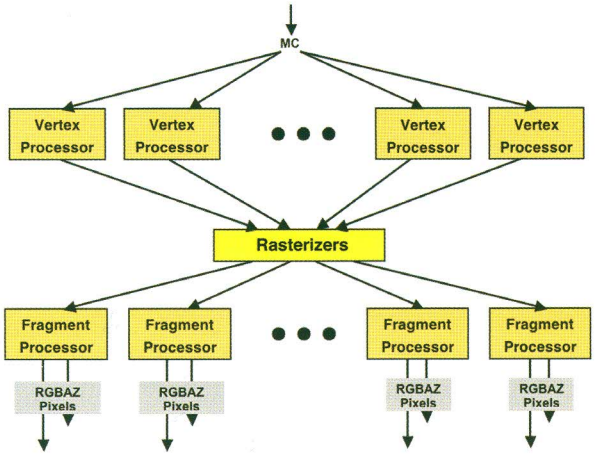
5 The GLSL Shader Language



As shader capabilities in graphics hardware have become more flexible, shader languages have been developed to give the graphics programmer access to these capabilities. The GLSL shading language was designed to be device independent and has been part of the OpenGL standardfrom OpenGL 2.0 forward. It accomplishes its device independence by having compilers built into the graphics card driver translate the GLSL code into the specific device instructions for that card. The actual process of attaching shaders to shader programs, compiling them, and linking them to be downloaded into the graphics card is part of the GLSL API, covered in Chapter 14.

GLSL is a very C-like language, with most of the same fundamental code structure and operators that are found in that language. Thus, there are no challenges to the graphics programmer in understanding the control flow, basic operations, or basic data types in the language. However, there are some areas where GLSL extends the capabilities of C, some areas where

GLSL shader capabilities are very much a moving target. This chapter and all our examples are based on GLSL 4.1. However, we also include many features that are deprecated in that standard but are available in compatibility mode, because they may be helpful to someone learning to work with shaders for the first time.

In order to keep current on GLSL, you should consult [32] from time to time.[1] You will not need a new copy of *glman*, however, because OpenGL will compile only the GLSL shaders, but you may need to get a new OpenGL driver.

GLSL omits some of the capabilities of C, and some areas where GLSL has language features that remind us of the best of earlier generations of computer languages. This chapter focuses on these differences and discusses why they are needed for the shader environment. There is a tendency for any discussion like this to have a strong flavor of a language manual, and you might find that you use this chapter more as a reference than as general reading.

We introduced a number of GLSL language features in Chapter 3, but here we take a more thorough approach to the language and describe it more formally. We are working from the GLSL language specification [23] and include those features and capabilities that we believe are most useful to you, but we are not absolutely complete in our coverage. Once you are familiar with a good working set of GLSL, you probably should read the GLSL specification to see what else is there—especially since the language will continue to evolve over time.[1]

We are indebted to the GLSL Shader Language Specification document both for the overall information it contains and for its excellent tables of GLSL functions and operations that we have borrowed from extensively.

Factors that Shape Shader Languages

Shader languages operate in a different environment and with different goals than general-purpose languages. Their environment is the processing capability of graphics cards, which differs in some important ways from the capability of a general CPU, and their goals are tightly focused on supporting graphics operations, rather than more general kinds of computations. These capabilities shape the language in significant ways, and it is important that you understand their impacts as you write shaders.

1. Good resources: "OpenGL." *Khronos*. Available at http://www.khronos.org/opengl/.
"OpenGL 4.2 API Quick Reference Card." *Khronos*. Available at http://www.khronos.org/files/opengl42-quick-reference-card.pdf, 2010.
"OpenGL Shading Language." *OpenGL*. Available at http://www.opengl.org/documentation/glsl/, 2011.

Graphics Card Capabilities

The first thing we should understand when we think of a language to support graphics shaders is that graphics cards, or GPUs, are not like standard CPUs in several ways. In some ways they are much more advanced than most processors, and in some ways they are more restricted. GPUs are meant to operate on streaming data, transforming it and passing it along a pipeline of processing stages. They hate exceptions, and exceptions can force a whole pipeline to be flushed and restarted. The GLSL shader language has added features that take advantage of graphics card capabilities, especially features that come from the increasingly general-purpose architecture of these cards. These changes are described throughout this chapter.

Parallelism in Graphics Cards

One of the main differences between graphics cards and standard processors is that graphics cards can be parallel processors. Certainly there are some kinds of data-level parallelism in modern processors and, in fact, it has become common for systems to offer parallelism through multiple processors or cores. But these are different kinds of parallelism. Today's graphics cards typically perform parallelism at four levels:

1. Device-Level Parallelism—multiple processors or multiple graphics cards can exist in the same system.
2. Core-Level Parallelism—each processor typically has multiple cores that are capable of independent execution.
3. Thread-Level Parallelism—each core can run multiple threads, that is, can have multiple instruction streams.
4. Data-Level Parallelism—many instructions can act on multiple data elements at once.

Much of the time, the details of these modes of parallelism are abstracted away from the application programmer, and are used as shown in Figure 5.1. This is a good thing. Most of the time, we don't care where or how the processing takes place, just that it happens with sufficient parallelism to handle the increasing demands of today's complex rendering tasks.

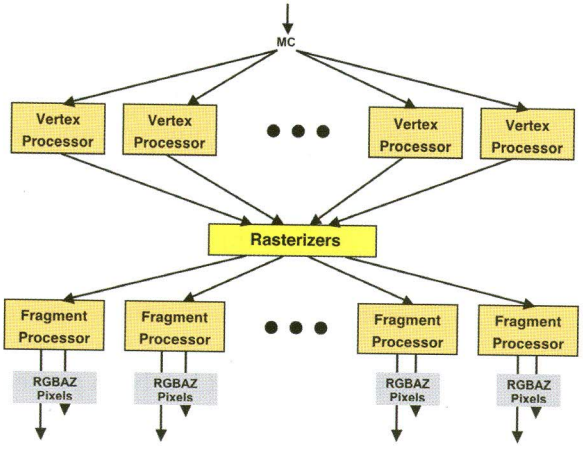


Figure 5.1. Abstracted parallelism in graphics processors.

