6.1: The Programmable Pipeline

OpenGL 1.1 used a fixed-function pipeline for graphics processing. Data is provided by a program and passes through a series of processing stages that ultimately produce the pixel colors seen in the final image. The program can enable and disable some of the steps in the process, such as the depth test and lighting calculations. But there is no way for it to change what happens at each stage. The functionality is fixed.

OpenGL 2.0 introduced a programmable pipeline. It became possible for the programmer to replace certain stages in the pipeline with their own programs. This gives the programmer complete control over what happens at that stage. In OpenGL 2.0, the programmability was optional; the complete fixed-function pipeline was still available for programs that didn’t need the flexibility of programmability. WebGL uses a programmable pipeline, and it is mandatory. There is no way to use WebGL without writing programs to implement part of the graphics processing pipeline.

The programs that are written as part of the pipeline are called shaders. For WebGL, you need to write a vertex shader, which is called once for each vertex in a primitive, and a fragment shader, which is called once for each pixel in the primitive. Aside from these two programmable stages, the WebGL pipeline also contains several stages from the original fixed-function pipeline. For example, the depth test is still part of the fixed functionality, and it can be enabled or disabled in WebGL in the same way as in OpenGL 1.1.

In this section, we will cover the basic structure of a WebGL program and how data flows from the JavaScript side of the program into the graphics pipeline and through the vertex and fragment shaders.

This book covers WebGL 1.0. Version 2.0 was released in January 2017. and it is compatible with version 1.0. At the end of 2017, WebGL 2.0 is available in some browsers, including Chrome and Firefox, but its new features are things that I would not cover in this book in any case. Also, I note that later versions of OpenGL have introduced additional programmable stages into the pipeline, in addition to the vertex and fragment shaders, but they are not part of WebGL and are not covered in this book.
The WebGL Graphics Context

To use WebGL, you need a WebGL graphics context. The graphics context is a JavaScript object whose methods implement the JavaScript side of the WebGL API. WebGL draws its images in an HTML canvas, the same kind of `<canvas>` element that is used for the 2D API that was covered in Section 2.6. A graphics context is associated with a particular canvas and can be obtained by calling the function `canvas.getContext("webgl")`, where `canvas` is a DOM object representing the canvas. A few browsers (notably Internet Explorer and Edge) require "experimental-webgl" as the parameter to `getContext`, so my code for creating a WebGL context often looks something like this:

```javascript
canvas = document.getElementById("webglcanvas");

gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
```

Here, `gl` is the WebGL graphics context. This code might require some unpacking. This is JavaScript code that would occur as part of a script in the source code for a web page. The first line assumes that the HTML source for the web page includes a canvas element with `id="webglcanvas"`, such as

```html
<canvas width="800" height="600" ></canvas>
```

In the second line, `canvas.getContext("webgl")` will return `null` if the web browser does not support "webgl" as a parameter to `getContext`; in that case, the second operand of the `||` operator will be evaluated. This use of `||` is a JavaScript idiom, which uses the fact that `null` is considered to be false when used in a boolean context. So, the second line, where the WebGL context is created, is equivalent to:

```javascript
if (!gl) {
  gl = canvas.getContext("experimental-webgl");
}
```

It is possible that `canvas.getContext("experimental-webgl")` is also `null`, if the browser supports the 2D canvas API but does not support WebGL. Furthermore, if the browser has no support at all for `<canvas>`, the code will throw an exception. So, I use a function similar to the following for initialization of my WebGL programs:

```javascript
function init() {
  try {
    canvas = document.getElementById("webglcanvas");
    gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
    if (!gl) {
      throw "Browser does not support WebGL";
    }
  } catch (e) {
    // report the error
    return;
  }
}
```
In this function, `canvas` and `gl` are global variables. And `initGL()` is a function defined elsewhere in the script that initializes the graphics context, including creating and installing the shader programs. The `init()` function could be called, for example, by the `onload` event handler for the `<body>` element of the web page:

```html
<body onload="init()">
```

Once the graphics context, `gl`, has been created, it can be used to call functions in the WebGL API. For example, the command for enabling the depth test, which was written as `glEnable(GL_DEPTH_TEST)` in OpenGL, becomes

```javascript
gl.enable( gl.DEPTH_TEST );
```

Note that both functions and constants in the API are referenced through the graphics context. The name “gl” for the graphics context is conventional, but remember that it is just an ordinary JavaScript variable whose name is up to the programmer.

