An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 18 Vertex Array

18.1 Function Calls

We have discussed that we can specify the shapes of a graphics object by calling a sequence of **glVertex*** commands, which are enclosed within a pair of **glBegin/glEnd** functions. Actually, each **glVertex*** command is also a function. This is not an efficient method because function calls are expensive as they involve pushing and popping parameters onto a stack, and this method involves a lot of redundant function calls; the shared vertices have to be specified again and again. For example, consider rendering a cube, which is shown in Figure 18-1 below. A cube has 6 faces and 8 vertices. We need 4 vertices to specify a face. To render a cube using **glBegin/glEnd**, we will process a total of $6 \times 4 = 24$ vertices though only 8 different vertices exist.

To avoid redundant function calls, OpenGL provides vertex array routines to process arrays of vertices with fewer function calls. Vertex arrays make graphics programming more efficient and effective.

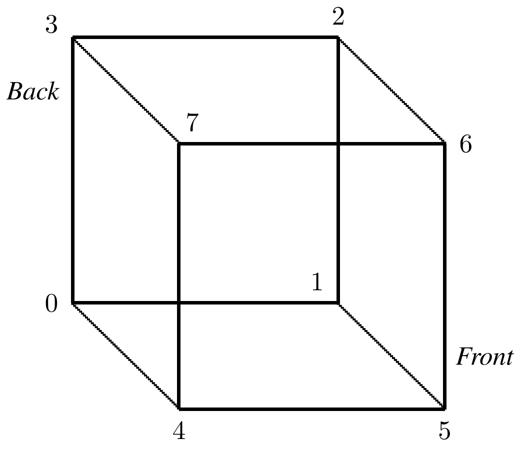

Figure 18-1 A Cube With Indexed Vertices

18.2 Vertex Array Process

There are three steps in the process of using vertex array routines:

1. **Enabling Arrays**.
   We enable the usage of vertex arrays with the command **glEnableClientState**, which has protocol:

void glEnableClientState(GLenum array)
Specifies the array to enable. Acceptable Symbolic constants: GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY, and GL_EDGE_FLAG_ARRAY

   For example, if we use lighting, we need to define a surface normal for each vertex. In this situation, we have to activate both the surface normal and vertex coordinate arrays with:

```
glEnableClientState ( GL_NORMAL_ARRAY );
glEnableClientState ( GL_VERTEX_ARRAY );
```

The lighting effect can be turned off by:

```
glDisableClientState(GL_NORMAL_ARRAY);
```

2. **Specifying Data for the Arrays**.
   We specify the data for a vertex array with the function **glVertexPointer**:

void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);
Specifies where spatial coordinate data can be accessed; *pointer* is the memory address of the first coordinate of the first vertex in the array; *type* specifies the data type (GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE) of each coordinate in the array; *size* is the number of coordinates per vertex, which must be 2, 3, or 4; *stride* is the byte offset between consecutive vertexes; if *stride* is 0, the vertices are tightly packed in the array.

   Commands for specifying other array data include:

```
void glColorPointer(GLint size,GLenum type,GLsizei stride,
                                        const GLvoid *pointer);
void glIndexPointer(GLenum type,GLsizei stride,
                                        const GLvoid *pointer);
void glNormalPointer(GLenum type,GLsizei stride,
                                        const GLvoid *pointer);
void glTexCoordPointer(GLint size,GLenum type,GLsizei stride,
                                        const GLvoid *pointer);
void glEdgeFlagPointer(GLsizei stride, const GLvoid *pointer);
```

   The following table shows various vertex array sizes (Values per Vertex) and data types:

Command	**Sizes**	**Values for type Argument**
glVertexPointer	2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glNormalPointer	3	GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glColorPointer	3, 4	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glIndexPointer	1	GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glTexCoordPointer	1, 2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glEdgeFlagPointer	1	no type argument (type of data must be GLboolean)

   The following is an example of enabling and loading vertex arrays, which specify 6 vertices, each with 2 coordinates and 6 different colors, each with 3 values for the RGB components:

```
int vertices[] = { 20, 20,          100, 300,
                  180, 20,          180, 300,
                  245, 20,         300, 300};
float colors[] = {0.8, 0.2, 0.2,    0.2, 0.8, 1.0,
                  0.6, 1.0, 0.2,    0.8, 0.8, 0.6,
                  0.3, 0.3, 0.3,    0.6, 0.6, 0.6};

glEnableClientState (GL_COLOR_ARRAY);
glEnableClientState (GL_VERTEX_ARRAY);

glColorPointer (3, GL_FLOAT, 0, colors);
glVertexPointer (2, GL_INT, 0, vertices);
```

