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.

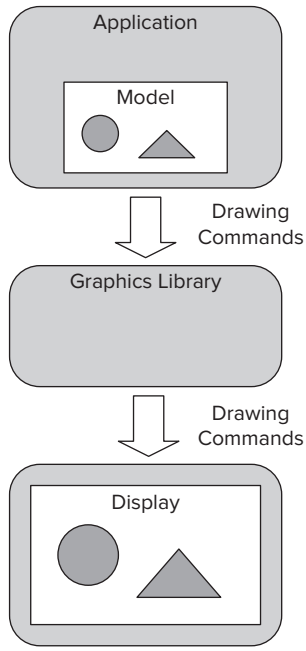

FIGURE 1-1: A diagram of how an immediate-mode API works

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.

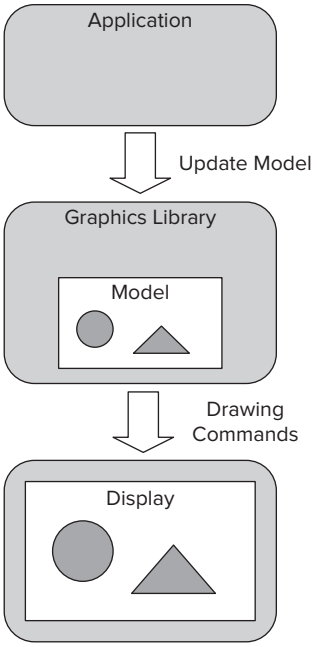

FIGURE 1-2: A diagram of how a retained-mode API works

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.

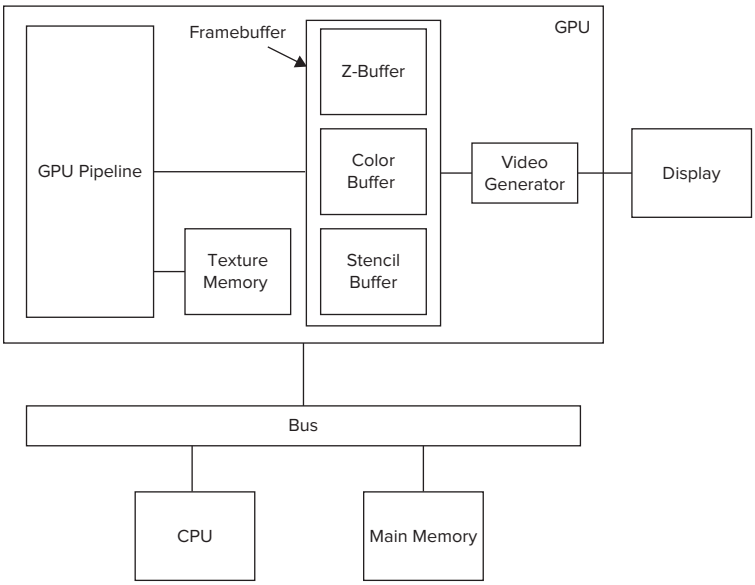

FIGURE 1-3: A simplified diagram of a graphics hardware and its relation to other hardware

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).

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.