Although I use `canvas.getContext("experimental-webgl")` in my sample programs, I will generally not include it in code examples in the text.

### The Shader Program

Drawing with WebGL requires a shader program, which consists of a vertex shader and a fragment shader. Shaders are written in the language GLSL ES 1.0 (the OpenGL Shader Language for Embedded Systems, version 1.0). GLSL is based on the C programming language. The vertex shader and fragment shader are separate programs, each with its own `main()` function. The two shaders are compiled separately and then “linked” to produce a complete shader program. The JavaScript API for WebGL includes functions for compiling the shaders and then linking them. To use the functions, the source code for the shaders must be JavaScript strings. Let’s see how it works. It takes three steps to create the vertex shader.

```javascript
var vertexShader = gl.createShader( gl.VERTEX_SHADER );
gl.shaderSource( vertexShader, vertexShaderSource );
gl.compileShader( vertexShader );
```

The functions that are used here are part of the WebGL graphics context, `gl`, and `vertexShaderSource` is the string that contains the source code for the shader. Errors in the source code will cause the compilation to fail silently. You need to check for compilation errors by calling the function

```javascript
gl.getShaderParameter( vertexShader, gl.COMPILE_STATUS )
```
which returns a boolean value to indicate whether the compilation succeeded. In the event that an error occurred, you
can retrieve an error message with

\[ \text{\texttt{gl.getShaderInfoLog(vsh)}} \]

which returns a string containing the result of the compilation. (The exact format of the string is not specified by the
WebGL standard. The string is meant to be human-readable.)

The fragment shader can be created in the same way. With both shaders in hand, you can create and link the program.
The shaders need to be "attached" to the program object before linking. The code takes the form:

```javascript
var prog = gl.createProgram();
geAttachShader( prog, vertexShader );
geAttachShader( prog, fragmentShader );
geLinkProgram( prog );
```

Even if the shaders have been successfully compiled, errors can occur when they are linked into a complete program.
For example, the vertex and fragment shader can share certain kinds of variable. If the two programs declare such
variables with the same name but with different types, an error will occur at link time. Checking for link errors is similar to
checking for compilation errors in the shaders.

The code for creating a shader program is always pretty much the same, so it is convenient to pack it into a reusable
function. Here is the function that I use for the examples in this chapter:

```javascript
/**
 * Creates a program for use in the WebGL context gl, and returns the
 * identifier for that program. If an error occurs while compiling or
 * linking the program, an exception of type String is thrown. The error
 * string contains the compilation or linking error.
 */
function createProgram(gl, vertexShaderSource, fragmentShaderSource) {
    var vsh = gl.createShader( gl.VERTEX_SHADER );
geShaderSource( vsh, vertexShaderSource );
geCompileShader( vsh );
    if ( ! gl.getShaderParameter(vsh, gl.COMPILE_STATUS) ) {
        throw "Error in vertex shader: " + gl.getShaderInfoLog(vsh);
    }
    var fsh = gl.createShader( gl.FRAGMENT_SHADER );
geShaderSource( fsh, fragmentShaderSource );
geCompileShader( fsh );
    if ( ! gl.getShaderParameter(fsh, gl.COMPILE_STATUS) ) {
        throw "Error in fragment shader: " + gl.getShaderInfoLog(fsh);
    }
    var prog = gl.createProgram();
geAttachShader( prog, vsh );
geAttachShader( prog, fsh );
geLinkProgram( prog );
    if ( ! gl.getProgramParameter( prog, gl.LINK_STATUS) ) {
        throw "Link error in program: " + gl.getProgramInfoLog(prog);
    }
}
```
There is one more step: You have to tell the WebGL context to use the program. If `prog` is a program identifier returned by the above function, this is done by calling

```javascript
gl.useProgram( prog );
```

It is possible to create several shader programs. You can then switch from one program to another at any time by calling `gl.useProgram`, even in the middle of rendering an image. (*Three.js*, for example, uses a different program for each type of `Material`.)

