google Chrome Web Store lets you publish your web applications so Google Chrome users can easily find them. If you are developing some new

WebGL-enabled applications, you might want to publish them in the Chrome Web Store.

In addition to the fact that users can find your application more easily, the Chrome Web Store also allows you to charge for your application. The online documentation for the Chrome Web Store is well written, and there are also introductory videos available, so it is not difficult to get started.

Using Additional Resources

An excellent way to learn more about WebGL is, of course, to develop your own WebGL application. You can also learn a lot by looking at other people's applications. Chrome Developer Tools and Firebug make this easy by enabling you to study the techniques and ideas other people have used for their WebGL applications.

Following is a list of excellent online resources that contain a lot of valuable information about WebGL:

- The Khronos main page for WebGL (www.khronos.org/webgl/) contains many useful links to WebGL-related information. From here, you can find links to:
 - The latest WebGL specification
 - The Khronos WebGL wiki
 - Several excellent WebGL demos
 - WebGL forums
- Giles Thomas's website (<http://learningwebgl.com/blog/>) contains several different tutorials on WebGL, as well as news on what is happening with WebGL around the web.
- Brandon Jones's blog (<http://blog.tojicode.com/>) contains several articles on WebGL as well as what to consider when making a game with WebGL.
- Developer conference presentations by Gregg Tavares, Kenneth Russell, Ben Vanik, and others. The easiest way to find these presentations is to search for them on Google. Most of them are also linked from the Khronos WebGL wiki.
- WebGL tutorials from Mozilla (<https://developer.mozilla.org/en/WebGL>).

- Chrome Experiments (www.chromeexperiments.com) contains many interesting WebGL demos from which you can learn new techniques. Note that not all demos are based on WebGL.
- HTML5 Rocks (www.html5rocks.com) contains many useful resources for HTML5. This is a great place to look when you want to combine WebGL with other HTML5 features. Even though WebGL is not part of the W3C HTML5 specification, the site contains some material about WebGL as well.

This list contains several resources that can be useful on your journey towards being a real WebGL expert. However, things change quickly, so to keep up-to-date, it's a good idea to perform your own web searches for WebGL to access new and interesting resources.

SUMMARY

This chapter offered a high-level view of what a WebGL implementation can look like under the hood. You also learned about some of the most important hardware and software components.

The major, and most important, part of the chapter covered WebGL performance optimizations. You learned how to perform different tests and how to draw conclusions from these tests on where in the system a bottleneck is located. The chapter contained both general performance advice for WebGL and specific advice that you can follow when you know the location of your current performance bottleneck.

Finally, you learned about blending, as well as what to consider when you need to draw a scene with both opaque and semi-transparent objects.



PROFESSIONAL

WebGL™ Programming

DEVELOPING 3D GRAPHICS FOR THE WEB

Andreas Anyuru



John Wiley & Sons, Ltd.

This edition first published 2012

© 2012 Andreas Anyuru

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex,
PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for
information about how to apply for permission to reuse the copyright
material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been
asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in
a retrieval system, or transmitted, in any form or by any means, electronic,
mechanical, photocopying, recording or otherwise, except as permitted by
the UK Copyright, Designs and Patents Act 1988, without the prior
permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some
content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often
claimed as trademarks. All brand names and product names used in this book
are trade names, service marks, trademarks or registered trademarks of their
respective owners. The publisher is not associated with any product or
vendor mentioned in this book. This publication is designed to provide
accurate and authoritative information in regard to the subject matter
covered. It is sold on the understanding that the publisher is not engaged in
rendering professional services. If professional advice or other expert
assistance is required, the services of a competent professional should be
sought.

