OpenGL® ES 2.0 Programming Guide

Aaftab Munshi ■ Dan Ginsburg ■ Dave Shreiner

Foreword by Neil Trevett, President, Khronos Group

OpenGL® ES 2.0 Programming Guide

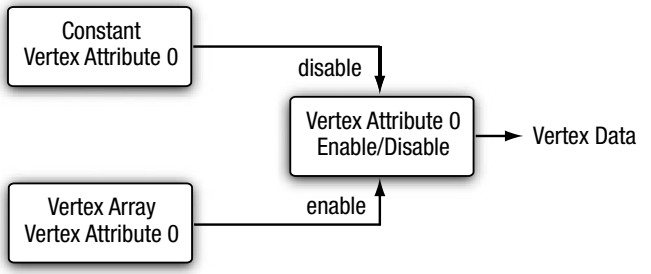

Figure 6-3 Selecting Constant or Vertex Array Vertex Attribute

The commands `glEnableVertexAttribArray` and `glDisableVertex-AttribArray` are used to enable and disable a generic vertex attribute array. If the vertex attribute array is disabled for a generic attribute index, the constant vertex attribute data specified for that index will be used.

```
void   glEnableVertexAttribArray(GLuint index);
void   glDisableVertexAttribArray(GLuint index);
```

index	specifies the generic vertex attribute index. This value is 0 to max vertex attributes supported – 1

Example 6-3 describes how to draw a triangle where one of the vertex attributes is constant and the other is specified using a vertex array.

Example 6-3 Using Constant and Vertex Array Attributes

```
GLbyte vertexShaderSrc[] =
      "attribute vec4 a_position;    \n"
      "attribute vec4 a_color;       \n"
      "varying vec4   v_color;       \n"
      "void main()                   \n"
      "{                             \n"
      "    v_color = a_color;        \n"
      "    gl_Position = a_position; \n"
      "}";

GLbyte fragmentShaderSrc[] =
      "varying vec4 v_color;         \n"
      "void main()                   \n"
```

```
      "{                                \n"
      "    gl_FragColor = v_color;      \n"
      "}";

GLfloat   color[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
GLfloat   vertexPos[3 * 3];  // 3 vertices, with (x,y,z) per-vertex
GLuint    shaderObject[2];
GLuint    programObject;

shaderObject[0] = LoadShader(vertexShaderSrc, GL_VERTEX_SHADER);
shaderObject[1] = LoadShader(fragmentShaderSrc, GL_FRAGMENT_SHADER);

programObject = glCreateProgram();
glAttachShader(programObject, shaderObject[0]);
glAttachShader(programObject, shaderObject[1]);

glVertexAttrib4fv(0, color);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, vertexPos);
glEnableVertexAttribArray(1);

glBindAttribLocation(programObject, 0, "a_color");
glBindAttribLocation(programObject, 1, "a_position");

glLinkProgram(programObject);
glUseProgram(programObject);

glDrawArrays(GL_TRIANGLES, 0, 3);
```

The vertex attribute `color` used in the code example is a constant value whereas the `vertexPos` attribute is specified using a vertex array. The value of `color` will be the same for all vertices of the triangle(s) drawn whereas the `vertexPos` attribute could vary for vertices of the triangle(s) drawn.

Declaring Vertex Attribute Variables in a Vertex Shader

We have looked at what a vertex attribute is, and how to specify vertex attributes in OpenGL ES. We now discuss how to declare vertex attribute variables in a vertex shader.

In a vertex shader, a variable is declared as a vertex attribute by using the `attribute` qualifier. The `attribute` qualifier can only be used in a vertex shader. If the `attribute` qualifier is used in a fragment shader, it should result in an error when the fragment shader is compiled.