It is advisable to create any shader programs that you need as part of initialization. Although `gl.useProgram` is a fast operation, compiling and linking are rather slow, so it’s better to avoid creating new programs while in the process of drawing an image.

Shaders and programs that are no longer needed can be deleted to free up the resources they consume. Use the functions `gl.deleteShader(shader)` and `gl.deleteProgram(program)`.

---

**Data Flow in the Pipeline**

The WebGL graphics pipeline renders an image. The data that defines the image comes from JavaScript. As it passes through the pipeline, it is processed by the current vertex shader and fragment shader as well as by the fixed-function stages of the pipeline. You need to understand how data is placed by JavaScript into the pipeline and how the data is processed as it passes through the pipeline.

The basic operation in WebGL is to draw a geometric primitive. WebGL uses just seven of the OpenGL primitives that were introduced in Subsection 3.1.1. The primitives for drawing quads and polygons have been removed. The remaining primitives draw points, line segments, and triangles. In WebGL, the seven types of primitive are identified by the constants `gl.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, and `gl.TRIANGLE_FAN`, where `gl` is a WebGL graphics context.

When WebGL is used to draw a primitive, there are two general categories of data that can be provided for the primitive. The two kinds of data are referred to as **attribute variables** (or just “attributes”) and **uniform variables** (or just “uniforms”). A primitive is defined by its type and by a list of vertices. The difference between attributes and uniforms is that a uniform variable has a single value that is the same for the entire primitive, while the value of an attribute variable can be different for different vertices.

One attribute that is always specified is the coordinates of the vertex. The vertex coordinates must be an attribute since each vertex in a primitive will have its own set of coordinates. Another possible attribute is color. We have seen that OpenGL allows you to specify a different color for each vertex of a primitive. You can do the same thing in WebGL, and in that case the color will be an attribute. On the other hand, maybe you want the entire primitive to have the same, “uniform” color; in that case, color can be a uniform variable. Other quantities that could be either attributes or uniforms, depending on your needs, include normal vectors and material properties. Texture coordinates, if they are used, are
almost certain to be an attribute, since it doesn’t really make sense for all the vertices in a primitive to have the same texture coordinates. If a geometric transform is to be applied to the primitive, it would naturally be represented as a uniform variable.

It is important to understand, however, that WebGL does not come with any predefined attributes, not even one for vertex coordinates. In the programmable pipeline, the attributes and uniforms that are used are entirely up to the programmer. As far as WebGL is concerned, attributes are just values that are passed into the vertex shader. Uniforms can be passed into the vertex shader, the fragment shader, or both. WebGL does not assign a meaning to the values. The meaning is entirely determined by what the shaders do with the values. The set of attributes and uniforms that are used in drawing a primitive is determined by the source code of the shaders that are in use when the primitive is drawn.

To understand this, we need to look at what happens in the pipeline in a little more detail. When drawing a primitive, the JavaScript program will specify values for any attributes and uniforms in the shader program. For each attribute, it will specify an array of values, one for each vertex. For each uniform, it will specify a single value. The values will all be sent to the GPU before the primitive is drawn. When drawing the primitive, the GPU calls the vertex shader once for each vertex. The attribute values for the vertex that is to be processed are passed as input into the vertex shader. Values of uniform variables are also passed to the vertex shader. Both attributes and uniforms are represented as global variables in the shader, whose values are set before the shader is called.

As one of its outputs, the vertex shader must specify the coordinates of the vertex in the clip coordinate system (see Subsection 3.3.1). It does that by assigning a value to a special variable named gl_Position. The position is often computed by applying a transformation to the attribute that represents the coordinates in the object coordinate system, but exactly how the position is computed is up to the programmer.

After the positions of all the vertices in the primitive have been computed, a fixed-function stage in the pipeline clips away the parts of the primitive whose coordinates are outside the range of valid clip coordinates (~1 to 1 along each coordinate axis). The primitive is then rasterized; that is, it is determined which pixels lie inside the primitive. The fragment shader is then called once for each pixel that lies in the primitive. The fragment shader has access to uniform variables (but not attributes). It can also use a special variable named gl_FragCoord that contains the clip coordinates of the pixel. Pixel coordinates are computed by interpolating the values of gl_Position that were specified by the vertex shader. The interpolation is done by another fixed-function stage that comes between the vertex shader and the fragment shader.

