OpenGL® ES™ 3.0 Programming Guide

Second Edition

Dan Ginsburg ▪ Budirijanto Purnomo

With Earlier Contributions from Dave Shreiner and Aaftab Munshi

Foreword by Neil Trevett, President, Khronos Group

Praise for *OpenGL® ES™ 3.0 Programming Guide, Second Edition*

"As a graphics technologist and intense OpenGL ES developer, I can honestly say that if you buy only one book on OpenGL ES 3.0 programming, then this should be the book. Dan and Budirijanto have written a book clearly by programmers for programmers. It is simply required reading for anyone interested in OpenGL ES 3.0. It is informative, well organized, and comprehensive, but best of all practical. You will find yourself reaching for this book over and over again instead of the actual OpenGL ES specification during your programming sessions. I give it my highest recommendation."

—Rick Tewell, Graphics Technology Architect, Freescale

"This book provides outstanding coverage of the latest version of OpenGL ES, with clear, comprehensive explanations and extensive examples. It belongs on the desk of anyone developing mobile applications."

—Dave Astle, Graphics Tools Lead, Qualcomm Technologies, Inc., and Founder, GameDev.net

"The second edition of *OpenGL® ES™ 3.0 Programming Guide* provides a solid introduction to OpenGL ES 3.0 specifications, along with a wealth of practical information and examples to help any level of developer begin programming immediately. We'd recommend this guide as a primer on OpenGL ES 3.0 to any of the thousands of developers creating apps for the many mobile and embedded products using our PowerVR Rogue graphics."

—Kristof Beets, Business Development, Imagination Technologies

"This is a solid OpenGL ES 3.0 reference book. It covers all aspects of the API and will help any developer get familiar with and understand the API, including specifically the new ES 3.0 functionality."

—Jed Fisher, Managing Partner, 4D Pipeline

"This is a clear and thorough reference for OpenGL ES 3.0, and an excellent presentation of the concepts present in all modern OpenGL programming. This is the guide I'd want by my side when diving into embedded OpenGL."

—Todd Furlong, President & Principal Engineer, Inv3rsion LLC

Example 6-1 Array of Structures *(continued)*

```
                        VERTEX_TEXC00RD1_SIZE,
                        GL_FLOAT, GL_FALSE,
                        VERTEX_ATTRIB_SIZE * sizeof(float),
                          (p + VERTEX_TEXC00RD1_0FFSET));
```

In Example 6-2, `position`, `normal`, and texture coordinates 0 and 1 are stored in separate buffers.

Example 6-2 Structure of Arrays

```
float *position  = (float*) malloc(numVertices *
   VERTEX_POS_SIZE * sizeof(float));
float *normal    = (float*) malloc(numVertices *
   VERTEX_NORMAL_SIZE * sizeof(float));
float *texcoordO = (float*) malloc(numVertices *
   VERTEX_TEXCOORD0_SIZE * sizeof(float));
float *texcoordl = (float*) malloc(numVertices *
   VERTEX_TEXC00RD1_SIZE * sizeof(float));

// position is vertex attribute 0
glVertexAttribPointer(VERTEX_POS_INDX, VERTEX_POS_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_POS_SIZE * sizeof(float),
                      position);

// normal is vertex attribute 1
glVertexAttribPointer(VERTEX_NORMAL_INDX, VERTEX_NORMAL_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_NORMAL_SIZE * sizeof(float),
                      normal);

// texture coordinate 0 is vertex attribute 2
glVertexAttribPointer(VERTEX_TEXCOORDO_INDX,
                      VERTEX_TEXCOORD0_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_TEXCOORD0_SIZE *
                      sizeof(float), texcoordO);

// texture coordinate 1 is vertex attribute 3
glVertexAttribPointer(VERTEX_TEXCOORDl_INDX,
                      VERTEX_TEXC00RD1_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_TEXC00RD1_SIZE * sizeof(float),
                      texcoordl);
```

Performance Hints

How to Store Different Attributes of a Vertex

We described the two most common ways of storing vertex attributes: an *array of structures* and a *structure of arrays*. The question to ask is which allocation method would be the most efficient for OpenGL ES 3.0 hardware implementations. In most cases, the answer is an *array of structures*. The reason is that the attribute data for each vertex can be read in sequential fashion, which will most likely result in an efficient memory access pattern. A disadvantage of using an array of structures becomes apparent when an application wants to modify specific attributes. If a subset of vertex attribute data needs to be modified (e.g., texture coordinates), this will result in strided updates to the vertex buffer. When the vertex buffer is supplied as a buffer object, the entire vertex attribute buffer will need to be reloaded. You can avoid this inefficiency by storing vertex attributes that are dynamic in nature in a separate buffer.

