GPU Programming

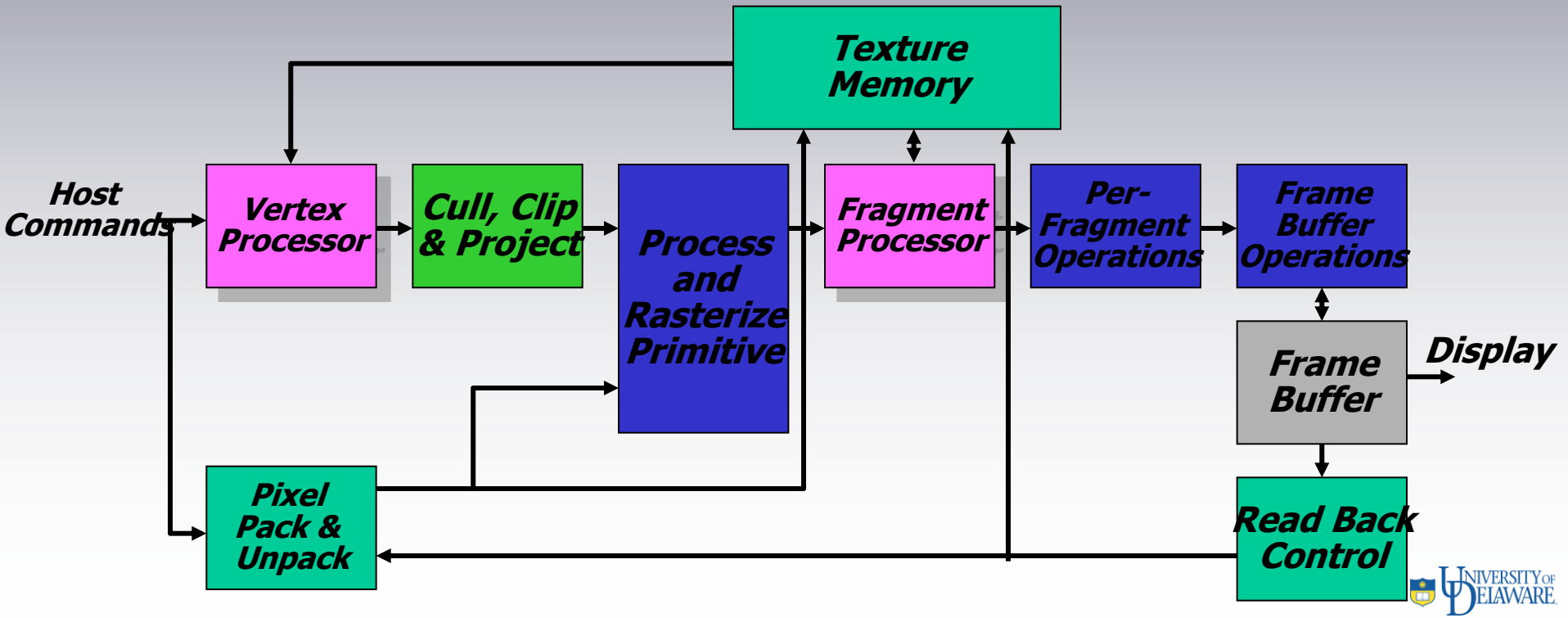
Course web page:
http://goo.gl/EB3aA

Outline

- Graphics pipeline review
- GLSL tutorial
  - Portions adapted from Qing Sun's notes for CIS 565 at Upenn

OpenGL 2.0 - Graphics Today

- Programmable Processing units (exposing what used to be fixed)
  - Programmable per-Vertex Processors
  - Programmable per-Fragment Processors
- Texture memory – general purpose data storage

