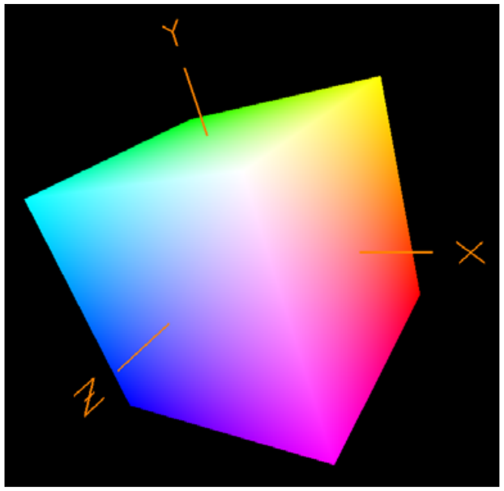
Vertex Buffer Objects

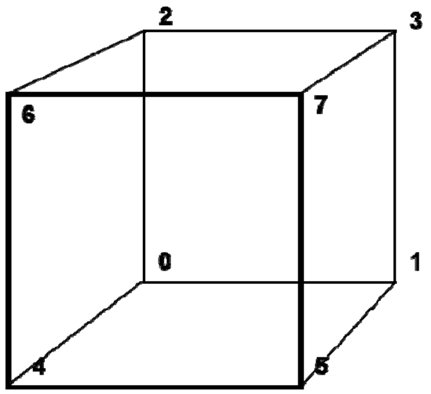
Mike Bailey

Oregon State University

Vertex Buffer Objects: The Big Idea

- Store vertex coordinates and vertex attributes on the ***graphics card***.
- Optionally store the connections on the graphics card too.
- Every time you go to redraw, coordinates will be pulled from GPU memory, avoiding a potentially significant amount of bus latency.



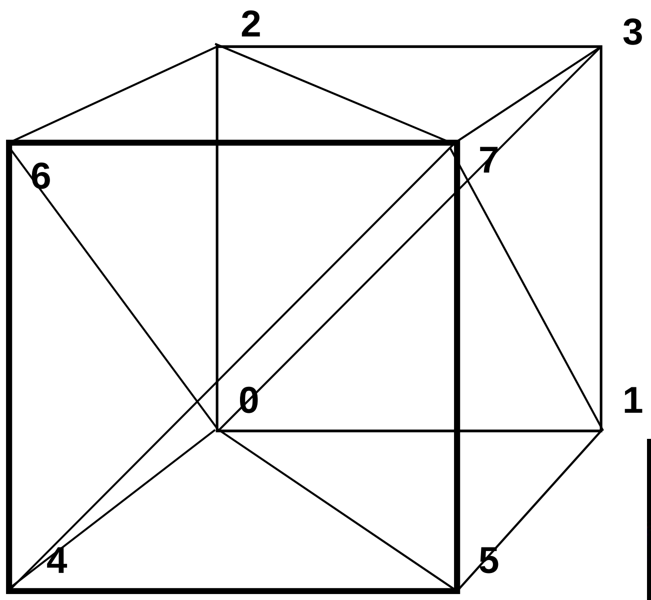


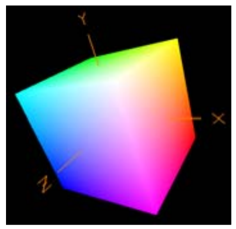
```
GLfloat  CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        {  1., -1., -1. },
        { -1.,  1., -1. },
        {  1.,  1., -1. },
        { -1., -1.,  1. },
        {  1., -1.,  1. },
        { -1.,  1.,  1. },
        {  1.,  1.,  1. }
};
```

```
GLfloat  CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```

```
GLuint CubeIndices[ ][4] =
{
        { 0, 2, 3, 1 },
        { 4, 5, 7, 6 },
        { 1, 3, 7, 5 },
        { 0, 4, 6, 2 },
        { 2, 6, 7, 3 },
        { 0, 1, 5, 4 }
};
```

The Cube Can Also Be Defined with Triangles



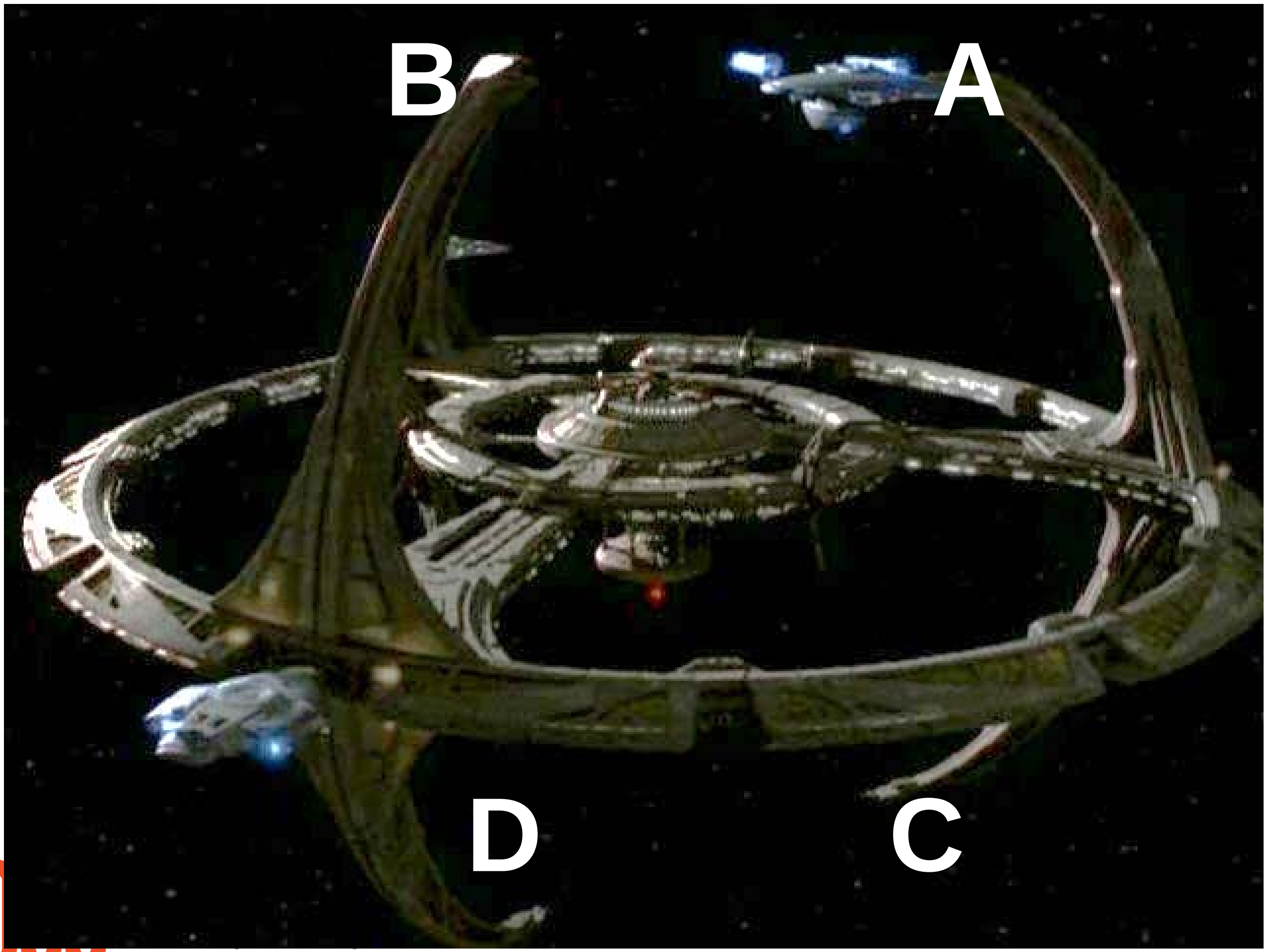


