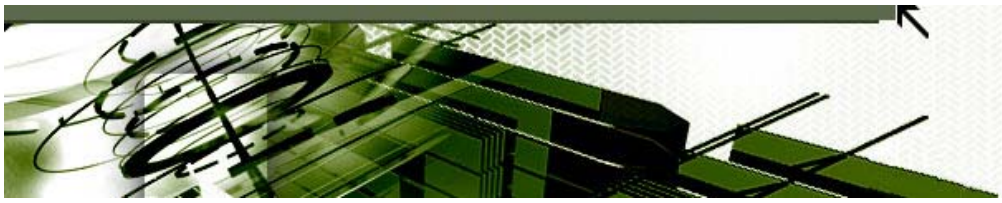

OpenGL Shading Language Course

Chapter 2 – GLSL Basics

By
Jacobo Rodriguez Villar
jacobo.Rodriguez@typhoonlabs.com

Chapter 2: GLSL Basics

INDEX

Introduction

This chapter will attempt to summarize the entire GLSL 1.10.59 specification.

GLSL itself is a C-like language, which borrows features from C++. Knowledge of C is assumed, so the fundamentals and basics of that language will not be explained further (with exception to the parts that differ between C and C++).

GLSL Language

Mechanisms Borrowed from C++:

- Function overloading based on argument types.
- Declaration of variables where they are required, instead of at the beginning of each block.

Character Set

The character set used for GLSL is a subset of ASCII, and includes the following characters:

1º	The letters **a-z**, **A-Z,** and the underscore (_)**.**
2º	The numbers **0-9**.
3º	The symbols period (**.**), plus (**+**), dash (**-**), slash (***/***), asterisk (*****), percent (**%**), angled brackets (**<** and **>**), square brackets (**[** and **]**), parentheses (**(** and **)**), braces (**{** and **}**), caret (**^**), vertical bar (**|**), ampersand (**&**), tilde (**~**), equals (**=**), exclamation point (**!**), colon (**:**), semicolon (**;**), comma (**,**), and question mark (**?**).
4º	The number sign (#) for pre-processor use.
5º	White space: the space character, horizontal tab, vertical tab, form feed, carriage-return, and linefeed.

The character set is case-sensitive, and there are no characters or string data types, so no quotation characters are included.

Also, the language does not allow any pointer types or any kind of pointer arithmetic.

Pre-processor Keywords

#	Operates as is standard for C++ pre-processors.
#define	Operates as is standard for C++ pre-processors.
#undef	Operates as is standard for C++ pre-processors.
#if	Operates as is standard for C++ pre-processors (only for integer and #defined types).
#ifdef	Operates as is standard for C++ pre-processors.
#ifndef	Operates as is standard for C++ pre-processors.
#else	Operates as is standard for C++ pre-processors.
#elif	Operates as is standard for C++ pre-processors (only for integer and #defined types).
#endif	Operates as is standard for C++ pre-processors.
#error	**error** will cause the implementation to put a diagnostic message in the shader's information log. The message will be the tokens following the **#error** directive, up to the first new line. The implementation must then consider the shader to be ill-formed.
#pragma	**#pragma** allows implementation-dependent compiler control. Tokens following **#pragma** are not subject to pre-processor macro expansion. The following pragmas are defined as part of the language: #pragma optimize(on) #pragma optimize(off) #pragma debug(on) #pragma debug(off)
#extension	For default, any extension to the language must be enabled using this keyword: `#extension` *extension_name* **:** *behaviour* `#extension all` **:** *behaviour* [Frame1] The initial state of the compiler is as if the directive #extension all : disable was issued, telling the compiler that all error and warning reporting must be done according to this specification, ignoring any extensions. Macro expansion is not done on lines containing **#extension** directive.
#version	Used to declare the GLSL version against the shader. Written (#version *number*), where *number* must be 110 for this specification's version of the language (following the same convention as *__VERSION__* above), in which case the directive will be accepted with no errors or warnings. Any *number* less than 110 will cause an error to be generated. Macro expansion is not done on lines containing **#version** directive.
#line	**#line** must have, after macro substitution, one of the following two forms: #line *line* **or** #line *line source-string-number* where *line* and *source-string-number* are constant integer expressions. After processing this directive (including its new-line), the implementation will behave as if it is compiling at line number *line+1* and source string number *source-string-number*. Subsequent source strings will be numbered sequentially, until another **#line** directive overrides that numbering.
defined	The **defined** operator can be used in either of the following ways: defined *identifier* or defined (*identifier*)
__LINE__	*__LINE__* will substitute a decimal integer constant that is one more than the number of preceding newlines in the current source string.
__FILE__	*__FILE__* will substitute a decimal integer constant that says which source string number is currently being processed.
__VERSION__	*__VERSION__* will substitute a decimal integer reflecting the version number of the OpenGL shading language. The version of the shading language described in this document will have *__VERSION__* substitute the decimal integer 110.