The Need to Support Graphics Operations

Another key fact about graphics cards is that they must carry out a large number of matrix operations at high speeds, so matrix and vector operations are native to the language, and most likely supported at some level in your hardware. Thus, the GLSL language is shaped by its goal of supporting the operations needed for computer graphics. This is done by adding specific support for matrix and vector data types and operations, including both operations and useful functions; supporting functions that are frequently used for geometric operations; adding language support for noise functions; and adding functions for texture and fragment operations. Some of these are included so they can be optimized, and some are included in anticipation of higher-level operations moving onto graphics cards. GLSL developments so far have extended the original scope of the language, and there is every reason to believe that when additional graphical capabilities are available, such as the recent development of geometry shaders, the language will be extended to support them.

Built-In Data

General-purpose processors have registers that can be used for many kinds of variables, so each must be capable of any kind of operation. Graphics cards designed as OpenGL 2.1 was being released, on the other hand, have a number of special-purpose registers that are loaded with specific data when information is received from the general OpenGL application program. This gives these graphics cards known environments that can be read or written by a shader program, leading to the use of specific names for variables that have particular information.

This aspect of the graphics environment is primarily handled in GLSL by a number of built-in variables that let you access standard data passed to the graphics card from the OpenGL API. This data describes geometry, lighting, transformations, and textures. By using the appropriate GLSL variables, you can use this information for computation in your shaders.

More recently, however, graphics cards have become much more general processors, and these special-purpose registers have been deprecated. A few specific variables have been retained, but the task of building the graphics environment has been passed to the graphics programmer. This increases the programmer's task, but returns significant improvements in performance and in the generality of graphics operations you can create. These changes are described below and in the chapters on each kind of shader.

General GLSL Language Concepts

GLSL is designed to be similar to C and maintains many of the familiar conventions of that language. The overall syntax is the same, with the same conventions for literals and identifiers, and the same preprocessor capabilities. You have have the full set of integer and unsigned integer operations, most of the same operators, and the same operator precedence. The looping and conditional structures are the same, including the `switch` statement. Overall, if you know C, you will find the basic nature of GLSL to be quite comfortable.

However, there are differences between GLSL and C that are driven by the differences in the special environment and the goals of the language, rather than by limitations of C. There are five fundamental ways in which GLSL differs from most conventional languages:

1. The range of conventional operators and functions is extended beyond those usually found in C or similar languages.
2. The language contains some capabilities, such as name sets and shared data namespaces that are implicit in the language, rather than explicitly specified.
3. Data passing between shaders is handled by specifically declaring which variables are input and which are output, and some variables must be explicitly passed along from a shader to subsequent shaders.
4. Function parameters are passed by value-return, rather than by value alone.
5. Some general-purpose language capabilities are omitted.

In GLSL, some conventional operators and functions have extended capabilities, and some new functions and operations are introduced that are convenient for graphics. GLSL has two new implicit capabilities that come from extending the variable types to include types that carry specific capabilities and from using a shared namespace to communicate between shaders. The GLSL function parameters and omitted capabilities from C come from changes in the processing environment. All these differences are described fully later in this chapter, but are briefly discussed in the sections below.

Shared Namespace

Shaders operate independently of each other, so an application can use any shader independently of any other. In order for shaders to communicate, they

must use memory on the graphics card, so the application and its shaders must create names for variables in on-card memory. Sharing these names between shaders that are linked into a single shader program then creates the between-shader communication that shaders need. The set of names of variables used by a set of shaders is called a *shared namespace*.

A namespace may hold *attribute variables*, created by the appliction to define per-vertex data and available only to the vertex shader as `in` variables; *uniform variables*, created by the application to be used as read-only variables by any shader; and *shader-defined variables*, created as `out` variables to pass on as `in` variables to later shaders. (The concept of `out` and `in` variables is discussed later in this chapter.) Some variables created in vertex-processing shaders are intended to be used by fragment shaders by being interpolated across a fragment as a geometric primitive is processed.

You can define attribute variables in your application through the OpenGL API function `glVertexAttrib*( )` and make them accessible to the vertex shader. This lets you define per-vertex data that can be used to define colors or other properties of vertices. You can also define uniform variables to communicate from your OpenGL application to vertex or fragment shaders. Because of limitations on the memory on the graphics card, there is a limit to the total amount of uniform data available to you. Defining and accessing user-defined attribute and uniform variables will be discussed when we present the GLSL API in Chapter 14.

The types and initializers of variables with the same name must match across all shaders that are linked into a single executable. It is legal for some shaders to provide an initializer for a particular variable, while other shaders do not, but all provided initializers must be equal. This is checked as the program is linked.

There are a few specific variables that GLSL uses for very specific capabilities; these may be seen as basic parts of the namespace for one or more kinds of shader. These are described in the chapters on the different shaders.

Extended Function and Operator Capabilities

GLSL extends some of the operators and functions of C to act on vectors and matrices. The standard scalar arithmetic operators are extended to vectors by applying the original operation componentwise. The additive operators are also extended to componentwise operations on matrices, but the multiply operator is taken to mean the standard linear algebra matrix multiplication. Many familiar functions on scalars are similarly extended to vectors by acting componentwise.

GLSL also adds several new vector and matrix operators. There are name set conventions for vectors that let you name components in computer-graphics ways, and there are operations to construct vectors and matrices, and to reorder vector components that give you much more flexible control over these data objects. Overall, GLSL treats vectors and matrices much more like data primitives than does C.

New Functions

GLSL includes many numeric functions that might be relatively easy to write for yourself, but that when included, make their capabilities more standardized across the developer world. These include `floor`, `ceil`, `fract`, `mod` (a generalized version of the familiar function), `min`, `max`, `clamp`, `mix`, `step`, and `smoothstep`. GLSL also includes several vector and matrix functions to support common operations in a uniform way. These include the dot and cross product for vectors, functions for the reflection and refraction vectors, and the transpose and outer product for matrices. These are described fully later.

New Variable Types

GLSL introduces some new variable types: `const`, `attribute`, and `uniform`. Const variables act as constants, much as if they were set with a `#define` statement, only more strongly typed as they are in C and C++. Attribute variables are per-vertex values passed to the vertex shader. Uniform variables let you define graphics variables that do not vary across a primitive and make them accessible to all shaders.