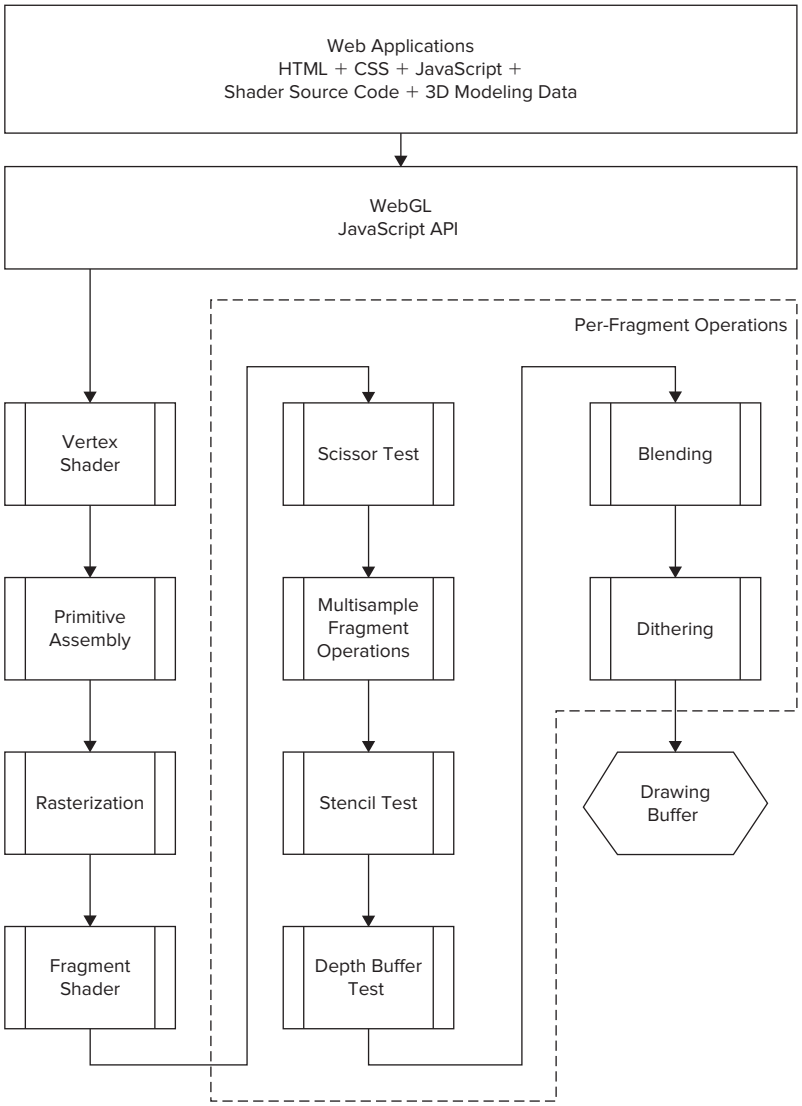

FIGURE 1-4: An overview of the WebGL graphics pipeline

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.

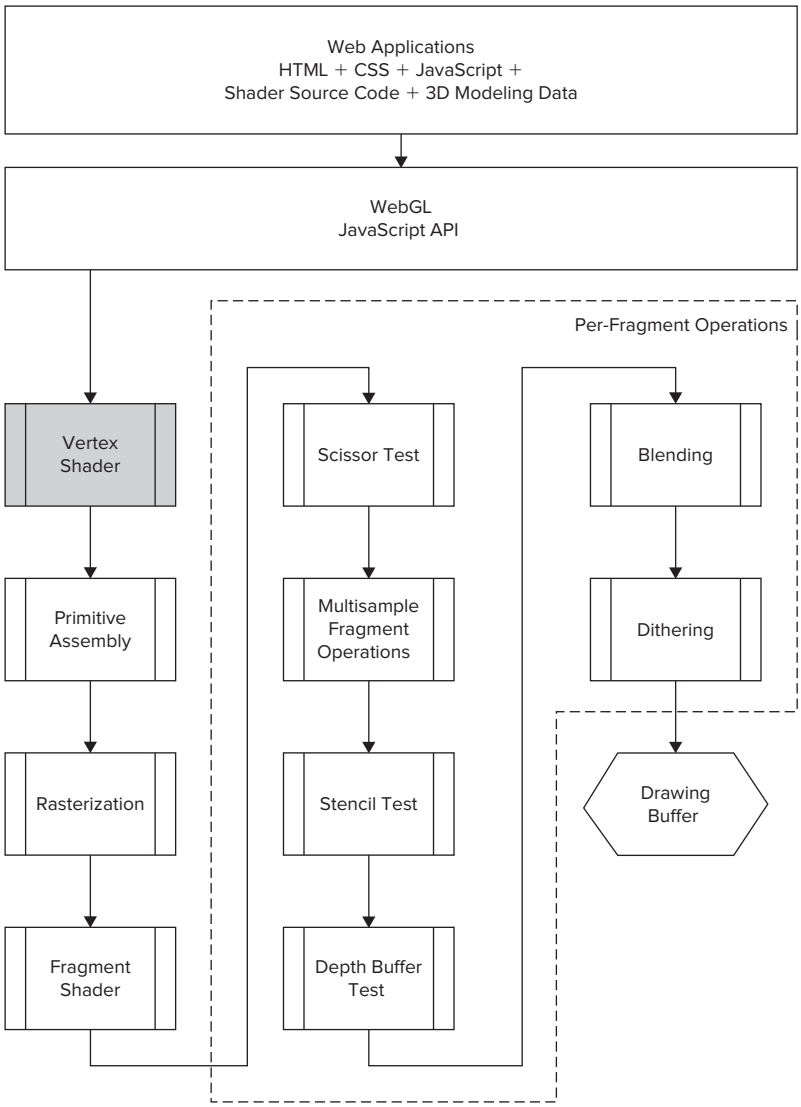

FIGURE 1-5: The location of the vertex shader in the WebGL graphics pipeline

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.

