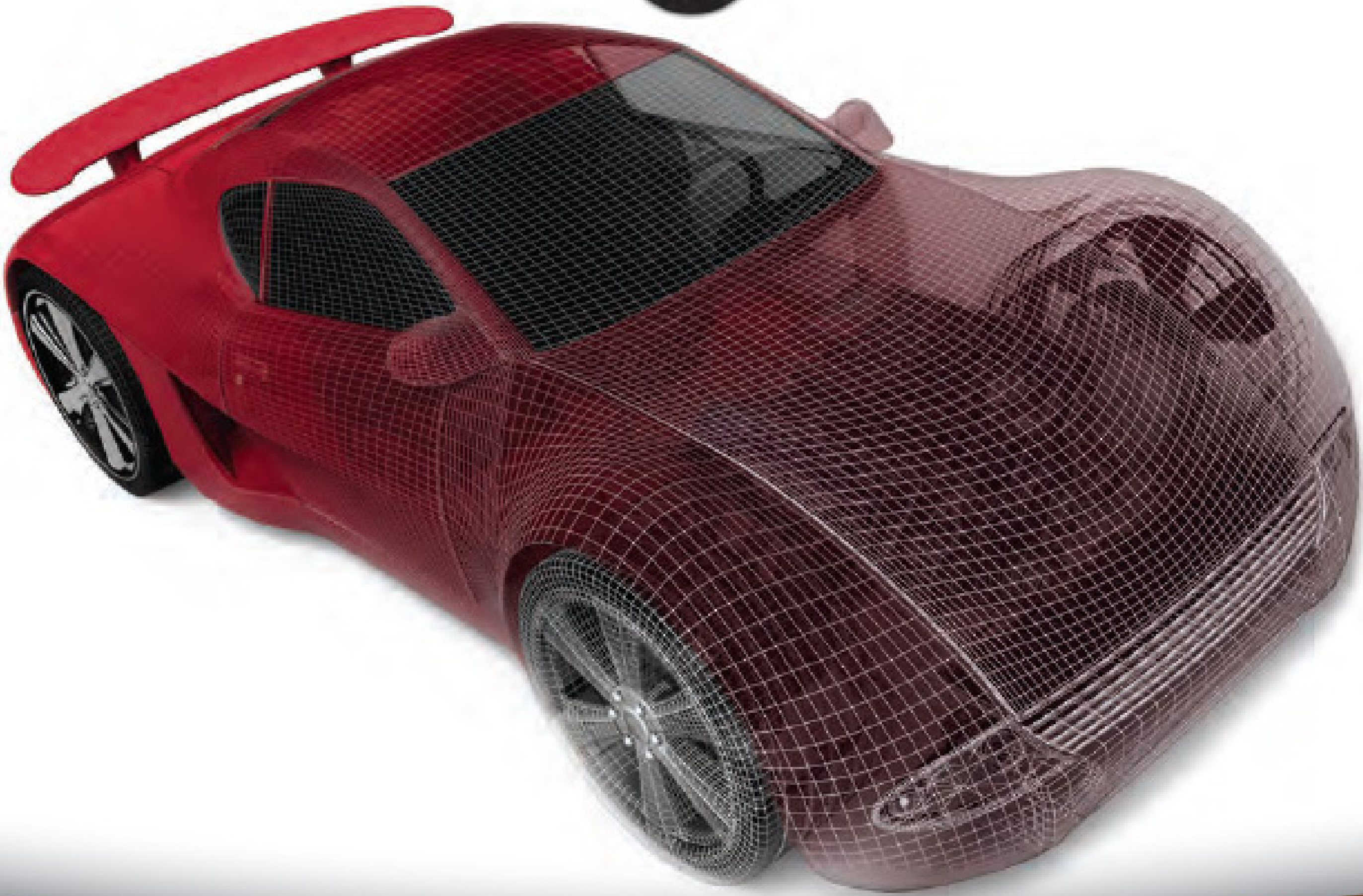
Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™



Professional

WebGL Programming

Developing 3D Graphics for the Web

Andreas Anyuru

Contents

Chapter 1

Introducing WebGL

WHAT'S IN THIS CHAPTER?

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

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

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

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

THE BASICS OF WEBGL

WebGL is an application programming interface (API) for advanced 3D graphics on the web. It is based on OpenGL ES 2.0, and provides similar rendering functionality, but in an

HTML and JavaScript context. The rendering surface that is used for WebGL is the HTML5 canvas element, which was originally introduced by Apple in the WebKit open-source browser engine. The reason for introducing the HTML5 canvas was to be able to render 2D graphics in applications such as Dashboard widgets and in the Safari browser on the Apple Mac OS X operating system.

