

# Shader

## GLSL Syntax

# OpenGL Reference Card Page 6ff

Page 6

The OpenGL® Shading Language is used to create shaders for each of the programmable processors contained in the OpenGL processing pipeline. The OpenGL Shading Language is actually several closely related languages. Currently, these processors are the vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders.

[n.n.n] and [Table n.n] refer to sections and tables in the OpenGL Shading Language 4.30 specification at [www.khronos.org/registry](http://www.khronos.org/registry).

OpenGL Shading Language 4.30 Reference Card

**Preprocessor [3.3]**  
**Preprocessor Directives**  
# define #undef #extension #version #ifdef #ifndef #else #endif #error #pragma  
**Preprocessor Operators**  
#version 430 Required when using version 4.30 profile is core, compatibility, or es  
#extension\_name : behavior extension\_name : behavior  
#extension\_name : behavior

**Predefined Macros**  
\_\_LINE\_\_ \_\_FILE\_\_ Decimal integer constants \_\_FILE\_\_ says which source string is being processed.  
\_\_VERSION\_\_ Decimal integer; e.g., 430  
GL\_core\_profile Defined as 1.  
GL\_es\_profile 1 if the implementation supports the es profile  
GL\_compatibility\_profile Defined as 1 if the implementation supports the compatibility profile.

**Operators and Expressions [5.1]**  
The following operators are numbered in order of precedence. Relational and equality operators evaluate to Boolean. Also see lessThan(), equal(), etc.

1. {}	parenthetical grouping
2. []	array subscript
3. ()	function call, constructor, structure field, selector, twiddle
++ --	prefix increment and decrement

4. ++ --	prefix increment and decrement unary
5. * %	multiplicative
6. + -	additive
7. << >>	bitwise shift
8. < > <= >=	relational
9. == !=	equality
10. &	bitwise and
11.	bitwise inclusive or
12. &&	logical and
13. ^	logical exclusive or
14.	logical inclusive or
15. ? :	selects an entire operand
16. = += -= *= /= %*= %/=	assignment arithmetic assignments
17. .	member

**Types [4.1]**  
**Transparent Types**  
void no function return value  
bool Boolean  
int, uint signed/unsigned integers  
float single-precision floating-point scalar  
double double-precision floating-point scalar  
vec2, vec3, vec4 floating-point vector  
dvec2, dvec3, dvec4 double-precision floating-point vectors  
bvec2, bvec3, bvec4 Boolean vectors  
ivec2, ivec3, ivec4 signed and unsigned integer vectors  
uvec2, uvec3, uvec4 unsigned integer vectors  
mat2, mat3, mat4 2x2, 3x3, 4x4 float matrix  
mat2x2, mat2x3, mat2x4 2 column float matrix of 2, 3, or 4 rows

**Floating-Point Opaque Types**  
sampler1D, sampler2D, sampler3D 1D, 2D, or 3D texture  
samplerCube cube mapped texture  
sampler2DRect rectangular texture  
sampler3DRect 3D rectangular texture  
sampler1DArray 1D or 2D array texture  
sampler2DArray 2D array texture  
samplerBuffer buffer texture  
samplerBufferImage buffer texture  
sampler2DMS 2D multi-sample texture  
sampler2DMSArray 2D multi-sample array texture  
sampler3DMSArray 3D multi-sample array texture  
samplerCubeArray cube map array texture  
samplerCubeArrayImage cube map array texture  
sampler1DShadow 1D or 2D depth texture with comparison  
sampler2DShadow 2D depth texture with comparison  
sampler3DShadow 3D depth texture with comparison

**Signed Integer Opaque Types (cont'd)**  
image2DRect int, 2D rectangular image  
isampler1DArray int, 1D array texture  
isampler2DArray int, 2D array texture  
isamplerBuffer int, buffer texture  
isamplerBufferImage int, buffer texture  
isampler2DMS int, 2D multi-sample texture  
isampler2DMSArray int, 2D multi-sample array texture  
isampler3DMSArray int, 3D multi-sample array texture  
isamplerCubeArray int, cube map array texture  
isamplerCubeArrayImage int, cube map array texture  
**Unsigned Integer Opaque Types**  
atomic\_uint uint atomic counter  
usampler1D, usampler2D, usampler3D, usamplerCube uint 1D, 2D, or 3D texture  
uimage1D, uimage2D, uimage3D, uimageCube uint 1D, 2D, or 3D image  
uimage2DRect, uimage3DRect, uimageCubeRect, uimageCubeRectImage uint 2D, 3D, or 4D image  
uimageCubeRectImage uint 3D image  
uimageCubeRectImage uint 3D image

