Teaching OpenGL in a Post-Deprecation World

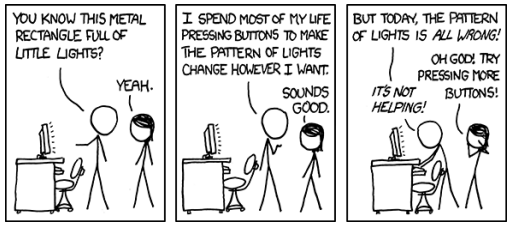
Mike Bailey
mjb@cs.oregonstate.edu

Oregon State University

Our motto: "We figure out the spec so you don't have to." ☺

http://cs.oregonstate.edu/~mjb/sig12

Life in the World of OpenGL – Always Adding More Buttons to Push ! :-)



http://xkcd.com

OpenGL / GLSL Release History

OpenGL Release	GLSL Release	When
1.0	---	1993
1.1	---	1997
1.2	---	1998
1.3	---	2001
1.4	---	2002
1.5	---	2003
2.0	1.10	2004
2.1	1.20	2006
3.0	1.30	July 2008
3.1, 3.2, 3.3	3.30	July 2009
4.0	4.00	March 2010
4.1	4.10	July 2010
4.2	4.20	August 2011
4.3	4.30	August 2012

OpenGL 3.x deprecated some features that we, as educators, care about

"Deprecate" doesn't mean it has gone away *now*, but means that it will go away "at some time", which is undefined so far. There is some evidence that "at some time" could mean "never". Or not.

Deprecated features include:

- The Fixed-Function pipeline (will need to use shaders for everything)
- glBegin / glEnd (use vertex buffer objects)
- Quads (use triangles)
- Polygons (use triangles)
- Built-in variables in GLSL (pass in everything yourself)

These are the 3 items that I think are most troubling from a teaching perspective

You could, of course, continue to use the old features by using OpenGL's compatibility mode, but if those features ever really *do* go away, you will be stuck with a bunch of code you need to modify. What is the right compromise here?

GLSL Deprecation – Transitioning from Built-in Variables

Variables like **gl_Vertex** and **gl_ModelViewMatrix** have been built-in to the GLSL language.

However, starting with Desktop OpenGL 3.0, they have been deprecated in favor of you defining your own variables and passing them in from the application yourself. The built-ins still work, but be prepared for them to maybe go away some day. Also, OpenGL-ES has already completely *eliminated* the built-ins.

What to do?

I now pretend that we have created variables in an application and have passed them in. So, lines of code would be changed to look like:

```
vec4 ModelCoords = gl_Vertex ;
vec4 ModelCoords = aVertex ;

vec4 EyeCoords = gl_ModelViewMatrix * gl_Vertex ;
vec4 EyeCoords = uModelViewMatrix * aVertex ;

vec4 ClipCoords = gl_ModelViewProjectionMatrix * gl_Vertex ;
vec4 ClipCoords = uModelViewProjectionMatrix * aVertex ;

vec3 TransfNorm = gl_NormalMatrix * gl_Normal ;
vec3 TransfNorm = uNormalMatrix * aNormal ;
```

Why do some of the variables begin with 'a'?
Why do some begin with 'u'?

My Own Variable Naming Convention

With 7 different places GLSL variables can be written from, I decided to adopt a naming convention to help recognize what variables came from what sources:

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 the vertex shader
tc	Came from the tessellation control shader
te	Came from the tessellation evaluation shader
g	Came from the geometry shader
f	Came from the fragment shader

This isn't part of "official" OpenGL – it is *my* way of handling the confusion

Handling the Transition Now

This is how I equivalence the new names to the deprecated (but still working) ones:

File *gstap.h*

```
// uniform variables:

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

// 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
```

This isn't part of "official" OpenGL – it is *my* way of handling the transition

GLSL Deprecation – Transitioning from glBegin-glEnd

glBegin-glEnd was my *favorite* OpenGL feature when it came time to teach it. It is so easy to understand and learn that students could go from no-knowledge-at-all to working-3D-program-to-smugly-show-their-friends in the first week.

Now everything is supposed to be done using vertex buffer objects, which is much harder to pick up and write a program around.

I have been trying the approach of using a C++ class that looks like glBegin-glEnd, but underneath really uses a vertex buffer object. The students can start here, and then eventually see how it should be formatted in the new way.

Using the Vertex Buffer Object Class

Setting Up:

```
VertexBufferObject   Blob( );
Blob.CollapseCommonVertices( true );
```

Filling:

```
Blob.glBegin( GL_TRIANGLES );    // can be any of the OpenGL topologies
        Blob.glColor3f(    r0, g0, b0 );
        Blob.glVertex3v( x0, y0, z0 );
        . . .
Blob.glEnd( );
```

Drawing:

```
Blob.Draw( );
```


Vertex Buffer Object Class Methods

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.