```
GLuint TriangleCubeIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 }
        { 0, 5, 4 }
};
```

```
GLuint CubeIndices[ ][4] =
{
        { 0, 2, 3, 1 },
        { 4, 5, 7, 6 },
        { 1, 3, 7, 5 },
        { 0, 4, 6, 2 },
        { 2, 6, 7, 3 },
        { 0, 1, 5, 4 }
};
```

Did any of you ever watch *Star Trek: Deep Space Nine*?

It was about life aboard a space station. Ships docked there to unload cargo and pick up supplies. When a ship was docked at docking port "**A**", for instance, the supply-loaders didn't need to know what ship it was. They could just be told, "send these supplies out docking port A, and pick up this cargo from docking port A".

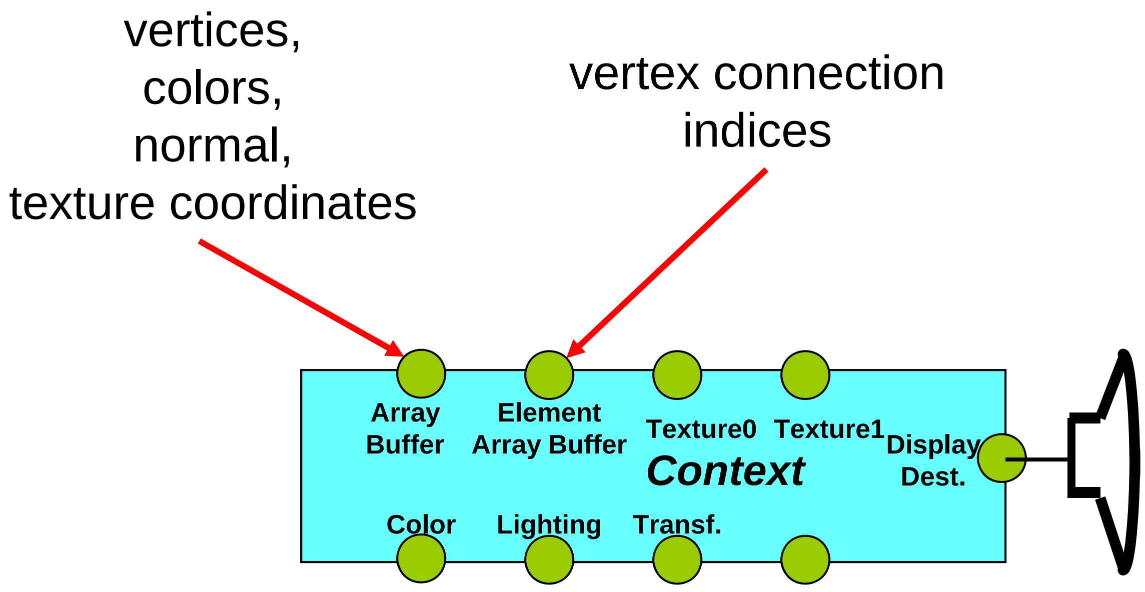


Surprisingly, this has something to do with computer graphics!

The OpenGL *Rendering Context*

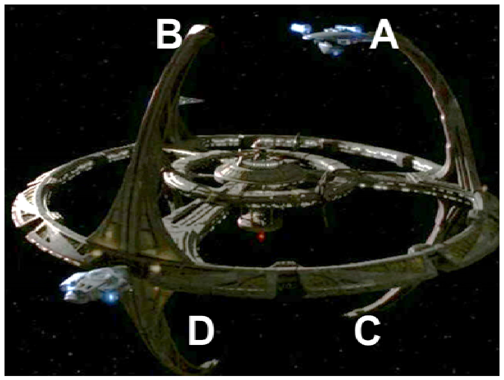
The OpenGL **Rendering Context** (also called “the **state**”) contains all the characteristic information necessary to produce an image from geometry. This includes the current transformation, color, lighting, textures, where to send the display, etc.

Each window (e.g., glutCreateWindow) has its own rendering context.

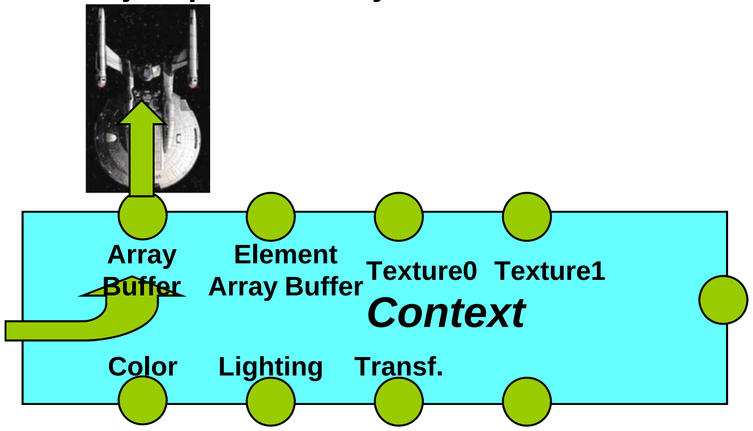


More Background – "Binding" to the Context

The OpenGL term "binding" refers to "attaching" or "docking" (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will "flow in" through the Context into the object.



Vertex Buffer Object pointed to by *bufA*

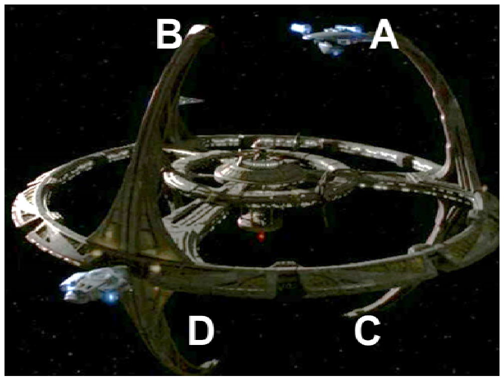


