Transitioning Students to Post-Deprecation OpenGL

Mike Bailey

Introduction

From an Educator's perspective, teaching OpenGL in the past has been a snap. The separation of geometry from topology in the glBegin-glEnd, the simplicity of glVertex3f, and the classic organization of the post-multiplied transformation matrices has been fast and easy to explain. This considerably excited the students because going from zero-knowledge to cool-3D-program-you-can-smugly-show-your-friends was a matter of a single day. This made motivation easy.

The Great OpenGL Deprecation has changed that. Creating and using vertex buffer objects is a lot more time-consuming to explain than glBegin-glEnd. [Amgel2011] It's also much more error-prone. Creating and maintaining matrices and matrix stacks now requires deft handling of matrix components and multiplication order. [GLM2011] In short, while post-deprecation OpenGL might be more streamlined and efficient, it has wreaked havoc on those who need to teach it, and even more on those who need to learn it.

So, the "old way" is not current, but the "new way" takes a long time to learn before one can see a single pixel. How can we keep students enthusiastic and motivated, but still move them along the road to learning things the new way?[1] This chapter discusses intermediate solutions to this problem by presenting C++ classes that ease the transition to post-deprecation OpenGL. These C++ classes are:

1. Create vertex buffers with methods that look suspiciously like glBegin-glEnd
2. Load, compile, link, and use shaders

This chapter also suggests a naming convention that can be instrumental to keeping shader variables untangled from each other.

Naming Shader Variables, I

This isn't exactly a transition issue. It's more of a confusion-prevention issue, which is always a good thing.

With seven different places that GLSL variables can be set, it is convenient to adopt a naming convention to help recognize what variables came from what sources. This works very well:

[1] One option, of course, is not to transition at all, where the current penalty is simply falling behind the OpenGL curve. However, in some instances, most notably OpenGL-ES 2.0 [Munshi2008], failing to transition is not even an option.

Beginning letter(s)	Means that the Variable …
a	Is a per-vertex attribute from the application
u	Is a uniform variable from the application
v	Came from a vertex shader
tc	Came from a tessellation control shader
te	Came from a tessellation evaluation shader
g	Came from a geometry shader
f	Came from a fragment shader

Table 1. Variable Name Prefix Convention

Naming Shader Variables, II

Variables like `gl_Vertex` and `gl_ModelViewMatrix` have been built-in to the GLSL language from the start. They are used like this:

```
vec4 ModelCoords = gl_Vertex;
vec4 EyeCoords   = gl_ModelViewMatrix * gl_Vertex;
vec4 ClipCoords  = gl_ModelViewProjectionMatrix * gl_Vertex;
vec3 TransfNorm  = gl_NormalMatrix * gl_Normal;
```

However, starting with OpenGL 3.0, they have been deprecated in favor of defining our own variables and passing them in from the application. The built-ins still work if compatibility mode is enabled, but we should all be prepared for them to go away some day. Also, OpenGL- ES has already completely eliminated the built-ins. What to do?

We have chosen to pretend that we have created variables in an application and have passed them in. So, the previous lines of code would be changed to look like this:

```
vec4 ModelCoords = aVertex;
vec4 EyeCoords   = uModelViewMatrix * aVertex;
vec4 ClipCoords  = uModelViewProjectionMatrix * aVertex;
vec3 TransfNorm  = uNormalMatrix * aNormal;
```

If they really are being passed in from the application, fine, go ahead and use these names. But, if you haven't made that transition yet, the new names can still be used (thus preparing for an eventual transition), by including a set of #defines at the top of their shader code:

```
// uniform variables:

#define uModelViewMatrix                gl_ModelViewMatrix
#define uProjectionMatrix               gl_ProjectionMatrix
#define uModelViewProjectionMatrix      gl_ModelViewProjectionMatrix
#define uNormalMatrix                   gl_NormalMatrix
#define uModelViewMatrixInverse         gl_ModelViewMatrixInverse

// per-vertex attribute variables:

#define aColor                          gl_Color
```

```
#define aNormal                         gl_Normal
#define aVertex                         gl_Vertex
#define aTexCoord0                      gl_MultiTexCoord0
#define aTexCoord1                      gl_MultiTexCoord1
#define aTexCoord2                      gl_MultiTexCoord2
#define aTexCoord3                      gl_MultiTexCoord3
#define aTexCoord4                      gl_MultiTexCoord4
#define aTexCoord5                      gl_MultiTexCoord5
#define aTexCoord6                      gl_MultiTexCoord6
#define aTexCoord7                      gl_MultiTexCoord7

#line 1
```