A few example declarations of vertex attributes are given here.

```
attribute vec4   a_position;
attribute vec2   a_texcoord;
attribute vec3   a_normal;
```

The `attribute` qualifier can be used only with the data types `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, and `mat4`. Attribute variables cannot be declared as arrays or structures. The following example declarations of vertex attributes are invalid and should result in a compilation error.

```
attribute foo_t  a_A;   // foo_t is a structure
attribute vec4   a_B[10];
```

An OpenGL ES 2.0 implementation supports `GL_MAX_VERTEX_ATTRIBS` `vec4` vertex attributes. A vertex attribute that is declared as a `float` or `vec2` or `vec3` will count as one `vec4` attribute. Vertex attributes declared as `mat2`, `mat3`, or `mat4` will count as two, three, or four `vec4` attributes, respectively. Unlike uniform and varying variables, which get packed automatically by the compiler, attributes do not get packed. Each component is stored internally by the implementation as a 32-bit single precision floating-point value. Please consider carefully when declaring vertex attributes with sizes less than `vec4`, as the maximum number of vertex attributes available is a limited resource. It might be better to pack them together into one `vec4` attribute instead of declaring them as individual vertex attributes in the vertex shader.

Variables declared as vertex attributes in a vertex shader are *read-only* variables and cannot be modified. The following code should cause a compilation error.

```
attribute vec4   a_pos;
uniform   vec4   u_v;

void main()
{
   a_pos = u_v; <--- cannot assign to a_pos as it is read-only
}
```

An attribute can be declared inside a vertex shader but if it is not used then it is not considered active and does not count against the limit. If the number of attributes used in a vertex shader is greater than `GL_MAX_VERTEX_ATTRIBS`, the vertex shader will fail to link.

Once a program has been successfully linked, we need to find out the number of active vertex attributes used by the vertex shader attached to this

program. The following line of code describes how to get the number of active vertex attributes.

```
glGetProgramiv(progam, GL_ACTIVE_ATTRIBUTES, &numActiveAttribs);
```

A detailed description of `glGetProgamiv` is given in Chapter 4, "Shaders and Programs."

The list of active vertex attributes used by a program and their data types can be queried using the `glGetActiveAttrib` command.

```
void   glGetActiveAttrib(GLuint program, GLuint index,
                         GLsizei bufsize, GLsizei *length,
                         GLint *size, GLenum *type,
                         GLchar *name)
```

---

program	name of a program object that was successfully linked previously
index	specifies the vertex attribute to query and will be a value between 0 … GL_ACTIVE_ATTRIBUTES – 1. The value of GL_ACTIVE_ATTRIBUTES is determined with glGetProgramiv
bufsize	specifies the maximum number of characters that may be written into name, including the null terminator
length	returns the number of characters written into name excluding the null terminator, if length is not NULL
type	returns the type of the attribute. Valid values are: GL_FLOAT GL_FLOAT_VEC2 GL_FLOAT_VEC3 GL_FLOAT_VEC4 GL_FLOAT_MAT2 GL_FLOAT_MAT3 GL_FLOAT_MAT4
size	returns the size of the attribute. This is specified in units of the type returned by *type*. If the variable is not an array, *size* will always be 1. If the variable is an array, then *size* returns the size of the array
name	name of the attribute variable as declared in the vertex shader

The glGetActiveAttrib call provides information about the attribute selected by *index*. As described above, *index* must be a value between 0 and GL_ACTIVE_ATTRIBUTES – 1. The value of GL_ACTIVE_ATTRIBUTES is queried using glGetProgramiv. An index of 0 selects the first active attributes and an *index* of GL_ACTIVE_ATTRIBUTES – 1 selects the last vertex attribute.

Binding Vertex Attributes to Attribute Variables in a Vertex Shader

We discussed that in a vertex shader, vertex attribute variables are specified by the attribute qualifier, the number of active attributes can be queried using glGetProgramiv and the list of active attributes in a program can be queried using glGetActiveAttrib. We also described that generic attribute indices that range from 0 to (GL_MAX_VERTEX_ATTRIBS – 1) are used to enable a generic vertex attribute and specify a constant or per-vertex (i.e., vertex array) value using the glVertexAttrib* and glVertexAttrib-Pointer commands. Now we describe how to map this generic attribute index to the appropriate attribute variable declared in the vertex shader. This mapping will allow appropriate vertex data to be read into the correct vertex attribute variable in the vertex shader.