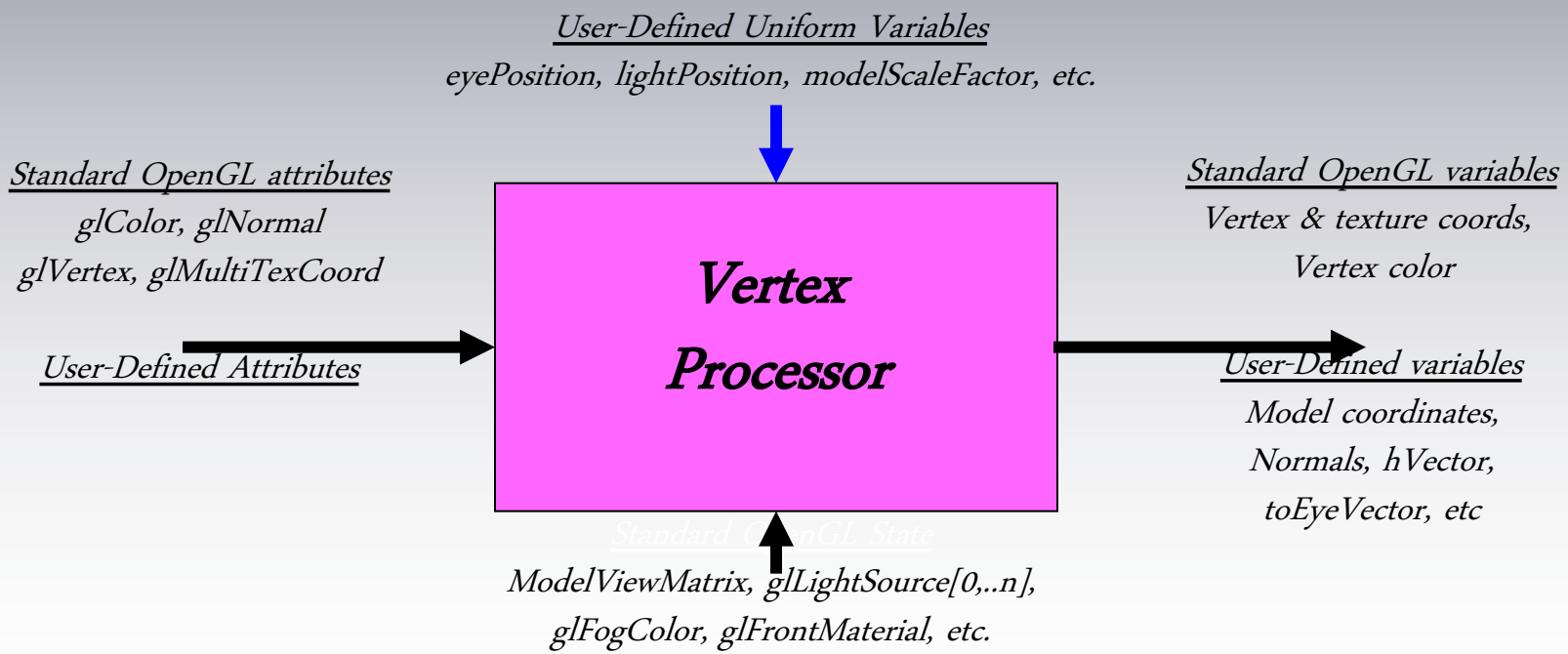


Vertex Processor Capabilities

- **Lighting, Material and Geometry flexibility**
- **Vertex programs replace the following parts of the pipeline:**
  - Vertex & Normal transformation
  - Normalization and rescaling
  - Per-Vertex Lighting Calculations
  - Color application & clamping
  - Texture coordinate generation & transformation
- **The vertex shader does NOT replace:**
  - Perspective divide and viewport (NDC) mapping
  - Clipping
  - Backface culling
  - Primitive assembly (Triangle setup, edge equations, etc.)

Vertex Processor Inputs & Outputs

- Vertex "Shader" has all of the primitive arguments available to it
- Fixed constants that are compiled into the shader
- Special variables that are rendering specific
- Writes its results into prearranged locations (registers) that are "understood" by later processing steps