Listing 1. #include file to translate new names to old names

If the graphics driver supports the `ARB_shading_language_include` extension[2], then these lines can be #include'd right into the shader code. If it is not supported, a #include can be "faked" by copying these lines into the first of the multiple strings that are used to load shader source code before compiling.

The #line statement is there so that compiler error messages give the correct line numbers and do not include these lines in the count.

Later on in this chapter, this set of #include lines will be referred to as *gstap.h*[3]

Indexed Vertex Buffer Object C++ Class

There is no question that using `glBegin-glEnd` is convenient, especially when beginning to learn OpenGL. With this in mind, here is a C++ class that looks like the application is using `glBegin-glEnd`, but inside, its data structures are preparing to use indexed VBOs [Shreiner2009] when the class's `Draw( )` method is called. The `Print( )` method's print format shows the data in VBO-table form so the students can see what they would have created if they had used VBOs in the first place.

The following methods are supported by the class:

```
void   CollapseCommonVertices( bool collapse );
void   Draw( );
void   Begin( GLenum type );
void   Color3f( GLfloat red, GLfloat green, GLfloat blue );
void   Color3fv( GLfloat *rgb );
void   End( );
void   Normal3f( GLfloat nx, GLfloat ny, GLfloat nz );
void   Normal3fv( GLfloat *nxyz );
void   TexCoord2f( GLfloat s, GLfloat t );
void   TexCoord2fv( GLfloat *st );
void   Vertex2f( GLfloat x, GLfloat y );
void   Vertex2fv( GLfloat *xy );
```

[2] … and if this line is placed at the top of the shader code:
`#extension GL_ARB_shading_language_include : enable`

[3] … which stands for *Graphics Shaders: Theory and Practice*, the book in which this file originally appeared (Second Edition, A.K. Peters, 2011).

```
void   Vertex3f( GLfloat x, GLfloat y, GLfloat z );
void   Vertex3fv( GLfloat *xyz );
void   Print( char *str = “”, FILE *out = stderr );
void   RestartPrimitive( );
void   SetTol( float tol );
```

Listing 2. Vertex Buffer Object Class methods

Usage Notes:

- This implements an indexed VBO, that is, it keeps track of the vertices’ index in the VBO and then uses `glDrawElements( )` to display the object.

- Passing a true to the `CollapseCommonVertices( )` method’s Boolean argument says that any vertices “close enough” to each other should be collapsed to be treated as a single vertex. “Close enough” is defined by the distance specified in `SetTol( )`. The advantage to this is that the single vertex gets transformed once per display update. The disadvantage is that the collapsing process takes time, especially for large lists of vertices.

- The `RestartPrimitive( )` method invokes an OpenGL-ism that restarts the current primitive topology without starting a new VBO. It is especially handy for triangle strips and line strips. For example, if the topology is triangle strip, then `RestartPrimitive( )` would allow the application to end one strip and start another, and have all the vertices end up in a single VBO. This saves overhead.

- The first call to the `Draw( )` method sends the VBO data to the graphics card and draws it. Subsequent calls to `Draw( )` just do the drawing.

