OpenGL Vertex Array

Overview

Instead you specify individual vertex data in immediate mode (between *glBegin()* and *glEnd()* pairs), you can store vertex data in a set of arrays including vertex positions, normals, texture coordinates and color information. And you can draw only a selection of geometric primitives by dereferencing the array elements with array indices.

Take a look at the following code to draw a cube with immediate mode.

Each face needs 6 times of glVertex*() calls to make 2 triangles, for example, the front face has v0-v1-v2 and v2-v3-v0 triangles. A cube has 6 faces, so the total number of glVertex*() calls is 36. If you also specify normals, texture coordinates and colors to the corresponding vertices, it increases the number of OpenGL function calls.

The other thing that you should notice is the vertex "v0" is shared with 3 adjacent faces; front, right and top face. In immediate mode, you have to provide this shared vertex 6 times, twice for each side as shown in the code.

```
glBegin(GL_TRIANGLES);  // draw a cube with 12 triangles

    // front face =================
    glVertex3fv(v0);    // v0-v1-v2
    glVertex3fv(v1);
    glVertex3fv(v2);

    glVertex3fv(v2);    // v2-v3-v0
    glVertex3fv(v3);
    glVertex3fv(v0);

    // right face =================
    glVertex3fv(v0);    // v0-v3-v4
    glVertex3fv(v3);
    glVertex3fv(v4);

    glVertex3fv(v4);    // v4-v5-v0
    glVertex3fv(v5);
    glVertex3fv(v0);

    // top face ===================
    glVertex3fv(v0);    // v0-v5-v6
    glVertex3fv(v5);
    glVertex3fv(v6);

    glVertex3fv(v6);    // v6-v1-v0
    glVertex3fv(v1);
    glVertex3fv(v0);

    ...                 // draw other 3 faces

glEnd();
```

Using vertex arrays reduces the number of function calls and redundant usage of shared vertices. Therefore, you may increase the performance of rendering. Here, 3 different

OpenGL functions are explained to use vertex arrays; **glDrawArrays()**, **glDrawElements()** and **glDrawRangeElements()**. Although, better approach is using vertex buffer objects (VBO) or display lists.

Initialization

OpenGL provides **glEnableClientState()** and **glDisableClientState()** functions to activate and deactivate 6 different types of arrays. Plus, there are 6 functions to specify the exact positions(addresses) of arrays, so, OpenGL can access the arrays in your application.

- **glVertexPointer()**: specify pointer to vertex coords array
- **glNormalPointer()**: specify pointer to normal array
- **glColorPointer()**: specify pointer to RGB color array
- **glIndexPointer()**: specify pointer to indexed color array
- **glTexCoordPointer()**: specify pointer to texture cords array
- **glEdgeFlagPointer()**: specify pointer to edge flag array

Each specifying function requires different parameters. Please look at OpenGL API manuals. Edge flags are used to mark whether the vertex is on the boundary edge or not. Hence, the only edges where edge flags are on will be visible if glPolygonMode() is set with GL_LINE.

glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid pointer)*

1. **size**: The number of vertex coordinates, 2 for 2D points, 3 for 3D points.
2. **type**: GL_FLOAT, GL_SHORT, GL_INT or GL_DOUBLE.
3. **stride**: The number of bytes to offset to the next vertex (used for interleaved array).
4. **pointer**: The pointer to the vertex array.

glNormalPointer(GLenum type, GLsizei stride, const GLvoid pointer)*

1. **type**: GL_FLOAT, GL_SHORT, GL_INT or GL_DOUBLE.
2. **stride**: The number of bytes to offset to the next normal (used for interleaved array).
3. **pointer**: The pointer to the vertex array.

Notice that vertex arrays are located in your application(system memory), which is on the client side. And, OpenGL on the server side gets access to them. That is why there are distinctive commands for vertex array; **glEnableClientState()** and **glDisableClientState()** instead of using **glEnable()** and **glDisable()**.

glDrawArrays()

glDrawArrays() reads vertex data from the enabled arrays by marching straight through the array without skipping or hopping. Because **glDrawArrays()** does not allows hopping around the vertex arrays, you still have to repeat the shared vertices once per face.