Other quantities besides coordinates can work in much that same way. That is, the vertex shader computes a value for the quantity at each vertex of a primitive. An interpolator takes the values at the vertices and computes a value for each pixel in the primitive. The value for a given pixel is then input into the fragment shader when the shader is called to process that pixel. For example, color in OpenGL follows this pattern: The color of an interior pixel of a primitive is computed by interpolating the color at the vertices. In GLSL, this pattern is implemented using varying variables.

A varying variable is declared both in the vertex shader and in the fragment shader. The vertex shader is responsible for assigning a value to the varying variable. The interpolator takes the values from the vertex shader and computes a value for each pixel. When the fragment shader is executed for a pixel, the value of the varying variable is the interpolated value for that pixel. The fragment shader can use the value in its own computations. (In newer versions of GLSL, the term “varying variable” has been replaced by “out variable” in the vertex shader and “in variable” in the fragment shader.)
Varying variables exist to communicate data from the vertex shader to the fragment shader. They are defined in the shader source code. They are not used or referred to in the JavaScript side of the API. Note that it is entirely up to the programmer to decide what varying variables to define and what to do with them.

We have almost gotten to the end of the pipeline. After all that, the job of the fragment shader is simply to specify a color for the pixel. It does that by assigning a value to a special variable named \textit{gl\_FragColor}. That value will then be used in the remaining fixed-function stages of the pipeline.

To summarize: The JavaScript side of the program sends values for attributes and uniform variables to the GPU and then issues a command to draw a primitive. The GPU executes the vertex shader once for each vertex. The vertex shader can use the values of attributes and uniforms. It assigns values to \textit{gl\_Position} and to any varying variables that exist in the shader. After clipping, rasterization, and interpolation, the GPU executes the fragment shader once for each pixel in the primitive. The fragment shader can use the values of varying variables, uniform variables, and \textit{gl\_FragCoord}. It computes a value for \textit{gl\_FragColor}. This diagram summarizes the flow of data:

![Diagram of WebGL pipeline]

The diagram is not complete. There are a few more special variables that I haven't mentioned. And there is the important question of how textures are used. But if you understand the diagram, you have a good start on understanding WebGL.
Values for Uniform Variables

It's time to start looking at some actual WebGL code. We will concentrate on the JavaScript side first, but you need to know a little about GLSL. GLSL has some familiar basic data types: `float`, `int`, and `bool`. But it also has some new predefined data types to represent vectors and matrices. For example, the data type `vec3` represents a vector in 3D. The value of a `vec3` variable is a list of three floating-point numbers. Similarly, there are data types `vec2` and `vec4` to represent 2D and 4D vectors.

Global variable declarations in a vertex shader can be marked as `attribute`, `uniform`, or `varying`. A variable declaration with none of these modifiers defines a variable that is local to the vertex shader. Global variables in a fragment can optionally be modified with `uniform` or `varying`, or they can be declared without a modifier. A varying variable should be declared in both shaders, with the same name and type. This allows the GLSL compiler to determine what attribute, uniform, and varying variables are used in a shader program.

The JavaScript side of the program needs a way to refer to particular attributes and uniform variables. The function `gl.getUniformLocation` can be used to get a reference to a uniform variable in a shader program, where `gl` refers to the WebGL graphics context. It takes as parameters the identifier for the compiled program, which was returned by `gl.createProgram`, and the name of the uniform variable in the shader source code. For example, if `prog` identifies a shader program that has a uniform variable named `color`, then the location of the `color` variable can be obtained with the JavaScript statement

```javascript
colorUniformLoc = gl.getUniformLocation( prog, "color" );
```

The location `colorUniformLoc` can then be used to set the value of the uniform variable:

```javascript
gl.uniform3f( colorUniformLoc, 1, 0, 0 );
```