Example Code:

```
#include “VertexBufferObject.h”

VertexBufferObject VB;
. . .

// this goes in the part of the program where graphics things
// get initialized once:

VB.CollapseCommonVertices( true );
VB.SetTol( .001f );              // how close need to be to collapse

VB.Begin( GL_QUADS );
for( int i = 0; i < 6; i++ )
{
       for( int j = 0; j < 4; j++ )
       {
              VB.Color3fv( . . . );
              VB.Vertex3fv( . . .] );
       }
}
```

```
VB.End( );
VB.Print( “VB:” );  // verify that vertices were really collapsed

. . .


// this goes in the display-callback part of the program:

VB.Draw( );
```
Listing 3. Vertex Buffer Object Class used to draw a colored cube

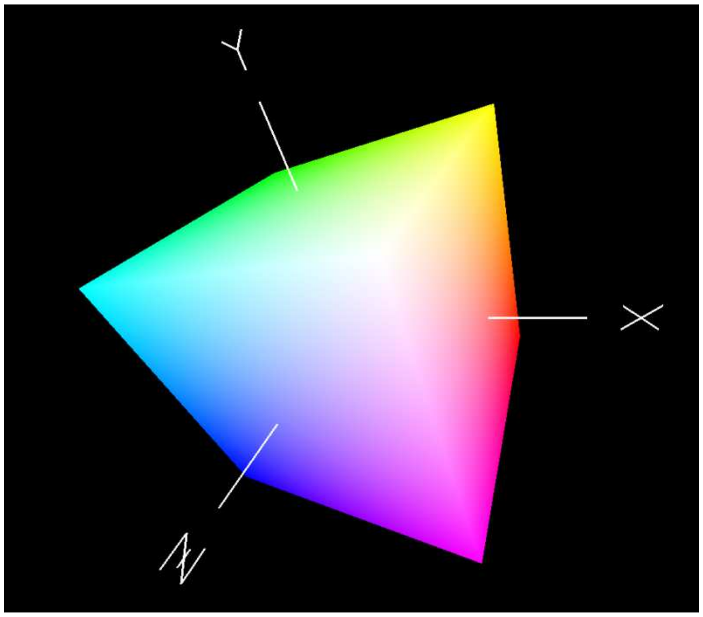

Figure 1. Colored Cube created with the VertexBufferObject class

coloredcube.tif

This next example shows drawing gridlines on a terrain map. The already-defined Heights[ ] array holds the terrain heights. This is a good example of using the `RestartPrimitive( )` method so that the next grid line doesn't have to be in a new line strip. The entire grid is saved as a *single* line strip, and is drawn by blasting a *single* VBO into the graphics pipeline.

```
// create an instance of the class:
// (the real “constructor” is in the Begin method)
VertexBufferObject VB;
VB.CollapseCommonVertices( true );
VB.SetTol( .001f );
      . . .

// this goes in the part of the program where graphics things
// get initialized once:

int x, y;           // loop indices
float ux, uy;// utm coordinates
```

```
VB.Begin( GL_LINE_STRIP );

for( y = 0, uy = meteryMin;  y < NumLats;  y++, uy += meteryStep )
{
	VB.RestartPrimitive(  );
	for( x = 0, ux = meterxMin; x < NumLngs  x++, ux += meterxStep )
	{
			float uz = Heights[  y*NumLngs + x  ];
			VB.Color3f( 1., 1., 0. ); // single color = yellow
			VB.Vertex3f( ux, uy, uz );
	}
}

for( x = 0, ux = meterxMin; x < NumLngs; x++, ux += meterxStep )
{
	VB.RestartPrimitive(  );
	for( y = 0, uy = meteryMin; y < NumLats; y++, uy += meteryStep )
	{
		float uz = Heights[  y*NumLngs + x  ];
		VB.Color3f( 1., 1., 0. );
		VB.Vertex3f( ux, uy, uz );
	}
}

VB.End(  );
VB.Print( "Terrain VBO:" );
. . .


// this goes in the display-callback part of the program:

VB.Draw( );
```
Listing 4. Vertex Buffer Object Class used to draw a wireframe terrain