In the above example, the *stride* values are 0, meaning that the data are closely packed. To show the usage of the parameter *stride*, which tells OpenGL how to access the data we provide, we consider another example where the vertex coordinates and color values are stored in the same array:

```
   float mixed_data[] = { 20.0, 20.0,0.0,    0.8,0.2,0.2,
                         100.0,300.0,0.0,    0.2,0.8,1.0,
                         180.0, 20.0,0.0,    0.6,1.0,0.2,
                         180.0,300.0,0.0,    0.8,0.8,0.6,
                         245.0, 20.0,0.0,    0.3,0.3,0.3,
                         300.0,300.0,0.0,    0.6,0.6,0.6};
```

In this *mixed_data* array, the left column contains the vertex coordinates and the right column contains the corresponding color values. The *stride* parameter allows a vertex array to access its desired data at regular intervals in the array, and its value should be the number of bytes between the starts of two successive pointer elements, or zero, which is the special case when the data are tightly packed. So in this example, we provide access to the vertex coordinates with the command:

```
glVertexPointer(3,GL_FLOAT,6*sizeof(float),mixed_data);
```

For the color values, we start from the fourth element of the array *mixed_data*. This can can be accomplished by the command:

```
glColorPointer(3,GL_FLOAT,6*sizeof(float),mixed_data+3);
```

3. **Dereferencing and Rendering**.
   We dereference **a single array element** with **glArrayElement**:

void glArrayElement(GLint ith)
Obtains the data of one (the *ith*) vertex for all currently enabled arrays. For the vertex coordinate array, the corresponding command would be **glVertex**[*size*][*type*]v(), where *size* is one of [2,3,4], and *type* is one of [s,i,f,d] for GLshort, GLint, GLfloat, and GLdouble respectively. Both *size* and *type* were defined by **glVertexPointer**(). For other enabled arrays, **glArrayElement**() calls **glEdgeFlagv**(), **glTexCoord**[*size*][*type*]v(), **glColor**[*size*][*type*]v(), **glIndex**[*type*]v(), and **glNormal**[*type*]v(). If the vertex coordinate array is enabled, the **glVertex*v**() routine is executed last, after the execution (if enabled) of up to five corresponding array values.

The following is an example of drawing a triangle using this command:

```
glEnableClientState (GL_COLOR_ARRAY);
glEnableClientState (GL_VERTEX_ARRAY);
glColorPointer (3, GL_FLOAT, 0, colors);
glVertexPointer (2, GL_INT, 0, vertices);

glBegin(GL_TRIANGLES);
  glArrayElement (2);
  glArrayElement (3);
  glArrayElement (4);
glEnd();
```

When we execute the above code, the last five statements has the same effect as the following code:

```
glBegin(GL_TRIANGLES);
  glColor3fv(colors+(2*3));
  glVertex2iv(vertices+(2*2));
  glColor3fv(colors+(3*3));
  glVertex2iv(vertices+(3*2));
  glColor3fv(colors+(4*3));
  glVertex2iv(vertices+(4*2));
glEnd();
```

We dereference **a list of array elements** with **glDrawElements**:

void glDrawElements(GLenum mode, GLsizei count, GLenum type, void *indices)
Specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL function to pass each vertex attribute, we can use **glVertexAttribPointer** to prespecify separate arrays of vertex attributes and use them to construct a sequence of primitives with a single call to **glDrawElements**. When **glDrawElements** is called, it uses count sequential elements from an enabled array, starting at *indices* to construct a sequence of geometric primitives; *type* specifies the type of *indices* values, which must be GL_UNSIGNED_BYTE or GL_UNSIGNED_SHORT. *mode* specifies what kind of primitives are constructed and how the array elements construct these primitives (GL_POLYGON, GL_POINTS,...). If more than one array is enabled, each is used. We can call **glEnableVertexAttribArray** and **glDisableVertexAttribArray** to enable and disable a generic vertex attribute array.