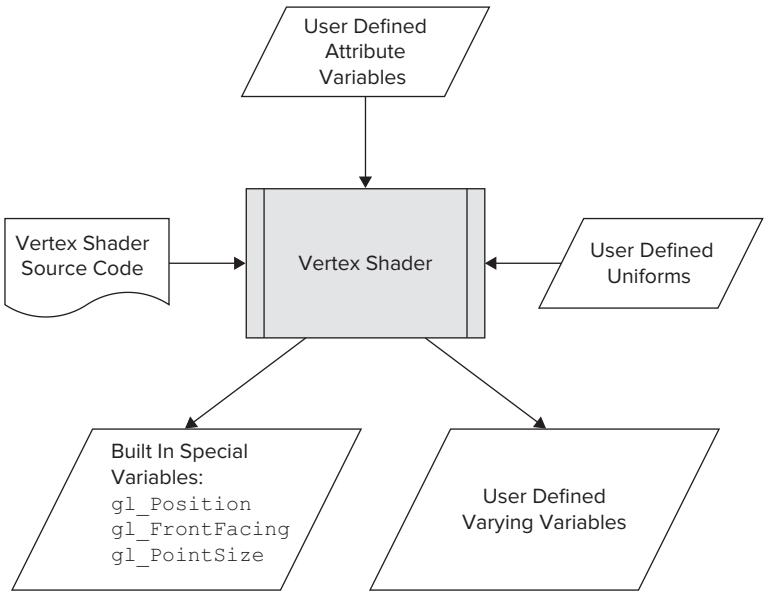

FIGURE 1-6: An overview of the vertex shader

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 uMVMatrix;
uniform mat4 uPMatrix;

varying vec4 vColor;

void main() {
  gl_Position = uPMatrix * uMVMatrix * 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 uMVMatrix;
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 * uMVMatrix * 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.

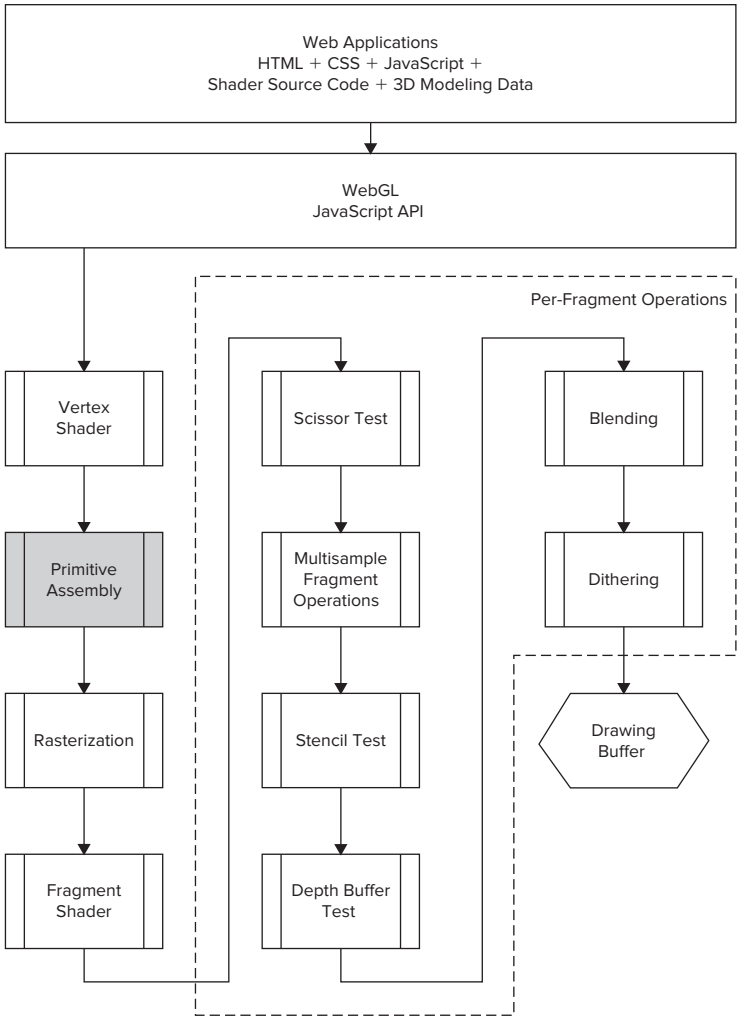

FIGURE 1-7: The location of the primitive assembly stage in the WebGL graphics pipeline

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.

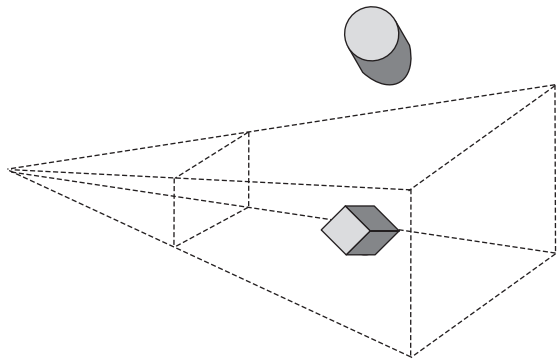

FIGURE 1-8: A view frustum with a cube that is inside the frustum and a cylinder that is outside the frustum

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).