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

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

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

SO WHY IS WEBGL SO GREAT?

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

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

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

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

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

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

The WebGL standard also offers a great way for students and others to learn and experiment with 3D graphics. There is no need to download and set up a toolchain like you have to do for most other 3D APIs. To create your WebGL application, you only need a text editor to write your code, and to view your creation you can just load your files into a web browser with WebGL support.

DESIGNING A GRAPHICS API

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

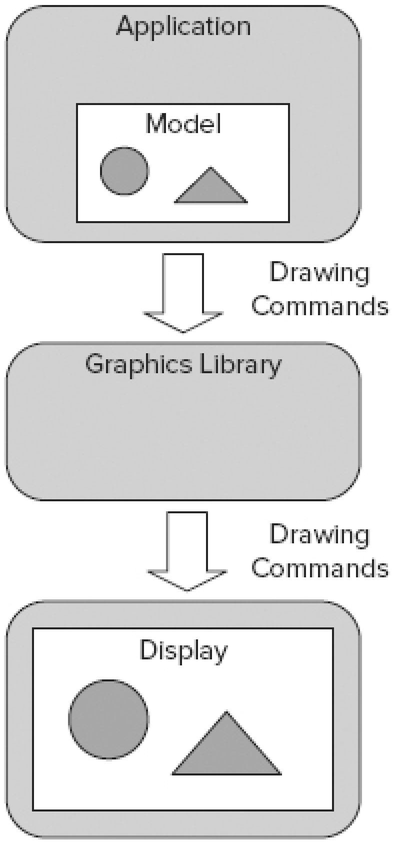
- Using an immediate-mode API
- Using a retained-mode API

WebGL is an immediate-mode API.

An Immediate-Mode API

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

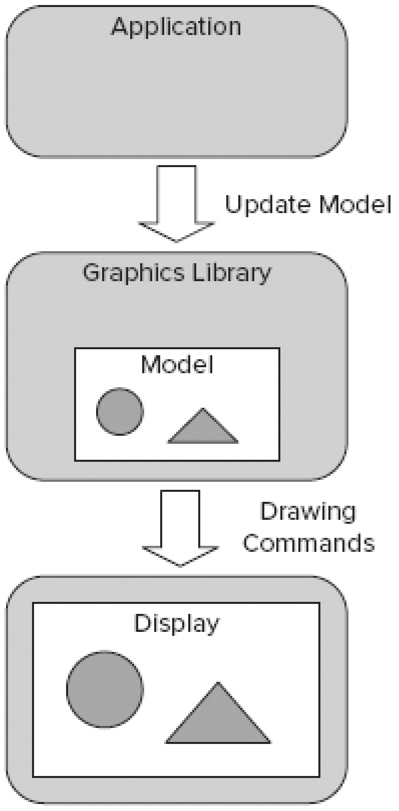
FIGURE 1-1: A diagram of how an immediate-mode API works