Comments

Comments are delimited by /* and */, or by // and a new-line. The begin comment delimiters (/* or //) are not recognized as delimiters when inside a comment, meaning nesting does not exist.

Variables and types

These are the basic GLSL types:

GLSL data type	C data type	Description
`bool`	int	A conditional type, taking on values of **true** or **false.**
`int`	int	Signed integer.
`float`	float	Single floating-point scalar.
`vec2`	float [2]	Two component floating-point vector.
`vect3`	float [3]	Three component floating-point vector.
`vec4`	float [4]	Four component floating-point vector.
`bvec2`	int [2]	Two component Boolean vector.
`bvec3`	int [3]	Three component Boolean vector.
`bvec4`	int [4]	Four component Boolean vector.
`ivec2`	int [2]	Two component signed integer vector.
`ivec3`	int [3]	Three component signed integer vector.
`ivec4`	int [4]	Four component signed integer vector.
`mat2`	float [4]	2×2 floating-point matrix.
`mat3`	float [9]	3×3 floating-point matrix.
`mat4`	float [16]	4×4 floating-point matrix.
`sampler1D`	int	Handle for accessing a 1D texture.
`sampler2D`	int	Handle for accessing a 2D texture.
`sampler3D`	int	Handle for accessing a 3D texture.
`samplerCube`	int	Handle for accessing a cubemap texture.
`sampler1DShadow`	int	A handle for accessing a 1D depth texture with comparison.
`Sampler2DShadow`	int	A handle for accessing a 2D depth texture with comparison.

Type Descriptions

Booleans
To make conditional execution of code easier to express, the type **`bool`** is supported. There is no expectation that hardware directly supports variables of this type. It is a genuine Boolean type, holding only one of two values (either true or false). Two keywords, **true** and **false**, can be used as Boolean constants.

Integers
Integers are mainly supported as a programming aid. At the hardware level, real integers would aid in the efficient implementation of loops and array indices, and in referencing texture units. However, there is no requirement that integers used within the language can map to integer types used within hardware. It is not expected that underlying hardware has full support for a wide range of integer operations. Because of their intended (limited) purpose, integers are limited to 16 bits of precision, plus a sign representation in both the vertex and fragment languages. There is no automatic promotion from **`int`** to **`float`** (so **`int`** `a=1.0;` would cause an error, for example).

Floats
Floats are available for use within a variety of scalar calculations. Floating-point variables are defined as follows: `float a = 1.8;`

Vector types
GLSL includes data types for generic 2, 3, and 4 component vectors of floating-point values, integers, or booleans. Defining vectors as part of the shading language allows for direct mapping of vector operations on graphics hardware that is capable of doing vector processing. In general, applications will be able to take better advantage of the parallelism in graphics hardware by doing computations on vectors rather than on scalar values.

Vector elements can be accessed in several ways:

For example, using **`vec4 myVector;`** we can access myVector's elements in the usual way:

```
myVector[1] = 6.4;
```

Another way of accessing the elements is to use the struct subscript:

`myVector.y = 6.4` or `myVector.s = 6.4;`

When using a struct subscript, we have the following access names for the indices (they are synonyms):

[0]	[1]	[2]	[3]	Useful when element looping is required.
x	y	z	w	Useful when accessing vectors that represent points.
s	t	p	q	Useful when accessing vectors that texture coordinates.
r	g	b	a	Useful when accessing vectors that represent colors.

Accessing components beyond those declared for the vector type is an error, meaning:

```
vec2 pos;
pos.x // is legal
pos.z // is illegal
```

The component selection syntax allows multiple components to be selected by appending their names (from the same name set) after the period character (**.**):

```
vec4 v4;
v4.rgba; // is a vec4 and the same as just using v4
v4.rgb; // is a vec3
v4.b; // is a float
v4.xy; // is a vec2
v4.xgba; // is illegal - the component names do not come from the same set
```

The order of the components can be changed or replicated via sizzling:

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz = pos.wzyx; // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy; // dup = (1.0, 1.0, 2.0, 2.0)
```

Component group notation can occur on the left-hand side of an expression:

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
pos.xw = vec2(5.0, 6.0); // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0); // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0); // illegal - 'x' used twice
pos.xy = vec3(1.0, 2.0, 3.0); // illegal - mismatch between vec2 and vec3
```

Some examples of vector type construction:

```
vec3 (float) // initializes each component of a vec3 with the float
vec4 (ivec4) // makes a vec4 from an ivec4, with component-wise conversion
vec2 (float, float) // initializes a vec2 with 2 floats
ivec3 (int, int, int) // initializes an ivec3 with 3 ints
bvec4 (int, int, float, float) // initializes with 4 Boolean conversions
vec2 (vec3) // drops the third component of a vec3
vec3 (vec4) // drops the fourth component of a vec4
vec3 (vec2, float) // vec3.x = vec2.x, vec3.y = vec2.y, vec3.z = float
vec3 (float, vec2) // vec3.x = float, vec3.y = vec2.x, vec3.z = vec2.y
vec4 (vec3, float)
vec4 (float, vec3)
vec4 (vec2, vec2)
```

Examples:

```
vec4 color = vec4(0.0, 1.0, 0.0, 1.0);
vec4 rgba = vec4(1.0); // sets each component to 1.0
```

Matrices

GLSL supports 2×2, 3×3, and 4×4 matrices of floating-point numbers. Matrices are read-from and written-to the column in major order.