Limit of Liability/Disclaimer of Warranty: The publisher and the author
make no representations or warranties with respect to the accuracy or
completeness of the contents of this work and specifically disclaim all
warranties, including without limitation warranties of fitness for a particular
purpose. No warranty may be created or extended by sales or promotional

materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Google Chrome is a trademark of Google Inc. Use of this trademark is subject to Google Permissions. All other trademarks are the property of their respective owners. John Wiley & Sons, Ltd., is not associated with any product or vendor mentioned in this book. The WebGL specification copyright is owned by the Khronos Group Inc. The OpenGL ES Man Pages copyright is owned by the Khronos Group Inc. and Silicon Graphics Inc. Khronos and WebGL are trademarks of the Khronos Group Inc. OpenGL is a registered trademark and the OpenGL ES logo is a trademark of Silicon Graphics International used under license by Khronos.

978-1-119-96886-3

978-1-119-94058-6 (ebk)

978-1-119-94059-3 (ebk)

978-1-119-94060-9 (ebk)

10 9 8 7 6 5 4 3 2 1

A catalogue record for this book is available from the British Library.

To Jessica, Tilde, and Victor

ABOUT THE AUTHOR

ANDREAS ANYURU has extensive experience with various web technologies, including WebGL. He has worked with web browser integration and optimizations for mobile devices for many years. Andreas is one of the contributors to the V8 JavaScript engine that is used in Google Chrome for desktops and in Android. He has developed new courses and taught at the Information and Communications Engineering division at Lund University.

Andreas is a Senior Member of Technical Staff at ST-Ericsson, where his primary area of expertise is web technologies. In this position, he is responsible for ensuring that existing and forthcoming web technologies work in an optimal way on ST-Ericsson's Linux-based mobile platforms.

Andreas holds a M.Sc. in Electrical Engineering from Lund University, Faculty of Engineering, Sweden.

ABOUT THE TECHNICAL EDITOR

PAUL BRUNT was working with JavaScript early on in the emergence of HTML5 to create some early games and applications that made extensive use of SVG, canvas, and a new generation of fast JavaScript engines. This work included a proof-of-concept platform game demonstration called “Berts Breakdown.” With a keen interest in computer art and an extensive knowledge of Blender, combined with knowledge of real-time graphics, the introduction of WebGL was the catalyst for the creation of GLGE. He began work on GLGE in 2009 when WebGL was still in its infancy, gearing it heavily towards the development of online games. Paul has also contributed to other WebGL frameworks and projects, as well as porting the jiglib physics library to javascript in 2010, demoing 3D physics within a browser for the first time.

CREDITS

VP CONSUMER AND TECHNOLOGY PUBLISHING DIRECTOR

Michelle Leete

ASSOCIATE DIRECTOR—BOOK CONTENT MANAGEMENT

Martin Tribe

ASSOCIATE PUBLISHER

Chris Webb

EXECUTIVE COMMISSIONING EDITOR

Birgit Gruber

ASSISTANT EDITOR

Ellie Scott

ASSOCIATE MARKETING DIRECTOR

Louise Breinholt

SENIOR MARKETING EXECUTIVE

Kate Parrett

EDITORIAL MANAGER

Jodi Jensen

SENIOR PROJECT EDITOR

Sara Shlaer

DEVELOPMENT EDITOR

Box Twelve Communications, Inc.

TECHNICAL EDITOR

Paul Brunt

COPY EDITOR

Marylouise Wiack

SENIOR PRODUCTION EDITOR

Debra Banninger

PROOFREADER

Nancy Carrasco

INDEXER

Robert Swanson

COVER DESIGNER

LeAndra Young

COVER IMAGE

© iStock / Mark Evans

ACKNOWLEDGMENTS

A BIG THANK YOU — goes to the Khronos WebGL Working Group, which created the WebGL specification and the browser vendors, including open source projects, which have implemented the specification. Without your work, this book would not exist. In addition, I want to thank all developers in the WebGL community who have contributed to WebGL as a technology with their discussions, experiments, and examples.

I want to thank the people at Wiley, especially Jeff Riley, Sara Shlaer, Ellie Scott, and Chris Webb. You have all done a great job supporting me.

I want to say thank you to all my great colleagues at ST-Ericsson — you make ST-Ericsson a great place to work. I especially want to thank Kent Olsson, Christian Bejram, Marco Cornero, Göran Roth, and Patrik Åberg for believing in me, supporting me, and coaching me.