Which Data Format to Use for Vertex Attributes

The vertex attribute data format specified by the `type` argument in `glVertexAttribPointer` can affect not only the graphics memory storage requirements for vertex attribute data, but also the overall performance, which is a function of the memory bandwidth required to render the frame(s). The smaller the data footprint, the lower the memory bandwidth required. OpenGL ES 3.0 supports a 16-bit floating-point vertex format named `GL_HALF_FLOAT` (described in detail in Appendix A). Our recommendation is that applications use `GL_HALF_FLOAT` wherever possible. Texture coordinates, normals, binormals, tangent vectors, and so on are good candidates to be stored using `GL_HALF_FLOAT` for each component. Color could be stored as `GL_UNSIGNED_BYTE` with four components per vertex color. We also recommend `GL_HALF_FLOAT` for vertex position, but recognize that this choice might not be feasible for quite a few cases. For such cases, the vertex position could be stored as `GL_FLOAT`.

How the Normalized Flag in glVertexAttribPointer Works

Vertex attributes are internally stored as a single-precision floating-point number before being used in a vertex shader. If the data *type* indicates that the vertex attribute is not a float, then the vertex attribute will be converted to a single-precision floating-point number before it is used in a vertex shader. The *normalized* flag controls the conversion of the non-float vertex attribute data to a single precision floating-point value. If the *normalized* flag is false, the vertex data are converted directly to a

floating-point value. This would be similar to casting the variable that is not a float type to float. The following code gives an example:

```
GLfloat  f;
GLbyte b;
f = (GLfloat)b;  // f represents values in the range [-128.0,
                 // 127.0]
```

If the *normalized* flag is true, the vertex data is mapped to the [–1.0, 1.0] range if the data *type* is `GL_BYTE`, `GL_SHORT`, or `GL_FIXED`, or to the [0.0, 1.0] range if the data *type* is `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT`.

Table 6-1 describes conversion of non-floating-point data types with the normalized flag set. The value *c* in the second column of Table 6-1 refers to a value of the format specified in the first column.

Table 6-1 Data Conversions

Vertex Data Format	Conversion to Floating Point
`GL_BYTE`	$\max(c / (2^7 - 1), -1.0)$
`GL_UNSIGNED_BYTE`	$c / (2^8 - 1)$
`GL_SHORT`	$\max(c / (2^{15} - 1), -1.0)$
`GL_UNSIGNED_SHORT`	$c / (2^{16} - 1)$
`GL_FIXED`	$c/2^{16}$
`GL_FLOAT`	c
`GL_HALF_FLOAT_OES`	c

It is also possible to access integer vertex attribute data as integers in the vertex shader rather than having them be converted to floats. In this case, the `glVertexAttribIPointer` function should be used and the vertex attribute should be declared to be of an integer type in the vertex shader.

Selecting Between a Constant Vertex Attribute or a Vertex Array

The application can enable OpenGL ES to use either the constant data or data from vertex array. Figure 6-3 describes how this works in OpenGL ES 3.0.

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

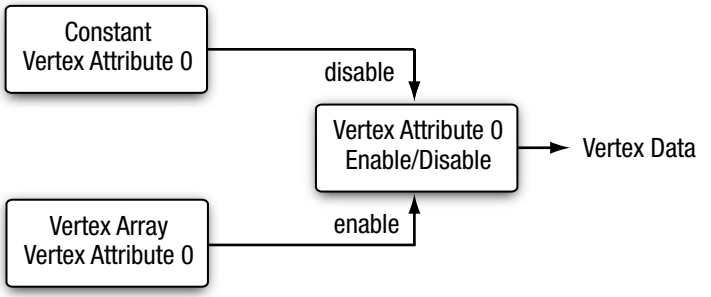

Figure 6-3 Selecting Constant or Vertex Array Vertex Attribute

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

index	specifies the generic vertex attribute index. This value ranges from 0 to the maximum vertex attributes supported minus 1.

