Vertex Buffer Objects

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

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.

The Cube Can Also Be Defined with Triangles

```gl
GLuint CubeQuadIndices[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.

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”.

```gl
glBindBuffer( GL_ARRAY_BUFFER, bufA);
glBufferData( GL_ARRAY_BUFFER, numBytes, data, usage);
```
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.

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:

```c
struct OpenGLState {
    float CurrentColor[3];
    float CurrentLineWidth;
    float CurrentMatrix[4][4];
    struct TextureObject *CurrentTexture0;
    struct ArrayBufferObject *CurrentArrayBuffer;
    // ...
};
TheState;
```

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

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

Loading data into the currently-bound Vertex Buffer Object

```c
gBufferData( 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.)

Adding data into the current Vertex Buffer Object

```c
gBufferSubData( type, offset, numBytes, data );
```

Use **GL_STATIC_DRAW** for what we are doing in most classes.
Step #1 – Fill the C/C++ Arrays with Drawing Data (vertices, colors, ...)

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

Step #2 – Transfer the Drawing Data

```c
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

```c
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:

```c
gDisableClientState( type );
```

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

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

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

```c
glVertexPointer( size, type, stride, offset);
gColorPointer( size, type, stride, offset);
gNormalPointer( type, stride, offset);
gTexCoordPointer( size, type, stride, offset);
```

- size is the "how many numbers per vertex", and can be: 2, 3, or 4
  - GL_SHORT
  - GL_INT
  - GL_FLOAT
  - GL_DOUBLE
- type can be:
  - GLSHORT
  - GL_UNSIGNED_SHORT
  - GL_UNSIGNED_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.

Step #6 – Draw!

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

Example:

```
Example: 3 1 2
2
2
```

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

Packed:

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

Interleaved:

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

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?

```c
GLfloat CubeVertices[3][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.}
};
```

```c
GLfloat CubeColors[3][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.}
};
```

```c
GLuint CubeQuadIndices[3][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

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

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

The `glDrawElements()` call

```c
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 );
```

```c
  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 );
```
Using a Vertex Buffer Object C++ Class

Declaring

```cpp
VertexBufferObject VB;
```

Filling:

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

Drawing:

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

Vertex Buffer Object Class Methods

```cpp
void CollapseCommonVertices( bool );

void Draw( ); // Draw the primitive.  If this is in 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 glPushMatrix( );
void glPopMatrix( );

void gldraw( );

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

void glEnd( );

void glVertex3fv( xyz[ 3 ] );
void glVertex3f( x, y, z);

void glVertex2fv( st[ 2 ] );
void glVertex2f( x, y );

void glTexCoord2fv( st[ 2 ] );
void glTexCoord2f( s, t );

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

Notes

- *If you want to print the contents of your data structure to a file (for debugging or curiosity), do this:*
  ```cpp
  FILE *fp = fopen( "debuggingfile.txt", "w" );
  if( fp == NULL )
  {
      fprintf(stderr, "Cannot create file 'debuggingfile.txt'
      return( fp );
  }
  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.*

- *In many cases, using standard glBegin() – glEnd() in a display list can be just 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 5° triangle strip, half of those points are coincident with points in the S-1st 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(logN), 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 Without Collapsing Identical Vertices

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.00</td>
<td>-1.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>1.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>-1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>-1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>-1.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>1.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>-1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### 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.

```glsl
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.

```glsl
glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );
gEnableClientState( GL_VERTEX_ARRAY );
gEnableClientState( GL_COLOR_ARRAY );
GLuint vertexLocation = glGetAttribLocation( program, "aVertex" );
GLuint colorLocation = glGetAttribLocation( program, "aColor" );
gVertexAttribPointer( vertexLocation, 3, GL_FLOAT, GL_FALSE, 0, (GLuchar *)0 );
gEnableVertexAttribArray( vertexLocation ); // dynamic attribute
glVertexAttrib3f( colorLocation, r, g, b ); // static attribute
gDisableVertexAttribArray( colorLocation );
gDrawArrays( GL_TRIANGLES, 0, 3*NumTris );
```

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

### Using Vertex Buffers with Shaders C++ Class

```cpp
gBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );
gEnableClientState( GL_VERTEX_ARRAY );
gEnableClientState( GL_COLOR_ARRAY );
Pattern->SetVertexAttributePointer3fv( "aVertex", (GLfloat *)0 );
Pattern->EnableVertexAttribArray( "aVertex" ); // dynamic attribute
Pattern->SetVertexAttributeVariable( "aColor", r, g, b ); // static attribute
Pattern->DisableVertexAttribArray( "aColor" );
gDrawArrays( GL_TRIANGLES, 0, 3*NumTris );
```

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

```cpp
gBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );
gEnableClientState( GL_VERTEX_ARRAY );
gEnableClientState( GL_COLOR_ARRAY );
Pattern->SetVertexAttributePointer3fv( "aVertex", (GLfloat *)0 );
Pattern->EnableVertexAttribArray( "aVertex" ); // dynamic attribute
Pattern->SetVertexAttributeVariable( "aColor", r, g, b ); // static attribute
Pattern->DisableVertexAttribArray( "aColor" );
gDrawArrays( GL_TRIANGLES, 0, 3*NumTris );
```