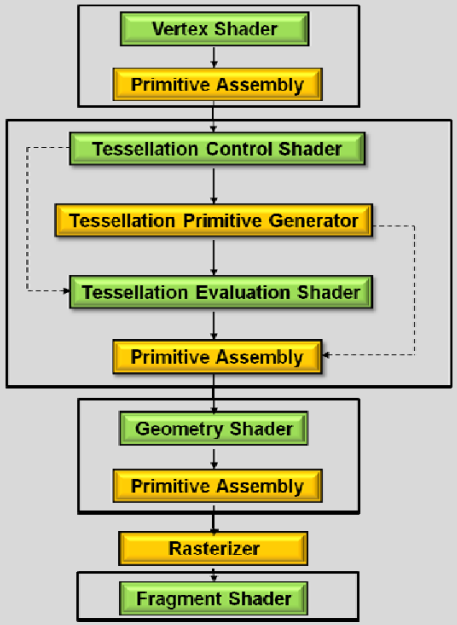
The Transition from RenderMan to the OpenGL Shading Language (GLSL)



Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

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

All Six RenderMan Shader Types

1. Displacement → ≈ GLSL Vertex Shader
2. Distortion / transformation → ≈ GLSL Vertex Shader
3. Surface → ≈ GLSL Fragment Shader
4. Lighting → ≈ GLSL Fragment Shader
5. Atmospheric / volumetric → ≈ GLSL Fragment Shader
6. Imaging → ≈ GLSL Fragment Shader

RenderMan Built-in Microfaceting ≈ Manual or GLSL Tessellation

Fundamental Differences Between RenderMan Shaders and OpenGL Shaders

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

GLSL Variable Types

Attribute	Assigned ***per-vertex*** and passed into the vertex shader, usually with the intent to interpolate through the rasterizer.
Uniform	"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.)
Out / In	Passed from one shader stage to the next shader stage.

GLSL Shaders Are Like C With Extensions for Graphics:

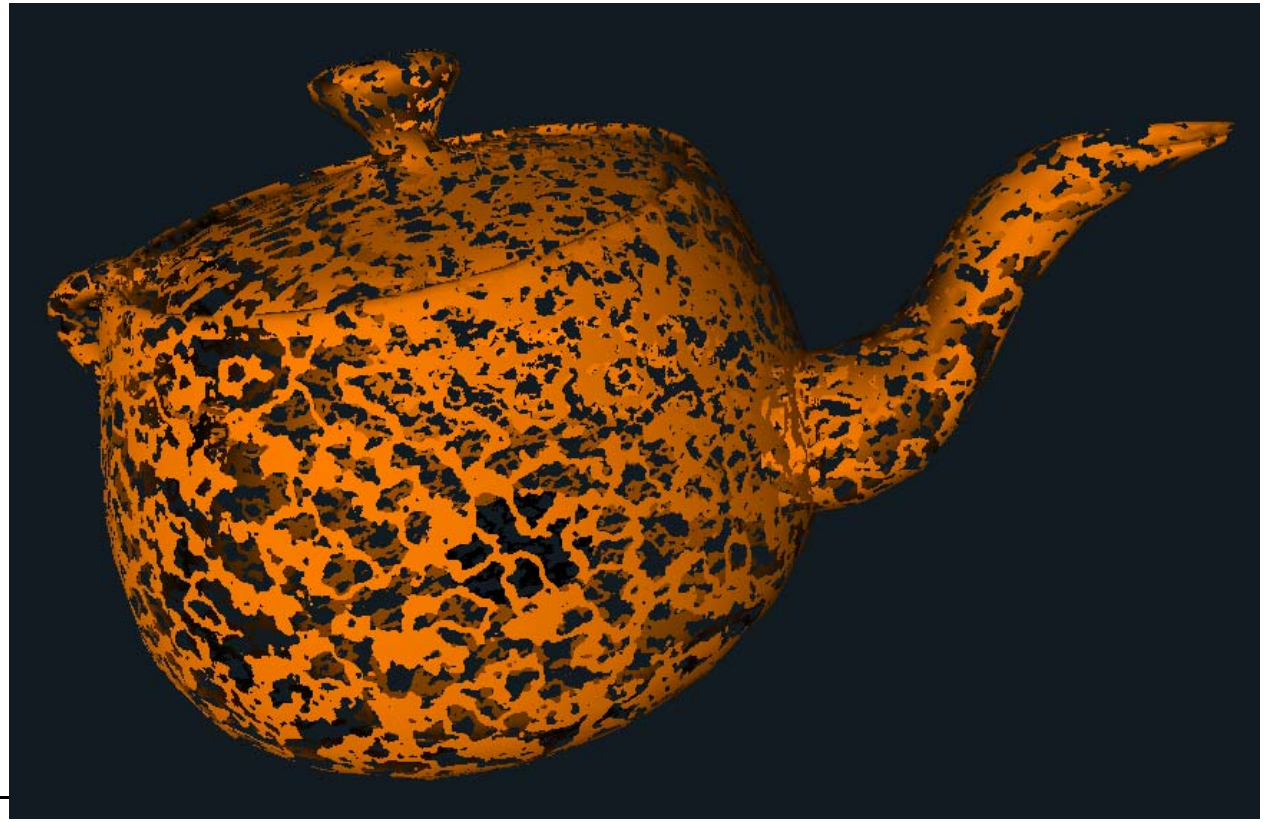
- Types include int, ivec2, ivec3, ivec4
- Types include float, vec2, vec3, vec4
- Types include mat2, mat3, mat4
- Types include bool, bvec2, bvec3, bvec4
- Types include sampler to access textures
- Vector components are accessed with [index], .rgba, .xyzw, or.stpq
- Can ask for parallel SIMD operations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

```
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