Shaders create a shared namespace, described above, by specifying the variables to be included in the namespace. They do this by declaring `out` variables, treated as write-only and used to give variables values to be used in the next shader in the pipeline, and `in` variables, treated as read-only and used to read values in from the previous shader in the pipeline. An `out` variable declared in, say, a vertex shader, can be used to set a value to be read in an `in` variable of the same name declared in, say, a fragment shader.

There are some keywords that modify the behavior of `in` variables for a fragment shader; these are `flat`, `noperspective`, and `centroid`. The keyword `flat` indicates that values of the input variables are not to be interpolated across a primitive. The usage is

```
flat in float variable_name;
```

as discussed in Chapter 8. The keyword `noperspective` indicates that these variables are interpolated in screen space, rather than being interpolated in a perspective-correct way. The usage is

```
noperspective in float variable_name;
```

The keyword `centroid` indicates that values are to be centroid sampled, that is, sampled at an implementation-defined position in the intersection of a pixel and a primitive, for the purpose of determining what value to apply to the pixel. This is an advanced topic, but it could be useful if you are applying functions across a primitive that may be discontinuous or highly non-linear.

New Function Parameter Types

GLSL function parameters are passed by *value-return*, rather than by value. This allows two-way communication between the calling function and the called function by copying values into and out of function parameters. Parameters are modified by the keywords `in`, `out`, and `inout`. The parameter keyword `in` describes the traditional pass-by-value of C, while the parameter keywords `out` and `inout`, described later in this chapter, replace the need for reference parameters. GLSL does not use pointers.

Language Details

In the sections below, we discuss specific features of the GLSL shader language. In most cases, it should be clear how these features support the kinds of computation needed for shaders. In a few cases, however, we will briefly discuss some examples, such as swizzle operations where the language features make capabilities possible that go beyond those implicit in the nature of the language.

Omitted Language Features

Because GLSL is not a general-purpose language, it does not have some capabilities we are used to seeing in C and other languages. In fact, it cannot have some of these features because the graphics processor does not support all the operations that a general-purpose processor must. The features that are omitted are probably less important for most processing than they are convenient, so you will probably not miss them too much. They include

- There are no `char`, `char *`, or `string` data types, and GLSL has no string-manipulation functions.
- There is no `sizeof( )` operator, because there is no need to deal with data in various sizes. There are standard constructors for arrays and matrices of all needed sizes.
- No implicit type conversions are allowed in GLSL. Conversions are supported by explicit type constructors.

Instead of implicit conversions, or type casts, there are three explicit constructors for simple types, as follows:

- `int(arg)`: converts the argument to an int; the argument may be a float or a bool.
- `float(arg)`: converts the argument to a float; the argument may be an int or a bool.
- `bool(arg)`: converts the argument to a boolean; the argument may be a float or an int.

The usual conversion operations are used: conversions from float to int simply drop the fractional part, nonzero floats or ints convert to the Boolean *true*, etc. This is a different syntax from the familiar cast operations, but it gives you the same functionality if you need it.

New Matrix and Vector Types

GLSL supports a number of predefined data types for vectors and matrices. Vectors may have a real, integer, or Boolean base type, but matrices must be real. Many familiar vector and matrix operations and functions can be applied to variables of these types, and a number of useful new functions are also provided. These are discussed in several sections later in this chapter.

GLSL's built-in floating-point scalar and vector types are `float`, `vec2`, `vec3`, and `vec4`. The storage for a variable of type `vecN` is simply that of a traditional array, but you want to use the built-in type rather than the traditional array type. Using the `vecN` types explicitly makes a much larger number of operations available for the data, and these operations can then take advantage of graphics card parallelism to work at a much higher speed.

GLSL's built-in integer, scalar, and vector types are `int`, `ivec2`, `ivec3`, and `ivec4`. Again, the storage for an `ivec` variable is the same as that for a traditional array, but the explicit `ivec` type can take a much larger set of operations.

GLSL's built-in Boolean scalar and vector types are `bool`, `bvec2`, `bvec3`, and `bvec4`. The main value in Boolean vectors is their ability to support logical operations on vectors, and thus to parallelize some logical tests.

GLSL supports a number of matrix types. For square matrices, `mat2`, `mat3`, and `mat4` can be used for square floating-point matrices of dimension 2×2, 3×3, or 4×4, respectively. Using explicit matrix types rather than simple arrays lets you take advantage of GLSL's many matrix operations and functions.

There are also matrix types that define the dimensions explicitly by listing both dimensions in the declaration. Thus, GLSL has `mat2x2`, `mat2x3`, `mat2x4`, `mat3x2`, `mat3x3`, `mat3x4`, `mat4x2`, `mat4x3`, and `mat4x4` floating-point matrix types. When the two dimensions are equal, this is the same as the declarations above (`mat2x2` is the same as `mat2`, for example). Using these matrix types lets you use GLSL's matrix operations on non-square matrices. Note that there is no declaration of `mat1xN` or `matNx1` arrays; when a one-dimensional array is needed, you can usually use a simple `vecN` in its place.

Name Sets

GLSL supports some standard name sets for vector components that are used for notational convenience. For a `vec4` variable, you can use (x, y, z, w) if you want to refer to components for geometry, (r, g, b, a) if you want to refer to components for color, or (s, t, p, q) if you want to refer to components for texture coordinates. The name set you choose need not depend on the context; you can use (x, y, z, w) to refer to colors if you like, for example. (Note that the letter r for texture coordinates has been replaced by p to avoid confusion with the letter r for red.) In general, you should be careful to avoid name sets that imply such meanings when choosing name sets for vectors other than geometry, RGBA color, or texture.

The component selection syntax allows multiple components to be selected by appending their names (which must be from the same name set) after the period (`.`). So with a declaration `vec4 v4`, for example, we have the examples given in the table below.

`v4.rgba`	Is a `vec4` and is the same as just using `v4`.
`v4.rgb`	Is a `vec3` made from the first three components of `v4`.
`v4.b`	Is a float whose value is the third component of `v4`; also `v4.z` or `v4.p`.
`v4.xz`	Is a `vec2` made from the first and third components of `v4`; also `v4.rb` or `v4.sp`.
`v4.xgba`	Is illegal because the component names do not come from the same set.