The function `gl.uniform3f` is one of a family of functions that can be referred to as a group as `gl.uniform*`. This is similar to the family `gl.Vertex*` in OpenGL 1.1. The `*` represents a suffix that tells the number and type of values that are provided for the variable. In this case, `gl.uniform3f` takes three floating point values, and it is appropriate for setting the value of a uniform variable of type `vec3`. The number of values can be 1, 2, 3, or 4. The type can be "f" for floating point or "i" for integer. (For a boolean uniform, you should use `gl.uniform1i` and pass 0 to represent `false` or 1 to represent `true`.) If a "v" is added to the suffix, then the values are passed in an array. For example,

```javascript
gl.uniform3fv( colorUniformLoc, [ 1, 0, 0 ] );
```

There is another family of functions for setting the value of uniform matrix variables. We will get to that later.

The value of a uniform variable can be set any time after the shader program has been compiled, and the value remains in effect until it is changed by another call to `gl.uniform*`.

Values for Attributes

Turning now to attributes, the situation is more complicated, because an attribute can take a different value for each
vertex in a primitive. The basic idea is that the complete set of data for the attribute is copied in a single operation from a JavaScript array into memory that is accessible to the GPU. Unfortunately, setting things up to make that operation possible is non-trivial.

First of all, a regular JavaScript array is not suitable for this purpose. For efficiency, we need the data to be in a block of memory holding numerical values in successive memory locations, and regular JavaScript arrays don't have that form. To fix this problem, a new kind of array, called typed arrays, was introduced into JavaScript. We encountered typed arrays briefly in Subsection 2.6.6. A typed array can only hold numbers of a specified type. There are different kinds of typed array for different kinds of numerical data. For now we will use Float32Array, which holds 32-bit floating point numbers.

A typed array has a fixed length, which is assigned when it is created by a constructor. The constructor takes two forms: One form takes an integer parameter giving the number of elements in the array; the other takes a regular JavaScript array of numbers as parameter and initializes the typed array to have the same length and elements as the array parameter. For example:

```javascript
var color = new Float32Array( 12 ); // space for 12 floats
var coords = new Float32Array( [ 0,0.7, -0.7,-0.5, 0.7,-0.5 ] );
```

Once you have a typed array, you can use it much like a regular array. The length of the typed array `color` is `color.length`, and its elements are referred to as `color[0]`, `color[1]`, `color[2]`, and so on. When you assign a value to an element of a `Float32Array`, the value is converted into a 32-bit floating point number. If the value cannot be interpreted as a number, it will be converted to `NaN`, the "not-a-number" value.

Before data can be transferred from JavaScript into an attribute variable, it must be placed into a typed array. When possible, you should work with typed arrays directly, rather than working with regular JavaScript arrays and then copying the data into typed arrays.

For use in WebGL, the attribute data must be transferred into a VBO (vertex buffer object). VBOs were introduced in OpenGL 1.5 and were discussed briefly in Subsection 3.4.4. A VBO is a block of memory that is accessible to the GPU. To use a VBO, you must first call the function `gl.createBuffer()` to create it. For example,

```
colorBuffer = gl.createBuffer();
```

Before transferring data into the VBO, you must "bind" the VBO:

```
gl.bindBuffer( gl.ARRAY_BUFFER, colorBuffer );
```

The first parameter to `gl.bindBuffer` is called the "target." It specifies how the VBO will be used. The target `gl.ARRAY_BUFFER` is used when the buffer is being used to store values for an attribute. Only one VBO at a time can be bound to a given target.

The function that transfers data into a VBO doesn't mention the VBO — instead, it uses the VBO that is currently bound. To copy data into that buffer, use `gl.bufferData()`. For example:
The first parameter is, again, the target. The data is transferred into the VBO that is bound to that target. The second parameter is the typed array that holds the data on the JavaScript side. All the elements of the array are copied into the buffer, and the size of the array determines the size of the buffer. Note that this is a straightforward transfer of raw data bytes; WebGL does not remember whether the data represents floats or ints or some other kind of data.