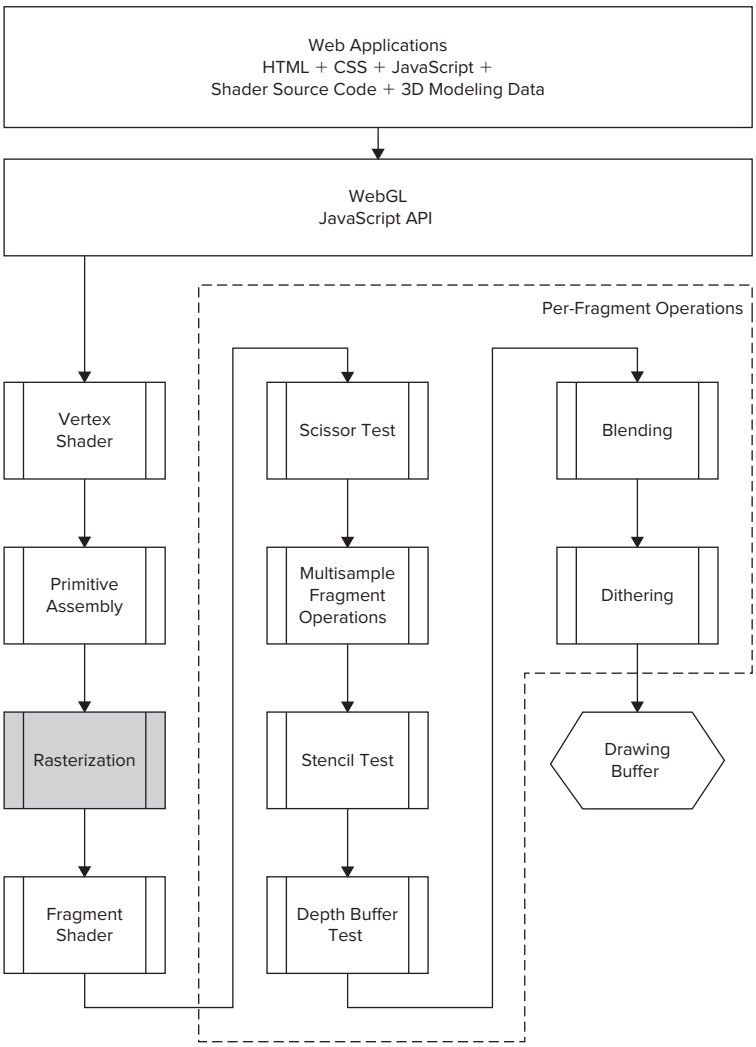

FIGURE 1-9: The location of rasterization in the WebGL graphics pipeline

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.

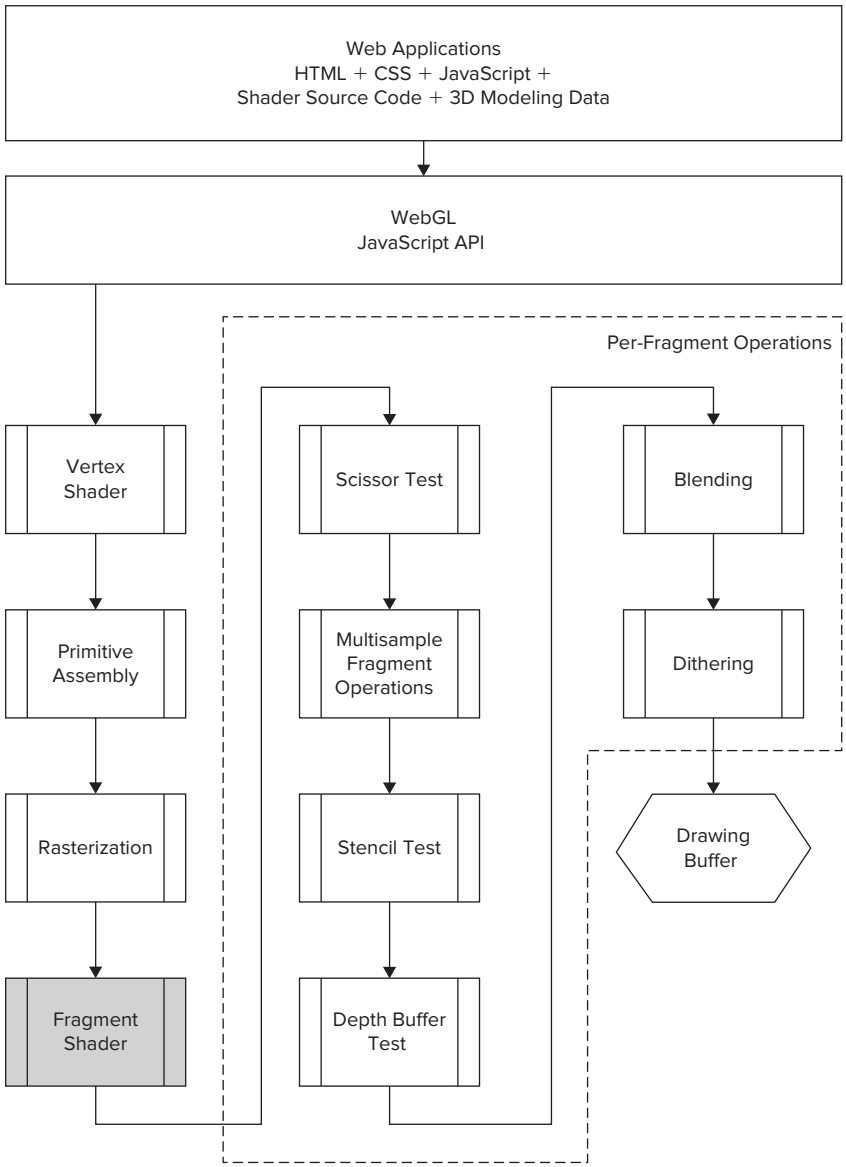

FIGURE 1-10: The Location of the fragment shader in the WebGL graphics pipeline

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.

