Vertex Buffer Objects

Mike Bailey
mjb@cs.oregonstate.edu

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

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

The Cube Can Also Be Defined with Triangles

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

Did you ever watch Star Trek: Deep Space Nine?

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”.

Impressively, this actually 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.

vertices, colors, normal, texture coordinates

vertex connection indices

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 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".

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.

How many "handles" to generate

The "array" to put them in

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

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:

GLuint bufA;
glGenBuffers( 1, &bufA );
glBindBuffer( GL_ARRAY_BUFFER, bufA );
glBufferData( GL_ARRAY_BUFFER, numBytes, data, usage );
glDrawArrays( GL_TRIANGLES, 0, numVertices );

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:

GLuint bufA;
glGenBuffers( 1, &bufA );
glBindBuffer( GL_ARRAY_BUFFER, bufA );
glBufferData( GL_ARRAY_BUFFER, numBytes, data, usage );
glDrawArrays( GL_TRIANGLES, 0, numVertices );
Loading data into the currently-bound Vertex Buffer Object

```c
void 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.)

For what we are doing, use **GL_STATIC_DRAW**

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
void 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
void glDisableClientState( 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
GLuint size, type;
GLuint stride, offset;

void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer);

void glColorAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer);

void glNormalAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer);

void glTexCoordAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer);
```

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.

---

Step #6 – Draw!

```c
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Example:

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

This is how you do it if your vertices need to be drawn in consecutive order
What if your vertices are to be accessed in random order?

```c
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 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 }
};
```

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

```c
glBindBuffer( GL_ARRAY_BUFFER, bufA );
gBufferData( GL_ARRAY_BUFFER, 3*sizeof(GLfloat)*numVertices, Vertices, GL_STATIC_DRAW );
gBindBuffer( GL_ELEMENT_ARRAY_BUFFER, bufB );
gBufferData( GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint)*numIndices, CubeIndices, GL_STATIC_DRAW );
```
The glDrawElements( ) call

```c
#include <gl.h>

int main()
{
    // Setup the vertex buffer
    GLuint vbo[2];
    glGenBuffers( 2, vbo );
    glBindBuffer( GL_ARRAY_BUFFER, vbo[0] );
    glEnableClientState( GL_VERTEX_ARRAY );
    glVertexPointer( 3, GL_FLOAT, 0, (GLuchar*) 0 );
    glDrawElements( GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, (GLuchar*) 0 );
    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, vbo[1] );
    glDrawElements( GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, (GLuchar*) 0 );
    glBindBuffer( GL_ARRAY_BUFFER, 0 );
    glDeleteBuffers( 2, vbo );
    return 0;
}
```

The glDrawElements( ) call

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

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

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

glMapBuffer Example

```c
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 a Vertex Buffer Object C++ Class

```c
using namespace std;

class VertexBufferObject {
public:
    VertexBufferObject() { }
    void Draw() { }
};
```

Declaring:
```
VertexBufferObject VB;
```

Filling:
```
VB.glBegin( GL_QUADS );
for( int i = 0; i < 6; i++ )
    for( int j = 0; j < 4; j++ )
        VB.glColor3fv( CubeColors[ CubeIndices[ i ][ j ] ] );
VB.glEnd( );
```

Drawing:
```
VB.Draw( );
```
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 glNormal3f( nx, ny, nz );
void glNormal3fv( nxyz[ 3 ] );

Specify a vertex’s normal.

void glEnd( );

Terminate the definition of a primitive.

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:

```c
FILE *fp = fopen( "debuggingfile.txt", "w" );
if( fp == NULL ) {
    fprintf( stderr, "Cannot create file 'debuggingfile.txt'
```

```c
else {
    VB.Print( "My Vertex Buffer ", fp );
    close( 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.

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 Sth 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. Even though the search is only O(log2N), 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

<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>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>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>
</tbody>
</table>

Drawing 24 array elements:

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

A Comparison

Not Collapsing Identical Vertices

Collapsing Identical Vertices

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.

```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 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

```c
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
glDisableVertexAttribArray( colorLocation );
gDrawArrays( 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

```cpp
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 );
```