Vertex Buffer Objects

1

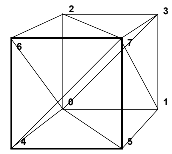
Oregon State University

Mike Bailey
mjb@cs.oregonstate.edu

1

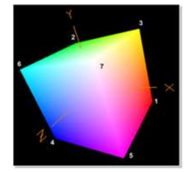
Remember this from the Geometric Modeling Notes?

2




```
static 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. }
};
```

```
GLuint CubeTriangleIndices[ ][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 }
};
```



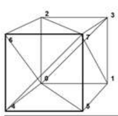
```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```


2

Vertex Buffer Objects: The Big Idea

3

- 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 instead of CPU memory, avoiding a significant amount of bus latency.



```
static 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. }
};
```

```
GLuint CubeTriangleIndices[ ][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 }
};
```

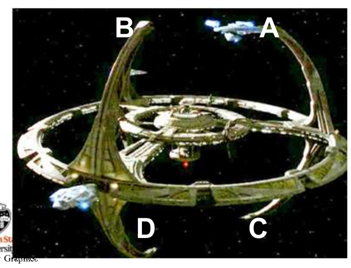
```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```


3

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

4

It was about life aboard a space station. Ships docked at Deep Space Nine 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 "bring this cargo in from docking port A".



Surprisingly, this actually has something to do with computer graphics! ☺

4

The OpenGL *Rendering Context*

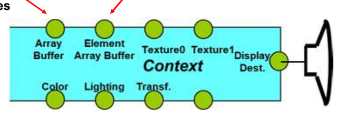
5

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.

vertices, colors, normal, texture coordinates

vertex connection indices

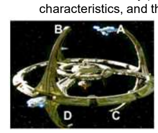


5

More Background – "Binding" to the Context

6

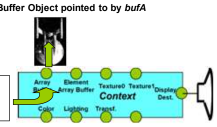
The OpenGL term "binding" refers to "attaching" or "docking" (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will "flow in" through the Context into the object.



Vertex Buffer Object pointed to by *bufA*

```
glBindBuffer( GL_ARRAY_BUFFER, bufA);

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



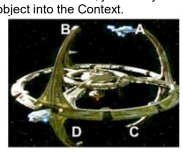
Ships docked at Deep Space Nine to unload cargo and pick up supplies. When a ship was docked at docking port "**A**", for instance, the supply-loaders didn't need to know what ship it was. They could just be told, "send these supplies out docking port A", and "pick up this cargo from docking port A".

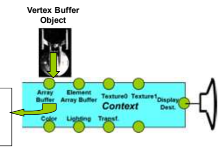
6

More Background – “Binding” to the Context

7

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 );
```

Oregon State University Computer Graphics

mjb – September 20, 2023

7

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

8

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:

How many “handles” to generate

The “array” to put them in

```
GLuint bufA;

glGenBuffers( 1, &bufA );
```

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

Oregon State University Computer Graphics

mjb – September 20, 2023

8

Loading data into the currently-bound Vertex Buffer Object

9

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.)

Oregon State University Computer Graphics

mjb – September 20, 2023

9

Loading data into the currently-bound Vertex Buffer Object

10

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, use **GL_STATIC_DRAW**

Oregon State University Computer Graphics

mjb – September 20, 2023

10

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

11

```
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 );
```

Oregon State University Computer Graphics

mjb – September 20, 2023

11

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

12

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);

Oregon State University Computer Graphics

mjb – September 20, 2023

12

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

13

glBindBuffer(GL_ARRAY_BUFFER, bufA);

13

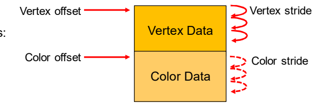
Step #5 – Then, specify how to get at each Data Type within that Buffer

14

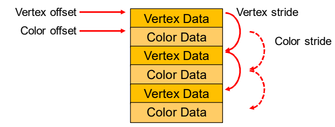
offset is the number of byte offsets from the start of the data array buffer to where the first element of this part of the data lives

stride is the number of bytes between the same type of data

Information can be stored as packed, like this:



Information can be stored as interleaved, like this:

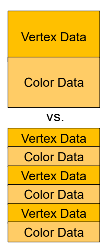


14

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

15

```
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.

15

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

16

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));
```

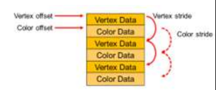
stride offset

Interleaved:

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

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

stride offset



16

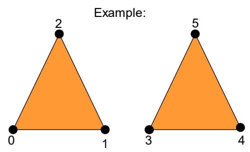
Step #6 – Draw!