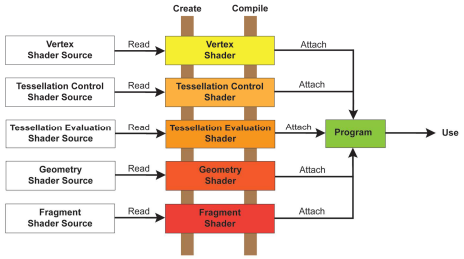
void Draw( );	Draw the primitive. If this is the first time Draw( ) is being called, it will setup all the proper buffer objects, etc. If it as a subsequent call, then it will just initiate the drawing.
void glBegin(topology);	Initiate the primitive.
void glColor3f(r, g, b); void glColor3fv(rgb[3]);	Specify a vertex's color.
void glEnd( );	Terminate the definition of this primitive.
void glNormal3f(nx, ny, nz); void glNormal3fv(nxyz[3]);	Specify a vertex's normal.

Vertex Buffer Object Class Methods

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(FILE *);	Prints the vertex, normal, color, texture coordinate, and connection element information to a file. 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 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.
void UseBufferObjects(bool);	*false* means to use vertex arrays instead of vertex buffer objects. The big advantage of buffer objects is that the data all lives on the graphics card so that it only ever needs to be transferred once. Vertex Array data is kept in host memory and so needs to be transferred each time it is drawn. The default is to use VBOs if they are supported on your graphics system, and vertex arrays if they are not.

GLSL Deprecation – Transitioning from a Fixed Function Pipeline

Getting a GLSL shader ready to go requires several steps, none of which has to do with understanding *computer graphics*.



Once again, a C++ class seems to be able to help smooth the transition.

Loading, Compiling, and Linking GLSL Shaders

```
int    Polar;
float K;

GLSLProgram   Hyper( );
Hyper.SetVerbose( true );
bool valid = Hyper.Create( "hyper.vert", "hyper.frag" );

if( ! valid ) { print something; do something; }
```

The constructor creates the *GLSLProgram* object, but doesn't put anything in it.

The *Create( )* method loads, compiles, and links all the shaders into one shader program.
The *Create( )* method prints error messages if something failed.
Your program can test for success by seeing if the *Create( )* return is *true*.

You can list as many individual shader file types in the *Create( )* method as you are using, up to the maximum number of shader types that OpenGL currently supports), **in any order**.

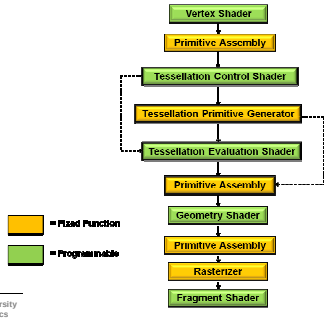
Recognizing the Different Types of Shader File

The *GLSLProgram Create( )* method recognizes shader types by their filename extensions. Recognized extensions are:

Type of Shader	Filename Extension
Vertex	.vert
Vertex	.vs
Tessellation Control	.tcs
Tessellation Evaluation	.tes
Geometry	.geom
Geometry	.gs
Fragment	.frag
Fragment	.fs

The Order that Shaders are used has nothing to do with the Order in which they are Passed to the *GLSLProgram Create( )* method

Shader files can be passed into the *GLSLProgram Create( )* method in any order.
They will still operate on your graphics per the OpenGL standard order shown here:



Making this Shader Program Active and Inactive

Make this the active shader program with:

```
Hyper.Use( );
```

Revert to the fixed-function pipeline with:

```
Hyper.UseFixedFunction( );
```


Passing in Uniform and Attribute Variables

```
GLfloat   Polar,   K;
          . . .
Hyper.SetUniformVariable( "Polar", Polar );
Hyper.SetUniformVariable( "K", K );
          . . .
Hyper.Use( );
glBegin( GL_TRIANGLES );

          Hyper.SetAttributeVariable( "Temperature", T0 );
          glVertex3f( x0, y0, z0 );

          Hyper.SetAttributeVariable( "Temperature", T1 );
          glVertex3f( x1, y1, z1 );

          Hyper.SetAttributeVariable( "Temperature", T2 );
          glVertex3f( x2, y2, z2 );

glEnd( );
```


And …

The GLSLProgram class will also handle the major new feature that is being announced in the OpenGL 4.3 press conference on Monday.

But, if I say anything about that now, they will kill me.

Khronos Group members:

Teaching OpenGL in a Post-Deprecation World

Mike Bailey
mjb@cs.oregonstate.edu

Oregon State University

Our motto: "We figure out the spec so you don't have to." ☺

http://cs.oregonstate.edu/~mjb/sig12

This page will be modified Monday evening…