Vertex Buffer Objects

1

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.

2

The Cube Can Also Be Defined with Triangles

3

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.

4

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 "bring this cargo in from docking port A".

Impressively, this actually has something to do with computer graphics! 😊

5

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.

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

Vertex Buffer
Object

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:

```c
GLuint bufA;
glGenBuffers( 1, &bufA );
```

How many “handles” to generate

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

Loading data into the currently-bound Vertex Buffer Object

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

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

```c
gEnableClientState( 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

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

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

```cpp
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`, or `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"

**Packed:**

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

**Interleaved:**

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

Step #6 – Draw!

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

Example:

```
Example:
```

Cube Example

```cpp
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. }
};
```
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );
gVertexPointer(  3, GL_FLOAT, 0, (Gluchar*) 0 );
gColorPointer(  3, GL_FLOAT, 0, (Gluchar*) (3*sizeof(GLfloat)*numVertices) );
gBegin( GL_QUADS );
gArrayElement( 0 );
gArrayElement( 2 );
gArrayElement( 3 );
gArrayElement( 1 );
gArrayElement( 4 );
gArrayElement( 5 );
gArrayElement( 7 );
gArrayElement( 6 );
gArrayElement( 1 );
gArrayElement( 3 );
gArrayElement( 7 );
gArrayElement( 5 );
gArrayElement( 0 );
gArrayElement( 4 );
gArrayElement( 6 );
gArrayElement( 2 );
gArrayElement( 2 );
gArrayElement( 6 );
gArrayElement( 7 );
gArrayElement( 3 );
gArrayElement( 0 );
gArrayElement( 1 );
gArrayElement( 5 );
gArrayElement( 4 );
gEnd( );

Vertex Data
Color Data

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

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

float * vertexArray = glMapBuffer( GL_ARRAY_BUFFER, usage );
Re-writing Data into a Buffer Object, Treating it as a C/C++ Array of Structures

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 );

Using a Vertex Buffer Object C++ Class

Declaring:
Filling:
Drawing:

glMapBuffer Example

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();

VB.glDraw();

void Point::glBegin( int topological )
    // can be any of the OpenGL topologies
    // for( int i = 0; i < E; ++i )
    //     for( int j = 0; j < 4; ++j )
    //         int k = CubeIndices[ i ][ j ];
    //         VB.glColor3fv( CubeColor[ k ] );
    //         VB.glVertex3fv( CubeVertices[ k ] );
    //     }
void glNormal3fv( nxyz[ 3]  );
void glNormal3f( nx, ny, nz);
void glEnd( );
void glColor3fv( rgb[ 3 ]);
void glColor3f( r, g, b);
void glBegin( topology);

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'
" );
    else
      {  
        VB.Print( "My Vertex Buffer :", fp );
        fclose( fp );
      }
  }

• You can call the gbsign 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 8th triangle strip, half of those points are coincident with points in the 6th-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 side 20 x 1,000.

Just a thought…

A Comparison

<table>
<thead>
<tr>
<th>Not Collapsing Identical Vertices</th>
<th>Collapsing Identical Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></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 Vertex Buffer Object.

```cpp
vec3 aVertex;
in vec3 aColor;

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.

```cpp
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

GLuint vertexLocation = glGetUniformLocation(program, "aVertex");
GLuint colorLocation = glGetUniformLocation(program, "aColor");

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

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

glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
```

We're assuming here that:
- we already have the shader program setup in program
- we already have the vertices in the vertexBuffer

Using Vertex Buffers with Shaders

```cpp
Pattern->SetVertexAttributePointer("aVertex", (GLfloat *)0);
Pattern->EnableVertexAttribArray("aVertex"); // dynamic attribute

Pattern->SetVertexAttributeVariable("aColor", r, g, b); // static attribute

Pattern->DrawArrays(GL_TRIANGLES, 0, 3*NumTris);
```

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
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

GLuint vertexLocation = glGetUniformLocation(program, "aVertex");
GLuint colorLocation = glGetUniformLocation(program, "aColor");

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

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

Pattern->DrawArrays(GL_TRIANGLES, 0, 3*NumTris);
```