Example 6-3 illustrates 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

```
int Init ( ESContext *esContext )
{
   UserData *userData = (UserData*) esContext->userData;
   const char vShaderStr[] =
      "#version 300 es                              \n"
      "layout(location = 0) in vec4 a_color;        \n"
      "layout(location = 1) in vec4 a_position;     \n"
      "out vec4 v_color;                            \n"
      "void main()                                  \n"
      "{                                            \n"
      "    v_color = a_color;                       \n"
      "    gl_Position = a_position;                \n"
      "}";


   const char fShaderStr[] =
      "#version 300 es            \n"
      "precision mediump float;   \n"
      "in vec4 v_color;           \n"
      "out vec4 o_fragColor;      \n"
```

(continues)

Example 6-3 Using Constant and Vertex Array Attributes *(continued)*

```
      "void main()                \n"
      "{                          \n"
      "    o_fragColor = v_color; \n"
      "}" ;

   GLuint programObject;

   // Create the program object
   programObject = esLoadProgram ( vShaderStr, fShaderStr );

   if ( programObject == 0 )
      return GL_FALSE;

   // Store the program object
   userData->programObject = programObject;

   glClearColor ( 0.0f, 0.0f, 0.0f, 0.0f );
   return GL_TRUE;
}

void Draw ( ESContext *esContext )
{
   UserData *userData = (UserData*) esContext->userData;
   GLfloat color[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
   // 3 vertices, with (x, y, z) per-vertex
   GLfloat vertexPos[3 * 3] =
   {
       0.0f,  0.5f, 0.0f, // v0
      -0.5f, -0.5f, 0.0f, // v1
       0.5f, -0.5f, 0.0f  // v2
   };

   glViewport ( 0, 0, esContext->width, esContext->height );

   glClear ( GL_COLOR_BUFFER_BIT );

   glUseProgram ( userData->programObject );

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

   glDrawArrays ( GL_TRIANGLES, 0, 3 );

   glDisableVertexAttribArray ( 1 );
}
```

The vertex attribute `color` used in the code example is a constant value specified with `glVertexAttrib4fv` without enabling the vertex attribute array 0. The `vertexPos` attribute is specified by using a vertex array with `glVertexAttribPointer` and enabling the array with `glEnableVertexAttribArray`. 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 considered 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 `in` qualifier. Optionally, the attribute variable can also include a layout qualifier that provides the attribute index. A few example declarations of vertex attributes are given here:

```
layout(location = 0) in vec4   a_position;
layout(location = 1) in vec2   a_texcoord;
layout(location = 2) in vec3   a_normal;
```

The `in` qualifier can be used only with the data types `float`, `vec2`, `vec3`, `vec4`, `int`, `ivec2`, `ivec3`, `ivec4`, `uint`, `uvec2`, `uvec3`, `uvec4`, `mat2`, `mat2x2`, `mat2x3`, `mat2x4`, `mat3`, `mat3x3`, `mat3x4`, `mat4`, `mat4x2`, and `mat4x3`. 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:

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

An OpenGL ES 3.0 implementation supports `GL_MAX_VERTEX_ATTRIBS` four-component vector vertex attributes. A vertex attribute that is declared as a scalar, two-component vector, or three-component vector will count as a single four-component vector attribute. Vertex attributes declared as two-dimensional, three-dimensional, or four-dimensional matrices will count as two, three, or four 4-component vector attributes, respectively. Unlike uniform and vertex shader output/fragment shader input variables, which are packed automatically by the compiler, attributes do not get packed. Please consider your choices carefully when declaring vertex attributes with sizes less than a four-component vector, as the maximum number of vertex attributes available is a limited resource. It might be better to pack