The Transition from RenderMan to the OpenGL Shading Language (GLSL)

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
All Six RenderMan Shader Types

1. Displacement  
≈ GLSL Vertex Shader

2. Distortion / transformation

3. Surface  
≈ GLSL Fragment Shader

4. Lighting

5. Atmospheric / volumetric

6. Imaging

RenderMan Built-in Microfaceting  
≈ Manual or GLSL Tessellation
### Fundamental Differences Between RenderMan Shaders and OpenGL Shaders

<table>
<thead>
<tr>
<th>Topic</th>
<th>RenderMan</th>
<th>GLSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goals</td>
<td>1. Image quality, 2. Speed</td>
<td>1. Speed, 2. Image quality</td>
</tr>
<tr>
<td>Shader Types</td>
<td>Surface, Displacement (+4 others)</td>
<td>Vertex, Fragment, Geometry, Tessellation, Compute</td>
</tr>
<tr>
<td>Surface Preprocessing</td>
<td>Microfacets</td>
<td>None [± Tessellation shaders]</td>
</tr>
<tr>
<td>Recompute Normals</td>
<td>CalculateNormal</td>
<td>None</td>
</tr>
<tr>
<td>Getting Rid of Pixels</td>
<td>Oi = 0.; discard;</td>
<td>discard;</td>
</tr>
<tr>
<td>Surface/Fragment shader sets</td>
<td>R, G, B, αr, αg, αb</td>
<td>R, G, B, A [,Z]</td>
</tr>
<tr>
<td>Shader Variables</td>
<td>Uniform, Varying</td>
<td>Attribute, Uniform, Out, In</td>
</tr>
<tr>
<td>Coordinate Systems</td>
<td>Shader (Object), World</td>
<td>Model (=OC), Eye (≈WC)</td>
</tr>
<tr>
<td>Noise</td>
<td>Built-in</td>
<td>Somewhat built-in or use a Texture</td>
</tr>
<tr>
<td>Compile Shaders</td>
<td>Must do yourself</td>
<td>Driver does it for you</td>
</tr>
<tr>
<td>Compiler messages</td>
<td>Cryptic</td>
<td>Cryptic</td>
</tr>
</tbody>
</table>
### GLSL Variable Types

<table>
<thead>
<tr>
<th><strong>Attribute</strong></th>
<th>Assigned <em>per-vertex</em> and passed into the vertex shader, usually with the intent to interpolate through the rasterizer.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Uniform</strong></td>
<td>“Global” values, assigned and left alone for a group of primitives. Read-only accessible from all of your shaders. (Cannot be written from a shader.)</td>
</tr>
<tr>
<td><strong>Out / In</strong></td>
<td>Passed from one shader stage to the next shader stage.</td>
</tr>
</tbody>
</table>
GLSL Shaders Are Like C With Extensions for Graphics:

- Types include int, ivec2, ivec3, ivec4
- Types include float, vec2, vec3, vec4
- Types include mat2, mat3, mat4
- Types include bool, bvec2, bvec3, bvec4
- Types include sampler to access textures
- Vector components are accessed with [index], .rgba, .xyzw, or .stpq
- Can ask for parallel SIMD operations (doesn’t necessarily do it in hardware):
  
  ```glsl
  vec4 a, b, c;
  a = b + c;
  ```

- Vector components can be “swizzled” (c1.rgba = c2.abgr)
- Type qualifiers: const, attribute, uniform, varying, in, out
- Variables can have “layout qualifiers” (more on this later)
- The `discard` operator is used in fragment shaders to get rid of the current fragment
The *discard* Operator

```cpp
if( alpha == 0. )
    discard;
```
GLSL Shaders Are Missing Some C-isms:

- No type casts (use constructors instead)
- Only some automatic promotion (don’t rely on it)
- No pointers
- No strings
- No enums
- Can only use 1-D arrays (no bounds checking)

**Warning:** integer division is still integer division!

```plaintext
float f = 2 / 4; // still gives 0.
```
The Shaders’ View of the Basic Computer Graphics Pipeline

- In general, you want to have a vertex and fragment shader as a minimum.
- A missing stage is OK. The output from one stage becomes the input of the next stage that is there.
- The last stage before the fragment shader feeds its output variables into the rasterizer. The interpolated values then go to the fragment shaders.

