Vertex Buffer Objects

Mike Bailey
mjb@cs.oregonstate.edu

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

Remember this from the Geometric Modeling Notes?

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

Did any of 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 "pick up this cargo in from docking port A".

Surprisingly, 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.

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.
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.

```cpp
glBindBuffer(GL_ARRAY_BUFFER, bufA);
gDrawArrays(GL_TRIANGLES, 0, numVertices);
```

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:

```cpp
GLuint bufA;
gGenBuffers(1, &bufA);
```

Loading data into the currently-bound Vertex Buffer Object

```cpp
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

```cpp
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

```cpp
 glEnableClientState(type);
```

- **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:

```cpp
 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**

- 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:
  - `glVertexAttribPointer(size, type, stride, offset);`
- Information can be stored as interleaved, like this:
  - `glVertexAttribPointer(size, type, stride, offset);`

### Example

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

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

- **Packed:**
  ```c
  glVertexAttribPointer(size, type, stride, offset);
  ```
- **Interleaved:**
  ```c
  glVertexAttribPointer(size, type, stride, offset);
  ```

**Step #6 – Draw!**

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

**Example:**

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

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

- Start with vertex #0
- Draw 6 vertices

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

```c
glDrawArrays(GL_TRIANGLES, [1, 2, 3, 4, 5, 0]);
```

- Cubes/Triangles/Vertices [3] =
  - (0, 2, 3)
  - (1, 2, 3)
  - (2, 3, 4)
  - (3, 4, 5)
  - (4, 0, 5)

- Colors/CubeColors [3] =
  - (0, 0, 0)
  - (1, 0, 0)
  - (1, 1, 1)
  - (1, 1, 1)
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);
```
### 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 1st triangle strip, half of those points are coincident with points in the S-1 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 N), where N is the number of points kept so far, it will add 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 VertexBufferObject classes, each of size 20 x 1,000.

Just a thought...

### 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)
      { printf( "Cannot create file debuggingfile.txt\n" );
        return( -1 );
    }
  else
      { FILE *fp = fopen( "debuggingfile.txt", "w" );
        VB.Print( "My Vertex Buffer ", fp );
        fclose( fp );
    }
  ```

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

### Drawing the Cube With Collapsing Identical Vertices

- Define 8 points
  - X Y Z
  - 0.00 0.00 0.00
  - 1.00 0.00 0.00
  - 0.00 1.00 0.00
  - 1.00 1.00 0.00
  - 0.00 0.00 1.00
  - 1.00 0.00 1.00
  - 0.00 1.00 1.00
  - 1.00 1.00 1.00

- Define 8 colors
  - R G B
  - 1.00 0.00 0.00
  - 0.00 1.00 0.00
  - 0.00 0.00 1.00
  - 1.00 1.00 1.00

- Draw 24 strip intersections
  - Top view
  - Front view
  - Camera view

- Victor](https://www.example.com)
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.

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

We're also saying 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 the Shaders C++ Class

```cpp
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(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
gVertexAttrib3f(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 set up in program
- we already have the vertices in the vertexBuffer
- we have already created a C++ GLSLProgram class object called Pattern