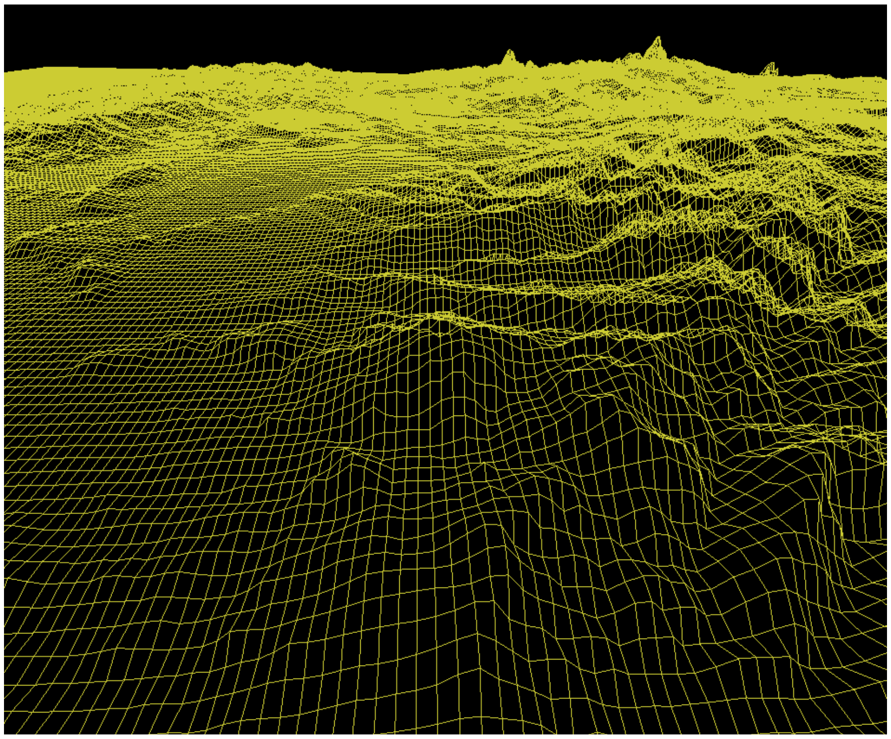
Figure 2. Wireframe Terraine created with the VertexBufferObject class

wireterrain.tif

Implementation Notes:

- This class uses the C++ Standard Template Library (STL) vector function to maintain the ever-expanding array of vertices.

- It also uses the C++ STL map function to speed the collapsing of common vertices.

GLSLProgram C++ Class

The act of creating, compiling, linking, using, and passing parameters to shaders is very repetitive. [Rost2009, Bailey2011] When teaching students, we have found it helpful to create a C++ class called *GLSLProgram* that implements this process. This class has the tools to manage all the steps of shader program development and use, including source file opening, loading, and compilation. It also has methods that implement setting attribute and uniform variables.

The following methods are supported by the class:

```
bool    Create( char *, char * = NULL, char * = NULL, char * = NULL, char * = NULL );
bool    IsValid( );
void    SetAttribute( char *name, int val );
void    SetAttribute( char *name, float val );
void    SetAttribute( char *name, float val0, float val1, float val2 );
void    SetAttribute( char *name, float *valp );
void    SetAttribute( char *name, Vec3& vec3 );
void   SetAttribute( char *name, VertexBufferObject& vb, GLenum which );
void    SetGstap( bool set );
void    SetUniform( char *name, int );
void    SetUniform( char *name, float );
void    SetUniform( char *name, float, float, float );
void    SetUniform( char *name, float[3] );
void    SetUniform( char *name, Vec3& );
void    SetUniform( char *name, Matrix4& );
void    Use( );
void    UseFixedFunction( );
```
Listing 5. GLSLProgram class methods

Usage Notes:

- The Create( ) method takes up to five shader file names as arguments. From the filename extensions, it figures out what type of shaders these are, loads them, compiles them, and links them all together. All errors are written to *stderr*[4]. It returns true is the resulting shader binary program is valid, or false if it is not. The `IsValid( )` method can be called later if the application wants to know if everything succeeded or not. The files listed in the `Create( )` call can be in any order.

- The filename extensions that the `Create( )` method is looking for are:

Extension	Shader Type
.vert	GL_VERTEX_SHADER
.vs	GL_VERTEX_SHADER
.frag	GL_FRAGMENT_SHADER
.fs	GL_FRAGMENT_SHADER
.geom	GL_GEOMETRY_SHADER
.gs	GL_GEOMETRY_SHADER
.tcs	GL_TESS_CONTROL_SHADER
.tes	GL_TESS_EVALUATION_SHADER

Table 2. Shader Type Filename Extensions

- The `SetAttribute( )` methods set attribute variables, to be passed to the vertex shader.