I want to thank my family for their support and understanding while writing this book. Most of all, I want to say thanks to my wonderful wife Jessica, whom I love very much. Thanks for everything, Jessica!

INTRODUCTION

TODAY USERS SPEND A MAJORITY OF THEIR TIME WITH THEIR COMPUTERS, tablets, and smartphones surfing with web browsers. They are writing documents, e-mailing, chatting, reading news, watching videos, listening to music, playing games, or buying things. The web browser has become the most important application on most devices. As a result, the innovation and development of web technologies are evolving very fast. Web browsers are getting faster, more stable, more secure, and more capable of handling new technologies every day.

This means very exciting times for web developers. If you want to develop an application that you want to distribute to as many users as possible, the web is definitely the platform to choose. But even if web applications have evolved tremendously in recent years, one disadvantage of web applications compared to native applications has been that it has not been possible to create real-time 3D graphics that games and other applications might need. This has changed completely with WebGL.

WebGL lets you hardware-accelerate your 2D and 3D graphics in a way that has not been possible on the web before now. You can get all the advantages that building a web application in HTML, CSS and JavaScript provides and still take advantage of the powerful graphics processor unit (GPU) that devices have. This book will start with the basics and teach you everything you need to know to start developing 2D or 3D graphics applications based on WebGL.

WHO THIS BOOK IS FOR

This book is primarily for web developers who have a basic understanding of HTML, CSS and JavaScript and want to get a deeper understanding of how to use WebGL to create hardware-accelerated 2D or 3D graphics for their web applications or web pages.

The second group of developers who can benefit from this book is composed of those who have some previous experience from a desktop 3D API (such as OpenGL or Direct3D) and want to make use of this knowledge on the web by using WebGL.

The book is also useful for students who study a 3D graphics course and want to use WebGL as an easy way to prototype and test things. Since you need only a web browser that supports WebGL and a text editor to get started to write your applications, the threshold to get started is much lower than with other 3D graphics APIs (such as desktop OpenGL or Direct3D), where you often need to set up a complete toolchain.

WHAT THIS BOOK COVERS

This book will teach you how to develop web applications based on WebGL. Even though the WebGL API can be used to hardware-accelerate both 2D graphics and 3D graphics, it is primarily an API used to create 3D graphics. Some books about other 3D graphics APIs describe only the API; they do not really teach you much about 3D graphics or how you use the API. This book does not require any previous knowledge about 3D graphics. Instead you learn about 3D graphics and using the WebGL API to build applications.

In addition, this book contains some fundamental linear algebra that is good to understand to get a deeper understanding of 3D graphics and WebGL. By having the relevant linear algebra available in the same book, you can concentrate on the parts of linear algebra that are important for 3D graphics. You do not need to dive into other generic linear algebra books consisting of hundreds of pages and wherein topics often are described in a very general and abstract way. If you're the type of person who just wants to get started writing code, you don't have to read all the linear algebra sections (which is mainly part of Chapter 1). You can skim them, and when you later notice that there is something related to linear algebra that you want to understand, you can go back and read them more thoroughly.

You should note that this book does not include all available methods of the WebGL API. For a complete reference of the API, you should have a look at the latest WebGL specification from Khronos at www.khronos.org/registry/webgl/specs/latest/

HOW THIS BOOK IS STRUCTURED

This book is organized in a logical order whereby most readers will benefit from reading the chapters in the order they appear. However, since all readers have different backgrounds, feel free to choose an order that suits you. The following overview might help you in this planning.

Chapter 1 is a theoretical chapter that gives you some background of WebGL and compares WebGL to some other graphics technologies. It introduces you to the WebGL graphics pipeline and also contains an overview of graphics hardware. Further, the chapter contains some linear algebra that is important for 3D graphics and gives you a deeper understanding of 3D graphics and WebGL. If you feel that you have enough knowledge about the topics presented in this chapter, feel free to skim the content and continue with Chapter 2.

