What You Really Need to Know About Recent Changes to OpenGL and GLSL

Mike Bailey
mjb@cs.oregonstate.edu

Oregon State University

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	2008
3.3	3.30	2009
4.0	4.00	2010

Features of OpenGL 2.0 / GLSL 1.1 Worth Knowing About
(in the order of what I think are most important)

- Programmable vertex and fragment shaders *Oh, yeah!*
- Vertex buffer objects Store vertex arrays in graphics memory
- Occlusion queries Ask how many pixels a particular scene element would occupy if displayed
- Texture-mapped point sprites Good for many small 2D objects
- Separate stencil operations for front and back faces Good for shadowing

Features of OpenGL 3.3 / GLSL 3.3 Worth Knowing About
(in the order of what I think are most important)

- Geometry shaders Primitive expansion
- Texture buffer objects Textures and parameters stored in graphics memory
- Named uniform variable blocks More efficient way t pass blocks of uniform variables
- Texture size query Ask the size of a texture so know how to advance to adjacent texels
- Centroid, flat, invariant, noperspectve qualifiers Affect how varying variables are interpolated
- Buffer object subimage mapping Able to memory-map part of a buffer object
- Texture arrays Keep arrays of textures, including cube ,maps
- Layout qualifiers Set some characteristics of named block variables
- 16-bit floats 16-bit floating point variables
- Rectangular textures Integer-addressed, reduced functionality texture, useful for video processing

OpenGL 3.x deprecated several things

"Deprecate" doesn't mean it has gone away now, but means that it will go away "at some time", which is undefined so far.

Deprecated features include:

- The Fixed-Function pipeline (will need to use shaders for everything)
- glBegin / glEnd (use vertex arrays and vertex buffers)
- Display lists (use vertex arrays and vertex buffers) [?????]
- Quads (use triangles)
- Polygons (use triangles)

What was Different about OpenGL 3.0?

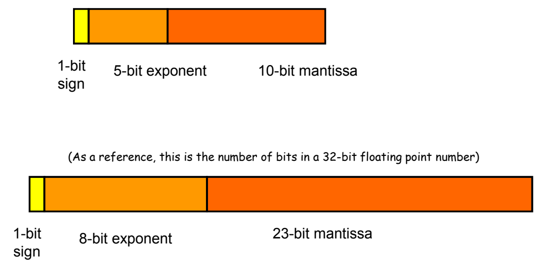
OpenGL 3.0 was the same as the OpenGL you knew with the following differences:

- There is no Fixed-Function pipeline. All graphics functionality needs to be implemented with GLSL shaders.
- There are no Display Lists
- There is no glBegin( ) - glEnd( ). All primitives are drawn with Vertex Arrays or Vertex Buffers.
- GLSL variables can have precision qualifiers These are `lowp, mediump`, and `highp`. These don't do anything, but makes the language compaticle with GLSL for OpenGL ES.
- GLSL variables can have the *invariant* qualifier so that the compiler will not use any optimizations when computing them. This is useful to be sure that successive rendering passes produce the same coordinates.

OpenGL 3.x Data Types

Type	Bits	Function Suffix
Byte	8	b
Unsigned byte	8	ub
Short	16	s
Unsigned short	16	us
Int	32	i
Unsigned int	32	ui
Fixed point	32 (16.16)	x
Floating point	32	f

OpenGL 3.x Optional Half-float Data Type



GLSL 3.30 deprecated several things

"Deprecate" doesn't mean it has gone away now, but means that it will go away "at some time", which is undefined so far.

Deprecated features include:

- The Fixed Function pipeline (in the future, all OpenGL programs will require you to use shaders)
- The attribute and varying keywords (replaced with *out* and *in*)
- *gl_ClipCoord* (replaced with *gl_ClipDistance[ ]*)
- The *ftransform( )* function
- Almost all built-in variables, such as *gl_ModelViewMatrix*, *gl_Color*, etc. These are replaced with variables that you define for yourself as inputs to your shaders.

What was Different about GLSL 3.30?

GLSL 3.30 was the same as the GLSL you knew with the following differences:

- Full integer support, including all standard C integer operations
- Full unsigned integer support, including all standard C unsigned integer operations
- Hyperbolic and inverse hyperbolic trigonometric functions
- Switch statements
- attribute variables in a vertex shader will now be declared *in*
- varying variables in a vertex shader will be declared *out*.
- varying variables in a fragment shader will be declared *in*
- *gl_FragColor* and *gl_FragData[ ]* in a fragment shader are no longer used. You define your own variable names and declare them *out*

What was Different about GLSL 3.30?

GLSL 3.30 was the same as the GLSL you knew with the following differences:

- varying in variables in a geometry shader are declared *in*
- varying out variables in a geometry shader are declared *out*
- Textures can be indexed by integers
- Textures can return integer values
- Texture sizes can be queried
- Texture arrays
- The preprocessor can perform token-pasting (##)

What was Different about GLSL 3.30?

GLSL 3.30 was the same as the GLSL you knew with the following differences:

- There is a new *gl_VertexId* variable which tells you which vertex this is in a vertex array
- User-clipping is performed with the *gl_ClipDistance[ ]* array
- An overloaded version of the *mix( )* function has a Boolean as the third argument, which lets it act as a switch between the first two arguments
- Where you used to used *ftransform( )* to get an exact *gl_Position* for multipass rendering, now use the *invariant* keyword.