The third parameter to `gl.bufferData` is one of the constants `gl.STATIC_DRAW`, `gl.STREAM_DRAW`, or `gl.DYNAMIC_DRAW`. It is a hint to WebGL about how the data will be used, and it helps WebGL to manage the data in the most efficient way. The value `gl.STATIC_DRAW` means that you intend to use the data many times without changing it. For example, if you will use the same data throughout the program, you can load it into a buffer once, during initialization, using `gl.STATIC_DRAW`. WebGL will probably store the data on the graphics card itself where it can be accessed most quickly by the graphics hardware. The second value, `gl.STREAM_DRAW`, is for data that will be used only once, then discarded. (It can be "streamed" to the card when it is needed.) The value `gl.DYNAMIC_DRAW` is somewhere between the other two values; it might be used for data that will be used a couple of times and then discarded.

Getting attribute data into VBOs is only part of the story. You also have to tell WebGL to use the VBO as the source of values for the attribute. To do so, first of all, you need to know the location of the attribute in the shader program. You can determine that using `gl.getAttribLocation`. For example,

```javascript
colorAttribLoc = gl.getAttribLocation(prog, "a_color");
```

This assumes that `prog` is the shader program and "a_color" is the name of the attribute variable in the vertex shader. This is entirely analogous to `gl.getUniformLocation`.

Although an attribute usually takes different values at different vertices, it is possible to use the same value at every vertex. In fact, that is the default behavior. The single attribute value for all vertices can be set using the family of functions `gl.vertexAttrib*`, which work similarly to `gl.uniform*`. In the more usual case, where you want to take the values of an attribute from a VBO, you must enable the use of a VBO for that attribute. This is done by calling

```javascript
gl.enableVertexAttribArray(colorAttribLoc);
```

where the parameter is the location of the attribute in the shader program, as returned by a call to `gl.getAttribLocation()`. This command has nothing to do with any particular VBO. It just turns on the use of buffers for the specified attribute. Generally, it is reasonable to call this method just once, during initialization.

Finally, before you draw a primitive that uses the attribute data, you have to tell WebGL which buffer contains the data and how the bits in that buffer are to be interpreted. This is done with `gl.vertexAttribPointer()`. A VBO must be bound to the `ARRAY_BUFFER` target when this function is called. For example,

```javascript
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);

gl.vertexAttribPointer(colorAttribLoc, 3, gl.FLOAT, false, 0, 0);
```
Assuming that \texttt{colorBuffer} refers to the VBO and \texttt{colorAttribLoc} is the location of the attribute, this tells WebGL to take values for the attribute from that buffer. Often, you will call \texttt{gl.bindBuffer()} just before calling \texttt{gl.vertexAttribPointer()}, but that is not necessary if the desired buffer is already bound.

The first parameter to \texttt{gl.vertexAttribPointer} is the attribute location. The second is the number of values per vertex. For example, if you are providing values for a \texttt{vec2}, the second parameter will be 2 and you will provide two numbers per vertex; for a \texttt{vec3}, the second parameter would be 3; for a \texttt{float}, it would be 1. The third parameter specifies the type of each value. Here, \texttt{gl.FLOAT} indicates that each value is a 32-bit floating point number. Other values include \texttt{gl.BYTE}, \texttt{gl.UNSIGNED_BYTE}, \texttt{gl.UNSIGNED_SHORT}, and \texttt{gl.SHORT} for integer values. Note that the type of data does not have to match the type of the attribute variable; in fact, attribute variables are always floating point. However, the parameter value does have to match the data type in the buffer. If the data came from a \texttt{Float32Array}, then the parameter must be \texttt{gl.FLOAT}. I will always use \texttt{false}, 0, and 0 for the remaining three parameters. They add flexibility that I won’t need; you can look them up in the documentation if you are interested.

There is a lot to take in here. Using a VBO to provide values for an attribute requires six separate commands, and that is in addition to generating the data and placing it in a typed array. Here is the full set of commands:

```javascript
let colorAttribLoc = gl.getAttribLocation( prog, "a color" );
let colorBuffer = gl.createBuffer();
gl.enableVertexAttribArray( colorAttribLoc );

gl.bindBuffer( gl.ARRAY_BUFFER, colorBuffer );
gl.vertexAttribPointer( colorAttribLoc, 3, gl.FLOAT, false, 0, 0 );
gl.bufferData( gl.ARRAY_BUFFER, colorArray, gl.STATIC_DRAW );
```

However, the six commands will not usually occur at the same point in the JavaScript code. the first three commands are often done as part of initialization. \texttt{gl.bufferData} would be called whenever the data for the attribute needs to be changed. \texttt{gl.bindBuffer} must be called before \texttt{gl.vertexAttribPointer} or \texttt{gl.bufferData}, since it establishes the VBO that is used by those two commands. Remember that all of this must be done for every attribute that is used in the shader program.