glDrawArrays() takes 3 arguments. The first thing is the primitive type. The second parameter is the starting offset of the array. The last parameter is the number of vertices to pass to rendering pipeline of OpenGL.
For above example to draw a cube, the first parameter is GL_TRIANGLES, the second is 0,

which means starting from beginning of the array. And the last parameter is 36: a cube has 6 sides and each side needs 6 vertices to draw 2 triangles, 6 × 6 = 36.

```
GLfloat vertices[] = {...}; // 36 of vertex coords
...
// activate and specify pointer to vertex array
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// draw a cube
glDrawArrays(GL_TRIANGLES, 0, 36);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
```

As a result of using **glDrawArrays()**, you can replace 36 **glVertex*()** calls with a single **glDrawArrays()** call. However, we still need to duplicate the shared vertices, so the number of vertices defined in the array is still 36 instead of 8. **glDrawElements()** is the solution to reduce the number of vertices in the array, so it allows transferring less data to OpenGL.

glDrawElements()

glDrawElements() draws a sequence of primitives by hopping around vertex arrays with the associated array indices. It reduces both the number of function calls and the number of vertices to transfer. Furthermore, OpenGL may cache the recently processed vertices and reuse them without resending the same vertices into vertex transform pipeline multiple times.

glDrawElements() requires 4 parameters. The first one is the type of primitive, the second is the number of indices of index array, the third is data type of index array and the last parameter is the address of index array. In this example, the parameters are, GL_TRIANGLES, 36, GL_UNSIGNED_BYTE and indices respectively.

```
GLfloat vertices[] = {...};          // 8 of vertex coords
GLubyte indices[] = {0,1,2, 2,3,0,   // 36 of indices
                     0,3,4, 4,5,0,
                     0,5,6, 6,1,0,
                     1,6,7, 7,2,1,
                     7,4,3, 3,2,7,
                     4,7,6, 6,5,4};
...
// activate and specify pointer to vertex array
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// draw a cube
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indices);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
```

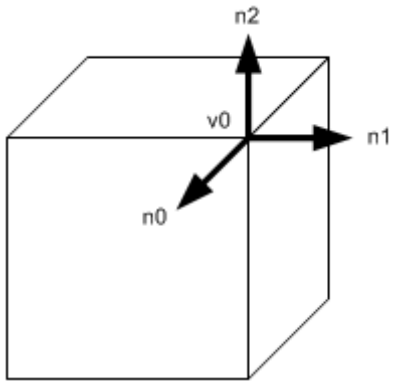
The size of vertex coordinates array is now 8, which is exactly same number of vertices in the cube without any redundant entries.

Note that the data type of index array is GLubyte instead of GLuint or GLushort. It should be the smallest data type that can fit maximum index number in order to reduce the size of index

array, otherwise, it may cause performance drop due to the size of index array. Since the vertex array contains 8 vertices, GLubyte is enough to store all indices. array, otherwise, it may cause performance drop due to the size of index array. Since the vertex array contains 8 vertices, GLubyte is enough to store all indices.

Another thing you should consider is the normal vectors at the shared vertices. If the normals of the adjacent polygons at a shared vertex are all different, then normal vectors should be specified as many as the number of faces, once for each face.

For example, the vertex v0 is shared with the front, right and top face, but, the normals cannot be shared at v0. The normal of the front face is n0, the right face normal is n1 and the top face is n2. For this situation, the normal is not the same at a shared vertex, the vertex cannot be defined only once in vertex array any more. It must be defined multiple times in the array for vertex coordinates in order to match the same amount of elements in the normal array. A typical cube with proper normals requires 24 unique vertices: 6 sides × 4 vertices per side. See the actual implementation in the example code.



Different normals at shared vertex

glDrawRangeElements()

Like **glDrawElements()**, **glDrawRangeElements()** is also good for hopping around vertex array. However, **glDrawRangeElements()** has two more parameters (*start* and *end* index) to specify a range of vertices to be prefetched. By adding this restriction of a range, OpenGL may be able to obtain only limited amount of vertex array data prior to rendering, and may increase performance.

The additional parameters in **glDrawRangeElements()** are *start* and *end* index, then OpenGL prefetches a limited amount of vertices from these values: *end - start + 1*. And the values in index array must lie in between *start* and *end* index. Note that not all vertices in range (*start*, *end*) must be referenced. But, if you specify a sparsely used range, it causes unnecessary process for many unused vertices in that range.