Example matrix declarations:

```
mat2 mat2D;
mat3 optMatrix;
mat4 view, projection;
```

Matrices are based on vectors, so syntax like `optMatrix[2]=vec3(1.0,1.0,1.0);` is allowed.

For example, initializing the diagonal of a matrix with all other elements set to zero:

```
mat2(float)
mat3(float)
mat4(float)
```

Initializing a matrix can be achieved by specifying vectors, or specifying all 4, 9, or 16 floats for `mat2`, `mat3`, and `mat4` respectively. The floats are then assigned to elements in column major order.

```
mat2(vec2, vec2);
mat3(vec3, vec3, vec3);
mat4(vec4, vec4, vec4, vec4);

mat2(float, float,
     float, float);

mat3(float, float, float,
     float, float, float,
     float, float, float);

mat4(float, float, float, float,
     float, float, float, float,
     float, float, float, float,
     float, float, float, float);
```

There are many possibilities here, as long as enough components are present to initialize the matrix. However, construction of a matrix from other matrices is currently reserved for future use.

Samplers

Sampler types (like **`sampler2D`**) are effectively opaque handles to textures. They are used with the built-in texture functions to access textures, and can only be declared as function parameters or uniforms. Samplers are not allowed to be used as operands within expressions, nor can they be assigned within. As uniforms, they are initialized through the OpenGL API using `int` type.

Note:

Samplers represent texture units, not texture objects. In order to use a texture within a shader, it must first be bound to a texture unit, and then the `tex` unit number must be passed to the shader. Some cards only have few texture units (for example, the GeForce FX5200 hardware only has four). If we only have access to GL_MAX_TEXTURE_UNITS textures within the shader this will not be an ideal situation.

However, this perceived limitation is not entirely correct:

A texture unit is composed of an image, a matrix, an interpolator, and a coordinate generation processor. Though we are limited by the number of texture images available, we are not limited by the number of texture units. This number can be obtained by querying GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS_ARB (with a FX5200 this number is 16). Graphics cards usually have more texture images than texture units.

The GLSL extensions (both vertex and fragment) specify that in order to set a texture to a texture image, the texture unit does not have to be enabled. The following example shows how to set-up texures within OpenGL, for use with uniform samplers:

```
for(int I = 0;I< my number of textures;I++)
{
  glActiveTextureARB(GL_TEXTURE0_ARB + I);
  glBindTextureARB(GL_TEXTURE_2D,my_texture_array[I]);

  // glEnable(GL_TEXTURE_2D); ← this command is not needed

 int location = glGetUniformLocationARB(myProgramObject, mySamplerNameArray[I]);
 glUniform1iARB(location, I );
}
glActiveTextureARB(GL_TEXTURE0_ARB);
```

Structures

User-defined types can be created by aggregating pre-defined types into a structure using the **struct** keyword. For example:

```
struct light
{
  float intensity;
  vec3 position;
} lightVar;
```

Here, *light* becomes the name of the new type, and *lightVar* becomes a variable of type *light*. To declare variables of the new type, simply use its name:

```
light lightVar2;
```

Arrays

Variables of the same type can be aggregated into arrays by declaring a name followed by brackets (**[ ]**) enclosing an optional size. When an array size is specified within a declaration, it must be an integral constant expression greater than zero.

All basic types and structures can be formed into arrays:

```
float frequencies[3];
uniform vec4
lightPosition[4];
light lights[];
const int numLights = 2;
light lights[numLights];
```

Type Qualifiers
Variable declarations can be accompanied by zero or more qualifiers:

<none>	Local read/write memory, or an input parameter to a function.
`const`	A compile-time constant, or a function parameter that is read-only.
`attribute`	Linkage between a vertex shader and OpenGL for per-vertex data (to define attributes).
`uniform`	Value does not change across the primitive being processed; uniforms form the linkage between a shader, OpenGL, and the application.
`varying`	Linkage between a vertex shader and a fragment shader for interpolated data.
`in`	For function parameters passed into a function.
`out`	For function parameters passed back out of a function, but not initialized for use when passed in.
`inout`	For function parameters passed both into and out of a function.

Here are some rules for these qualifiers:

- Global variables can only use the qualifiers **`const`**, **`attribute`**, **`uniform`**, or **`varying`**. Only one may be specified.
- Local variables can only use the qualifier **`const`**.
- Function parameters can only use the **`in`**, **`out`**, **`inout`**, or **`const`** qualifiers.
- Function return types and structure fields do not use qualifiers.
- Data types for communication from one execution of a shader to the next (to communicate between fragments or between vertices) do not exist. This would prevent parallel execution of the same shader on multiple vertices or fragments.