A Retained-Mode API

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

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

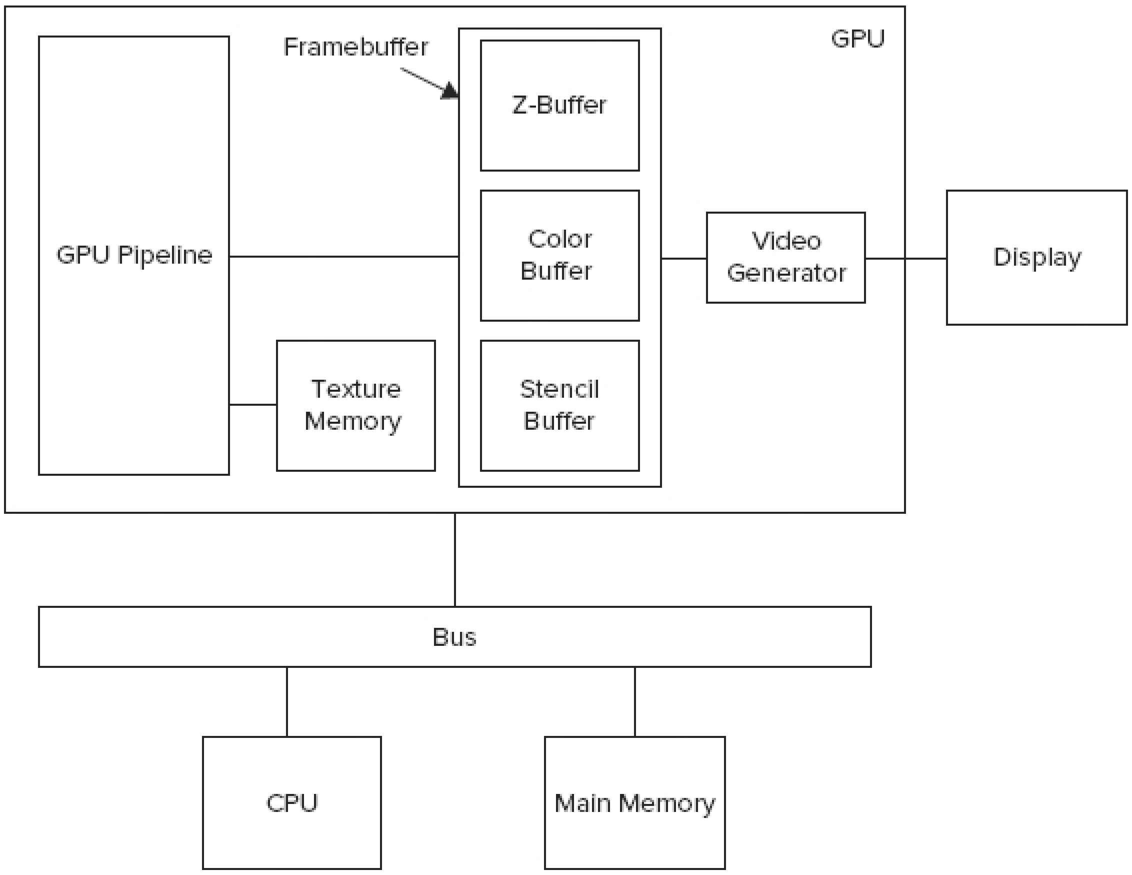


AN OVERVIEW OF GRAPHICS HARDWARE

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

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

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



GPU

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

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

Framebuffer

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

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

- Color buffer
- Z-buffer
- Stencil buffer

Color Buffer

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

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

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

In the same way, a framebuffer with a color depth of 24 bits per pixel usually allocates 8 bits for red, 8 bits for green, and 8 bits for blue. This gives you more than 16 million colors and no alpha channel in the framebuffer.

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

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

You will learn more about alpha blending in Chapter 8.

Z-Buffer

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

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

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

Stencil Buffer

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

Texture Memory

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

You will learn more about texturing in Chapter 5.

Video Controller

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

UNDERSTANDING THE WEBGL GRAPHICS PIPELINE

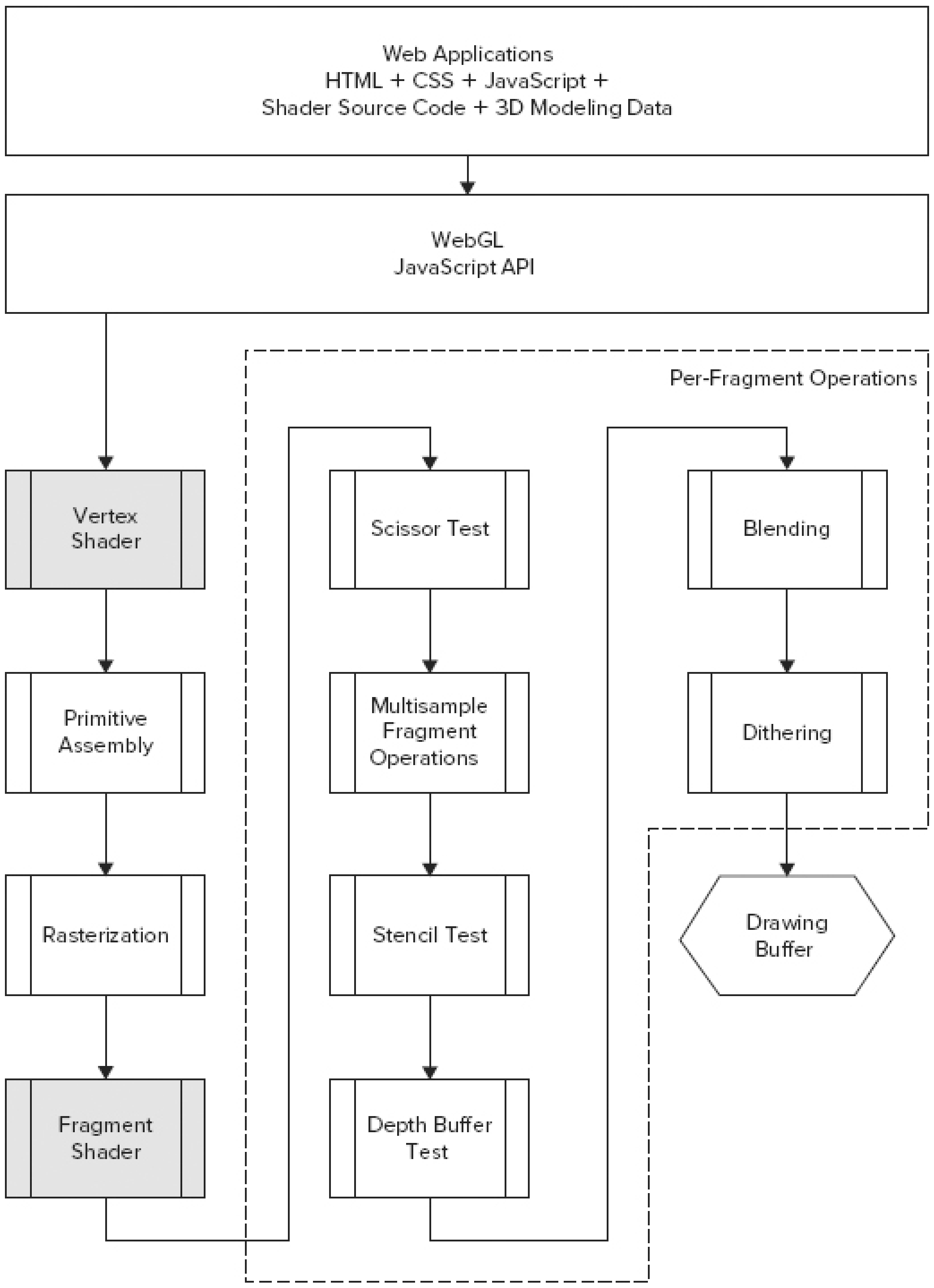
A web application that uses WebGL typically consists of HTML, CSS, and JavaScript files that execute within a web browser. In addition to this classical web application content, a WebGL application also contains source code for its shaders and some sort of data representing the 3D (or possibly 2D) models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