17

glDrawArrays(GL_TRIANGLES, first, numVertices);

This is how you do it *if your vertices can be drawn in consecutive order*

Example:



Start with vertex #0

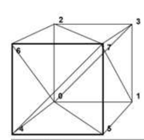
glDrawArrays(GL_TRIANGLES, 0, 6);

Draw 6 vertices

17

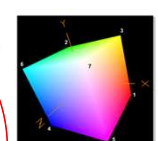
What if your vertices need to be accessed in random order?

18




```
static 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. }
};
```

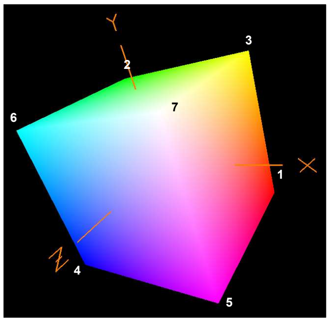
```
GLuint CubeTriangleIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 }
        { 0, 5, 4 }
};
```




```
static 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. },
};
```


18

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_TRIANGLES );
        glArrayElement( 0 );
        glArrayElement( 2 );
        glArrayElement( 3 );
        glArrayElement( 0 );
        glArrayElement( 3 );
        glArrayElement( 1 );
        glArrayElement( 4 );
        glArrayElement( 5 );
        glArrayElement( 7 );
        glArrayElement( 4 );
        glArrayElement( 7 );
        glArrayElement( 6 );
        glArrayElement( 1 );
        glArrayElement( 3 );
        glArrayElement( 7 );
        glArrayElement( 1 );
        glArrayElement( 7 );
        glArrayElement( 5 );
        . . .
```

Vertex Data

Color Data

```
        . . .
        glArrayElement( 0 );
        glArrayElement( 4 );
        glArrayElement( 6 );
        glArrayElement( 0 );
        glArrayElement( 6 );
        glArrayElement( 2 );
        glArrayElement( 2 );
        glArrayElement( 6 );
        glArrayElement( 7 );
        glArrayElement( 2 );
        glArrayElement( 7 );
        glArrayElement( 3 );
        glArrayElement( 0 );
        glArrayElement( 1 );
        glArrayElement( 5 );
        glArrayElement( 0 );
        glArrayElement( 5 );
        glArrayElement( 4 );
glEnd( );
```

```
GLuint CubeTriangleIndices[ ][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 }
};
```

But, this requires that all these glArrayElement( ) calls happen on the CPU and get transmitted across the bus to the GPU!

It would be better if that index array was stored over on the GPU as well

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

glBindBuffer(**GL_ELEMENT_ARRAY_BUFFER**, bufB);
glBufferData(**GL_ELEMENT_ARRAY_BUFFER**, sizeof(GLuint)*numIndices, CubeTriangleIndices, GL_STATIC_DRAW);

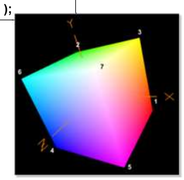
The glDrawElements( ) call


```
glBindBuffer( GL_ARRAY_BUFFER, bufA );
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_TRIANGLES, 36, GL_UNSIGNED_INT, (Gluchar*) 0 );
```



Writing Data into a Buffer Object, Treating it as a C/C++ Array of Structures

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 of structures.

When you are done, be sure to call:

glUnMapBuffer(GL_ARRAY_BUFFER);

glMapBuffer Example


```
struct Point
{
        float x, y, z;
        float nx, ny, nz;
        float r, g, b;
        float s, t;
};

. . .