Example:

```
attribute vec3 tangent;
attribute vec3 binormal;
uniform sampler2D bump_map;
const int  val = 0;
void ComputeSomeVector(in vec3 input, out vec3 result);
void transformVector(inout vec3 vector);
varying float lightIntensity;
varying vec4 texture_coord_aux;
ivec3 variable;
```

Operators and Preference
Operator and preference are very similar to C and C++, but with some restrictions:

- There is no pointer operator or de-reference operator.
- There is no `sizeof` operator.
- Bits-wise operations are illegal: no shift left/right (<< or >>), no exclusive/inclusive or (^, |).
- Modulus operation is illegal (%).
- Unary NOT (~) is illegal.

Subroutines
As indicated previously, a valid shader is a sequence of global declarations and function definitions. An example of function declaration could be:

```
// Prototype
returnType functionName (type0 arg0, type1 arg1, ..., typen argn);

// A function is defined like:
returnType functionName (type0 arg0, type1 arg1, ..., typen argn)
{
  // Do some computation.
  return returnValue;
}
```

- **`returnType`** must be present and include a type. Each `type*` must include a type and can optionally include the qualifier **`in`**, **`out`**, **`inout`**, and/or **`const`**.

- Arrays are allowed as arguments, but not as return types. When arrays are declared as formal parameters, their size must be included. An array is passed to a function by using the array name without any sub-scripting or brackets. The size of the array argument passed within must match the size specified in the formal parameter declaration.

- Structures are also allowed as arguments. The return type can also be structure.

Functions can be overloaded, as long as the argument list differs. For example, the built-in `dot` product function has the following prototypes:

```
float dot (float x, float y);
float dot (vec2 x, vec2 y);
float dot (vec3 x, vec3 y);
float dot (vec4 x, vec4 y);
```

The function `main` is used as the entry point to a shader. The shader does not need to have a `main` method, as one of the other shaders within the linked set will have one.

This function accepts no arguments, returns no value, and must be declared as type **`void`**..

Flow Control

GLSL provides the same mechanisms for flow control as C does, except that GLSL doesn't have the switch structure or `goto` statement. The rest is enumerated here:

- `for`
- `while`
- `do – while`
- `if`
- `if – else`
- `?: (`*`selection`*`)`
- `continue`
- `break`
- `return;`
- `return` *`expression`*`;`
- `discard` (only allowed within fragment shaders) can be used to abandon the current fragment operation. This keyword causes the fragment to be discarded and ceases buffer updates. It would typically be used within a conditional statement:

```
If(texture.a > 0.5)
   discard;
```

Built-in Variables

GLSL provides a set of variables and constants to access the OpenGL fixed function states, such as light parameters (position, diffuse, and color). Some of these variables are mandatory and need to be written (i.e gl_Position and gl_FragColor), while others are read-only (uniform built-in variables).

Vertex Shader Special Built-in Variables

`gl_Position` is only available within vertex shaders, and is intended for writing the homogeneous vertex position. Shaders need to write this variable to be valid, and it can be written at any time during shader execution. It may also be read-back by the shader after being written. This value will be used by the primitive assembly, clipping, culling, and other fixed functionality operations that operate on primitives after vertex processing has occurred. Compilers may generate a diagnostic message if they detect that **`gl_Position`** has not written, or was read before being written, but not all such cases are detectable. Results are undefined if a vertex shader is executed and does not write **`gl_Position`**.

`gl_PointSize` is only available within vertex shaders, and is intended to write the size of the point to be rasterized (measured in pixels).

`gl_ClipVertex` is only available within vertex shaders, and provides a place for vertex shaders to write the coordinates to be used with the clipping planes. The user must ensure the clip vertex and user clipping planes are defined in the same coordinate space. User clipping planes only function correctly under linear transform, though what happens under non-linear transform is undefined.

The built-in vertex shader variables for communicating with fixed functionality are intrinsically declared with the following types:

```
vec4 gl_Position; // Must be written to.
float gl_PointSize; // May be written to.
vec4 gl_ClipVertex; // May be written to.
```

If **`gl_PointSize`** or **`gl_ClipVertex`** are not written to, their values are undefined. Any of these variables can be read back by the shader after writing to them to retrieve what was written. Reading them before writing the results is undefined behavior. If they are written to more than once, the last value written is the one that is used/assumed.

These built-in variables have a global scope.

Fragment Shader Built-in Variables

There are three variables that can be written by the fragment shader, though only one is mandatory (unless the `discard` keyword is executed):

```
vec4 gl_FragCoord;  // May be written to.
bool gl_FrontFacing; // May be written to.
float gl_FragDepth;   // May be written to.

One of these must be written:
vec4 gl_FragColor;
vec4 gl_FragData[gl_MaxDrawBuffers];
// gl_FragData[0] is a synonym of gl_FragColor if GL_ARB_draw_buffers or
GL_ATI_draw_buffers extension is present.
```

`gl_FragData[*]` is only allowed if the extension `GL_ARB_draw_buffer` is implemented in the current GLSL implementation. If it is implemented, the symbol `GL_ARB_draw_buffer` must be defined as 1 within the fragment shader.

