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

Mike Bailey
mbj@cs.oregonstate.edu
Oregon State University

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
• Vertex buffer objects
• Occlusion queries
• Texture-mapped point sprites
• Separate stencil operations for front and back faces

Oh, yeah!
Store vertex arrays in graphics memory
Good for many small 2D objects
Ask how many pixels a particular scene element would occupy if displayed
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
• Texture buffer objects
• Named uniform variable blocks
• Texture size query
• Centroid, flat, invariant, noperspective qualifiers
• Buffer object subimage mapping

Textures and parameters stored in graphics memory
Ask the size of a texture so know how to advance to adjacent texels
More efficient way to pass blocks of uniform variables
Affect how varying variables are interpolated

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)
• Quadrics (use triangles)
• Polygons (use triangles)

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 compatible 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Function Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8</td>
<td>b</td>
</tr>
<tr>
<td>Unsigned byte</td>
<td>8</td>
<td>ub</td>
</tr>
<tr>
<td>Short</td>
<td>16</td>
<td>s</td>
</tr>
<tr>
<td>Unsigned short</td>
<td>16</td>
<td>us</td>
</tr>
<tr>
<td>Int</td>
<td>32</td>
<td>i</td>
</tr>
<tr>
<td>Unsigned int</td>
<td>32</td>
<td>ui</td>
</tr>
<tr>
<td>Fixed point</td>
<td>32 (16.16)</td>
<td>x</td>
</tr>
<tr>
<td>Floating point</td>
<td>32</td>
<td>f</td>
</tr>
</tbody>
</table>

### OpenGL 3.x Optional Half-float Data Type

<table>
<thead>
<tr>
<th>1-bit sign</th>
<th>5-bit exponent</th>
<th>10-bit mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>(As a reference, this is the number of bits in a 32-bit floating point number)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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
  - Divide geometry into smaller pieces for smoothness and displacement mapping

- Subroutines
  - Allow 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
  - Like C++, fma(a,b,c) performs (a*b)+c but in a single instruction without losing precision that happens with an intermediate result

- 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 use 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

- Vertex Shader
- Primitive Assembly
- Geometry Shader
- Rasterizer
- Fragment Shader

Pipeline Organization with Tessellation

- Vertex Shader
- Primitive Assembly
- Tessellation Control Shader
- Tessellation Primitive Generator
- Tessellation Evaluation Shader
- Primitive Assembly
- Geometry Shader
- Rasterizer
- Primitive Assembly
- Fragment Shader

Tessellation Shader Organization

- Tessellation Control Shader
  - One call per output vertex. Consumes the entire patch. Determines how much to tessellate.
  - Transformed vertex coordinates from the Vertex Shader

- Tessellation Primitive Generator
  - One call per patch. Tessellate curve or surface into u-v-w coordinates.
  - u-v-w vertices for the tessellated primitives

- Tessellation Evaluation Shader
  - One call per generated u-v-w vertex. Evaluate the curve or surface. Possibly apply a displacement map.
  - u-v-w vertices

- Primitive Assembly
  - 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

```gl
begin( GL_PATCHES );
vertex3f( ... );
vertex3f( ... );
gend();
```

These have no implied topology

```gl
GLuint tcs = glCreateShader( GL_TESS_CONTROL_SHADER );
GLuint tes = glCreateShader( GL_TESS_EVALUATION_SHADER );
```

In the GLSL Code

```gl

define SetColor collection of functions
```

```gl
vec3 SetRed( float s )
```

```gl
vec3 return vec3( s, 0., 0. );
```

```gl
vec3 SetGreen( float s )
```

```gl
vec3 return vec3( 0., s, 0. );
```

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

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.

```gl
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 );
}
```

In the OpenGL Code

```gl
GLuint where = glGetSubroutineUniformLocation( program, shader_type, "WhichColor" );
if( where < 0 ) { ... }
GLuint setred = glGetSubroutineIndex( program, shader_type, "SetRed" );
GLuint setgreen = glGetSubroutineIndex( program, shader_type, "SetGreen" );
gUniformSubroutinesuiv( shader_type, 1, &setgreen );
```

Where "WhichColor" is in the shader symbol table.

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