```
glBindBuffer( GL_ARRAY_BUFFER, bufA);

glBufferData( GL_ARRAY_BUFFER, numBytes, data, usage );
```

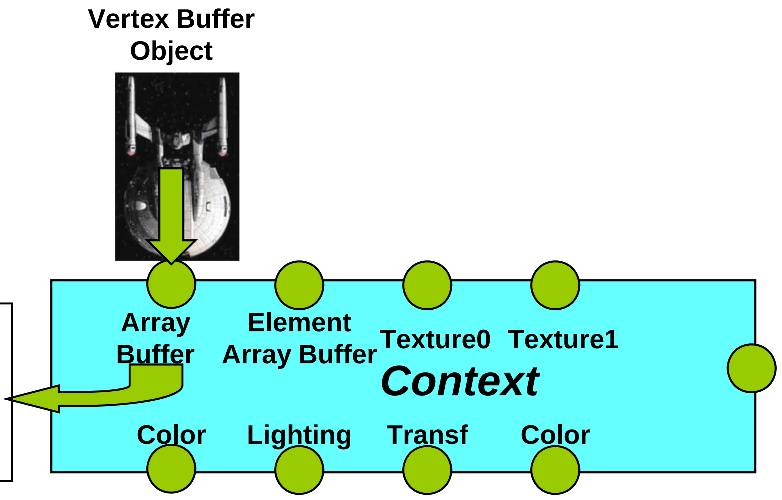
Ships docked there to unload cargo and pick up supplies. When a ship was docked at docking port "**A**", for instance, the supply-loaders didn't need to know what ship it was. They could just be told, "send these supplies out docking port A, and pick up this cargo from docking port A".

More Background – “Binding” to the Context

When you want to *use* that Vertex Buffer Object, just bind it again. All of the characteristics will then be active, just as if you had specified them again. Its contents will “flow out” of the object into the Context.



```
glBindBuffer( GL_ARRAY_BUFFER, bufA);

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



If you were writing OpenGL in C, how would you implement the State?

You would probably make all of that information into a big C struct

```
struct  OpenGLState
{
            float  CurrentColor[3];
            float  CurrentLineWidth;
            float  CurrentMatrix[4][4];


            struct TextureObject       *CurrentTexture0;

            struct ArrayBufferObject  *CurrentArrayBuffer;

            . . .

}  TheState;
```

If you were writing OpenGL in C, how would you implement the State?

Then, you could create any number of Array Buffer Objects, each with its own data and parameters. When you want to make a particular Array Buffer current, you just need to “bind” it to the state:

```
TheState.CurrentArrayBuffer = GpuAddressOf(  bufA  );
```

and, when you want to do something with it or to it, you just need to do an indirection:

```
TheState.CurrentBuffer->data[0].x = 42.;
```

More Background –
How do you Create an OpenGL “Buffer Object”?

When creating data structures in C++, objects are pointed to by their addresses.

In OpenGL, objects are pointed to by an unsigned integer “handle”. You can assign a value for this handle yourself (not recommended), or have OpenGL generate one for you that is guaranteed to be unique. For example:

the “array” to put them in

how many “handles” to generate

```
GLuint bufA;


glGenBuffers( 1, &bufA );
```

OpenGL then uses these handles to determine the actual GPU memory addresses to use.

Loading data into the current Vertex Buffer Object

```
glBufferData( type,  numBytes,  data,  usage );
```

type is the type of buffer object this is:

Use **GL_ARRAY_BUFFER** to store floating point vertices, normals, colors, and texture coordinates

numBytes is the number of bytes to store all together. It's not the number of numbers, not the number of coordinates, not the number of vertices, but the number of ***bytes***!

data is the memory address of (i.e., pointer to) the data to be transferred from CPU memory to the graphics memory. (This is allowed to be NULL, indicating that you will transfer the data over later.)

Loading data into the current Vertex Buffer Object

```
glBufferData( type,  numBytes,  data,  usage );
```

usage is a hint as to how the data will be used: GL_xxx_yyy

where xxx can be:

STATIC	this buffer will be re-written seldom
DYNAMIC	this buffer will be re-written often

and yyy can be:

DRAW	this buffer will be used for drawing
READ	this buffer will be copied into

For what we are doing in most classes, use **GL_STATIC_DRAW**

Remember this for the Vulkan notes!

Adding data into the current Vertex Buffer Object

```
glBufferSubData( type,  offset, numBytes,  data );
```

Step #1 – Fill the C/C++ Arrays with Drawing Data (vertices, colors, …)

```
GLfloat  Vertices[  ][3] =
{
                { 1.,  2.,  3. },
                { 4.,  5.,  6. },
                . . .
};
```

Step #2 – Transfer the Drawing Data

```
glGenBuffers( 1, &bufA );


glBindBuffer( GL_ARRAY_BUFFER, bufA );
glBufferData( GL_ARRAY_BUFFER, 3*sizeof(GLfloat)*numVertices, Vertices, GL_STATIC_DRAW );
```

Step #3 – Activate the Drawing Data Types That You Are Using

```
glEnableClientState( type )
```

where *type* can be any of:

```
GL_VERTEX_ARRAY
GL_COLOR_ARRAY
GL_NORMAL_ARRAY
GL_TEXTURE_COORD_ARRAY
```

- Call this as many times as you need to enable all the drawing data types that you are using.
- To deactivate a type, call:

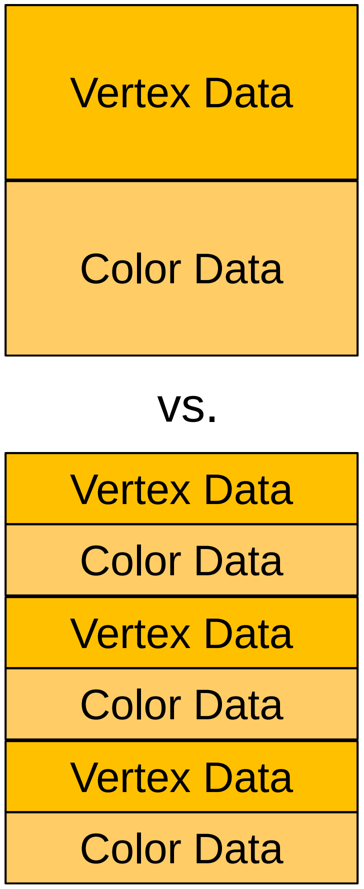
```
glDisableClientState( type )
```

Step #4 – To start the drawing process, bind the Buffer that holds the Drawing Data

```
glBindBuffer( GL_ARRAY_BUFFER, bufA );
```

Step #5 – Then, specify how to get at each Data Type within that Buffer

```
glVertexPointer( size, type, stride, offset);

glColorPointer( size, type, stride, offset);

glNormalPointer( type, stride, offset);

glTexCoordPointer( size, type, stride, offset);
```



size is the "how many numbers per vertex", and can be: 2, 3, or 4

type can be:

GL_SHORT
GL_INT
GL_FLOAT
GL_DOUBLE

stride is the byte offset between consecutive entries in the buffer (0 means tightly packed)

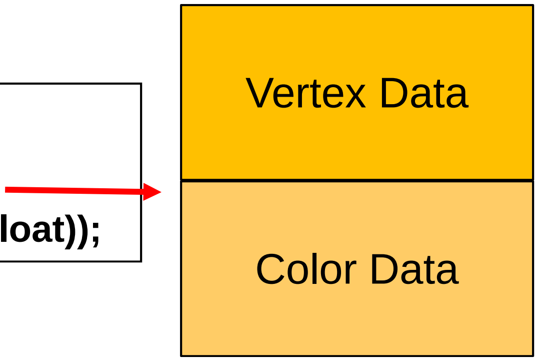
offset is the byte offset from the start of the data array buffer to where the first element of this part of the data lives.

The Data Types in a vertex buffer object can be stored either as “packed” or “interleaved”

```
gl*Pointer( size, type, stride, offset);
```

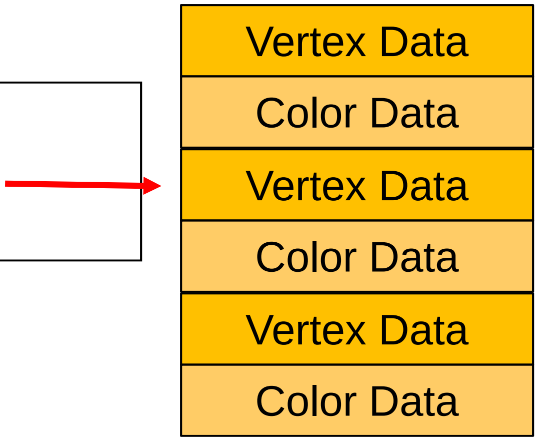
Packed:

```
glVertexPointer( 3, GL_FLOAT, 3*sizeof(GLfloat), 0 );

glColorPointer( 3, GL_FLOAT, 3*sizeof(GLfloat), 3*numVertices*sizeof(GLfloat));
```



Interleaved:

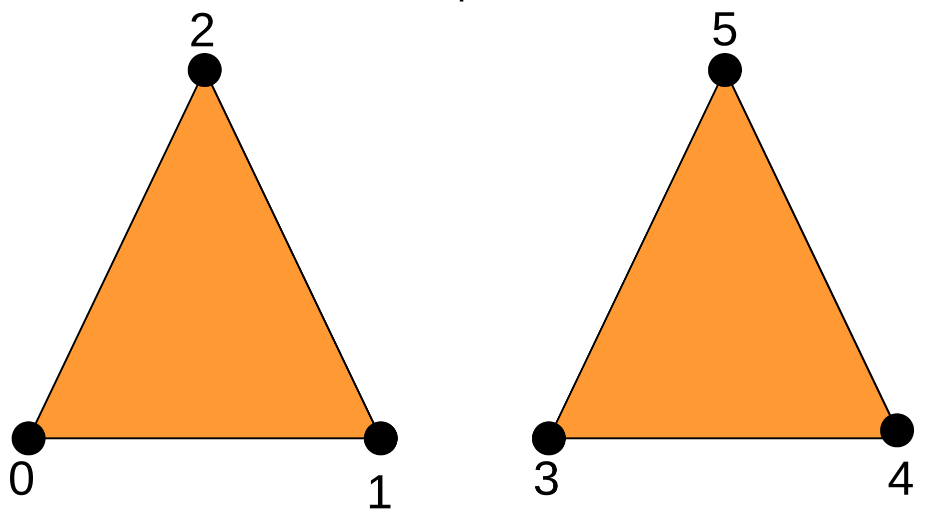
```
glVertexPointer( 3, GL_FLOAT, 6*sizeof(GLfloat), 0 );

glColorPointer( 3, GL_FLOAT, 6*sizeof(GLfloat), 3*sizeof(GLfloat) );
```



Step #6 – Draw!

```
glDrawArrays( GL_TRIANGLES, first, numVertices );
```

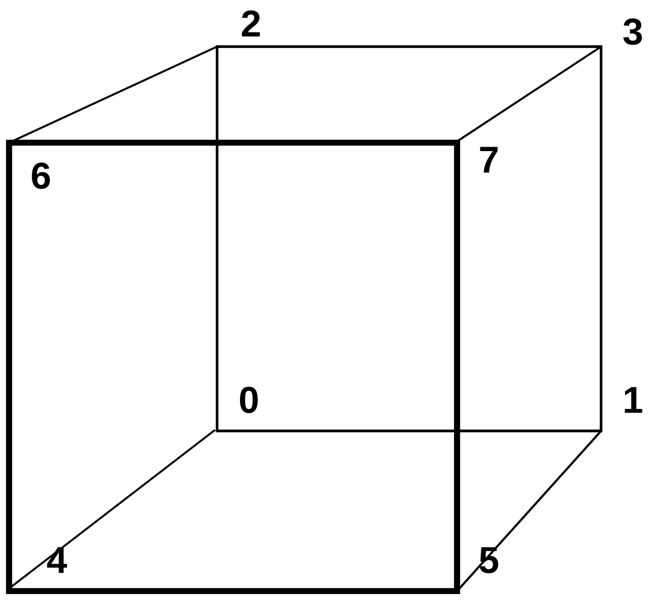
Example:



```
glDrawArrays( GL_TRIANGLES, 0, 6 );
```

This is how you do it if your vertices are to be drawn in consecutive order

What if your vertices are to be accessed in random order?