Vector Constructors

GLSL has a number of constructors that let you create new vectors from a mix of scalars and other vectors. These constructors have the same name as the vector types and serve to construct a vector of the named type. Some examples are given in the table below.

`vec3(float, float, float)`	Initializes each component of a vector with the explicit floats provided.
`vec4(ivec4)`	Makes a `vec4` with component-wise conversion.
`vec2(float)`	Initializes a `vec2` with the float value in each position.
`ivec3(int, int, int)`	Initializes an `ivec3` with three ints.
`bvec4(int,int,float,float)`	Performs four Boolean conversions.
`vec2(vec3)`	Drops the third component of a `vec3`.
`vec3(vec4)`	Drops the fourth component of a `vec4`.
`vec3(vec2, float)`	`vec3.xy = vec2` `vec3.z = float`
`vec3(float, vec2)`	`vec3.x = float` `vec3.yz = vec2`
`vec4(vec3, float)`	`vec4.xyz = vec3` `vec4.w = float`
`vec4(float, vec3)`	`vec4.x = float` `vec4.yzw = vec3`
`vec4(vec2a, vec2b)`	`vec4.xy = vec2a` `vec4.zw = vec2b`

To initialize a matrix by using specified vectors or scalars, we recall that matrices are stored in column-major order (unlike in C), so the components are assigned to the matrix elements in that order.

`mat2(vec2, vec2)`
`mat3(vec3, vec3, vec3)`
`mat4(vec4, vec4, vec4, vec4)`
`mat3x2(vec2, vec2, vec2)`
	Each matrix is filled using one column per argument.

`mat2(float, float, float, float)`	Rows are first column and second column, respectively.

`mat3(float, float, float, float, float, float, float, float, float)`	Rows are first column, second column, and third column, respectively.
`mat4(float,float,float,float, float,float,float,float, float,float,float,float, float,float,float,float)`	Rows are first column, second column, third column, and fourth column, respectively.
`mat2x3(vec2, float, vec2, float)`	Rows are first column and second column, respectively.

Even though GLSL offers these 2D matrix formats, it is sometimes convenient to use simpler 1D arrays. For example, we can represent a 3 × 3 matrix M as three separate `vec3` variables and then multiply M by a matrix V by using three dot products.

There are many other ways to construct a matrix from vectors and scalars, as long as there are enough components to initialize the matrix. The construction acts as though the matrix begins as an identity matrix (or a subset of an identity matrix), and the new elements that are specified replace the originals. For example, to construct a matrix from a matrix we might have the possibilities given in the following table.

`mat3x3(mat4x4)`	Uses the upper-left 3 × 3 submatrix of the `mat4x4` matrix.
`mat2x3(mat4x2)`	Takes the upper-left 2 × 2 submatrix of the `mat4x2`, and sets the last column to `vec2(0.)`.
`mat4x4(mat3x3)`	Puts the `mat3x3` matrix in the upper-left submatrix and sets the lower right component to 1 and the rest to 0.

Functions Extended to Matrices and Vectors

Standard programming languages tend to have a number of numeric functions and operators, including trigonometric functions, exponential functions, number manipulation functions, and relational operators. In GLSL, most of these can operate on vectors, as well as on scalar values.

The familiar bitwise integer functions <<, >>, %, &, |, ^, and ~ are all available in GLSL and apply to both simple integer and `ivecN` data.

In the lists of functions below, we use the term `genType` to refer to any scalar or vector data type that is appropriate for each function. In general, these functions use `float` or `vecN` data, but you can use an integer type anywhere a float type is needed, because GLSL allows that implicit type conversion.

GLSL supports the familiar set of trigonometric and inverse trigonometric functions. As with all the other functions, these can operate componentwise on vectors. Arguments identified with *angle* are assumed to be in radians.

`genType radians( genType degrees)`	Converts degrees to radians: $(\pi/180)$**degrees*.
`genType degrees( genType radians)`	Converts radians to degrees: $(180/\pi)$**radians*.
`genType sin( genType angle)` `genType cos( genType angle)` `genType tan( genType angle)`	The standard trigonometric sine, cosine, and tangent functions, with the argument *angle* in radians.
`genType asin(genType x)`	Arc sine. Returns the primary radian value of the angle whose sine is x. The range of returned values is $[-\pi/2,\pi/2]$. Undefined if $\|x\|>1$.
`genType acos(genType x)`	Arc cosine. Returns the primary radian value of the angle whose cosine is x. The range of returned values is $[0,\pi]$. Results are undefined if $\|x\|>1$.
`genType atan(genType y, genType x)`	Arc tangent. Returns the primary radian value of the angle whose tangent is y/x. The signs of x and y determine the angle's quadrant. The range of returned values is $[-\pi,\pi]$. Undefined if x and y are both 0.
`genType atan( genType y_over_x)`	Arc tangent. Returns the primary radian value of the angle whose tangent is *y_over_x*. The range of returned values is $[-\pi/2,\pi/2]$.

GLSL also supports the full range of hyperbolic trigonometric functions, sinh, cosh, and tanh, and their inverses.

GLSL has the usual exponential, logarithmic, and square root functions, including exponential and logarithmic functions of base 2. These can also operate componentwise on vectors.

`genType pow(genType x, genType y)`	Power function. Returns x raised to the y power, x^y. Undefined if $x < 0$, or if $x = 0$ and $y \le 0$.
`genType exp(genType x)`	Returns the natural exponentiation of x, e^x.
`genType log(genType x)`	Returns the natural logarithm of x, the value y for which $x = e^y$. Undefined if $x \le 0$.
`genType exp2(genType x)`	Returns 2 raised to the x power: 2^x.
`genType log2(genType x)`	Returns the base 2 logarithm of x, the value y for which $x = 2^y$. Undefined if $x <= 0$.
`genType sqrt(genType x)`	Returns the nonnegative square root of x. Undefined if $x < 0$.
`genType inversesqrt( genType x)`	Returns $1/\sqrt{x}$. Undefined if $x \le 0$.