**Unsigned Integer Opaque Types (cont'd)**  
uimage2DMSArray uint 2D multi-sample array texture  
usamplerCubeArray uint cube map array texture  
uimageCubeArray uint cube map array texture  
**Implicit Conversions**  
int -> uint  
int, uint -> float  
int, uint, float -> double  
ivec2 -> uvec2  
ivec3 -> uvec3  
ivec4 -> uvec4  
vec2 -> dvec2  
vec3 -> dvec3  
vec4 -> dvec4  
bvec2 -> ivec2  
bvec3 -> ivec3  
bvec4 -> ivec4  
mat2 -> dmat2  
mat3 -> dmat3  
mat4 -> dmat4  
mat2x2 -> dmat2x2  
mat2x3 -> dmat2x3  
mat2x4 -> dmat2x4  
mat3x2 -> dmat3x2  
mat3x3 -> dmat3x3  
mat3x4 -> dmat3x4  
mat4x2 -> dmat4x2  
mat4x3 -> dmat4x3  
mat4x4 -> dmat4x4

see [www.khronos.org/registry/OpenGL/4.3/quick-reference-card.pdf](http://www.khronos.org/registry/OpenGL/4.3/quick-reference-card.pdf)

# GLSL Syntax Overview

- GLSL is like C without
  - Pointers
  - Recursion
  - Dynamic memory allocation
- GLSL is like C with
  - Built-in vector, matrix and sampler types
  - Constructors
  - A math library
  - Input and output qualifiers

# GLSL Syntax Overview

- GLSL has a preprocessor

```
#version 330
#ifdef FAST_EXACT_METHOD
    FastExact();
#else
    SlowApproximate();
#endif
```

- All shaders have main()

```
void main() {
    ...
}
```

# Vectors

- Scalar types: **float**, **int**, **uint**, and **bool**
- Vectors are also built-in types:
  - **vec2**, **vec3**, and **vec4**
  - Also **ivec\***, **uvec\***, and **bvec\***
- Access components three ways:
  - **.x, .y, .z, .w** ← position or direction
  - **.r, .g, .b, .a** ← color
  - **.s, .t, .p, .q** ← texture coordinate

# Vectors

- Vectors have constructors

```
vec3 xyz = vec3(1.0, 2.0, 3.0);
```

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

```
vec3 xyz = (vec3)1.0; // error
```

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

# Swizzling

- 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]
      rgb = c.xyz;    // same thing! [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 = vec2(0.0);      // [0.0, 1.0, 0.0, 0.5]

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

# Matrices

- Matrices are built-in types:
  - Square: `mat2`, `mat3`, and `mat4`
  - Rectangular: `matmxn`. `m` columns, `n` rows
    - `mat2x3`
- Stored column major



# Matrices

- Matrix Constructors

```
mat3 i = mat3(1.0); // 3x3 identity matrix  
  
mat2 m = mat2(1.0, 2.0, // [1.0 3.0] column major!  
              3.0, 4.0); // [2.0 4.0]
```

- Accessing Elements

```
float f = m[column][row]; // m some 3x3 matrix  
  
float x = m[0].x; // x component of first column  
  
vec2 yz = m[1].yz; // yz components of second column
```

# 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(v0, v1, v2);    // give columns
mat3 m2 = mat3(2.0);          // diagonal all 2's

mat3 m3 = 3.0 * m;            // scale a matrix
mat3 mm2 = m * m2;            // matrix * matrix
vec3 xyz2 = mm2 * xyz;        // matrix * vector
```

# Built-in Functions

- Selected Trigonometry Functions