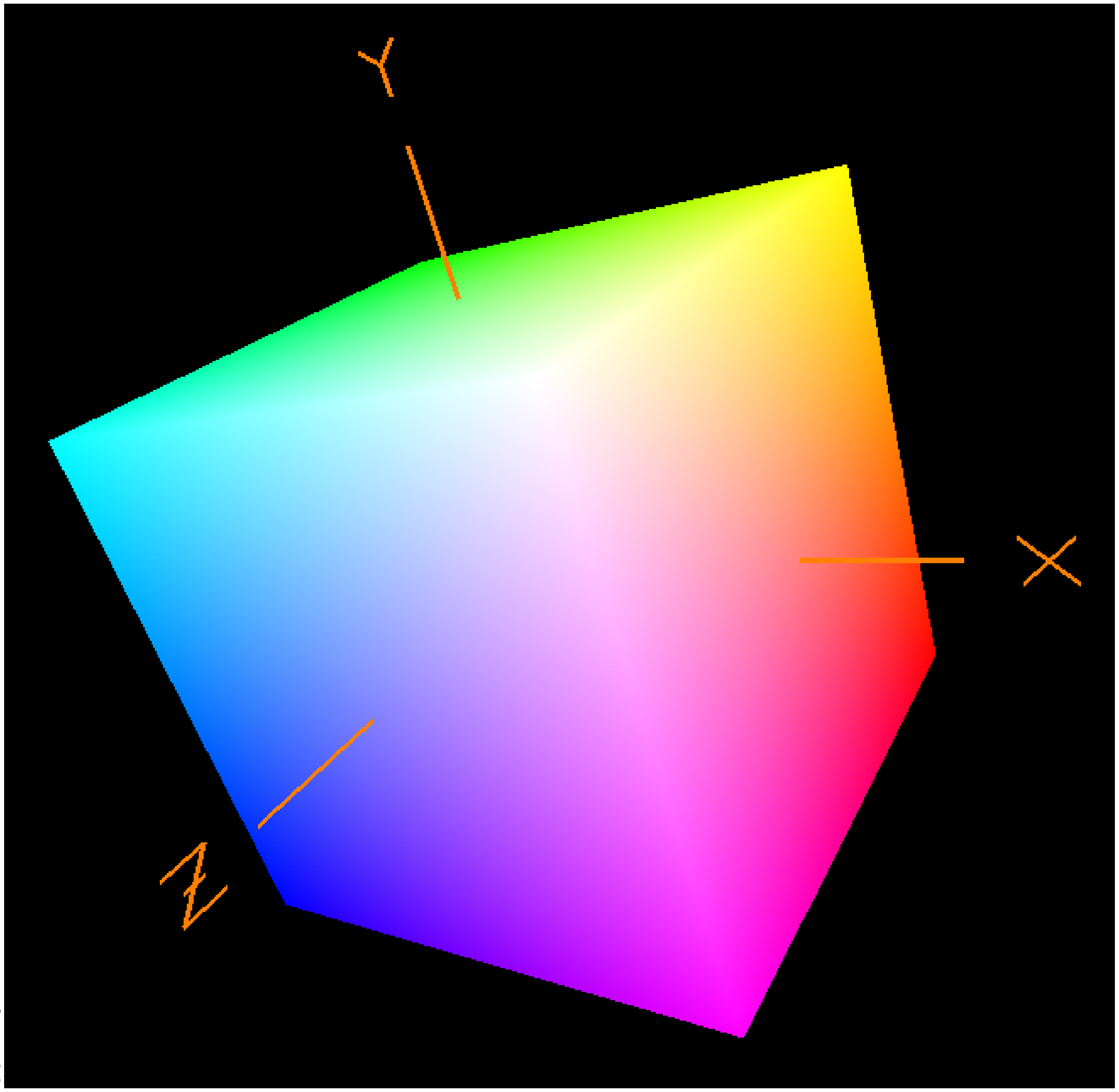


```
GLfloat CubeVertices[ ][3] =
{
                { -1., -1., -1. },
                {  1., -1., -1. },
                { -1.,  1., -1. },
                {  1.,  1., -1. },
                { -1., -1.,  1. },
                {  1., -1.,  1. },
                { -1.,  1.,  1. },
                {  1.,  1.,  1. }
};
```

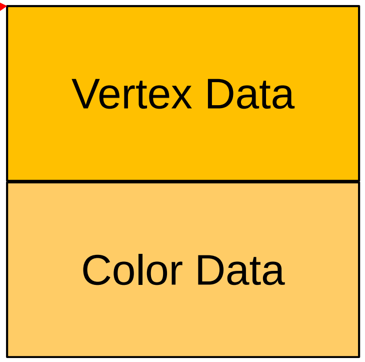
```
GLfloat CubeColors[ ][3] =
{
                { 0., 0., 0. },
                { 1., 0., 0. },
                { 0., 1., 0. },
                { 1., 1., 0. },
                { 0., 0., 1. },
                { 1., 0., 1. },
                { 0., 1., 1. },
                { 1., 1., 1. },
};
```

```
GLuint CubeIndices[ ][4] =
{
                { 0, 2, 3, 1 },
                { 4, 5, 7, 6 },
                { 1, 3, 7, 5 },
                { 0, 4, 6, 2 },
                { 2, 6, 7, 3 },
                { 0, 1, 5, 4 }
};
```