Chapter 2 provides a lot of practical knowledge. You create your first complete WebGL application and learn how to write a very basic vertex shader and a fragment shader. To make your journey towards learning WebGL as smooth as possible, this chapter introduces some useful development and debugging tools. You also learn what to think of when you troubleshoot your WebGL application.

Chapter 3 teaches you the details about which different options you have for drawing in WebGL. You learn about the different WebGL methods and primitives you can use to handle drawing.

Chapter 4 introduces you to three small JavaScript libraries that are useful to handle vector and matrix mathematics in your WebGL application. You also learn how to use transformations to position and orient the objects in your 3D world. This is a very important chapter, and if you have no previous experience of 3D graphics, you should be prepared to spend some extra time on it.

Chapter 5 walks through how you perform texturing in WebGL. Texturing is an important step towards making your 3D objects look more realistic. In

addition, this chapter describes how to make your applications more robust by handling what is referred to as *lost context* in WebGL. Make sure you read this part before you start to build real-world applications that have high requirements on robustness.

Chapter 6 teaches you how you use animation to create motion in your WebGL scenes. You will also learn the details of how event-handling in JavaScript works and how you can use key events and mouse events to affect your WebGL scene.

Chapter 7 teaches you a lot about the exciting topic of lighting in WebGL. You learn about different lighting models and how to write vertex shaders and fragment shaders for these. Lighting is a very important ingredient in making your WebGL scenes look more realistic.

Chapter 8 contains useful guidelines, tips, and tricks that make sure your WebGL application gets the best possible performance. You also learn how to analyze performance problems in a WebGL application, how to find the bottleneck, and how to try to eliminate it. This chapter also features a description of how WebGL works under the hood. You learn about the key software components and the key hardware components. An example from the mobile industry is used.

WHAT YOU NEED TO USE THIS BOOK

To run the examples in this book, you need a web browser that supports WebGL. If you don't have a browser that supports WebGL, you can download one for free. For a current list of which web browsers that support WebGL, visit www.khronos.org/webgl/wiki/. At the time of this writing, the following browsers support WebGL:

- Apple Safari
- Google Chrome
- Mozilla Firefox
- Opera

To write your own WebGL applications, simply use your favorite text editor. The book also describes some other tools that are helpful for debugging and trouble shooting of WebGL applications. These are, for example, Chrome Developer Tools, Firebug, and WebGL Inspector. These tools are all open source and can be downloaded for free.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.



The pencil icon indicates notes, tips, hints, tricks, or and asides to the current discussion.

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show file names, URLs, and code within the text like so:
`persistence.properties`.
- We present code in two different ways:

We use a monofont type with no highlighting for most code examples.

We use bold to emphasize code that's particularly important in the present context.

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code corresponding to the code listings in this book is available for download at www.wrox.com. You will find the code listings from the source code are accompanied by a download icon and listing number so you know it's available for download and can easily locate it in the download file. Once at the site, simply locate the book's title (either

by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.



Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-1-119-96886-3.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors.



A complete book list including links to each book's errata is also available at www.wrox.com/misic-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.



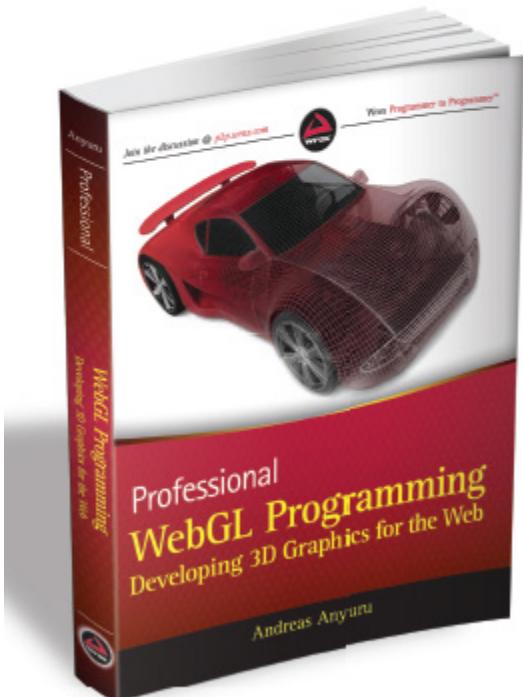
You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

Try Safari Books Online FREE for 15 days + 15% off for up to 12 Months*

Read thousands of books for free online with this 15-day trial offer.