```
float s = sin(theta);  
float c = cos(theta);  
float t = tan(theta);  
  
float as = asin(theta);  
  
vec3 angles = vec3(/* ... */);  
vec3 vs = sin(angles); //vector version
```

# 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); // single GPU instr.
```

# 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(),
// step(), smoothstep(), ...
```

# Built-in Functions

- Rewrite with one function call

```
float minimum = // ...  
float maximum = // ...  
float x = // ...  
  
float f = min(max(x, minimum), maximum);  
  
float f = clamp(x, minimum, maximum);
```

# Built-in Functions

- Rewrite this without the **if** statement

```
float x = // ...  
float f;  
  
if (x > 0.0) {  
    f = 2.0;  
}  
else {  
    f = -2.0;  
}  
  
f = 2.0 * sign(x);
```

# Built-in Functions

- Rewrite this without the **if** statement

```
float root1 = // ...  
float root2 = // ...  
  
if (root1 < root2) {  
    return vec3(0.0, 0.0, root1);  
}  
else {  
    return vec3(0.0, 0.0, root2);  
}  
  
return vec3(0.0, 0.0, min(root1, root2));
```



# Built-in Functions

- Selected Geometric Functions

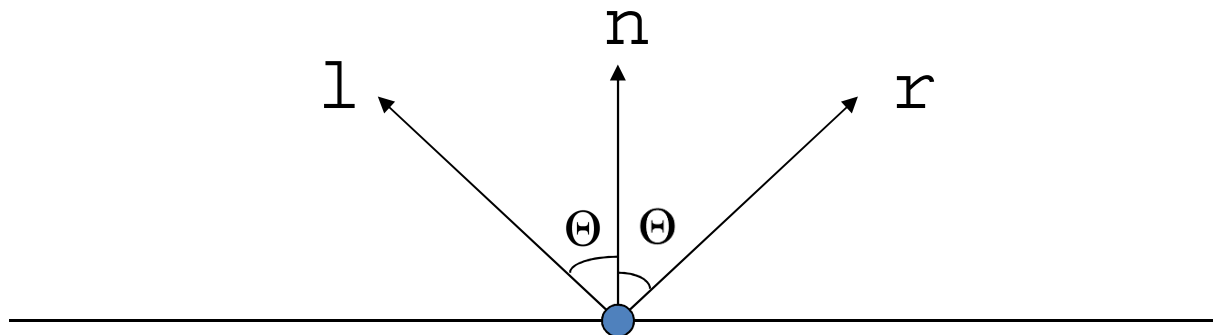
```
vec3 l = // ...
vec3 n = // ...
vec3 p = // ...
vec3 q = // ...

float f = length(l);           // vector length
float d = distance(p, q);      // point dist.
float d2 = dot(l, n);          // dot product
vec3 v2 = cross(l, n);         // cross product
vec3 v3 = normalize(l);        // normalize
vec3 v3 = reflect(l, n);       // reflect

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

# Built-in Functions

- **reflect**(  $-l$  ,  $n$  )
  - Given  $l$  and  $n$ , find  $r$
  - Angle in = angle out



# Built-in Functions

- Rewrite without **length**

```
vec3 p = // ...  
vec3 q = // ...  
  
vec3 v = length(p - q);  
  
vec3 v = distance(p, q);
```

# Built-in Functions

- What is wrong with this code?

```
vec3 n = // ...  
normalize(n);
```

# Built-in Functions

- Selected Matrix Functions

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

# Built-in Functions

- Selected Vector Relational Functions

```
vec3 p = vec3(1.0, 2.0, 3.0);  
vec3 q = vec3(3.0, 2.0, 1.0);  
  
bvec3 b = equal(p, q);           // (false, true, false)  
bvec3 b2 = lessThan(p, q);       // (true, false, false)  
bvec3 b3 = greaterThan(p, q);    // (false, false, true)  
  
bvec3 b4 = any(b);               // true  
bvec3 b5 = all(b);              // false
```

# Built-in Functions

- Rewrite this 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));
```

# Samplers

- *Opaque* types for accessing textures
- Always **uniform**