This command almost has the same effect as:

```
int i;
glBegin (mode);
  for (i = 0; i < count; i++)
     glArrayElement(indices[i]);
glEnd();
```

18.3 Drawing a Cube

As an example of illustrating the usage of the vertex array commands discussed above, we discuss drawing a colored cube here. Suppose the vertices of the cube is numbered as shown in Figure 18-1 above, and the vertex and color data are defined by:

```
    GLint vertices[] = {-1, -1, -1,   //vertex 0
                         1, -1, -1,   //vertex 1
                         1,  1, -1,   // 2
                        -1,  1, -1,   // 3
                        -1, -1,  1,   // 4
                         1, -1,  1,   // 5
                         1,  1,  1,   // 6
                        -1,  1,  1};  // 7

     GLfloat colors[] = {1.0, 0.2, 0.2,  //color at vertex 0
                         0.2, 0.2, 1.0,  //color at vertex 1
                         0.8, 1.0, 0.2,
                         0.7, 0.7, 0.7,
                         0.3, 0.3, 0.3,
                         0.5, 0.5, 0.5,
                         1.0, 0.0, 0.0,
                         0.0, 1.0, 0.0}; //color at vertex 7

    glVertexPointer (3, GL_INT, 0, vertices);
    glColorPointer (3, GL_FLOAT, 0, colors);
```

Suppose we have made the appropriate setup of viewing and have enabled the usage of vertex array. We can render the cube with the commands:

```
   glBegin( GL_QUADS );
     glArrayElement ( 4 );    // front face (see notes)
     glArrayElement ( 5 );
     glArrayElement ( 6 );
     glArrayElement ( 7 );

     glArrayElement ( 0 );    // back face
     glArrayElement ( 3 );
     glArrayElement ( 2 );
     glArrayElement ( 1 );

     glArrayElement ( 1 );    // right face
     glArrayElement ( 2 );
     glArrayElement ( 6 );
     glArrayElement ( 5 );
     .....
   glEnd();
```

In this method, we have to call **glArrayElement** 24 times. A better method is to define the indices of each face and draw it with the **glDrawElements** command:

```
  GLubyte frontIndices[] = {4, 5, 6, 7};
  GLubyte backIndices[]  = {0, 3, 2, 1};
  GLubyte rightIndices[] = {1, 2, 6, 5};
  GLubyte leftIndices[]  = {0, 4, 7, 3};
  GLubyte topIndices[]   = {2, 3, 7, 6};
  GLubyte bottomIndices[]= {0, 1, 5, 4};

  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);
  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);
  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);
  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

Using **glDrawElements**, we only need to call the function 6 times. Also note that we do not need to enclose the functions with the **glBegin/End** pair. Better still, we can put all indices in one array and draw the cube with only one command:

```
GLubyte allIndices[] = {4, 5, 6, 7, 0, 3, 2, 1, 1, 2, 6, 5,
                        0, 4, 7, 3, 2, 3, 7, 6, 0, 1, 5, 4};
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIndices);
```

Figure 18-2 shows two cubes that are rendered with these methods.

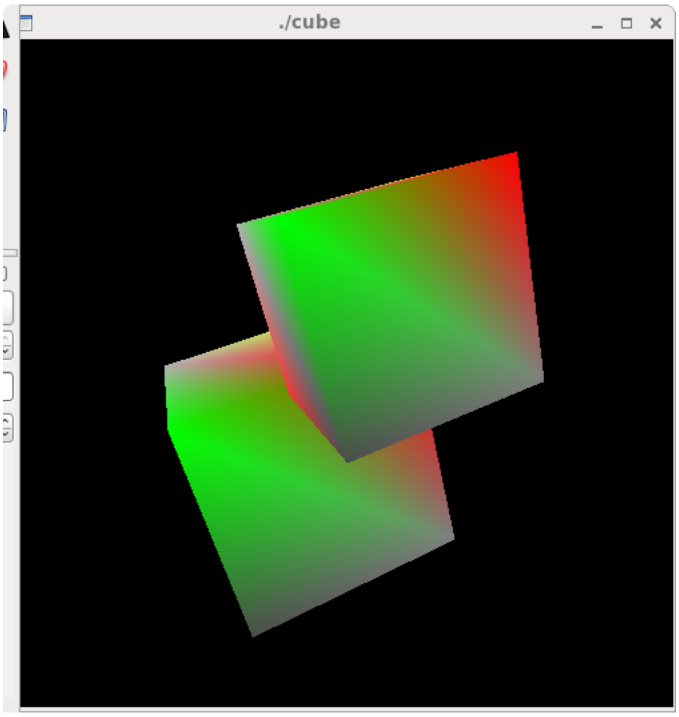

Figure 18-2 Two Colored Cubes Rendered Usinge **glDrawElements**