Usually, **`gl_FragColor`** is the variable that will be written, unless the shader uses MRT (multiple rendering targets).

- **`gl_FragColor`** refers to the current fragment that will appear within the framebuffer.
- **`gl_FragData[*];`** refers to the current fragment that will appear within the indexed buffer.
- **`gl_MaxDrawBuffers`** is a built-in constant that holds the maximum number of buffers available to be written to.
- **`gl_FrontFacing`** holds a boolean value that tells if the current fragment corresponds to a front face or to a back face.

Writing to **`gl_FragDepth`** will establish the depth value to be used for the fragment being processed. If depth buffering is enabled, and a shader does not write a **`gl_FragDepth`** value, then the fixed function value for depth will be used as the fragment's depth value. If a shader statically contains a write to **`gl_FragDepth`**, then it is responsible for always writing it, meaning if a path writes **`gl_FragDepth`**, then all paths must write to it as well. Otherwise, the value of the fragment's depth may be undefined for executions of the shader that take that path.

`gl_FragCoord` is available as a read-only variable from within fragment shaders and holds the window relative x, y, z coordinates, along with the 1/w values for the fragment. This value is the result of the fixed functionality that interpolates primitives after vertex processing to generate fragments. The *z* component is the depth value that would be used for the fragment's depth, should the shader not write to **`gl_FragDepth`**.

Built-in Constants

These constants are available from both vertex and fragment shaders. The displayed values are the minimum required for a successful GLSL implementation:

```
const int gl_MaxLights = 8;
const int gl_MaxClipPlanes = 6;
const int gl_MaxTextureUnits = 2;
const int gl_MaxTextureCoords = 2;
const int gl_MaxVertexAttribs = 16;
const int gl_MaxVertexUniformComponents = 512
const int gl_MaxVaryingFloats = 32;
const int gl_MaxVertexTextureImageUnits = 0;
const int gl_MaxCombinedTextureImageUnits = 2;
const int gl_MaxTextureImageUnits = 2;
const int gl_MaxFragmentUniformComponents = 64
const int gl_MaxDrawBuffers = 1; // Proposed ARB_draw_buffers.
```

Built-in Attributes

The following attribute names are built into the GLSL language, and can be used from within vertex shaders to access the current values of attributes declared by OpenGL:

```
attribute vec4 gl_Color; // Filled with glColorxx OpenGL call.
attribute vec4 gl_SecondaryColor; // Filled with glSecondaryColorxx OpenGL call.
attribute vec3 gl_Normal; // Filled  with glNormalxx OpenGL call.
attribute vec4 gl_Vertex; // Filled with glVertexxx OpenGL call.

// gl_MultiTexCoordx are filled with the glTexCoordxx or glMultiTexCoordxx OpenGL call.
attribute vec4 gl_MultiTexCoord0;
attribute vec4 gl_MultiTexCoord1;
attribute vec4 gl_MultiTexCoord2;
attribute vec4 gl_MultiTexCoord3;
attribute vec4 gl_MultiTexCoord4;
attribute vec4 gl_MultiTexCoord5;
attribute vec4 gl_MultiTexCoord6;
attribute vec4 gl_MultiTexCoord7;

attribute float gl_FogCoord; // Filled with glFogCoordxx OpenGL call.
// All of these values can be filled also, using all types of vertex arrays (VBO, VAR,
etc.) .
```

General Built-in Uniform States

These states are initialized using standard OpenGL calls, or derived from them:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
uniform mat3 gl_NormalMatrix; // Transpose of the inverse of the upper leftmost 3x3
of gl_ModelViewMatrix.
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixTranspose;
uniform mat4 gl_ProjectionMatrixTranspose;
uniform mat4 gl_ModelViewProjectionMatrixTranspose;
uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixInverseTranspose;
uniform mat4 gl_ProjectionMatrixInverseTranspose;
uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4 gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];

// Normal scaling.
uniform float gl_NormalScale;

// Depth range in window coordinates.
struct gl_DepthRangeParameters
{
    float near;
    float far;
    float diff; // far - near
};
uniform gl_DepthRangeParameters gl_DepthRange;

uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];

struct gl_PointParameters
{
    float size;
    float sizeMin;
    float sizeMax;
    float fadeThresholdSize;
    float distanceConstantAttenuation;
    float distanceLinearAttenuation;
    float distanceQuadraticAttenuation;
};
uniform gl_PointParameters gl_Point;
```