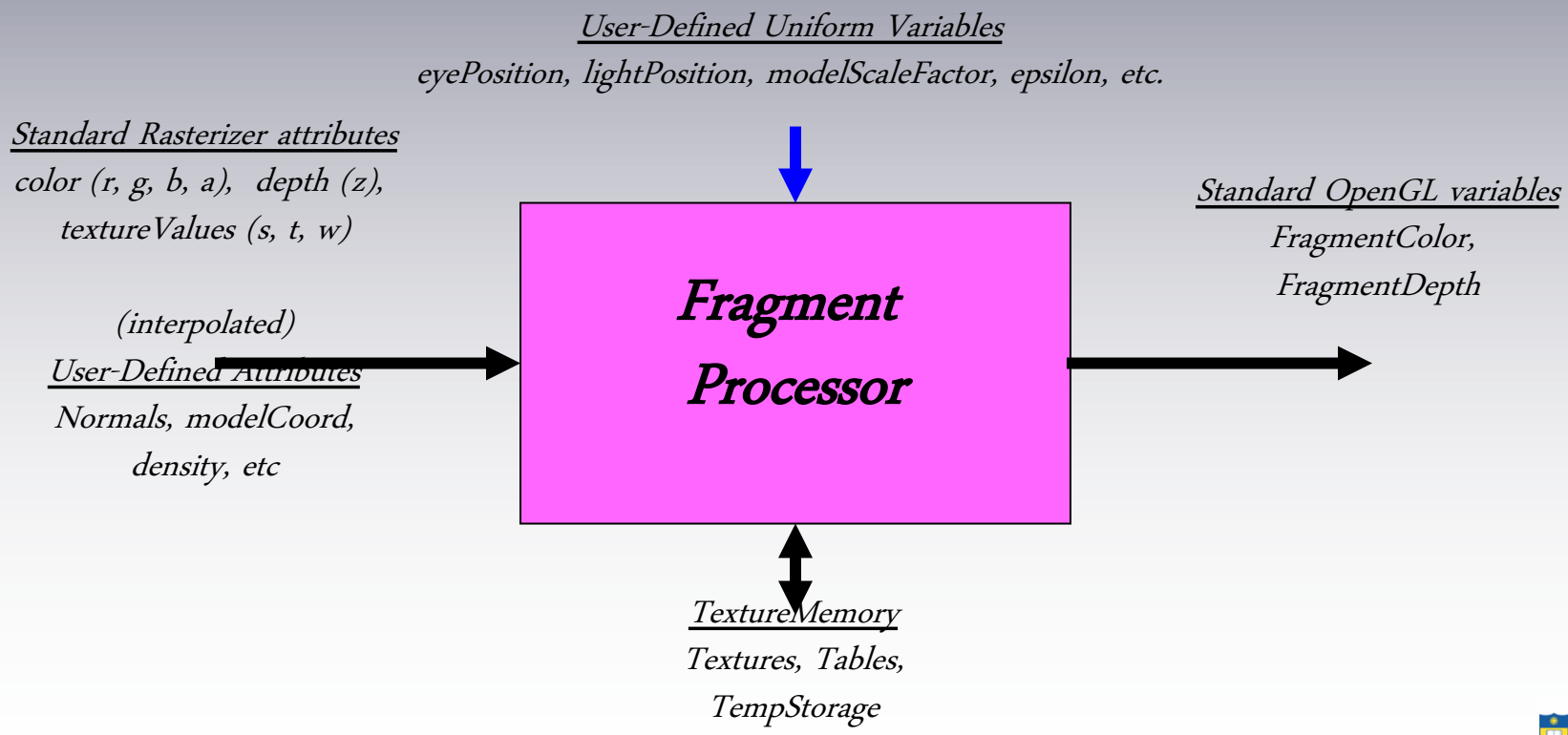


Fragment Processor Capabilities

- **Flexibility for texturing and per-pixel operations**
- **Fragment programs replace the following parts of the OpenGL pipeline:**
  - Operations on interpolated values
  - Texture access
  - Texture application (modulate, add)
  - Fog (color(depth))
  - Color sums (blends, mattes)
  - Perspective divide
  - Pixel zoom
  - Scale and bias
  - Color table lookup
  - Convolution
  - Color matrix
- **The Fragment shader does NOT replace:**
  - Scan Conversion
  - Coverage
  - Scissor
  - Alpha test
  - Stencil test
  - Logical ops
  - Plane masking
  - Histogram
  - Pixel packing and unpacking
  - Stipple
  - Depth test
  - Alpha blending
  - Dithering
  - Z-buffer replacement test