The browser does not require a plug-in to execute WebGL; the support is natively built into the browser. It is the JavaScript that calls the WebGL API to send in information to the WebGL pipeline for how the 3D models should be rendered. This information consists of not only the source code for the two programmable stages in the WebGL pipeline, the vertex shader, and the fragment shader, but also information about the 3D models that should be rendered.

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

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

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



Vertex Shader

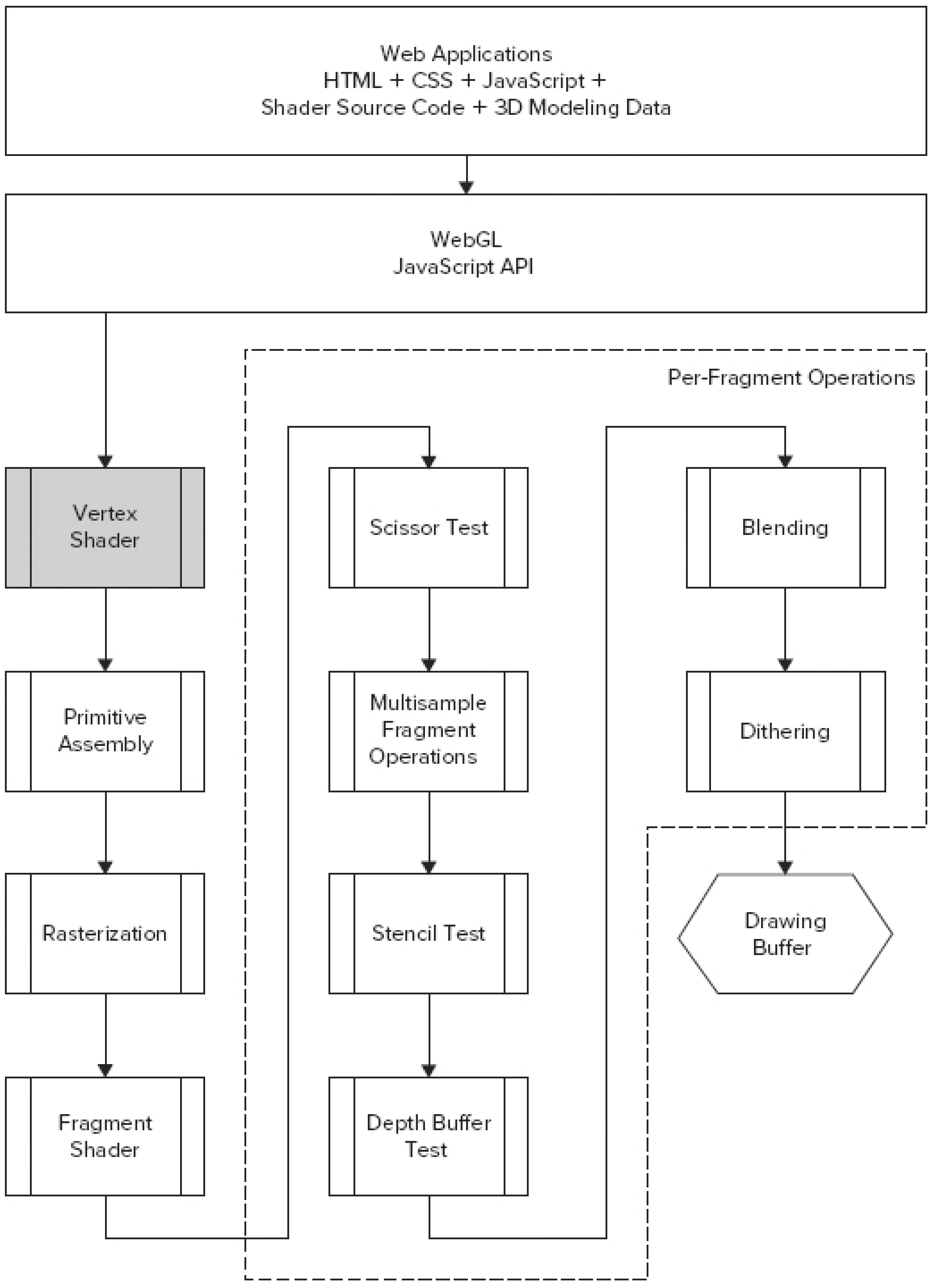
To get a realistic 3D scene, it is not enough to render objects at certain positions. You also need to take into account things like how the objects will look when light sources shine on them. The general term that is used for the process of determining the effect of light on different materials is called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