Cube Example



```
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );
glVertexPointer(  3, GL_FLOAT, 0, (Gluchar*) 0 );
glColorPointer(   3, GL_FLOAT, 0, (Gluchar*) (3*sizeof(GLfloat)*numVertices) );
glBegin( GL_QUADS );
            glArrayElement( 0 );
            glArrayElement( 2 );
            glArrayElement( 3 );
            glArrayElement( 1 );
            glArrayElement( 4 );
            glArrayElement( 5 );
            glArrayElement( 7 );
            glArrayElement( 6 );
            glArrayElement( 1 );
            glArrayElement( 3 );
            glArrayElement( 7 );
            glArrayElement( 5 );
            glArrayElement( 0 );
            glArrayElement( 4 );
            glArrayElement( 6 );
            glArrayElement( 2 );
            glArrayElement( 2 );
            glArrayElement( 6 );
            glArrayElement( 7 );
            glArrayElement( 3 );
            glArrayElement( 0 );
            glArrayElement( 1 );
            glArrayElement( 5 );
            glArrayElement( 4 );
glEnd( );
```



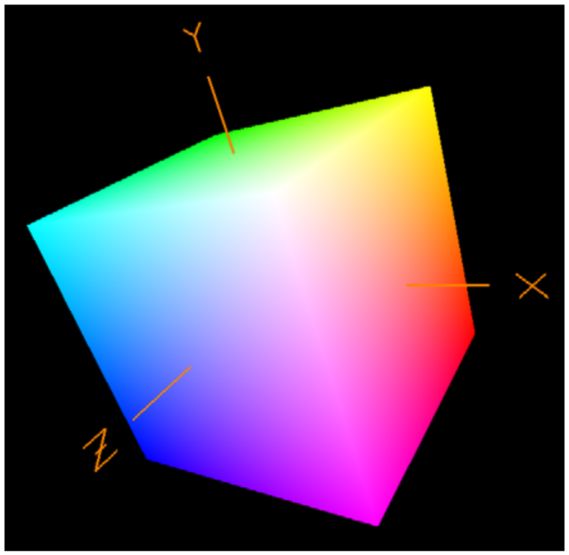
```
GLuint CubeIndices[ ][4] =
{
            { 0, 2, 3, 1 },
            { 4, 5, 7, 6 },
            { 1, 3, 7, 5 },
            { 0, 4, 6, 2 },
            { 2, 6, 7, 3 },
            { 0, 1, 5, 4 }
};
```

But, it would be better if that index array was over on the GPU as well

```
glBindBuffer( GL_ARRAY_BUFFER, bufA );

glBufferData( GL_ARRAY_BUFFER, 3*sizeof(GLfloat)*numVertices,
                              Vertices, GL_STATIC_DRAW );




glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, bufB );

glBufferData( GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint)*numIndices,
                              CubeIndices, GL_STATIC_DRAW );
```

The glDrawElements( ) call

```
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, bufB );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

glVertexPointer( 3, GL_FLOAT, 0, (Gluchar*) 0 );
glColorPointer(  3,  GL_FLOAT, 0, (Gluchar*) (3*sizeof(GLfloat)*numVertices) );

glDrawElements( GL_QUADS, 24, GL_UNSIGNED_INT, (Gluchar*) 0 );
```



Re-writing Data into a Buffer Object, Treating it as a C/C++ Array

```
float * vertexArray = glMapBuffer( GL_ARRAY_BUFFER,  usage );
```

usage is how the data will be accessed:

GL_READ_ONLY	the vertex data will be read from, but not written to
GL_WRITE_ONLY	the vertex data will be written to, but not read from
GL_READ_WRITE	the vertex data will be read from and written to

You can now use **vertexArray[ ]** like any other C/C++ floating-point array.

When you are done, be sure to call:

```
glUnMapBuffer( GL_ARRAY_BUFFER );
```

Using a Vertex Buffer Object C++ Class

Declaring:

```
VertexBufferObject  VB ;
```

Filling:

```
VB.glBegin( GL_QUADS );                    // can be any of the OpenGL topologies
for( int i = 0; i < 6; i++ )
{
        for( int j = 0; j < 4; j++ )
        {
                int k = CubeIndices[ i ][ j ];
                VB.glColor3fv( CubeColors[ k ] );
                VB.glVertex3fv( CubeVertices[ k ] );
        }
}
VB.glEnd(  );
```

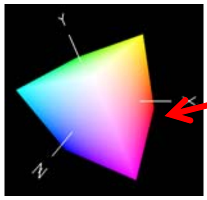
Drawing:

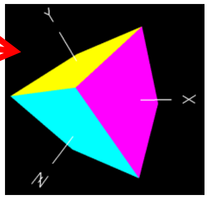
```
VB.Draw(  );
```

Vertex Buffer Object Class Methods

void CollapseCommonVertices(bool);

true means to not replicate common vertices in the internal vertex table. This is good if all uses of a particular vertex will have the same normal, color, and texture coordinates, like this – instead of like this.





void Draw( );

Draw the primitive. If this is the first time Draw( ) is being called, it will setup all the proper buffer objects, etc. If it as a subsequent call, then it will just initiate the drawing.

void glBegin(topology);

Initiate the primitive.

void glColor3f(r, g, b);
void glColor3fv(rgb[3]);

Specify a vertex's color.

void glEnd( );

Terminate the definition of a primitive.

void glNormal3f(nx, ny, nz);
void glNormal3fv(nxyz[3]);

Specify a vertex's normal.

Vertex Buffer Object Class Methods

void glTexCoord2f(s, t);
void glTexCoord2fv(st[2]);
	Specify a vertex's texture coordinates.

void glVertex3f(x, y, z);
void glVertex3fv(xyz[3]);
	Specify a vertex's coordinates.

void Print(char *text, FILE *);	Prints the vertex, normal, color, texture coordinate, and connection element information to a file, along with some preliminary text. If the file pointer is not given, standard error (i.e., the console) is used.
void RestartPrimitive( );	Causes the primitive to be restarted. This is useful when doing triangle strips or quad strips and you want to start another one without getting out of the current one. By doing it this way, all of the strips' vertices will end up in the same table, and you only need to have one *VertexBufferObject* class going.

Notes

• If you want to print the contents of your data structure to a file (for debugging or curiosity), do this:

```
FILE *fp = fopen( “debuggingfile.txt”, “w” );
if( fp == NULL )
{
        fprintf( stderr, “Cannot create file ‘debuggingfile.txt’\n” );
}
else
{
        VB.Print( “My Vertex Buffer :”, p );
        fclose( fp );
}
```

• You can call the *glBegin* method more than once. Each call will wipe out your original display information and start over from scratch. This is useful if you are interactively editing geometry, such as sculpting a curve.