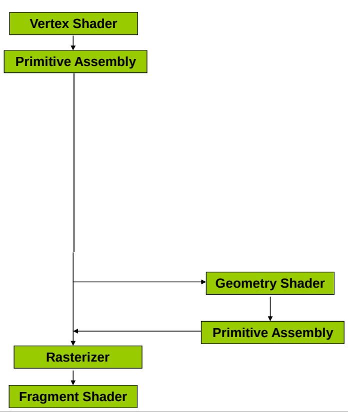
Features of OpenGL 4.0 / GLSL 4.0 Worth Knowing About
(in the order of what I think are most important)

- Tessellation shaders Subdivide geometry into smaller pieces for smoothness and displacement mapping
- Subroutines Keep multiple ways of doing things in a single shader, but avoid if-statements by using function jump tables
- Instanced geometry shaders Able to do multiple iterations through a single geometry shader to recursively subdivide
- Precise qualifier Optionally prevents the compiler from optimizing an expression – useful to maintain computational consistency in multipass algorithms
- Function overloading Just like C++
- Fused multiply-add **fma(a,b,c)** performs (a*b)+c but in a single instruction without the loss of precision that happens with an intermediate result
- #include Finally!
- Geometry shader streams Transform feedback from a geometry shader
- Double precision 64-bit IEEE floating point variables
- Texture gather Grab the four surrounding texel values and interpolate them yourself
- Timer query Asynchronous timing of individual pipeline instructions

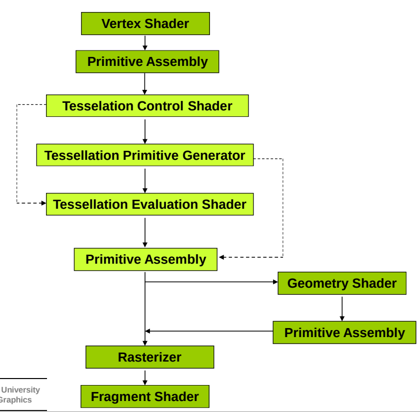
Why do we need a Tessellation step right in the pipeline?

- You can perform adaptive subdivision based on a variety of criteria
- You can provide coarser models (≈ geometric compression)
- You can apply detailed displacement maps without supplying equally detailed geometry
- You can adapt visual quality to the required level of detail
- You can create smoother silhouettes
- You can perform skinning easier

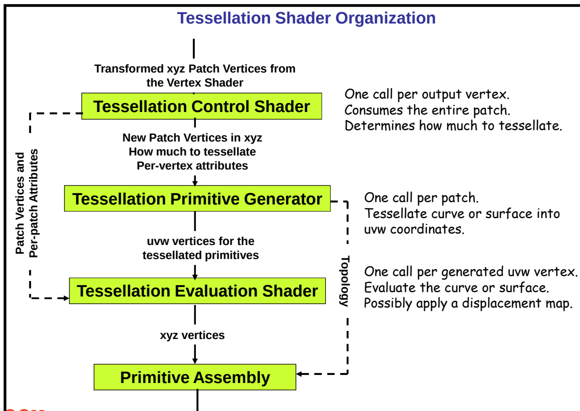
Pipeline Organization without Tessellation



Pipeline Organization with Tessellation



Tessellation Shader Organization



Tessellation Shader Organization

The **Tessellation Control Shader (TCS)** transforms the input coordinates to a regular surface representation. It also computes the required tessellation level based on distance to the eye, screen space spanning, hull curvature, or displacement roughness. There is one invocation per output vertex.

The Fixed-Function **Tessellation Primitive Generator (TPG)** generates semi-regular u-v-w coordinates. There is one invocation per patch.

The **Tessellation Evaluation Shader (TES)** evaluates the surface in *uvw* coordinates. It interpolates attributes and applies displacements. There is one invocation per generated vertex.

There is a new “Patch” primitive – it is the face and its neighborhood:
glBegin(GL_PATCHES)
There is no implied order – that is user-given.

In the OpenGL Program

```
glBegin( GL_PATCHES );
        glVertex3f( … );
        glVertex3f( … );
glEnd( );
```

These have no implied topology

```
GLuint tcs = glCreateShader( GL_TESS_CONTROL_SHADER );

GLuint tes = glCreateShader( GL_TESS_EVALUATION_SHADER );
```


What are GLSL Subroutines?

- Essentially, they are "jump tables" through which you can make an indexed function call.
- This is important in some applications because if-statements are so costly in a SIMD environment
- An example might be different kinds of lighting. Rather than changing the shader program or doing sets of if-tests, you could have functions that do the different types of lighting, and just decide which set of functions need to get called
- GLSL subroutines are context-state, not program-state like normal uniform variables. This is because it has been anticipated that these will be used across a number of GLSL programs.

In the GLSL Code

```
#extension GL_ARB_shader_subroutine : required;
subroutine vec3 SetColor( float );
vec3 SetRed( float );
vec3 SetGreen( float );

main( )
{
        subroutine uniform SetColor   WhichColor;


        vec3 color = WhichColor( Scale );
}

subroutine( SetColor )
vec3
SetRed( float s )
{
        return vec3( s, 0., 0. );
}

subroutine( SetColor )
vec3
SetGreen( float s )
{
        return vec3( 0., s, 0. );
}
```

Define the *SetColor* collection of functions

Define GLSL functions as usual

Declare a uniform variable which the proper function will be "jumped through"

Do the indirect function call

The indirect function call will really call one of these

Note: undefined things will happen if the WhichColor variable is not assigned to by the OpenGL program !

In the OpenGL Code

```
GLint  where = glGetSubroutineUniformLocation( program, shader_type, "WhichColor" );

if( where < 0 ) { . . . }

GLuint setred     = glGetSubroutineIndex( program, shader_type, "SetRed"    );
GLuint setgreen = glGetSubroutineIndex( program, shader_type, "SetGreen" );

glUniformSubroutinesuiv( shader_type, 1, &setgreen );
```

Where "WhichColor" is in the shader symbol table

Set which of the SetColor functions will get called.

What the index numbers of the different "SetColor" functions are