```
struct gl_MaterialParameters
{
    vec4 emission; // Ecm
    vec4 ambient; // Acm
    vec4 diffuse; // Dcm
    vec4 specular; // Scm
    float shininess; // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial, gl_BackMaterial;

struct gl_LightSourceParameters
{
    vec4 ambient; // Acli
    vec4 diffuse; // Dcli
    vec4 specular; // Scli
    vec4 position; // Ppli
    vec4 halfVector; // Derived: Hi
    vec3 spotDirection; // Sdli
    float spotExponent; // Srli
    float spotCutoff; // Crli  (range: [0.0,90.0], 180.0)
    float spotCosCutoff; // Derived: cos(Crli) (range: [1.0,0.0],-1.0)
    float constantAttenuation; // K0
    float linearAttenuation; // K1
    float quadraticAttenuation;// K2
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];

struct gl_LightModelParameters
{
    vec4 ambient; // Acs
};
uniform gl_LightModelParameters gl_LightModel;

// Derived state from products of light and material.
struct gl_LightModelProducts
{
    vec4 sceneColor; // Derived. Ecm + Acm * Acs
};
uniform gl_LightModelProducts gl_FrontLightModelProduct, gl_BackLightModelProduct;

struct gl_LightProducts
{
    vec4 ambient; // Acm * Acli
    vec4 diffuse; // Dcm * Dcli
    vec4 specular; // Scm * Scli
};
uniform gl_LightProducts gl_FrontLightProduct[gl_MaxLights],
gl_BackLightProduct[gl_MaxLights];
```

```
uniform vec4 gl_TextureEnvColor[gl_MaxTextureImageUnits];
uniform vec4 gl_EyePlaneS[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneT[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneR[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneQ[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneS[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneT[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneR[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneQ[gl_MaxTextureCoords];

struct gl_FogParameters
{
    vec4 color;
    float density;
    float start;
    float end;
    float scale; // Derived: 1.0 / (end - start)
};
uniform gl_FogParameters gl_Fog;
```

More information on the other uniform states GLSL states can be found at:
http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf

Varying Variables

Unlike user-defined varying variables, the built-in varying variables don't have a strict one-to-one correspondence between the vertex language and the fragment language. Two sets are provided, one for each language. Their relationship is described below:

Vertex Shader Varying Variables

The following built-in varying variables are writable from vertex shaders. The one that should be written to (if any) should link to a corresponding fragment shader or fixed pipeline which uses it or a state derived from it. Otherwise, behavior is undefined:

```
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;
varying vec4 gl_FrontSecondaryColor;
varying vec4 gl_BackSecondaryColor;
varying vec4 gl_TexCoord[]; // At most will be gl_MaxTextureCoords.
varying float gl_FogFragCoord;
```

Fragment Shader Varying Variables

The following varying variables are readable from fragment shaders. **`gl_Color`** and **`gl_SecondaryColor`** are the same names as the attributes passed to the vertex shader. However, there is no name conflict, as the attributes are visible only within vertex shaders, and the following are only visible in a fragment shader:

```
varying vec4 gl_Color;
varying vec4 gl_SecondaryColor;
varying vec4 gl_TexCoord[]; // At most will be gl_MaxTextureCoords.
varying float gl_FogFragCoord;
```

The fragment shader values **`gl_Color`** and **`gl_SecondaryColor`** will be automatically derived from the system through **`gl_FrontColor`**, **`gl_BackColor`**, **`gl_FrontSecondaryColor`**, and **`gl_BackSecondaryColor`**, based on the visible face.

Built-in Functions

GLSL defines an assortment of built-in convenience functions for scalar and vector operations. Many of these built-in functions can be used in more than one type of shader, but some are intended to provide a direct mapping to hardware, and as such are only available for specific types of shaders.

The built-in functions fall into three main categories:

- They expose some necessary hardware functionality in a convenient way, such as accessing texture maps. There is no way in the language for these functions to be emulated by a shader.
- They represent a trivial operation (clamp, mix, etc.) that is very simple for the user to write, but is very common and may have direct hardware support. It is a very difficult problem for the compiler to map expressions into complex assembler instructions.
- They represent an operation that graphics hardware will likely be able to accelerate at some point. Trigonometry functions would fall into this category, for example.

When using the built-in functions specified below, the input arguments (and corresponding output) can be **`float`**, **`vec2`**, **`vec3`**, **`vec4`**, or *`genType`* as used as the argument. For any specific use of a function, the actual type has to be the same for all arguments and for the return type. This is similar to *`mat`*, which can be a **`mat2`**, **`mat3`**, or **`mat4`**.

Trigonometric Functions

```
genType radians (genType degrees)
genType degrees (genType radians)
genType sin (genType angle)
genType cos (genType angle)
genType tan (genType angle)
genType asin (genType x)
genType acos (genType x)
genType atan (genType y, genType x)
genType atan (genType y_over_x)
```

Exponential Functions

```
genType pow (genType x, genType y)
genType exp (genType x)
genType log (genType x)
genType exp2 (genType x)
genType log2 (genType x)
genType sqrt (genType x)
genType inversesqrt (genType x)
```

Common Functions