Safari
Books Online

With Safari Books Online, you can experience searchable, unlimited access to thousands of technology, digital media and professional development books and videos from dozens of leading publishers. With one low monthly or yearly subscription price, you get:

- Access to hundreds of expert-led instructional videos on today's hottest topics
- Sample code to help accelerate a wide variety of software projects
- Robust organizing features including favorites, highlights, tags, notes, mash-ups and more
- Mobile access using any device with a browser
- Rough Cuts pre-published manuscripts

START YOUR FREE TRIAL TODAY!

Visit www.safaribooksonline.com/wrox35 to get started.

*Available to new subscribers only. Discount applies to the Safari Library and is valid for first 12 consecutive monthly billing cycles. Safari Library is not available in all countries.



An Imprint of **WILEY**
Now you know.

Visit www.safaribooksonline.com/wrox35 to get started



Programmer to Programmer™

Connect with Wrox.

Participate

Take an active role online by participating in our P2P forums @ p2p.wrox.com

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble

Join the Community

Sign up for our free monthly newsletter at newsletter.wrox.com

Wrox.com

Browse the vast selection of Wrox titles, e-books, and blogs and find exactly what you need

User Group Program

Become a member and take advantage of all the benefits

Wrox on twitter

Follow @wrox on Twitter and be in the know on the latest news in the world of Wrox

Wrox on facebook

Join the Wrox Facebook page at facebook.com/wroxpress and get updates on new books and publications as well as upcoming programmer conferences and user group events

Contact Us.

We love feedback! Have a book idea? Need community support?
Let us know by e-mailing wrox-partnerwithus@wrox.com

Related Wrox Books

Game and Graphics Programming for iOS and Android with OpenGL ES 2.0

ISBN: 978-1-119-97591-5

The smart phone app market is progressively growing, and there is a new market gap to fill that requires more graphically sophisticated applications and games. *Game and Graphics Programming for iOS and Android with OpenGL ES 2.0* quickly gets you up to speed on understanding how powerful OpenGL ES 2.0 technology is in creating apps and games for amusement and effectiveness. Leading you through the development of a real-world mobile app with live code, this text lets you work with all the best features and tools that OpenGL ES 2.0 has to offer.

HTML5 24-Hour Trainer

ISBN: 978-0-470-64782-0

HTML is the core technology for building websites. Today, with HTML5 opening the Internet to new levels of rich content and dynamic interactivity, developers are looking for information to learn and utilize HTML5. *HTML5 24-Hour Trainer* provides that information, giving new and aspiring web developers the knowledge they need to achieve early success when building websites.

Beginning HTML, XHTML, CSS, and JavaScript

ISBN: 978-0-470-54070-1

This beginner guide shows you how to use XHTML, CSS, and JavaScript to create compelling websites. While learning these technologies, you will discover coding practices such as writing code that works on multiple browsers including mobile devices, how to use AJAX frameworks to add interactivity to your pages, and how to ensure your pages meet accessible requirements.

Professional Augmented Reality Browsers for Smartphones: Programming for junaio, Layar and Wikitude

ISBN: 978-1-119-99281-3

Professional Augmented Reality Browsers for Smartphones guides you through creating your own augmented reality apps for the iPhone, Android, Symbian, and bada platforms, featuring fully workable and downloadable source code. You will learn important techniques through hands-on applications, and you will build on those skills as the book progresses.

Professional Papervision3D

ISBN: 978-0-470-74266-2

Professional Papervision3D describes how Papervision3D works and how real world applications are built, with a clear look at essential topics such as building websites and games, creating virtual tours, and Adobe's Flash 10. Readers learn important techniques through hands-on applications, and build on those skills as the book progresses. The companion website contains all code examples, video step-by-step explanations, and a collada repository.