```
if( alpha == 0. )
        discard;
```



GLSL Shaders Are Missing Some C-isms:

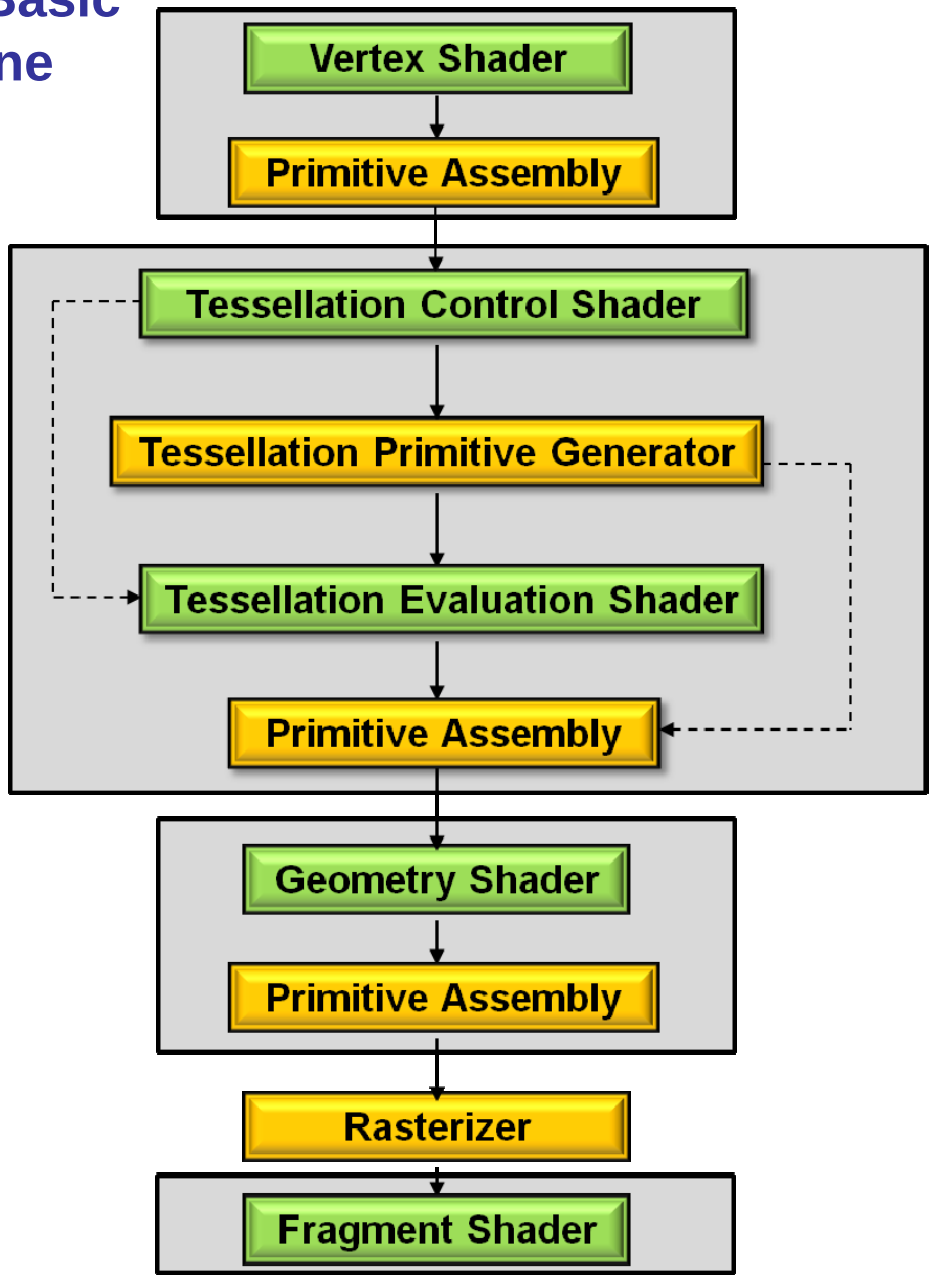
- No type casts (use constructors instead)
- Only some automatic promotion (don't rely on it)
- No pointers
- No strings
- No enums
- Can only use 1-D arrays (no bounds checking)

Warning: integer division is still integer division !