glGenBuffers(  1, &pbuffer );
glBindBuffer( GL_ARRAY_BUFFER, pbuffer );
glBufferData(  GL_ARRAY_BUFFER, numPoints * sizeof(struct  Point), NULL, GL_STATIC_DRAW );
struct Point * parray = (struct Point *) glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
for( int i = 0; i < numPoints; i++ )
{
                parray[i].x    = PointVec[i].x;
                parray[i].y    = PointVec[i].y;
                parray[i].z    = PointVec[i].z;
                parray[i].nx  = PointVec[i].nx;
                parray[i].ny  = PointVec[i].ny;
                parray[i].nz  = PointVec[i].nz;
                parray[i].r     = PointVec[i].r;
                parray[i].g    = PointVec[i].g;
                parray[i].b    = PointVec[i].b;
                parray[i].s    = PointVec[i].s;
                parray[i].t     = PointVec[i].t;
}
glUnmapBuffer( GL_ARRAY_BUFFER );
```

Using our Vertex Buffer Object C++ Class

25

Declaring a Global:

```
VertexBufferObject  VB ;
```

Filling:

```
VB.Init( );
VB.glBegin( GL_TRIANGLES );                    // can be any of the OpenGL topologies
for( int i = 0; i < 12; i++ )
{
        for( int j = 0; j < 3; j++ )
        {
                int k = CubeTriangleIndices[ i ][ j ];
                VB.glColor3fv( CubeColors[ k ] );
                VB.glVertex3fv( CubeVertices[ k ] );
        }
}
VB.glEnd(  );
```

This is available from our Class Resources page

Drawing:

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

Oregon State University Computer Graphics

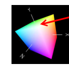
25

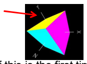
Vertex Buffer Object Class Methods

26

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 is a subsequent call, then it will just initiate the drawing.

void DrawInstanced(numInstances);

Same as Draw( ), but will draw multiple instances. Must be using shaders to make this worthwhile.

void glBegin(topology);

Begin the definition of a primitive.

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

Specify a vertex's color.

void glEnd( );

Terminate the definition of a primitive.

Oregon State University Computer Graphics

26

Vertex Buffer Object Class Methods

27

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

Specify a vertex's normal.

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.

Oregon State University Computer Graphics

27

Notes

28

- 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 :", fp );
        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.

Oregon State University Computer Graphics

28

A Caveat

29

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-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…

Oregon State University Computer Graphics

29

Drawing the Cube With Collapsing Identical Vertices

30

```
GLuint CubeTriangleIndices[ ][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 }
};
```

Define 8 points / **Define 8 colors**

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

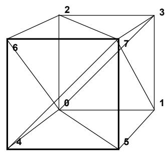
Draw 36 array indices:

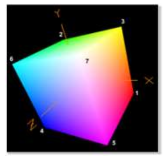
0	2	3
0	3	1
4	5	7
4	7	6
1	3	7
1	7	5
0	4	6
0	6	2
2	6	7
2	7	3
0	1	5
0	5	4

Oregon State University Computer Graphics

30

Drawing the Cube Without Collapsing Identical Vertices





Define 36 vertices and 36 colors

X	Y	Z	R	G	B
-1.	-1.	-1.	0.	0.	0.
. . .			. . .		

Using Vertex Buffers with Shaders

Let's say that we have the following **vertex shader** and we want to supply the vertices from a Vertex 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 Vertex Buffer Object as well, but for right now, the color will be uniform.

Using Vertex Buffers with Shaders

We're assuming here that

- we already have the shader program setup in *program*
- we 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
glVertexAttribArray( colorLocation );

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


Using Vertex Buffers with the Shaders C++ Class

We're assuming here that

- we already have the vertices in the *vertexBuffer*
- we have already created a C++ GLSLProgram class object called *Pattern*

```
glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

Pattern.SetVertexAttributePointer3fv( "aVertex", (GLfloat *)0 );
Pattern.EnableVertexAttribArray( "aVertex" );          // dynamic attribute

Pattern.SetVertexAttributeVariable( "aColor", r, g, b ); // static attribute
Pattern.EnableVertexAttribArray( "aColor" );

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