```
// fragment shader
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, textureLod
```



# Samplers

- Returns **vec4**
- Coordinate access differs by sampler type

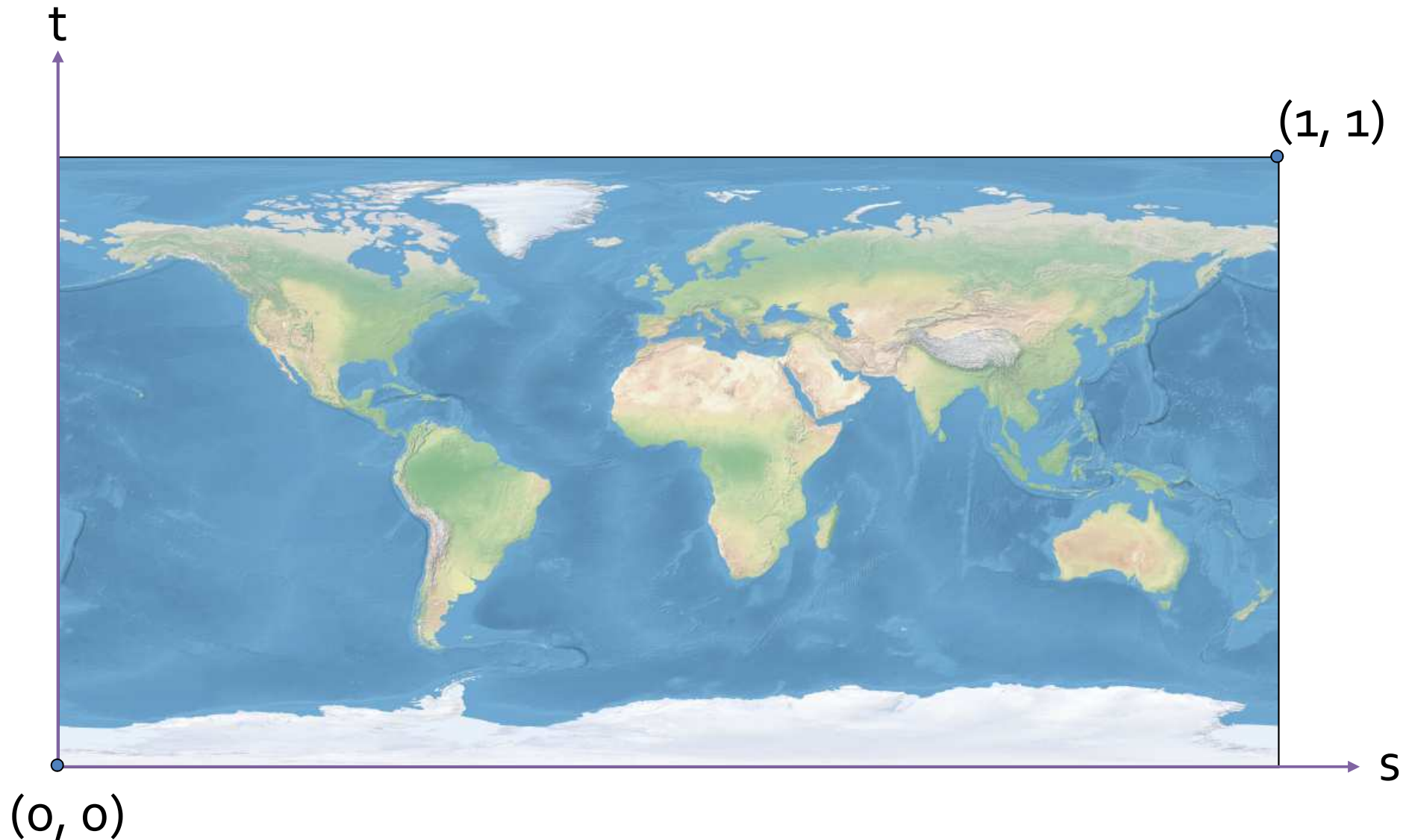
```
// fragment shader
uniform sampler2D colorMap; // 2D texture

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, textureLod
```

# Samplers – Texture Coordinates



Images from: <http://www.naturalearthdata.com/>