Don't use `inversesqrt( )` to normalize a vector! Use `normalize( )` instead.

GLSL supports a familiar set of common functions, as well as some that are not as familiar. Among the less-familiar functions are some that are very useful in combining colors or geometry. These functions can all operate componentwise on any vector. Note that the `mod` function is generalized to real numbers as well as integers. The functions `abs`, `clamp`, `min`, `max`, and `sign` can be applied to integers as well as to real numbers.

`genType abs(genType x)`	Returns x if $x \geq 0$, otherwise returns $-x$.
`genType sign(genType x)`	Returns 1.0 if $x > 0$, 0.0 if $x = 0$, or –1.0 if $x < 0$.
`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 the fraction part of x: $x - floor(x)$.
`genType truncate (genType x)`	Returns the integer closest to x whose absolute value is not larger than $abs(x)$.
`genType round(genType x)`	Returns the integer closest to x.
`genType mod(genType x, float y)` `genType mod(genType x, genType y)`	Generalized modulus. Returns $x - y * floor(x/y)$.
`genType min(genType x, genType y)` `genType min(genType x, float y)`	Minimum. Returns y if $y < x$, otherwise returns x.
`genType max(genType x, genType y)` `genType max(genType x, float y)`	Maximum. Returns y if $x < y$, otherwise returns x.

`genType clamp(genType x,`
`genType minVal,`
`genType maxVal)`
`genType clamp(genType x,`
`float minVal,`
`float maxVal)`
	Clamped value; Returns $min(max(x, minVal), maxVal)$. Undefined if $minVal > maxVal$.

`genType mix(genType x,`
`genType y,`
`genType a)`
`genType mix(genType x,`
`genType y,`
`float a)`
	Proportional mix. Returns a linear combination of x and y: $a * x + (1 - a) * y$.

`genType mix(genType x,`
`genType y,`
`bool b)`
	Select the value of either x or y, depending on the value of b.

`genType step(genType edge,`
`genType x)`
`genType step(float edge,`
`genType x)`
	Step function at the value of edge. Returns 0.0 if $x < edge$, otherwise 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 3.*t*t - 2.*t*t*t;`
	Results are undefined if $edge0 > edge1$.

Operations Extended to Matrices and Vectors

The traditional primitive operators, sum (+), difference (–), product (*), and quotient (/), operate only on scalar data in most languages. In GLSL, this is extended in a natural way to vector and matrix data. There are two different cases to consider here.

Sums, differences, and quotients act componentwise when

- One operand is a scalar and one is either a vector or matrix, or
- Both are vectors or matrices.

Products act componentwise when

- One operand is a scalar and one is either a vector or matrix, or
- Both are vectors.

Note that if `u` and `v` are vectors, `u*v` is *not* a dot product! This product `u*v` is just a componentwise product and is still a vector. If you are trying to get a dot product, use `dot( )` instead. In order to compute a product of vectors or matrices, of course, both operands must have the same dimensions and appropriate types. The result is a vector or matrix of the appropriate size and type.

Products of a vector and a matrix, or of two matrices, are different; they do not perform scalar operations, but perform the correct linear algebra operations on their operands. For vectors `u`, `v` and matrices `m`, `n`, `r` (always assuming appropriate dimensions so the operations make sense),

- We can write `u = v * m;` this treats `v` as if it were a $1 \times d$ matrix and performs the correct operation of the dot product of `v` with each column of `m`.
- We can write `u = m * v;` this treats `u` and `v` as if they were $d \times 1$ matrices and performs the correct operation of the dot product of `v` with each row of `m`.
- We can write `r = m * n;` this performs the dot product of each row of `m` with each column of `n` to produce the matrix `r`.

In addition, the assignment operator = and relational equality and inequality operators == and != can be applied to entire arrays or structs, but the operands must be of the same size and, for structs, the same declared types. Other relational functions are available for vectors, but they differ from the familiar built-in relational operators. These are described later in this chapter.

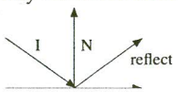
Other familiar vector operations, the dot and cross products, are available through the built-in dot and cross product operations that are described fully later in this chapter when we present GLSL's matrix functions. These include several other useful capabilities. For example, if you should want the componentwise scalar product of two matrices, you will need to use the new matrix function `matrixCompMult`. Or if you should want to do an outer product of two vectors (the outer product of two vectors `u`, `v` of dimension `n` is defined as though `u` has dimension $n \times 1$ and `v` has dimension $1 \times n$ and you are computing the matrix product [`u` times `v`]), you can use the new matrix function `outerProduct`.

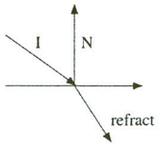
New Functions

As described in previous sections, many common functions from C are also available in GLSL. However, languages such as C do not focus on graphics and so have few functions to handle geometry and matrix data. GLSL provides several new functions to do this. The list here is long, but is broken out into several different areas, as they are in the language specification.

Geometric Functions

GLSL supports a number of functions to support geometric operations. These have an obvious application for graphics, since many of the basic graphical operations basically manipulate geometry. These functions include the familiar scalar functions for length and dot product, and the familiar vector operations for cross product and normalization. They also include less familiar vector operations for reflection, refraction, and faceforward that can be very useful.

`float length(genType x)`	Returns the length of the vector x, $\sqrt{(x[0]^2 + x[1]^2 + \ldots)}$.
`float distance(genType p0, genType p1)`	Returns the distance between *p0* and *p1*: `length(p0 - p1)`.
`float dot(genType x, genType y)`	Returns the dot product of x and y: `x[0]*y[0]+x[1]*y[1]+...` .
`vec3 cross(vec3 x, vec3 y)`	Returns the cross product of x and y, $\begin{Bmatrix} x[1]y[2] - y[1]x[2] \\ x[2]y[0] - y[2]x[0] \\ x[0]y[1] - y[0]x[1] \end{Bmatrix}$.
`genType normalize(genType x)`	Returns a vector in the same direction as x, but with a length of 1, or $\frac{x}{\|x\|}$.
`genType faceforward( genType N, genType I, genType Nref)`	Make N face in the direction of *Nref*. If $dot(Nref, I) < 0$ return N, otherwise return $-N$.
`genType reflect(genType I, genType N)`	For the incident vector I and surface orientation N, returns the reflection direction: $I - 2 * dot(N,I) * N$. The normal vector N must already be normalized in order to achieve the correct result. 