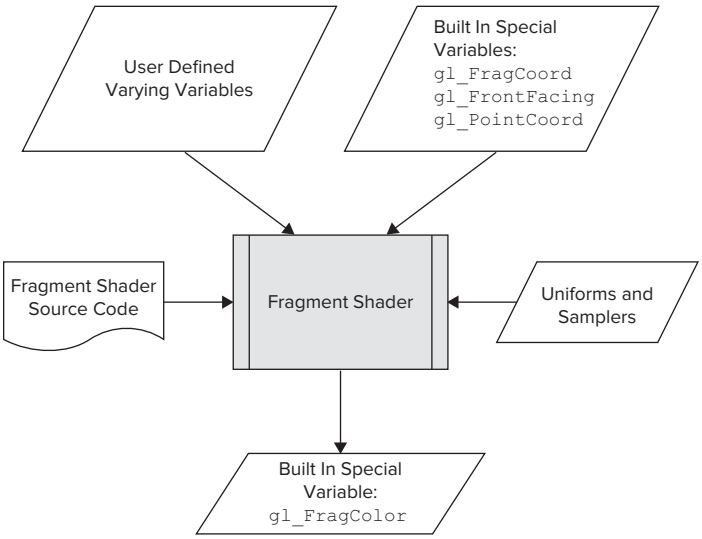

FIGURE 1-11: An overview of the fragment shader

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

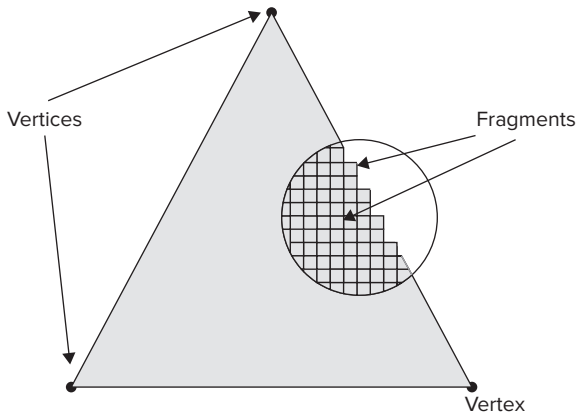

FIGURE 1-12: The relation between vertices and fragments

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.

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 out 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.

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.