Figure 6-4 describes how generic vertex attributes are specified and bound to attribute names in a vertex shader.

There are two approaches that OpenGL ES 2.0 enables to map a generic vertex attribute index to an attribute variable name in the vertex shader. These approaches can be categorized as follows:

- OpenGL ES 2.0 will bind the generic vertex attribute index to the attribute name.
- The application can bind the vertex attribute index to an attribute name.

The glBindAttribLocation command can be used to bind a generic vertex attribute index to an attribute variable in a vertex shader. This binding takes effect when the program is linked the next time. It does not change the bindings used by the currently linked program.

```
void   glBindAttribLocation(GLuint program, GLuint index,
                            const GLchar *name)
```

program	name of a program object
index	generic vertex attribute index
name	name of the attribute variable

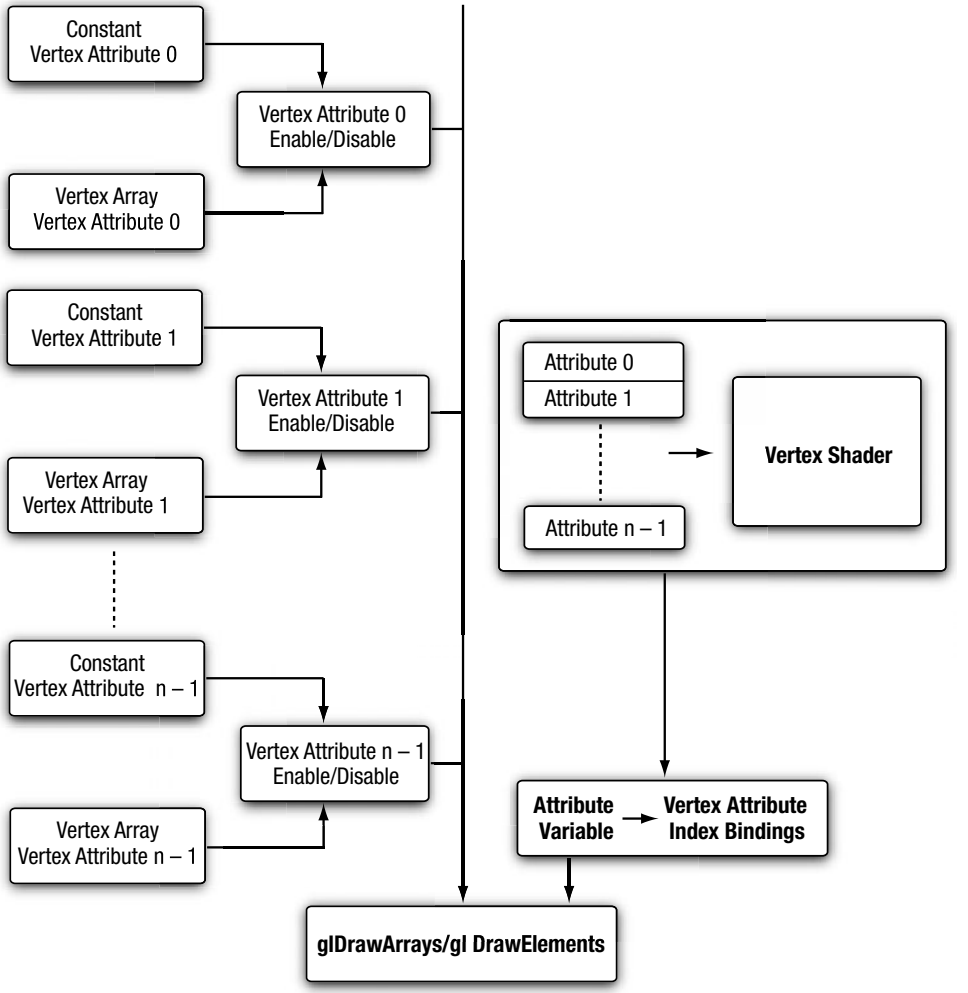

Figure 6-4 Specifying and Binding Vertex Attributes for Drawing a Primitive(s)