For WebGL, the shading is done in two stages:

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

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

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



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

JavaScript API, it can actually manipulate a vertex in many ways.

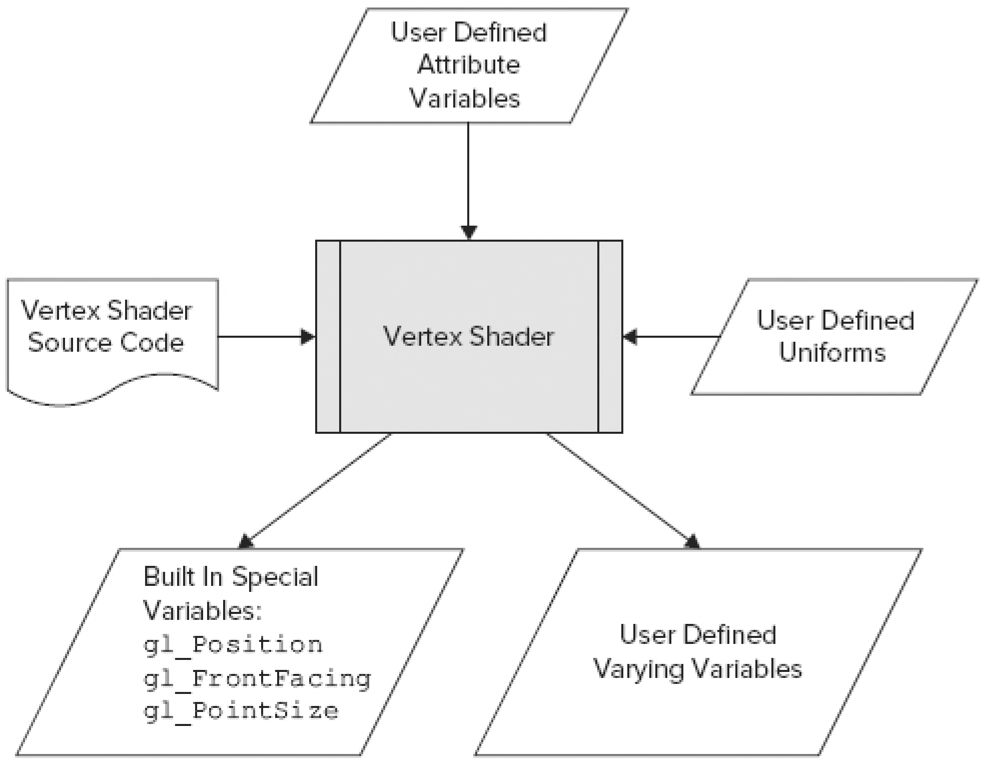
Before the actual shading starts, it often transforms the vertex by multiplying it with a transformation matrix. By multiplying all vertices of an object with the transformation matrix, the object can be placed at a specific position in your scene. You will learn more about transformations later in this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

The vertex shader uses the following input:

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

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

FIGURE 1-6: An overview of the vertex shader


User Defined Attribute Variables
Vertex Shader Source Code
Vertex Shader
User Defined Uniforms
Built In Special Variables:
gl_Position
gl_FrontFacing
gl_PointSize
User Defined Varying Variables

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);
```

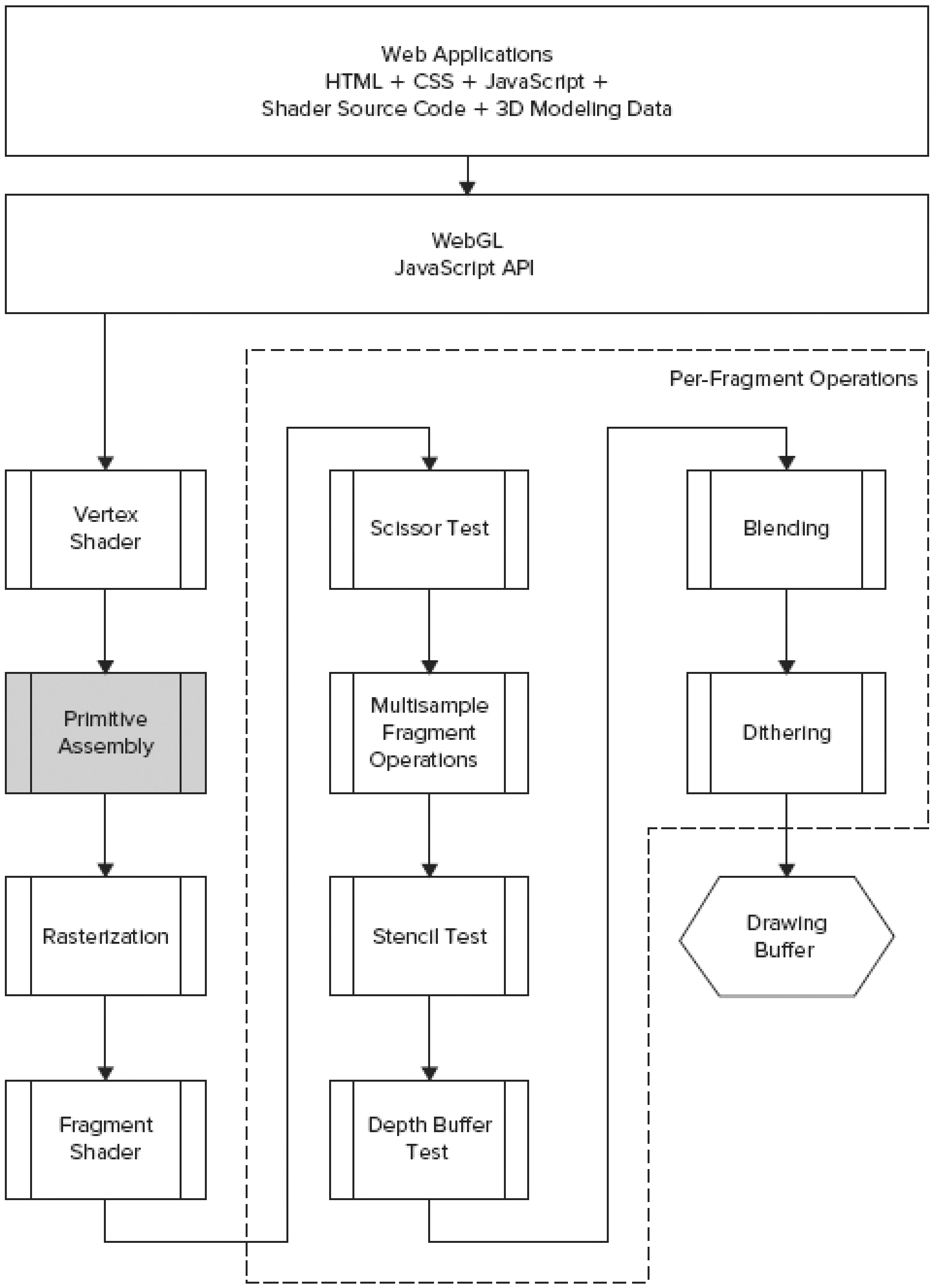