`genType refract(genType I, genType N, float eta)`	For the incident vector I and surface normal N, and the ratio of indices of refraction *eta*, return the refraction vector. The result is computed by
	`k=1.0-eta*eta*(1.0-dot(N,I)*dot(N,I))`
	`if (k < 0.0)`
	`  return genType(0.0)`
	`else`
	`  return eta*I-(eta*dot(N,I)+sqrt(k))*N.`
	The incident vector I and the normal vector N must already be normalized in order to achieve the correct result. 

Matrix Functions

GLSL has several useful functions for matrices, including componentwise multiplication, the outer product, and the transpose. If no matrix type is otherwise specified, `mat` is used for any matrix type.

`mat matrixCompMult(mat x, mat y)`	Multiply matrix *x* by matrix *y* component-wise, so that *result[i][j]* is the scalar product of *x[i][j]* and *y[i][j]*. Note: to get linear algebraic matrix multiplication, use the multiply operator (*).

`mat2 outerProduct(vec2 c, vec2 r)`
`mat3 outerProduct(vec3 c, vec3 r)`
`mat4 outerProduct(vec4 c, vec4 r)`
`mat2x3 outerProduct(vec3 c, vec2 r)`
`mat3x2 outerProduct(vec2 c, vec3 r)`
`mat2x4 outerProduct(vec4 c, vec2 r)`
`mat4x2 outerProduct(vec2 c, vec4 r)`
`mat3x4 outerProduct(vec4 c, vec3 r)`
`mat4x3 outerProduct(vec3 c, vec4 r)`
	Treats the first parameter *c* as a column vector (matrix with one column) and the second parameter *r* as a row vector (matrix with one row) and does a linear algebraic matrix multiply *c * r*, yielding a matrix whose number of rows is the number of components in *c* and whose number of columns is the number of components in *r*.

`mat2 transpose(mat2 m)`
`mat3 transpose(mat3 m)`
`mat4 transpose(mat4 m)`
	Returns a matrix that is the transpose of *m*; *m* need not be square, as is shown. The input matrix *m* is not modified.

`mat2x3 transpose(mat3x2 m)`
`mat3x2 transpose(mat2x3 m)`
`mat2x4 transpose(mat4x2 m)`
`mat4x2 transpose(mat2x4 m)`
`mat3x4 transpose(mat4x3 m)`
`mat4x3 transpose(mat3x4 m)`	

Relational Functions for Vectors

GLSL extends the familiar relational operators for scalars to a set of relational functions for vectors. These compare vectors componentwise and return a `bvec` result that can be used for parallel comparisons. There are also several functions that can convert a `bvec` result to a single Boolean scalar. In these descriptions, *vec* is a real vector, *ivec* is an integer vector, and *bvec* is a Boolean vector, and their lengths are arbitrary except that in each case the lengths are equal.

`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)`	The vector equivalent of the logical *or*, |—returns true if any component of *x* is true.
`bool all(bvec x)`	The vector equivalent of the logical *and*, &—returns true only if all components of *x* are true.
`bvec not(bvec x)`	The vector equivalent of the logical *not*, !—returns the component-wise logical complement of *x*.

Texture Lookup Functions

The built-in texture lookup functions give you access to textures through samplers, as set up through the OpenGL API. A texture sampler is a GLSL uniform variable that has been previously associated with a particular texture unit. The texture unit acts as a pointer to the texture data itself and its sampling information, such as size, pixel format, number of dimensions, filtering methods, and number of mip-map levels. These texture properties are taken into account as the texture is accessed.

Texture lookup functions can be used by both vertex and fragment shaders. However, 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 additional functions support texture lookups for shadow textures or for *level-of-detail* ("LOD") in shaders. Functions whose names include `Lod` are allowed only in vertex shaders. The bias term is optional for fragment shaders, but is not accepted for vertex shaders. If it is included, it is added to the level of detail before the texture access.

`vec4 texture( sampler1D sampler, float coord [, float bias])`
`vec4 textureProj( sampler1D sampler, vec{2,4} coord [, float bias])`
`vec4 textureLod( sampler1D sampler, float coord, float lod)`
`vec4 textureProjLod( sampler1D sampler, vec{2,4} coord, float lod)`
	Use the texture coordinate `coord` to do a texture lookup in the 1D texture currently bound to `sampler`. For the projective (`Proj`) versions, the texture coordinate `coord.s` is divided by the last component of `coord`.

`vec4 texture( sampler2D sampler, vec2 coord [, float bias])`
`vec4 textureProj( sampler2D sampler, vec{3,4} coord [, float bias])`
`vec4 textureLod( sampler2D sampler, vec2 coord, float lod)`
`vec4 textureProjLod( sampler2D sampler, vec{3,4} coord, float lod)`
	Use the texture coordinate `coord` to do a texture lookup in the 2D texture currently bound to `sampler`. For the projective (`Proj`) versions, the texture coordinate (`coord.s, coord.t`) is divided by the last component of `coord`. The third component of `coord` is ignored for the `vec4 coord` variant.

`vec4 texture( sampler3D sampler, vec3 coord [, float bias])`
`vec4 textureProj( sampler3D sampler, vec4 coord [, float bias])`
`vec4 textureLod( sampler3D sampler, vec3 coord, float lod)`
`vec4 textureProjLod( sampler3D sampler, vec4 coord, float lod)`
	Use the texture coordinate `coord` to do a texture lookup in the 3D texture currently bound to `sampler`. For the projective (`Proj`) versions, the texture coordinate is divided by `coord.q`.

`vec4 texture( samplerCube sampler, vec3 coord [, float bias])` `vec4 textureLod( samplerCube sampler, vec3 coord, float lod)`	Use the texture coordinate `coord` to do a texture lookup in the cube map texture currently bound to `sampler`. The direction of `coord` is used to select in which face to do a two-dimensional texture lookup.