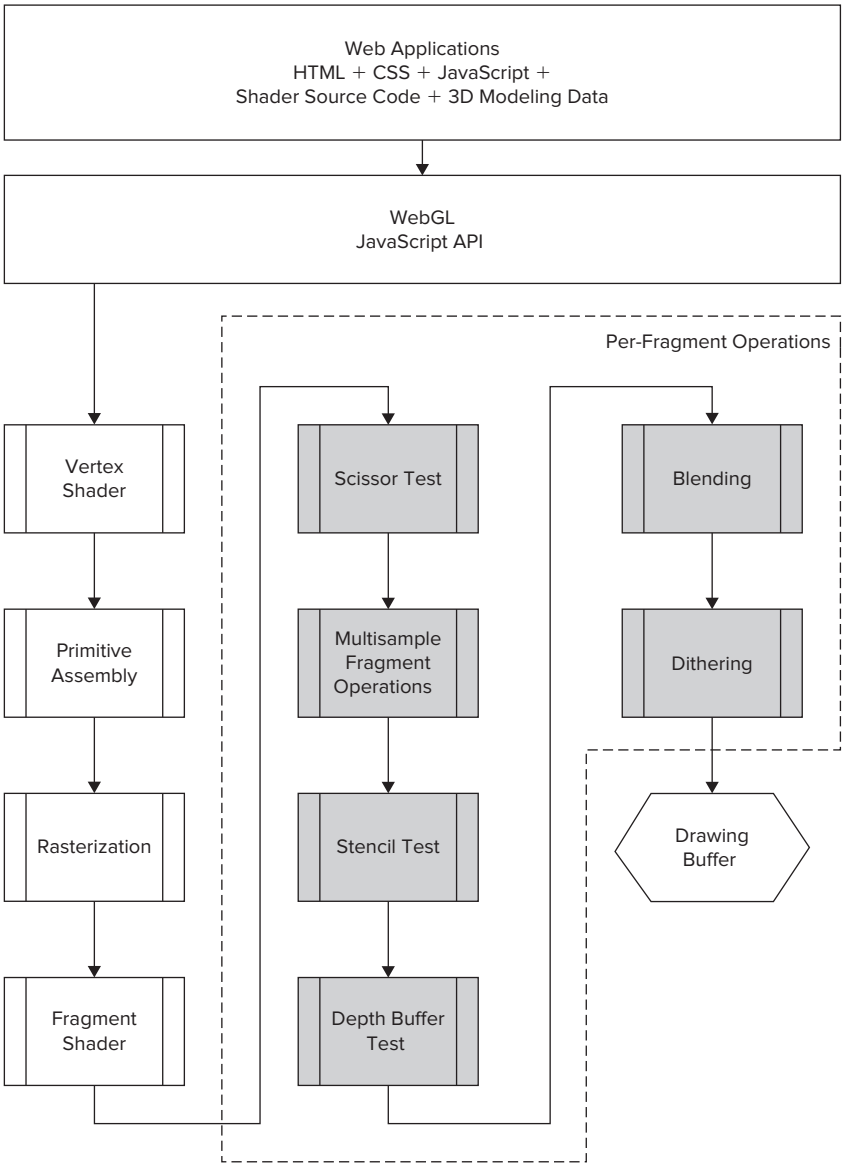

FIGURE 1-13: The location of the per-fragment operations in the WebGL graphics pipeline

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.

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.

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);