Both for the vertex shader and the fragment shader, the entry point is the `main()` function. The first statement in the `main()` function takes the vertex position `aVertexPos` and multiplies it by the two transformation matrices to perform the transformation. The result is written to the special built-in 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

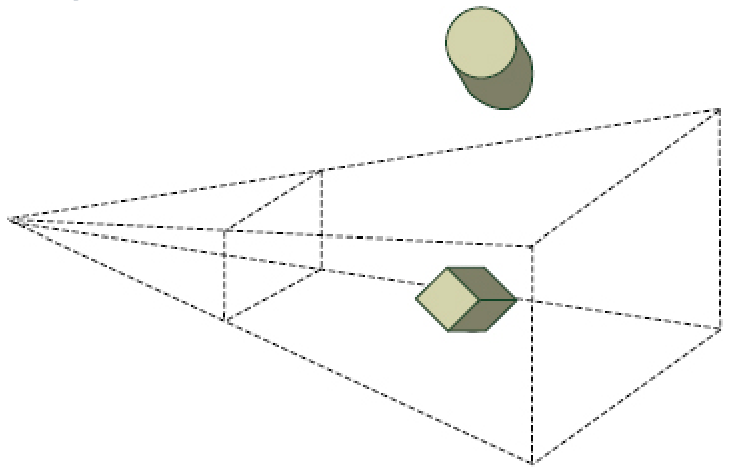
In the step after the vertex shader — known as *primitive assembly* (see Figure 1-7) — the WebGL pipeline needs to assemble the shaded vertices into individual geometric primitives such as triangles, lines, or