`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 do a depth comparison lookup on the depth texture bound to `sampler`, as described in Section 3.8.14 of Version 1.4 of the OpenGL specification. The third component of `coord (coord.p)` is used as the *R* value. The texture bound to `sampler` must be a depth texture, or results are undefined. For the projective (`Proj`) version of each built-in, the texture coordinate is divided by `coord.q`, giving a depth value *R* of `coord.p/coord.q`. The second component of `coord` is ignored for the `1D` variants.

Fragment Processing Functions

GLSL fragment shaders can antialias procedural textures using a variety of techniques, including analytic prefiltering. To support this, GLSL includes functions that let you calculate the gradient of any parameter in screen space, and a function that gives you a value for the upper bound of the width of the sampling filter needed to eliminate aliasing.

`genType dFdx(genType p)`	Returns the derivative in *x* using local differencing for the input argument *p*.
`genType dFdy(genType p)`	Returns the derivative in *y* using local differencing for the input argument *p*.
These two functions are commonly used to estimate the filter width used to antialias procedural textures. It is assumed that the expression is being evaluated in parallel on a SIMD array, so that at any given point in time the value of the function is known at the grid points represented by the array. Local differencing between array elements can therefore be used to derive `dFdx`, `dFdy`, etc.	
`genType fwidth(genType p)`	Returns the sum of the absolute derivative in *x* and *y* using local differencing for the input argument *p*, `abs(dFdx(p)) + abs(dFdy(p));`

Noise Functions

The GLSL language specification defines the noise functions shown here, but at this writing, it has not actually been implemented in all GLSL systems. While everyone agrees that there need to be built-in noise functions available, not everyone agrees on what would be the best specific implementation. This is why *glman* builds them in using 2D and 3D textures.

GLSL includes the built-in noise functions below, which can be used by both fragment and vertex shaders. The noise functions are pseudo-random stochastic functions that are C^1 continuous with range [–1., 1.] and mean 0.0, and they are deterministic for a given input. The output has the same statistical character if the domain is rotated or translated. The noise functions can readily be used to create textures that add to the visual complexity of a scene.

`float noise1(genType x)`	Returns a 1D noise value based on the input value *x*. At this time, this function is not available in GLSL.
`vec2 noise2(genType x)`	Returns a 2D noise value based on the input value *x*. At this time, this function is not available in GLSL.
`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*.

Swizzle

An operation that is probably new to you is the *swizzle* operation. This lets you rearrange or reorganize the components of a vector in any way you want. This operation is specified by simply writing the components of a vector in any order you want, using one of the component name sets. For example, if *m* is a `vec4`, you can reverse the order of the components of *m* by writing `m.wzyx`, or you can duplicate some of the components of *m* by writing `m.rrbb`.

New Function Parameter Types

GLSL function parameters differ from the standard C "pass by value" approach. GLSL parameters are passed by *value-return*. This means that parameters' values may be copied into a function or may be returned by the function, or both, but unlike with "pass by reference" variables, there is no change to any of the actual parameters until the function returns. The function parameters are preceded by one of the keywords `in`, `out`, or `inout`; this comes before the type of the parameter.

The keywords' meanings are

- `const`: The value of the input parameter is copied to the formal parameter, but no change to the formal parameter is allowed in the function.
- `in`: The value of the actual parameter is copied to the formal parameter, but no changed value will be returned. The actual parameter may be an expression that sets the value to be copied into the function. The formal parameter may be changed during the execution of the function. The keyword `in` may be preceded by `const`, in which case the formal parameter will be treated as a const in the function.
- `out`: The formal parameter must be an lvalue and will have no value until it is set inside the function. Any function operations may use this parameter, but a value must be set in the function. The value of the formal parameter in the function is copied to the actual parameter when the function terminates.
- `inout`: The formal parameter must be an lvalue and is assumed to have a value when it is copied to the function, and this value may be used or changed during the function execution. When the function terminates, the value of the formal parameter is copied to the actual parameter.

Const

The *const* data type lets you declare named compile-time constants. Any variables qualified by const are read-only variables for that shader and must be initialized when declared; the initial values must be constant expressions. The `const` qualifier can be used with any of the basic data types. As in C++, using `const` is good programming style because it is strongly typed and it will cause the compiler to throw an error if you attempt to re-assign a value to something you originally expected should never get reassigned.

GLSL has several built-in const variables for vertex and fragment shaders. The values given for initialization are implementation-dependent and are the minimum values allowed.

```
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;
```

Compatibility Mode

OpenGL 4.1 has replaced a number of features of the 2.x and 3.x standards with much more general functionality. This has increased the power, efficiency, and generality of the standard, but requires much more planning and setup than the earlier standard. If you are maintaining OpenGL code that was based on the 2.x and 3.x standards, or if you simply want to write quick shaders to test out some ideas, you may want to work in what is called *compatibility mode*: a mode in which you can use the earlier OpenGL functionality.

Defining Compatibility Mode

It is quite straightforward to specify that a shader is to be run in compatibility mode. If you are working in OpenGL 4.x, you can simply put the line

```
#version 400 compatibility
```

at the top of any shader source. If you are working in OpenGL 3.3, a similar command can be used:

```
#version 330 compatibility
```

Then you can use any functionality you like from the OpenGL 2.1 standard.

Among the things you might find most useful from the earlier standard is the set of built-in data. These let you pick up attribute and uniform variables that are defined by OpenGL 2.1 functions so you can use them easily in your shaders.

OpenGL 2.1 Built-in Data Types

GLSL originally included some completely new data types that correspond to functions needed to manage data flow across the spectrum of an application, the OpenGL API, the onboard data on a graphics card, and the needs of vertex and fragment shaders. These types are available in OpenGL 2.1 or in compatibility mode for later versions, and are named *attribute*, *uniform*, and *varying*. Their function is described in this section.

In general, you can differentiate these data types by how often the data they represent change. *Uniform* data changes infrequently and never within a graphics primitive; *attribute* data changes frequently, often as frequently as each vertex; and *varying* data changes most frequently, with each fragment as it's interpolated by the rasterizer.

Attribute

The *attribute* data qualifier lets you access per-vertex data passed to the graphics card by the OpenGL API functions. Attribute variables have only *float*, *vec*, and *mat* data types, and cannot be declared as arrays or structs. Attribute variables are only accessible in a vertex shader and are read-only for that shader. They must have global scope and must be declared outside of function bodies before they are first used.