        // 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>
```

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.

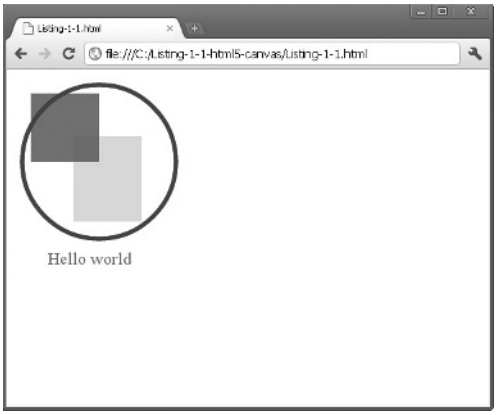

FIGURE 1-14: A simple drawing with HTML5 canvas

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.

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.

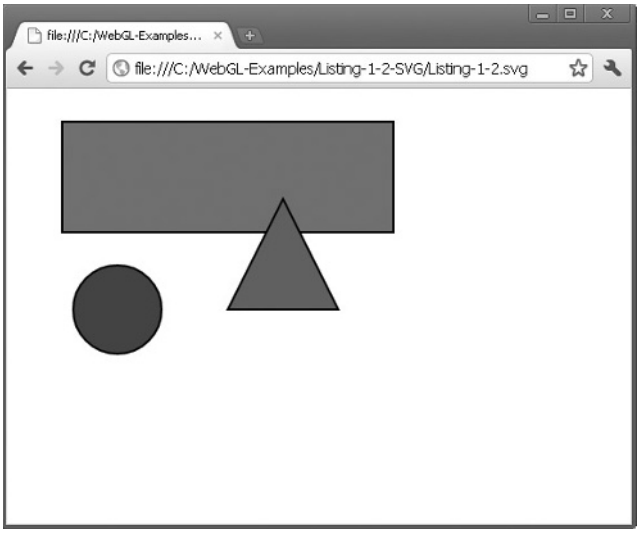

FIGURE 1-15: A simple drawing with SVG

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.

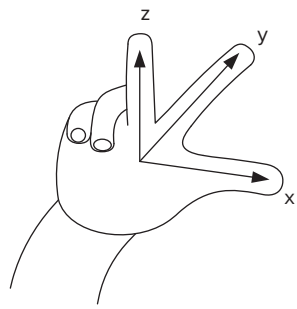

FIGURE 1-16: A way to visualize a right-handed coordinate system

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.

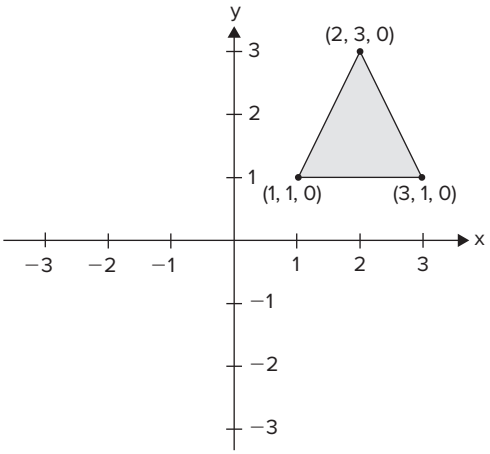

FIGURE 1-17: Three vertices defining a triangle

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 **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.

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 **u** that starts at the point **p** = (1, 1, 0) and ends at the point **q** = (3, 4, 0). You get this vector by subtracting the start point **p** from the end point **q** like this:

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

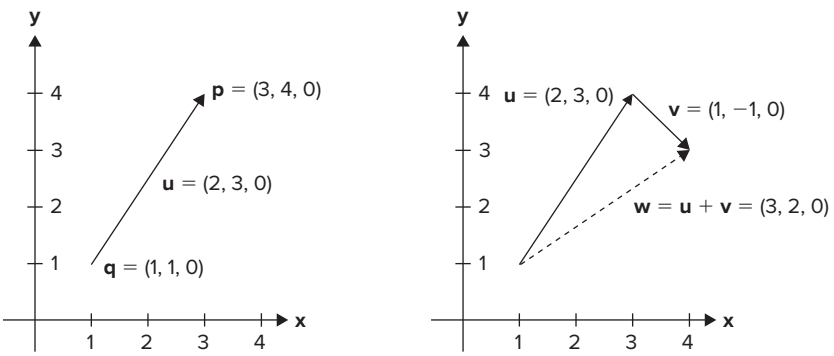

FIGURE 1-18: A Graphical visualization of a single vector to the left. To the right, two vectors are added with a new vector as the result

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 **u** and **v** can be graphically visualized as shown to the right in Figure 1-18. Draw the "tail" of vector **v** joined to the "nose" of vector **u**. The vector **w** = **u** + **v** is from the "tail" of **u** to the "nose" of **v**. In this example, you see how two vectors are added in 3D (but with z = 0). Adding **u** = (2, 3, 0) with **v** = (1, –1, 0) results in the third vector **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 **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 **u** and **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° 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.

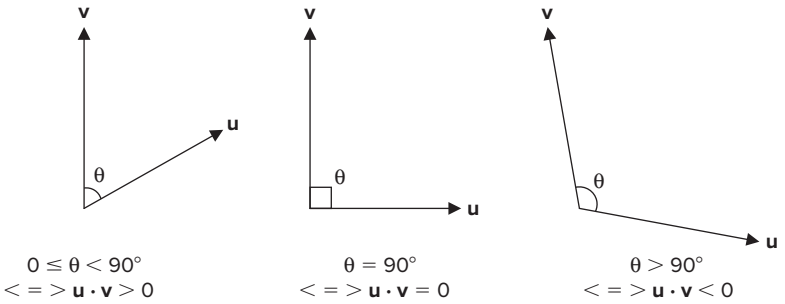

FIGURE 1-19: The geometry used for the dot product or scalar product

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:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}_x\mathbf{v}_x + \mathbf{u}_y\mathbf{v}_y + \mathbf{u}_z\mathbf{v}_z$$

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 **u** and **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 **w** with the following properties:

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

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

The second and third bullets specify the direction of the new vector **w**. It follows from the third bullet that the order of **u** and **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.

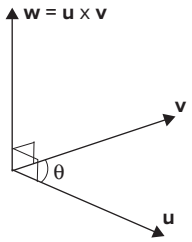

FIGURE 1-20: The geometry used in the cross product

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, $\mathbf{w} = 0$, and if $\mathbf{w} \neq 0$, 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:

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

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:

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

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:

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

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,

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

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

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

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:

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

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:

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

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

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

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