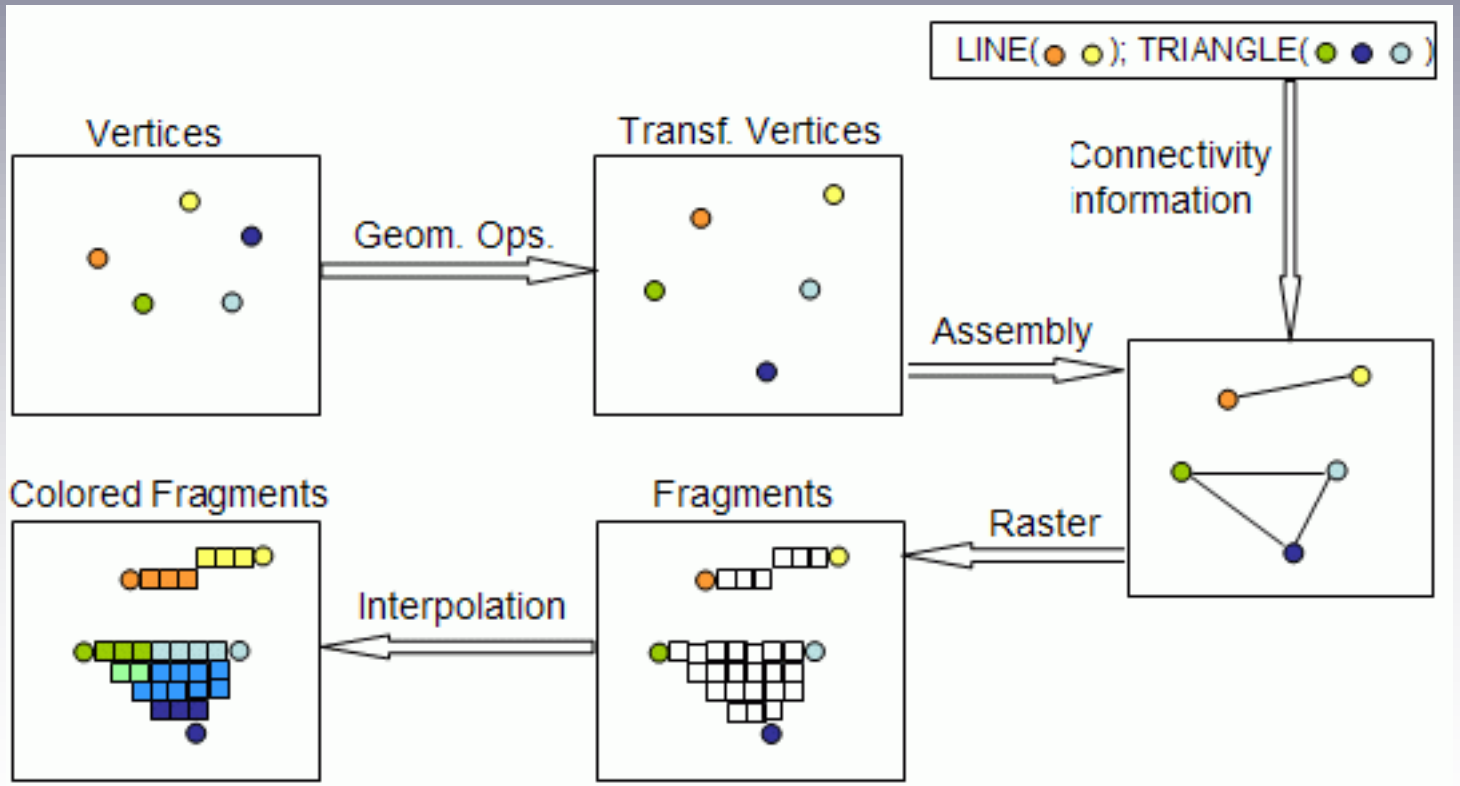
Fragment Processor Inputs & Outputs

- Fragment "Shader" has all of the rasterization arguments available to it
- Fixed constants that are compiled into the shader
- Special variables that are rendering specific
- Writes its results into prearranged locations (registers) that are "understood" by later processing steps



Pipeline Visual Summary

- From lighthouse3d



Vertex Shaders

- Operates on single vertex
- Input includes position, color, normals, etc.
- Must take care of
  - Modelview and perspective transformations
  - Normal transformations
  - Texture coordinate generation & transformations
  - Lighting per vertex
  - Etc.