Originally, GLSL had built-in variable names for all the standard OpenGL vertex attributes to give you easy access to data defined by OpenGL vertex functions. These are

```
attribute vec4 gl_Color;
attribute vec3 gl_Normal;
attribute vec4 gl_Vertex;
attribute vec4 gl_MultiTexCoordi; // i = 0..7
```

Uniform

The *uniform* qualifier identifies global variables whose values are constant across a graphics primitive. This can be used with any of the basic data types, or when declaring a variable whose type is a structure, or an array of any of these. Uniform variables are read-only for all shaders and are initialized externally either at link time or through the OpenGL API.

GLSL has a large set of built-in uniform variables that let you access the graphics states set by the OpenGL API in your application. These are listed below in groups that access similar states.

Primary matrices. OpenGL maintains four primary matrices that are available to your shaders:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
```

Derived matrices. OpenGL computes a number of other matrices that are used in various geometry processing steps. Some of these are inverses or transposes of the primary matrices, and you should be aware that if the primary matrix is poorly conditioned, the inverses may have unpredictable values. These derived matrices are available to your shaders:

```
uniform mat3 gl_NormalMatrix; // transpose of 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. If your application does its own normal scaling instead of relying on the normalization operation, you can access that normal scaling factor:

```
uniform float gl_NormalScale;
```

Front and back clipping planes. When you specify your projection in OpenGL, you specify the front and back clipping planes, and hence the depth of these planes. This data is available to your shaders:

```
struct gl_DepthRangeParameters
{
  float near; // n
  float far; // f
  float diff; // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;
```

Clip planes. OpenGL allows you to define clipping planes in your scene by specifying the equation of the plane as four real numbers. This data is available to your shaders:

```
uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];
```


Point parameters. In OpenGL you can specify the properties of a geometric point. This data is available to your shaders:

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

In the items below, we introduce some shortcut names for a number of properties of materials and lights. These are used to show how the later derived materials states are computed.

Materials. When you use the OpenGL lighting model, you specify properties of the materials that make up a graphics primitive. This data is available to your shaders:

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

Lights. When you specify a light in OpenGL, you specify a number of properties, from the light's colors to the light's position, to the type of light it is to be. You also specify the kind of light model to be used. This data is available to your shaders:

```
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 materials state. These states are products of the light and material that are used for actual color computations:

```
struct gl_LightModelProducts
{
  vec4 sceneColor;  // Derived. Ecm + Acm * Acs
};
uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

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

Texture environment. This set of GLSL built-in uniform variables gives you the colors that are produced by each texture unit and the coordinates of the eye plane or object plane for eye-linear or object-linear textures, respectively.


```
uniform vec4 gl_TextureEnvColor[gl_MaxTextureUnits];
uniform vec4 gl_EyePlaneS[gl_MaxTextureCoords];
    // eye linear
uniform vec4 gl_EyePlaneT[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneR[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneQ[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneS[gl_MaxTextureCoords];
    // object linear
uniform vec4 gl_ObjectPlaneT[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneR[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneQ[gl_MaxTextureCoords];
```

Fog. All the GLSL fog parameters set by the graphics API are available to your shaders:

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

Varying

GLSL's varying variables provide communication from vertex shaders to fragment shaders. Vertex shaders compute information for each vertex and write them to varying variables to be interpolated across a graphics primitive and then used by a fragment shader. GLSL specifies that default interpolations of varying variables must be done in a perspective-correct manner, so the problems of perspective correction that we saw in Chapter 1 are not part of GLSL. Only those varying variables used in the fragment shader must be written by the previous shader in the shader pipeline, but that previous shader may also declare other varying variables. A fragment shader cannot write to a varying variable.

The `varying` qualifier can be used only with float variables, floating-point vectors, matrices, or arrays of these. Structures cannot be varying. Varying variables must have global scope and must be declared outside of function bodies.

Varying variables may be defined using a modifier that describes how they are interpolated across a fragment. These modifiers are `flat`, `noperspective`, and `centroid`, and were discussed earlier in this chapter.

Summary

We have seen that GLSL is a language that looks familiar enough to be used easily, but that it has a significant number of new features that make writing shaders possible—and that are easy enough to use that it's straightforward to get started by writing shaders that do interesting things. Like OpenGL, it has enough capability that you will likely never run out of ways to add sophistication and new features to your shaders, or to create every effect that it can give you. This chapter should familiarize you with the basic operation of GLSL and should be a useful reference for you, but only when you actually begin to use the language to write shaders will you really understand the graphical power it gives you.

Exercises

1. For the following table of operator and operand type, indicate which operator can legally operate on the operands given. For each one that is legal, create an example of the two operands and show the result of the operation.

Operator	Left Operand	Right Operand	Result
+	mat2	float	:
+	vec3	ivec3	:
*	vec2	mat2	:
*	mat3x4	mat4x3	:

2. For the following table of functions and parameter type(s), indicate whether the function can legally act on the parameter(s) given. For each case where the function can legally act, identify the type of the return value, give an example of the function applied to the parameter(s) and show the returned value of the function.

	Function name	Parameter(s)
a.	pow	vec4, vec4
b.	mod	vec3, float
c.	cross	vec3, vec3
d.	outerProduct	mat2x3, mat3x4
e.	notEqual	float, vec4

3. Use GLSL operators to write three different ways to calculate the distance between two points.
4. Diagram the data flow that describes how geometric data gets from an application, through the OpenGL API, to the graphics card, to a vertex shader, to a fragment shader, and finally to a single pixel that is output to the graphics color buffer.
5. Write constructors to create new variables from old ones, using either the scalar, vector, or matrix constructors described in this chapter.
   a. Construct an integer `I` from a float `F`.
   b. Construct a `vec3` from three ints.
   c. Construct a `vec2` as the middle two components of a `vec4`.
   d. Construct a `mat4` with the first row a set of four floats and with the remaining part of each column given by a `vec3`.
6. Write statements you could use in a GLSL shader to convert a `vec4` color to a grayscale color with the same alpha value. Is there a difference in how you would do this for a vertex shader and for a fragment shader?