```
genType abs (genType x)
genType sign (genType x)
genType floor (genType x) // Returns a value equal to the nearest integer that is
less than or equal to x.
genType ceil (genType x)  // Returns a value equal to the nearest integer that is
greater than or equal to x.
genType fract (genType x)  // Returns x - floor (x).
genType mod (genType x, float y)   // Modulus. Returns x - y . floor (x/y).
genType mod (genType x, genType y)  // Modulus. Returns x - y . floor (x/y).
genType min (genType x, genType y)
genType min (genType x, float y)
genType max (genType x, genType y)
genType max (genType x, float y)
genType clamp (genType x, genType minVal, genType maxVal)
genType clamp (genType x, float minVal, float maxVal) // Note that colors and depths
written by fragment shaders will be clamped by the implementation after the fragment
shader runs.
genType mix (genType x, genType y, genType a)
genType mix (genType x, genType y, float a) //  Returns x * (1 - a) + y * a, i.e., the
linear blend of x and y.
genType step (genType edge, genType x)
genType step (float edge, genType x) // Returns 0.0 if x < edge, otherwise it returns
1.0.
genType smoothstep (genType edge0, genType edge1, genType x)
genType smoothstep (float edge0, float edge1, genType x) // Returns 0.0 if x <= edge0
and 1.0 if x >= edge1 and performs smooth Hermite interpolation between 0 and 1 when
edge0 < x < edge1. This is useful in cases where you would want a threshold function
with a smooth transition. This is equivalent to: genType t; t = clamp ((x - edge0) /
(edge1 - edge0), 0, 1); return t * t * (3 - 2 * t);
```

Geometric Functions

```
float length (genType x)
float distance (genType p0, genType p1)
float dot (genType x, genType y)
vec3 cross (vec3 x, vec3 y)
genType normalize (genType x)
genType faceforward (genType N, genType I, genType Nref)
genType reflect (genType I, genType N)
genType refract(genType I, genType N, float eta)
vec4 ftransform() // For vertex shaders only. This function will ensure that
the incoming vertex value will be transformed in a way that produces exactly
the same result as would be produced by OpenGL's fixed functionality
transform. It is intended to be used to compute gl Position (gl Position =
ftransform(); ).
```

Matrix Functions

The following functions work on operating component-wise over the vectors. For vector results, use the following built-in functions: ***bvec*** is a placeholder for one of **bvec2**, **bvec3**, or **bvec4**; ***ivec*** is a placeholder for one of **ivec2, ivec3**, or **ivec4**; and ***vec*** is a placeholder for **vec2, vec3**, or vec4. In all cases, the sizes of the input and return vectors for any particular call must match.

```
bvec lessThan(vec x, vec y)
bvec lessThan(ivec x, ivec y)
//Returns the component-wise compare of x < y.

bvec lessThanEqual(vec x, vec y)
bvec lessThanEqual(ivec x, ivec y)
//Returns the component-wise compare of x <= y.

bvec greaterThan(vec x, vec y)
bvec greaterThan(ivec x, ivec y)
//Returns the component-wise compare of x > y.

bvec greaterThanEqual(vec x, vec y)
bvec greaterThanEqual(ivec x, ivec y)
//Returns the component-wise compare of x >= y.

bvec equal(vec x, vec y)
bvec equal(ivec x, ivec y)
bvec equal(bvec x, bvec y)
//Returns the component-wise compare of x == y.

bvec notEqual(vec x, vec y)
bvec notEqual(ivec x, ivec y)
bvec notEqual(bvec x, bvec y)
//Returns the component-wise compare of x != y.

bool any(bvec x)
//Returns true if any component of x is true.

bool all(bvec x)
//Returns true only if all components of x are true.