```
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:
uThreshhold, etc.

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

Special:
gl_Position,
gl_PointSize,
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

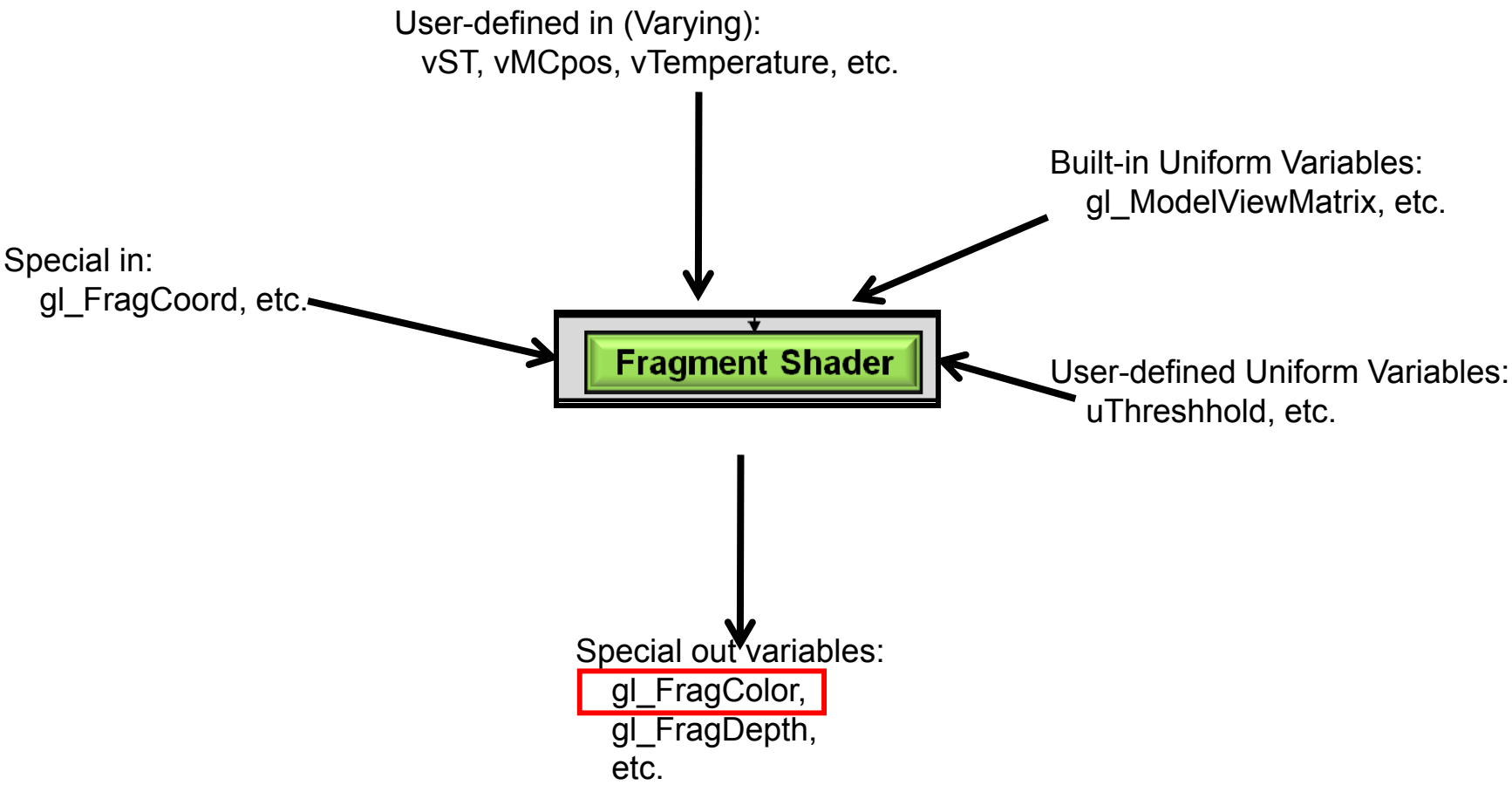
mat4 gl_ProjectionMatrix

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



= 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

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

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:

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.