• In many cases, using standard glBegin( ) – glEnd( ) in a display list can be just about as fast as using vertex buffer objects if the vendor has written the drivers to create the display list on the graphics card. But, the vendors don’t always do this. You’re better off using vertex buffer objects because they are *always* fast.

A Caveat

Be judicious about collapsing common vertices! The good news is that it saves space and it might increase speed some (by having to transform fewer vertices). But, the bad news is that it takes much longer to create large meshes. Here's why.

Say you have a 1,000 x 1,000 point triangle mesh, drawn as 999 triangle strips, all in the same *VertexBufferObject* class (which you can do using the *RestartPrimitive* method) .

When you draw the S^{th} triangle strip, half of those points are coincident with points in the $S\text{-}1^{st}$ strip. But, to find those 1,000 coincident points, it must search through 1000*S points first. There is no way to tell it to only look at the last 1,000 points. Even though the search is only $O(\log_2 N)$, where N is the number of points kept so far, it still adds up to a lot of time over the course of the entire mesh.

It starts out fast, but slows down as the number of points being held increases.

If you did have a 1,000 x 1,000 mesh, it might be better to not collapse vertices at all. Or, a compromise might be to collapse vertices, but break this mesh up into 50 *VertexBufferObjects*, each of size 20 x 1,000.

Just a thought…

Drawing the Cube With Collapsing Identical Vertices

Drawing 8 points

X	Y	Z	R	G	B
-1.00	-1.00	-1.00	0.00	0.00	0.00
-1.00	1.00	-1.00	0.00	1.00	0.00
1.00	1.00	-1.00	1.00	1.00	0.00
1.00	-1.00	-1.00	1.00	0.00	0.00
-1.00	-1.00	1.00	0.00	0.00	1.00
1.00	-1.00	1.00	1.00	0.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00
-1.00	1.00	1.00	0.00	1.00	1.00

Drawing 24 array elements:

0	1	2	3
4	5	6	7
3	2	6	5
0	4	7	1
1	7	6	2
0	3	5	4

Drawing the Cube Without Collapsing Identical Vertices

Drawing 24 points

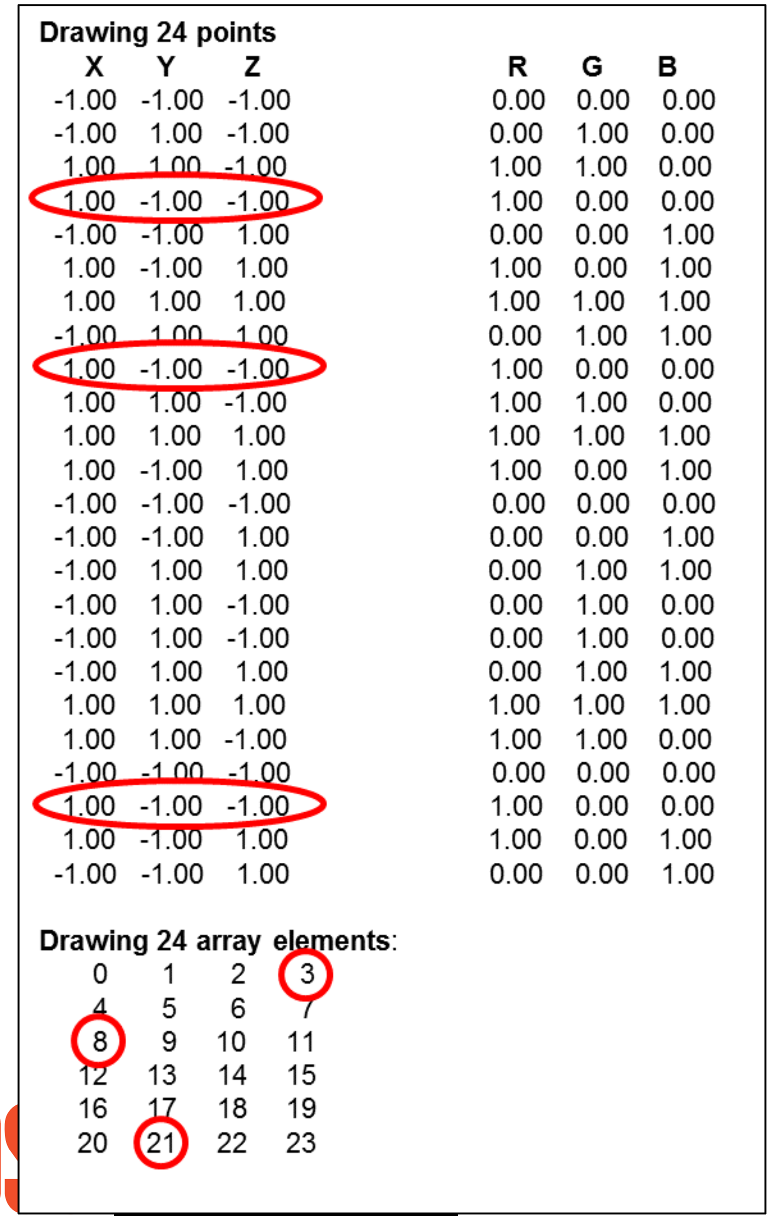
X	Y	Z	R	G	B
-1.00	-1.00	-1.00	0.00	0.00	0.00
-1.00	1.00	-1.00	0.00	1.00	0.00
1.00	1.00	-1.00	1.00	1.00	0.00
1.00	-1.00	-1.00	1.00	0.00	0.00
-1.00	-1.00	1.00	0.00	0.00	1.00
1.00	-1.00	1.00	1.00	0.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00
-1.00	1.00	1.00	0.00	1.00	1.00
1.00	-1.00	-1.00	1.00	0.00	0.00
1.00	1.00	-1.00	1.00	1.00	0.00
1.00	1.00	1.00	1.00	1.00	1.00
1.00	-1.00	1.00	1.00	0.00	1.00
-1.00	-1.00	-1.00	0.00	0.00	0.00
-1.00	-1.00	1.00	0.00	0.00	1.00
-1.00	1.00	1.00	0.00	1.00	1.00
-1.00	1.00	-1.00	0.00	1.00	0.00
-1.00	1.00	-1.00	0.00	1.00	0.00
-1.00	1.00	1.00	0.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	-1.00	1.00	1.00	0.00
-1.00	-1.00	-1.00	0.00	0.00	0.00
1.00	-1.00	-1.00	1.00	0.00	0.00
1.00	-1.00	1.00	1.00	0.00	1.00
-1.00	-1.00	1.00	0.00	0.00	1.00