---

### Drawing a Primitive

After the shader program has been created and values have been set up for the uniform variables and attributes, it takes just one more command to draw a primitive:

```javascript
gl.drawArrays( primitiveType, startVertex, vertexCount );
```

The first parameter is one of the seven constants that identify WebGL primitive types, such as \texttt{gl.TRIANGLES}, \texttt{gl.LINE_LOOP}, and \texttt{gl.POINTS}. The second and third parameters are integers that determine which subset of available vertices is used for the primitive. Before calling \texttt{gl.drawArrays}, you will have placed attribute values for some number of vertices into one or more VBOs. When the primitive is rendered, the attribute values are pulled from the VBOs. The \texttt{startVertex} is the starting vertex number of the data within the VBOs, and \texttt{vertexCount} is the number of vertices in the primitive. Often, \texttt{startVertex} is zero, and \texttt{vertexCount} is the total number of vertices for which data is available. For example, the command for drawing a single triangle might be
gl.drawArrays( gl.TRIANGLES, 0, 3 );

The use of the word “array” in gl.drawArrays and gl.ARRAY_BUFFER might be a little confusing, since the data is stored in vertex buffer objects rather than in JavaScript arrays. When glDrawArrays was first introduced in OpenGL 1.1, it used ordinary arrays rather than VBOs. Starting with OpenGL 1.5, glDrawArrays could be used either with ordinary arrays or VBOs. In WebGL, support for ordinary arrays was dropped, and gl.drawArrays can only work with VBOs, even though the name still refers to arrays.

We encountered the original version of glDrawArrays in Subsection 3.4.2. That section also introduced an alternative function for drawing primitives, glDrawElements, which can be used for drawing indexed face sets. A gl.drawElements function is also available in WebGL. With gl.drawElements, attribute data is not used in the order in which it occurs in the VBOs. Instead, there is a separate list of indices that determines the order in which the data is accessed.

To use gl.drawElements, an extra VBO is required to hold the list of indices. When used for this purpose, the VBO must be bound to the target gl.ELEMENT_ARRAY_BUFFER rather than gl.ARRAY_BUFFER. The VBO will hold integer values, which can be of type gl.UNSIGNED_BYTE or gl.UNSIGNED_SHORT. The values can be loaded from a JavaScript typed array of type Uint8Array or Uint16Array. Creating the VBO and filling it with data is again a multi-step process. For example,

```javascript
let elementBuffer = gl.createBuffer();
gl.bindBuffer( gl.ELEMENT_ARRAY_BUFFER, elementBuffer );
var data = new Uint8Array([ 2,0,3, 2,1,3, 1,4,3 ]);
gl.bufferData( gl.ELEMENT_ARRAY_BUFFER, data, gl.STREAM_DRAW );
```

Assuming that the attribute data has also been loaded into VBOs, gl.drawElements can then be used to draw the primitive. A call to gl.drawElements takes the form

```javascript
gl.drawElements( primitiveType, count, dataType, startByte );
```

The VBO that contains the vertex indices must be bound to the ELEMENT_ARRAY_BUFFER target when this function is called. The first parameter to gl.drawElements is a primitive type such as gl.TRIANGLE_FAN. The count is the number of vertices in the primitive. The dataType specifies the type of data that was loaded into the VBO; it will be either gl.UNSIGNED_SHORT or gl.UNSIGNED_BYTE. The startByte is the starting point in the VBO of the data for the primitive; it is usually zero. (Note that the starting point is given in terms of bytes, not vertex numbers.) A typical example would be

```javascript
gl.drawElements( gl.TRIANGLES, 9, gl.UNSIGNED_BYTE, 0 );
```

We will have occasion to use this function later. If you find it confusing, you should review Subsection 3.4.2. The situation is much the same in WebGL as it was in OpenGL 1.1.