- The vertex buffer version of the `SetAttribute( )` method lets a `VertexBufferObject` be specified along with which data inside it is to be assigned to this attribute name. For example, if one might say:
```
GLSLProgram  Ovals;
VertexBufferObject VB;
```

[4] Standard error is used for these messages because it is unbuffered. If a program crashes, the helpful messages sent to standard output might still be trapped in a buffer and will not be seen. Those messages sent to standard error were seen right away.

```
    ...
Ovals.SetAttribute( “aNormal”, VB, GL_NORMAL_ARRAY );
```

- The `SetUniform( )` methods set uniform variables, destined for any of the shaders.

- The Use( ) method makes this shader program active, so that it affects any subsequent drawing. If someone insists on using the fixed functionality, the `UseFixedFunction( )` method returns the state of the pipeline to use the fixed-functionality.

- The `SetGstap( )` method is there to give the option to have the *gstap.h* code included automatically. Just pass true as the argument. Call this before the call to the `Create( )` method.

Example Code:

```
#include "GLSLProgram.h"

float        Ad, Bd, NoiseAmp, NoiseFreq, Tol;
GLSLProgram  Ovals;
VertexBufferObject VB;

. . .

// set everything up once:

Ovals.SetVerbose( true );
Ovals.SetGstap( true );
bool good = Ovals.Create( "ovalnoise.vert", "ovalnoise.frag" );
if( ! good  )
{
      fprintf( stderr, "GLSL Program Ovals wasn’t created.\n" );
      . . .
}

. . .


// do this in the display callback:

Ovals.Use( );
Ovals.SetUniform( "uAd", Ad );
Ovals.SetUniform( "uBd", Bd );
Ovals.SetUniform( "uNoiseAmp",  NoiseAmp );
Ovals.SetUniform( "NoiseFreq",  NoiseFreq );
Ovals.SetUniform( "uTol", Tol );
Ovals.SetAttribute(   “aVertex”,  VB,  GL_VERTEX_ARRAY );
Ovals.SetAttribute(   “aColor”,   VB,  GL_COLOR_ARRAY );
Ovals.SetAttribute(   “aNormal”,  VB,  GL_NORMAL_ARRAY );

VB.Draw( );
```
Listing 6. GLSLProgram class application example

Implementation Notes:

- The `SetAttribute( )` and `SetUniform( )` methods use the C++ Standard Template Library *map* function to relate variable names to variable locations in the shader program symbol table. It only ever really looks them up once.

Conclusions

From a teaching perspective, the simplicity of explanation and the speed to develop an application have long been advantages of using OpenGL in its fixed-function, pre-deprecation state. Students of OpenGL *do* need to learn how to use OpenGL in the post-deprecation world. However, they don't need to learn it right from the start.

These notes have presented a way of starting students out in the ways that are easier for them to learn, but are still using the recommended ways underneath. As they get comfortable with graphics programming, the "underneath" can be revealed to them. This sets the students up for using shaders and vertex buffer objects.

These notes have also suggested a shader-variable naming convention. As shaders become more complex, and as more variables are being passed between the shaders, we have found this useful to keep shader variables names untangled from each other. This naming convention, along with the gstap.h file, sets students up for passing their own quantities into their shaders.

Bibliography

Angel2011 Edward Angel and Dave Shreiner, *Interactive Computer Graphics: A Top-down Approach with OpenGL,* 6th Edition, Addison-Wesley, 2011.

Bailey2011 Mike Bailey and Steve Cunningham, *Computer Graphics Shaders: Theory and Practice*, Second Edition, AK Peters, 2011.

GLM2011 GLM: OpenGL Mathematics, http://glm.g-truc.net/

Munshi2008 Aaftab Munshi, Dan Ginsburg, and Dave Shreiner, *OpenGL ES 2.0*, Addison-Wesley, 2008.

Rost2009 Randi Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen, *OpenGL Shading Language*, Addison-Wesley, 2009. (3rd Edition)

Shreiner2009 Dave Shreiner, *OpenGL 3.0 Programming Guide*, Addison-Wesley, 2009 (7th edition).