Fragment Shaders

- Operates on single fragment, executed in parallel
- Alpha blending, depth test still in fixed pipeline
- Can be used to
  - Compute colors
  - Apply textures
  - Compute fog
  - Per-pixel lighting (e.g., Phong shading)

GLSL

- High-level shading language for vertex & fragment programs
- Compiler, linker tightly integrated with OpenGL driver
- Alternatives
  - Cg (created by Nvidia, cross platform, works with OpenGL and DirectX)
  - HLSL (Microsoft, only on Windows with DirectX)

GLSL Syntax Overview

- GLSL is like C **without**
  - Pointers, objects (but structs are still there)
  - Recursion
  - Gotos, switches
  - Dynamic memory allocation
- GLSL is like C with
  - Built-in vector, matrix and sampler types
  - Constructors
  - A great math library
  - Input and output qualifiers

GLSL Syntax: Types

- Scalar types: `float`, `int`, `uint`, `bool`
  - Can do casts: `int i = int(0.0);`
- Vectors are first-class types:
  - `vec2`, `vec3`, `vec4`
  - `ivec`, `bvec`: Integer, boolean vectors
- Matrices
  - mat2, mat3, mat4 (2x2, 3x3, 4x4)
  - Can also do m x n; stored in column-major format
- Samplers
  - sampler1D, sampler2D, sampler3D

GLSL Syntax: Vectors

- Constructors

```
vec3 xyz = vec3(1.0, 2.0, 3.0);
vec3 xyz = vec3(1.0); // [1.0, 1.0, 1.0]
vec3 xyz = vec3(vec2(1.0, 2.0), 3.0);
```

- Access components three ways
  - .x, .y, .z, .w position or direction
  - .r, .g, .b, .a color
  - .s, .t, .p, .q texture coordinate
  - But don't mix
    - `OK: myColor.xyz`
    - `No! myColor.xgb or vec3.xrs`

GLSL Syntax: Vectors

- Swizzle: select or rearrange components

```
vec4 c = vec4(0.5, 1.0, 0.8, 1.0);

vec3 rgb = c.rgb;  // [0.5, 1.0, 0.8]

vec3 bgr = c.bgr;  // [0.8, 1.0, 0.5]

vec3 rrr = c.rrr;  // [0.5, 0.5, 0.5]

c.a = 0.5;         // [0.5, 1.0, 0.8, 0.5]
c.rb = 0.0;        // [0.0, 1.0, 0.0, 0.5]

float g = rgb[1];  // 0.5, indexing, not swizzling
```

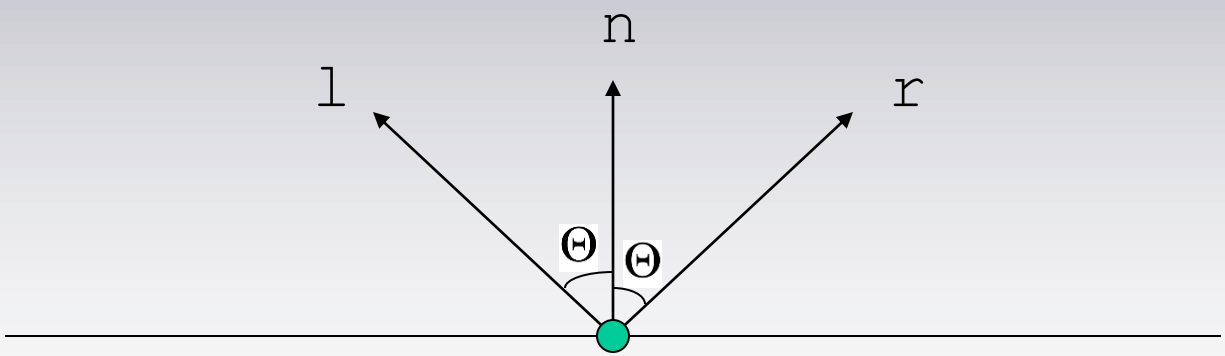
GLSL Built-in Functions

- Selected geometric functions

```
vec3 l, n; // = . . .
vec3 p, q; // = . . .

float f = length(l);          // vector length
float d = distance(p, q); // dist between pts

float d2 = dot(l, n);         // dot product
vec3 v2 = cross(l, n);        // cross product
vec3 v3 = normalize(l);       // "unitize"

vec3 v3 = reflect(l, n);      // reflect

// also:  refract() and faceforward() (?)
```

GLSL Built-in Functions

- `reflect(-l, n)`
  - Given **l** and **n**, find **r**. Angle in equals angle out



GLSL Built-in Functions

- Selected vector rational functions

```
vec3 p = vec3(1.0, 2.0, 3.0);
vec3 q = vec3(3.0, 2.0, 1.0);

bvec3 b = equal(p, q);          // (F, T, F)
bvec3 b2 = lessThan(p, q);      // (T, F, F)
bvec3 b3 = greaterThan(p, q); // (F, F, T)

bvec3 b4 = any(b);              // T
bvec3 b5 = all(b);              // F
```