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

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 **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 **M** is multiplied by the identity matrix **I**, the result is always the matrix **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:

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

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 $\mathbf{M}^{\mathrm{T}}$. 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 $\mathbf{M}^{\mathrm{T}}$ is given by the following:

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

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:

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

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:

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

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:

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

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

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

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

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

Also, the second condition is met and the transform $\mathbf{f(u)} = 3\mathbf{u}$ 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 4×4 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:

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

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:

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

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).

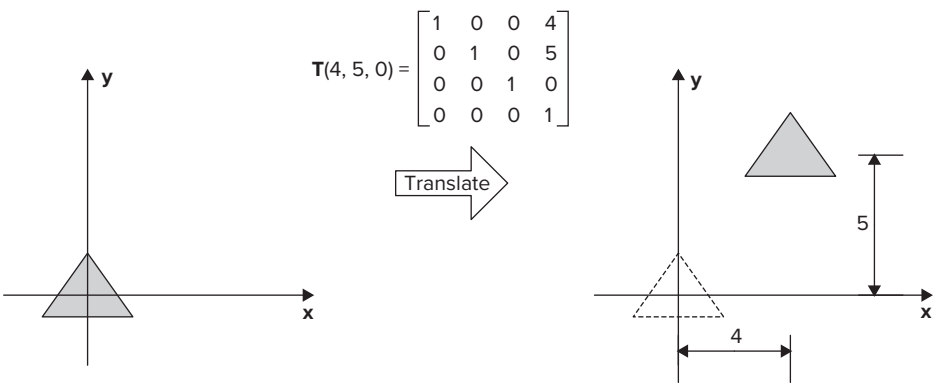

FIGURE 1-21: The translation of a triangle by 4 in the ***x***-direction and 5 in the ***y***-direction

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 **p** written on homogeneous notation:

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

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

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 **v** is unaffected when it is multiplied by the translation matrix:

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

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

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:

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

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

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

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.

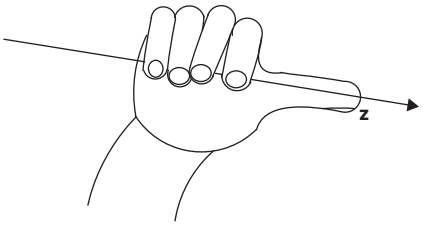

FIGURE 1-22: The right-hand grip rule helps you to remember the positive direction for rotation around an axis

Figure 1-23 illustrates a triangle that is rotated by 90° around the *z*-axis by applying the rotation matrix $\mathbf{R}_z$. To the left, the triangle is shown before the rotation. To the right, the triangle is shown after the transformation matrix $\mathbf{R}_z(90°)$ has been applied and the triangle has been rotated 90° around the *z*-axis that is pointing perpendicular out from the paper.

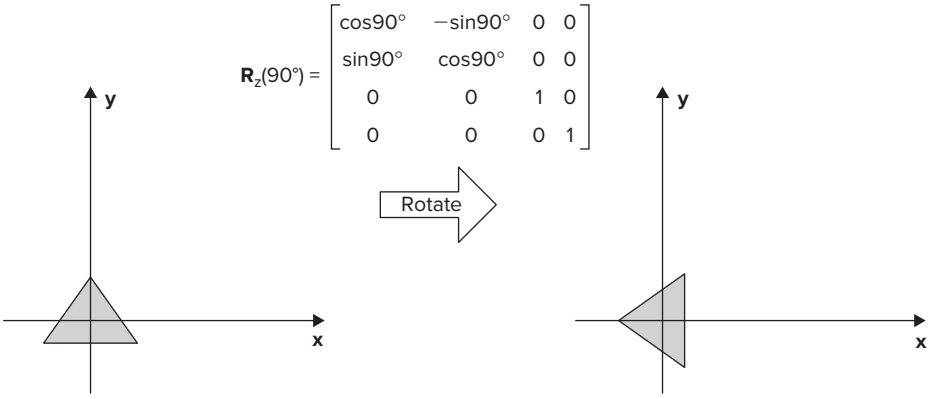

FIGURE 1-23: The rotation of a triangle 90° around the ***z***-axis

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.

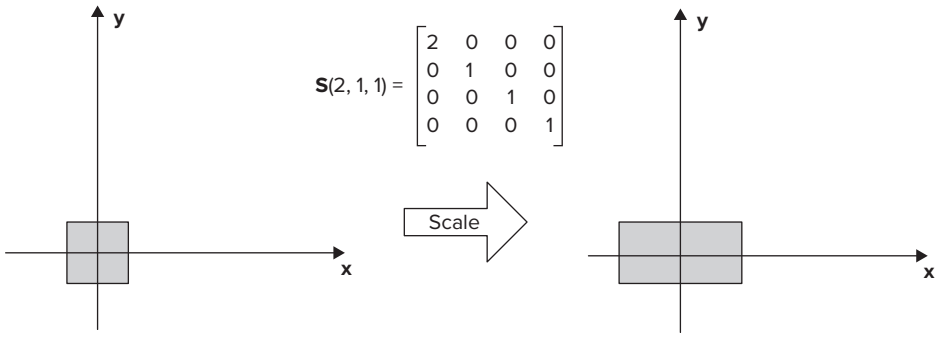

FIGURE 1-24: Scaling by two in the ***x***-direction

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 $\mathbf{H}_{xy}(s)$, $\mathbf{H}_{xz}(s)$, $\mathbf{H}_{yx}(s)$, $\mathbf{H}_{yz}(s)$, $\mathbf{H}_{zx}(s)$, and $\mathbf{H}_{zy}(s)$. 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 $\mathbf{H}_{xy}(s)$ is shown here:

$$\mathbf{H}_{xy}(s) = \begin{bmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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

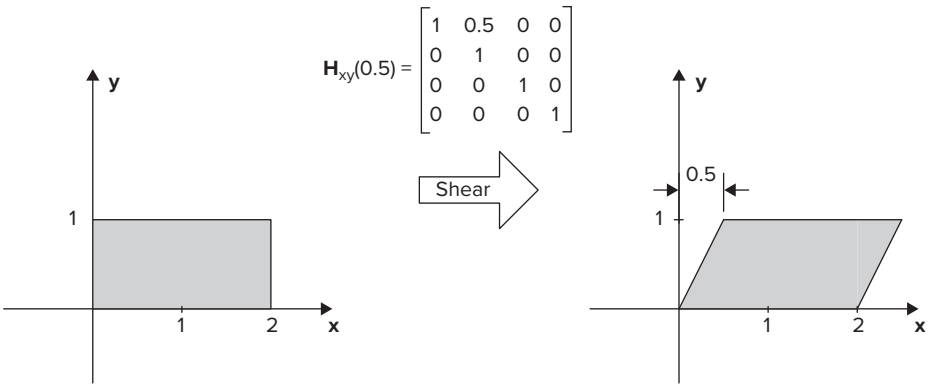

FIGURE 1-25: An example of shearing

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.

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.