point sprites. Then for each triangle, line, or point sprite, WebGL needs to decide whether 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.

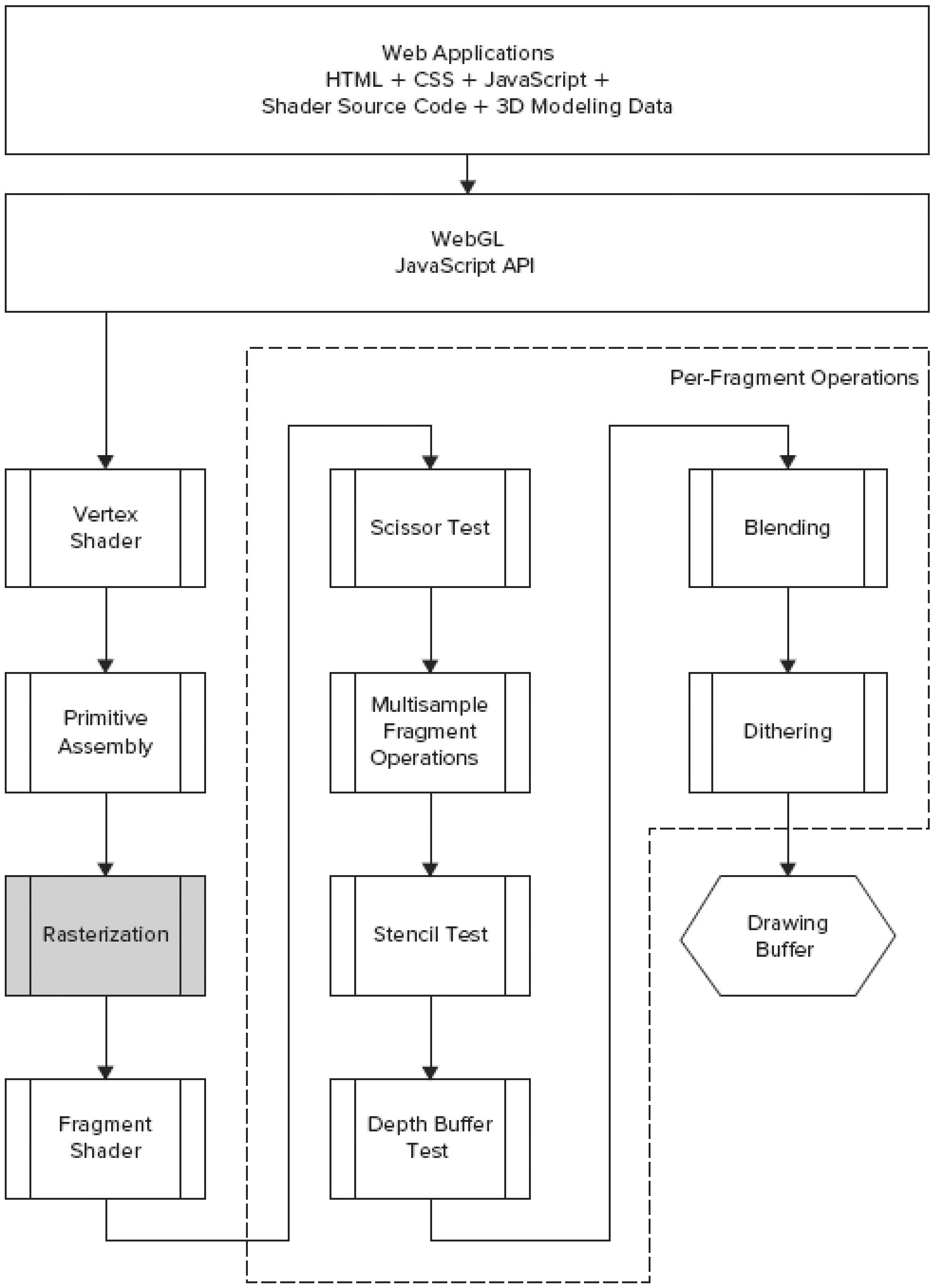
FIGURE 1-8: A view frustum with a cube that is inside the frustum and a cylinder that is outside the frustum



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).

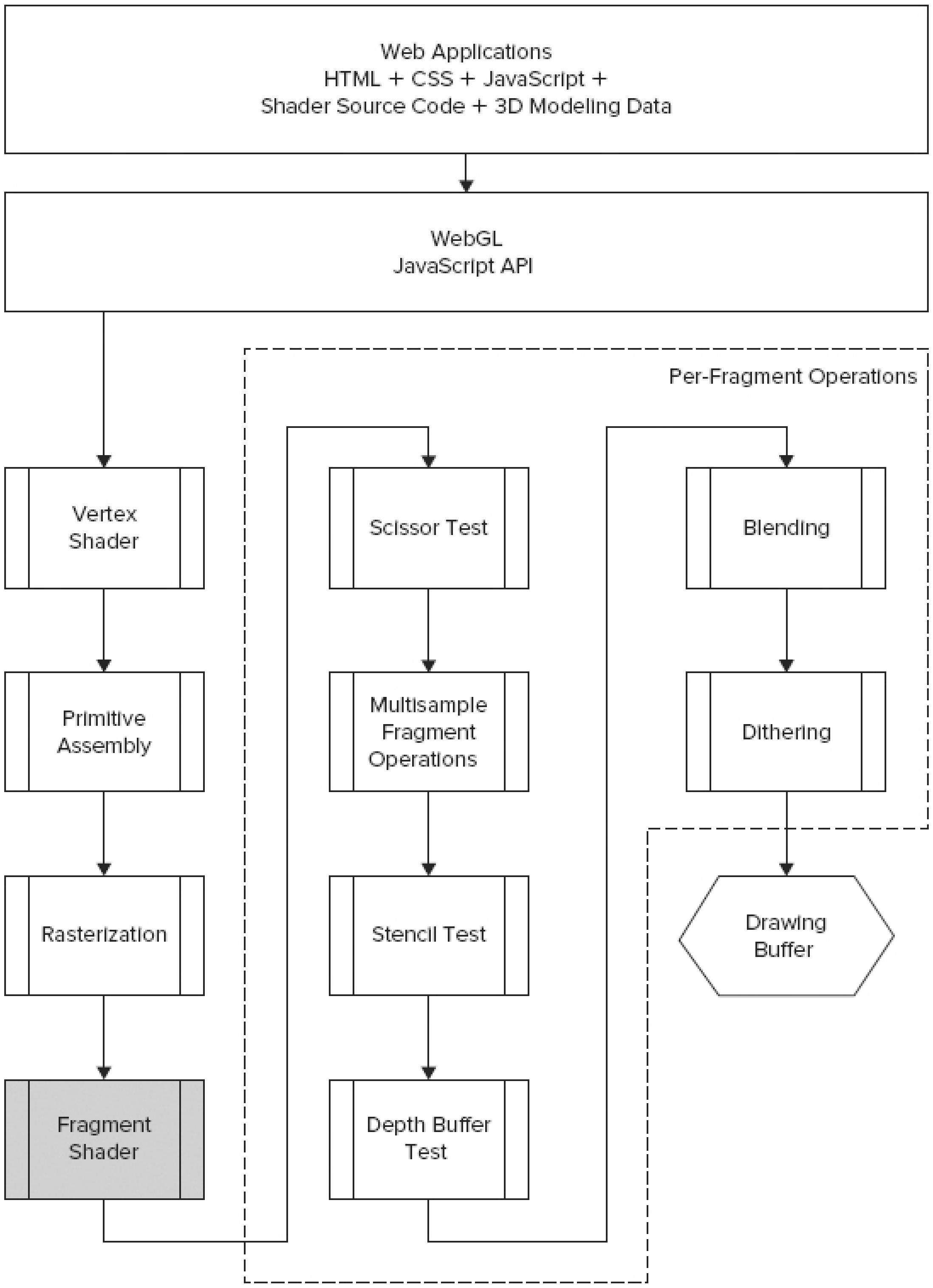
FIGURE 1-9: The location of rasterization in the WebGL graphics pipeline



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.