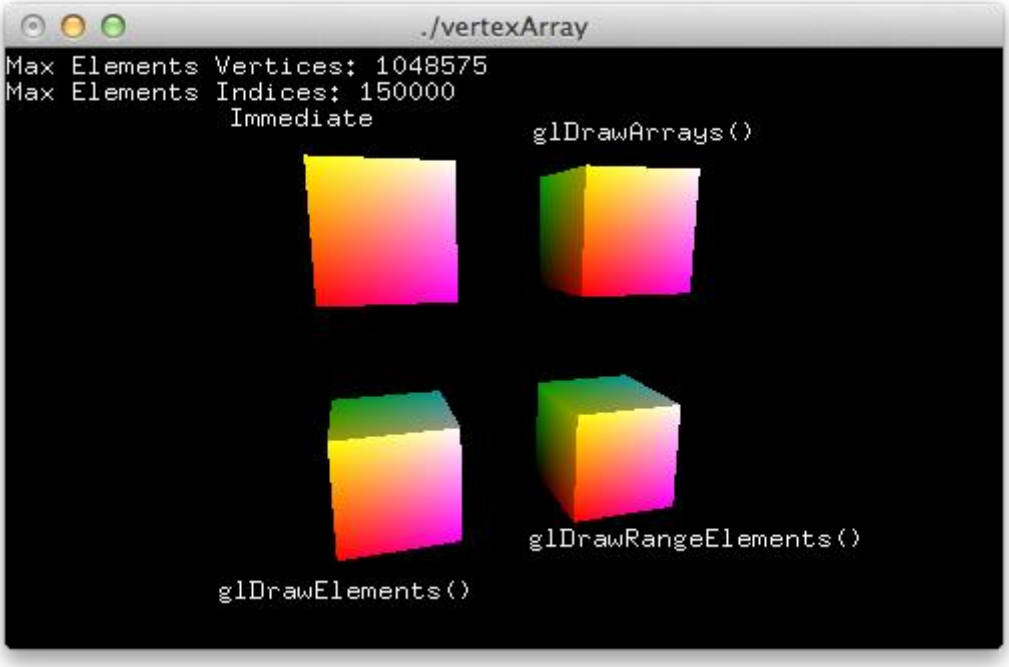
```
GLfloat vertices[] = {...};           // 8 of vertex coords
GLubyte indices[] = {0,1,2, 2,3,0,    // first half (18 indices)
                     0,3,4, 4,5,0,
                     0,5,6, 6,1,0,

                     1,6,7, 7,2,1,    // second half (18 indices)
                     7,4,3, 3,2,7,
                     4,7,6, 6,5,4};
...
```

```
// activate and specify pointer to vertex array
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// draw first half, range is 6 - 0 + 1 = 7 vertices used
glDrawRangeElements(GL_TRIANGLES, 0, 6, 18, GL_UNSIGNED_BYTE, indices);

// draw second half, range is 7 - 1 + 1 = 7 vertices used
glDrawRangeElements(GL_TRIANGLES, 1, 7, 18, GL_UNSIGNED_BYTE, indices+18);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
```

You can find out maximum number of vertices to be prefetched and the maximum number of indices to be referenced by using **glGetIntegerv()** with GL_MAX_ELEMENTS_VERTICES and GL_MAX_ELEMENTS_INDICES.

Note that glDrawRangeElements() is available OpenGL version 1.2 or greater.

Example



This demo application renders a cube with 4 different ways; immediate mode, **glDrawArrays()**, **glDrawElements()** and **glDrawRangeElements()**.

- **draw1()**: Draw a cube with immediate mode.
- **draw2()**: Draw a cube with glDrawArrays().
- **draw3()**: Draw a cube with glDrawElements().
- **draw4()**: Draw a cube with glDrawRangeElements().
- **draw5()**: Draw a cube with glDrawElements() and interleaved vertex array.

Download the source and binary: vertexArray.zip.

In order to run this program properly, video card must support OpenGL v1.2 or greater because of glDrawRangeElements(). Make sure your video driver supports version 1.2 or higher with glinfo.

This file contains a project file for Code::Blocks and Dev-C++, plus makefiles for Mac and Linux system. For example, you can compile the source code on Linux system by typing the following command in the terminal;

```
> make -f Makefile.linux
```

Update: Since OpenGL v3.1+ and ES (Embedded Systems) do not support **GL_QUADS** primitive, this article is modified using **GL_TRIANGLES** instead.