= Fixed Function

= Programmable
GLSL Vertex Shader Inputs and Outputs

- **Standard OpenGL per-vertex Attribute variables:**
  - `gl_Vertex`
  - `gl_Normal`, `gl_Color`, etc.

- **User-defined per-vertex in (Attribute) variables:**
  - `aVertex`, `aColor`, `aTemperature`, etc.

- **Built-in Uniform Variables:**
  - `gl_ModelViewMatrix`, etc.

- **User-defined Uniform Variables:**
  - `uThreshold`, etc.

- **User-defined out (Varying):**
  - `vST`, `vMCpos`, `vTemperature`, etc.

- **Special:**
  - `gl_Position`, `glPointSize`, etc.

= **Must Fill!**
A GLSL Vertex Shader Replaces These Operations:

- Vertex transformations
- Normal transformations
- Normal normalization (unitization)
- Handling of per-vertex lighting
- Handling of texture coordinates

A GLSL Vertex Shader Does Not Replace These Operations:

- View volume clipping
- Homogeneous division (divide by w)
- Viewport mapping
- Backface culling
- Polygon mode
- Polygon offset
Built-in Vertex Shader Variables You Will Use a Lot:

- `vec4 gl_Vertex`
- `vec3 gl_Normal`
- `vec4 gl_Color`
- `vec4 gl_MultiTexCoordi (i=0, 1, 2, …)`
- `mat4 gl_ModelViewMatrix`
- `mat4 gl_ProjectionMatrix`
- `mat4 gl_ModelViewProjectionMatrix`
- `mat4 gl_NormalMatrix` (this is the transpose of the inverse of the MV matrix)

- `vec4 gl_Position`
GLSL Fragment Shader Inputs and Outputs

User-defined in (Varying):
- vST, vMCpos, vTemperature, etc.

Built-in Uniform Variables:
- gl_ModelViewMatrix, etc.

User-defined Uniform Variables:
- uThreshold, etc.

Special in:
- gl_FragCoord, etc.

Special out variables:
- gl_FragColor,
- gl_FragDepth,
- etc.

= Must Fill!
A GLSL Fragment Shader Replaces These Operations:

- Color computation
- Texturing
- Color arithmetic
- Handling of per-pixel lighting
- Fog
- Blending
- Discarding fragments

A GLSL Fragment Shader Does Not Replace These Operations:

- Stencil test
- Z-buffer test
- Stippling
Built-in Fragment Shader Variables You Will Use a Lot:

vec4 gl_FragColor
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:

```glsl
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:

<table>
<thead>
<tr>
<th>Beginning letter(s)</th>
<th>Means that the variable …</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Is a per-vertex attribute from the application</td>
</tr>
<tr>
<td>u</td>
<td>Is a uniform variable from the application</td>
</tr>
<tr>
<td>v</td>
<td>Came from the vertex shader</td>
</tr>
<tr>
<td>tc</td>
<td>Came from the tessellation control shader</td>
</tr>
<tr>
<td>te</td>
<td>Came from the tessellation evaluation shader</td>
</tr>
<tr>
<td>g</td>
<td>Came from the geometry shader</td>
</tr>
<tr>
<td>f</td>
<td>Came from the fragment shader</td>
</tr>
</tbody>
</table>

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:

```c
// 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
```

File `gstap.h`

This isn’t part of “official” OpenGL – it is my way of handling the transition.
One Additional Warning: There will be times that the Shader Compiler will appear to have gone insane

A uniform variable that exists, but is not actually needed by the shaders, gets eliminated by the shader compiler and appears, to the main program, to be non-existent. You will then get an error when you try to set it from the glib file, even though you are positive it exists!

Glib file:
Program Bad LightPos [0. 0. 10. 1.]

Frag file:
uniform vec4 LightPos;
void main()
{
    gl_FragColor = vec4( 0., 0., 1., 1.);
    // LightPos never affects a pixel
}

Message:
Cannot find Uniform array variable 'LightPos'

BTW, a uniform variable that never gets set will not generate any sort of error – it will just quietly screw up with an undefined value in your shaders.