Drawing 24 array elements:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

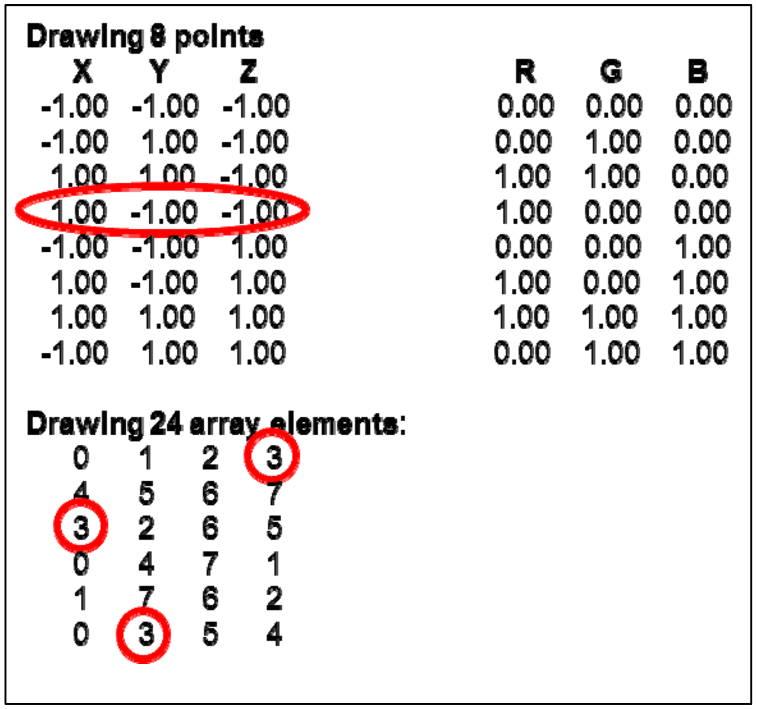
A Comparison

Not Collapsing Identical Vertices



```
Drawing 24 points
    X      Y      Z                          R      G      B
  -1.00  -1.00  -1.00                      0.00   0.00   0.00
  -1.00   1.00  -1.00                      0.00   1.00   0.00
   1.00   1.00  -1.00                      1.00   1.00   0.00
   1.00  -1.00  -1.00                      1.00   0.00   0.00
  -1.00  -1.00   1.00                      0.00   0.00   1.00
   1.00  -1.00   1.00                      1.00   0.00   1.00
   1.00   1.00   1.00                      1.00   1.00   1.00
  -1.00   1.00   1.00                      0.00   1.00   1.00
   1.00  -1.00  -1.00                      1.00   0.00   0.00
   1.00   1.00  -1.00                      1.00   1.00   0.00
   1.00   1.00   1.00                      1.00   1.00   1.00
   1.00  -1.00   1.00                      1.00   0.00   1.00
  -1.00  -1.00  -1.00                      0.00   0.00   0.00
  -1.00  -1.00   1.00                      0.00   0.00   1.00
  -1.00   1.00   1.00                      0.00   1.00   1.00
  -1.00   1.00  -1.00                      0.00   1.00   0.00
  -1.00   1.00  -1.00                      0.00   1.00   0.00
  -1.00   1.00   1.00                      0.00   1.00   1.00
   1.00   1.00   1.00                      1.00   1.00   1.00
   1.00   1.00  -1.00                      1.00   1.00   0.00
  -1.00  -1.00  -1.00                      0.00   0.00   0.00
   1.00  -1.00  -1.00                      1.00   0.00   0.00
   1.00  -1.00   1.00                      1.00   0.00   1.00
  -1.00  -1.00   1.00                      0.00   0.00   1.00

Drawing 24 array elements:
    0    1    2    3
    4    5    6    7
    8    9   10   11
   12   13   14   15
   16   17   18   19
   20   21   22   23
```

Collapsing Identical Vertices



```
Drawing 8 points
    X      Y      Z                          R      G      B
  -1.00  -1.00  -1.00                      0.00   0.00   0.00
  -1.00   1.00  -1.00                      0.00   1.00   0.00
   1.00   1.00  -1.00                      1.00   1.00   0.00
   1.00  -1.00  -1.00                      1.00   0.00   0.00
  -1.00  -1.00   1.00                      0.00   0.00   1.00
   1.00  -1.00   1.00                      1.00   0.00   1.00
   1.00   1.00   1.00                      1.00   1.00   1.00
  -1.00   1.00   1.00                      0.00   1.00   1.00

Drawing 24 array elements:
    0    1    2    3
    4    5    6    7
    3    2    6    5
    0    4    7    1
    1    7    6    2
    0    3    5    4
```

Using Vertex Buffers with Shaders

Let’s say that we have the following vertex shader and we want to supply the vertices from a Buffer Object.

```
in vec3 aVertex;
in vec3 aColor;

out vec3 vColor;

void
main( )
{
		vColor = aColor;
		gl_Position = gl_ModelViewProjectionMatrix * vec4( aVertex, 1. );
}
```

Let’s also say that, at some time, we want to supply the colors from a Buffer Object as well, but for right now, the color will be constant.

Using Vertex Buffers with Shaders

We’re assuming here that we already have the shader program setup in *program*, and already have the vertices in the *vertexBuffer*.

```
glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY  );

GLuint vertexLocation = glGetAttribLocation( program, “aVertex” );
GLuint colorLocation   = glGetAttribLocation( program, “aColor” );

glVertexAttribPointer( vertexLocation, 3, GL_FLOAT, GL_FALSE, 0, (GLuchar *)0 );
glEnableVertexAttribArray( vertexLocation );          // dynamic attribute

glVertexAttrib3f( colorLocation,  r, g, b );                // static attribute
glDisableVertexAttribArray( colorLocation );

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

Using Vertex Buffers with the Shaders C++ Class

We’re assuming here that we already have the shader program setup in *program*, and already have the vertices in the *vertexBuffer*.

```
glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY  );

Pattern->SetAttributePointer3fv( “aVertex”, (GLfloat *)0 );
Pattern->EnableVertexAttribArray( “aVertex” );                // dynamic attribute

Pattern->SetAttributeVariable( “aColor”,  r, g, b );          // static attribute
Pattern->DisableVertexAttribArray( “aColor” );

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