GLSL Built-in Functions

- Rewrite in one line of code

```
bool foo(vec3 p, vec3 q)
{
  if (p.x < q.x) {
    return true;
  }
  else if (p.y < q.y) {
    return true;
  }
  else if (p.z < q.z) {
    return true;
  }
  return false;
}
```

```
return
any(lessThan(p, q));
```

GLSL Syntax: Matrices

- Constructors

```
mat3 i = mat3(1.0); // 3x3 identity matrix

mat2 m = mat2(1.0, 2.0,  // [1.0 3.0]
              3.0, 4.0); // [2.0 4.0]
```

- Accessing elements

```
float f = m[column][row];

float x = m[0].x; // x component of first column

vec2 yz = m[1].yz; // yz components of second
   column
```

Treat matrix as array of column vectors

Can swizzle too!

GLSL Syntax: Vectors and Matrices

- Matrix and vector operations are easy and fast:

```
vec3 xyz = // ...

vec3 v0 = 2.0 * xyz; // scale
vec3 v1 = v0 + xyz;  // component-wise
vec3 v2 = v0 * xyz;  // component-wise

mat3 m = // ...
mat3 v = // ...

mat3 mv = m * v;        // matrix * matrix
mat3 xyz2 = mv * xyz; // matrix * vector
```

GLSL Built-in Functions

- Selected matrix functions

```
mat4 m = // ...

mat4 t = transpose(m);
float d = determinant(m);
mat4 d = inverse(m);
```

GLSL Built-in Functions

- Selected trigonometry functions

```
float s = sin(theta);
float c = cos(theta);
float t = tan(theta);

float theta = asin(s);
// ...

vec3 angles = vec3(/* ... */)
vec3 vs = sin(angles);
```

Angles are measured in radians

Works on vectors component-wise

GLSL Built-in Functions

- Exponential Functions

```
float xToTheY = pow(x, y);
float eToTheX = exp(x);
float twoToTheX = exp2(x);

float l = log(x);   // ln
float l2 = log2(x); // log2

float s = sqrt(x);
float is = inversesqrt(x);
```

GLSL Built-in Functions

- Selected common functions

```
float ax = abs(x);  // absolute value
float sx = sign(x); // -1.0, 0.0, 1.0

float m0 = min(x, y); // minimum value
float m1 = max(x, y); // maximum value
float c = clamp(x, 0.0, 1.0);

// many others:  floor(), ceil(), …
```

GLSL Built-in functions

- `mix(x, y, a)` // linear interpolation
  - x, y: start, end of range
  - a: Interpolation value in range [0, 1]
- `step(edge, x)`
  - Returns 0 if x < edge, 1 otherwise
- `smoothstep(edge1, edge2, x)`
  - edge1 = value of lower edge of step
  - edge2 = value of upper
  - x = value to be interpolated along S-curve

Type qualifiers

- **const**: Compile time constant
- **attribute**: Variables that may change per vertex
  - Passed from OpenGL app to vertex shaders
  - Read-only variable
- **uniform**: Variables that may change per primitive
  - Can't be set inside glBegin()/glEnd()
  - Passed from OpenGL app to shaders
  - Can be used in vertex or fragment shaders
- **varying**: Interpolated data between vertex shader & fragment shader
  - Can be written in vertex shader
  - Read-only in fragment shader

GLSL Syntax: Samplers

- Opaque types for accessing textures
- Basically like 2D arrays

```
uniform sampler2D colorMap; // 2D texture

vec3 color = texture(colorMap, vec2(0.5, 0.5)).rgb;

vec2 size = textureSize(colorMap, 0);


// Lots of sampler types:  sampler1D,
// sampler3D, sampler2DRect, samplerCube,
// isampler*, usampler*, ...

// Lots of sampler functions: texelFetch, textureLoad
```

Simple Shaders

- Vertex program

```
void main(void)
{
  gl_Position =
    gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Fragment program

```
void main(void)
{
  gl_FragColor = Vec4(0.0,0.0,1.0,1.0);
}
```

Texturing

- To perform texturing we need access to the texture coordinates
- GLSL provides attribute variables, one for each texture object
  - `attribute vec4 gl_MultiTexCoord0;`
  - …
  - `attribute vec4 gl_MultiTexCoord7;`
- Object stores texture coordinates and texture states