bvec not(bvec x)
//Returns the component-wise logical complement of x.
```

Texture Lookup Functions

Texture lookup functions are available to both vertex and fragment shaders. However, LOD (level of detail) is not computed by fixed functionality for vertex shaders, so there are some differences in operation between vertex and fragment texture lookups. The functions in the table below provide access to textures through samplers, as setup through the OpenGL API. Texture properties such as size, pixel format, number of dimensions, filtering method, number of mip-map levels, and depth comparison are also defined through OpenGL. These properties are taken into account when the texture is accessed via the built-in functions defined below.

The `bias` parameter is optional for fragment shaders, and is not accepted in a vertex shader. If *`bias`* is present, it is added to the calculated LOD prior to performing the texture access operation. If the *`bias`* parameter is not provided, then the implementation automatically selects the LOD. For textures that are not mip-mapped, the texture is used directly. If the textures are mip-mapped and running within a fragment shader, the LOD computed by the implementation is used to do the texture lookup. If running on the vertex shader, then the base texture is simply used. The built-in suffixes with **`Lod`** are allowed only within vertex shaders. *`lod`* is directly used as the level of detail..

One Dimension Textures

`vec4 texture1D (sampler1D sampler, float coord [, float bias])`
`vec4 texture1DProj (sampler1D sampler, vec2 coord [, float bias])`
`vec4 texture1DProj (sampler1D sampler, vec4 coord [, float bias])`
`vec4 texture1DLod (sampler1D sampler, float coord, float lod)`
`vec4 texture1DProjLod (sampler1D sampler, vec2 coord, float lod)`
`vec4 texture1DProjLod (sampler1D sampler, vec4 coord, float lod)`
	Use the texture coordinate *`coord`* to perform a texture lookup within the 1D texture currently bound to *`sampler`*. For the projective (**`Proj`**) versions, the texture coordinate *`coords`* is divided by the last component of *`coord`*.

Two Dimensions Textures

`vec4 texture2D (sampler2D sampler, vec2 coord [, float bias])`
`vec4 texture2DProj (sampler2D sampler, vec3 coord [, float bias])`
`vec4 texture2DProj (sampler2D sampler, vec4 coord [, float bias])`
`vec4 texture2DLod (sampler2D sampler, vec2 coord, float lod)`
`vec4 texture2DProjLod (sampler2D sampler, vec3 coord, float lod)`
`vec4 texture2DProjLod (sampler2D sampler, vec4 coord, float lod)`
	Use the texture coordinate *`coord`* to perform a texture lookup within the 2D texture currently bound to *`sampler`*. For the projective (**`Proj`**) versions, the texture coordinate *`coords`* is divided by the last component of *`coord`*. The third component of `coord` is ignored for the `vec4 coord` variant.

Three Dimensions Textures

`vec4 texture3D (sampler3D sampler, vec3 coord [, float bias])`
`vec4 texture3DProj (sampler3D sampler, vec4 coord [, float bias])`
`vec4 texture3DLod (sampler3D sampler, vec3 coord, float lod)`
`vec4 texture3DProjLod (sampler3D sampler, vec4 coord, float lod)`
	Use the texture coordinate *coord* to perform a texture lookup within the 3D texture currently bound to *sampler*. For the projective (**Proj**) versions, the texture coordinate is divided by *coord*.

Cube Map Textures

`vec4 textureCube (samplerCube sampler, vec3 coord [, float bias] )` `vec4 textureCubeLod (samplerCube sampler, vec3 coord, float lod)`	Use the texture coordinate *coord* to perform a texture lookup within the cube-map texture currently bound to *sampler*. The direction of *coord* is used to select which face to perform a 2-dimensional texture lookup on, as described within section 3.8.6 of the OpenGL 1.4 specification.

Shadow Textures

`vec4 shadow1D (sampler1DShadow sampler, vec3 coord [, float bias] )`
`vec4 shadow2D (sampler2DShadow sampler, vec3 coord [, float bias] )`
`vec4 shadow1DProj (sampler1DShadow sampler, vec4 coord [, float bias])`
`vec4 shadow2DProj (sampler2DShadow sampler, vec4 coord [, float bias])`
`vec4 shadow1DLod (sampler1DShadow sampler, vec3 coord, float lod)`
`vec4 shadow2DLod (sampler2DShadow sampler, vec3 coord, float lod)`
`vec4 shadow1DProjLod(sampler1DShadow sampler, vec4 coord, float lod)`
`vec4 shadow2DProjLod(sampler2DShadow sampler, vec4 coord, float lod)`
	Use the texture coordinate `coord` to perform a depth comparison lookup on the depth texture bound to *sampler*, as described in section 3.8.14 of the OpenGL 1.4 specification. The 3rd component of `coord` (`coord.p`) is used as the R value. The texture bound to *sampler* must be a depth texture, or the results are undefined. For the projective (**Proj**) version of each built-in variable, the texture coordinate is divided by `coord`, giving a depth value (R) of `coord.p`/`coord.q`. The second component of `coord` is ignored for the “1D” variants.

.

Fragment Processing Functions

Fragment processing functions are only available within shaders intended for use on the fragment processor. Derivatives may be expensive (from a computation point-of-view) and/or numerically unstable. Therefore, an OpenGL implementation may approximate the true derivatives by using a fast, though not entirely accurate, derivative computation:

`genType dFdx (genType p)`	Returns the derivative within x, using local differencing for the input argument p.
`genType dFdy (genType p)`	Returns the derivative within y, using local differencing for the input argument p. These two functions are commonly used to estimate the filter width used to anti-alias procedural textures.
`genType fwidth (genType p)`	Returns the sum of the absolute derivative within x and y, using local differencing for the input argument p, i.e: `return = abs (dFdx (p)) + abs (dFdy (p));`

Noise Functions

Noise functions are available to both vertex and fragment shaders. Other than the following, we will take a further look at noise within a later chapter.

`float noise1 (genType x)`	Returns a 1D noise value based on the input value x.
`vec2 noise2 (genType x)`	Returns a 2D noise value based on the input value x.
`vec3 noise3 (genType x)`	Returns a 3D noise value based on the input value x.
`vec4 noise4 (genType x)`	Returns a 4D noise value based on the input value x.