7 Introduction to Shading Language

This chapter provides a refresher on the RenderMan Shading Language. However, it is not a tutorial on programming in general, nor is it intended to be a substitute for *The RenderMan Companion* or the *RenderMan Interface Specification*. But rather it is meant to serve as a handy quick reference guide to Shading Language itself.

Shading Language is loosely based on the C programming language. We will use this to our advantage in this chapter by assuming that (1) you already know about general programming concepts such as variables, loops, and so on; (2) you are reasonably familiar with C; (3) your mathematical background is sufficient to allow casual discussion of trigonometry and vector algebra; (4) you have sufficient background in computer graphics to understand intermediate-level concepts related to illumination computations. If you are lacking in any of these areas, you should review the material in Chapter 2 and its references.

7.1 Shader Philosophy

Many renderers have a fixed shading model. This means that a single equation is used to determine the appearance of surfaces and the way that they respond to light. For example, many renderers use simple Phong illumination, which looks like this:

$$C_{\text{output}} = K_a C_{\text{amb}} + \sum_{i=1}^{\text{nlights}} (K_d C_{\text{diff}}(N \cdot L_i) Cl_i + K_s C_{\text{spec}}(R \cdot L_i)^n)$$

where

- L_i and Cl_i are the direction and color, respectively, of light number i
- K_a, K_d, K_s, n, C_{amb}, C_{diff}, and C_{spec} are user-specified parameters to the equation. By changing these parameters, the user can make different objects look as if they are made of different materials
- N is the surface normal and R is the mirror reflection direction from the point of view of the camera
- C_{output} is the resulting color of the surface

This particular equation is especially common and tends to make objects appear as though they are made of plastic if C_{spec} is white and somewhat like metal if both C_{spec} and C_{diff} are set to the same color.

Because a world made of flat-colored plastic would hardly be interesting, a common extension to this scheme is to allow the use of stored image files to determine the value of C_{diff} as it varies across the surface (this is called "texture mapping") or to modulate the surface normal N ("bump mapping"). Somewhat more sophisticated renderers may allow an image file to modulate any of the user-supplied parameters, but this still does not change the fundamental form of the shading equation, and therefore the resulting materials have a rather narrow range of appearances. Furthermore, even when using stored images to modulate the surface parameters, you are limited to the few kinds of modulations allowed by the renderer, and stored textures have a variety of limitations including limited resolution, obvious tiling and repetition artifacts, storage costs, and the problem of how the image textures get created in the first place.

7.1.1 Shading Language Overview

In contrast to this scheme, RenderMan-compliant renderers do not use a single shading equation. Rather, a programming language is used to describe the interactions of lights and surfaces. This idea was pioneered by Rob Cook (Cook, 1984), and further elaborated by Pat Hanrahan in the *RenderMan Specification* itself (Pixar, 1989; Hanrahan and Lawson, 1990) and by the *PRMan* product. The programs describing the output of light sources, and how the light is attenuated by surfaces and

volumes, are called *shaders*, and the programming language that we use is known as *Shading Language*.

The *RenderMan Interface Specification* describes several types of shaders, distinguished by what quantities they compute and at what point they are invoked in the rendering pipeline:

Surface shaders describe the appearance of surfaces and how they react to the lights that shine on them.
Displacement shaders describe how surfaces wrinkle or bump.
Light shaders describe the directions, amounts, and colors of illumination distributed by a light source in the scene.
Volume shaders describe how light is affected as it passes through a participating medium such as smoke or haze.
Imager shaders describe color transformations made to final pixel values before they are output. (Programmable imager shaders are supported by *BMRT*, but not by *PRMan*.)

All shaders answer the question "What is going on at this spot?" The execution model of the shader is that you (the programmer) are only concerned with a single point on the surface and are supplying information about that point. This is known as an *implicit* model, as compared to an *explicit* model, which would be more of the flavor "draw feature *X* at position *Y*." The job of a surface shader is to calculate the color and opacity at a particular point on some surface. To do this, it may calculate any function, do texture map lookups, gather light, and so on. The shader starts out with a variety of data about the point being shaded but cannot find out about any other points.

The RenderMan Shading Language is a C-like language you can use to program the behavior of lights and surfaces. Shading Language gives you

- basic types useful for manipulating points, vectors, or colors
- mathematical, geometric, and string functions
- access to the geometric state at the point being shaded, including the position, normal, surface parameters, and amount of incoming light
- parameters supplied to the shader, as specified in the declaration of the shader or alternatively attached to the geometry itself

With this information, the goal of the surface shader is to compute the resulting color, opacity, and possibly the surface normal and/or position at a particular point.

The remainder of this chapter will give a quick introduction to the RenderMan Shading Language, with an emphasis on the basic functionality you will need to write surface and displacement shaders. The vast majority of shaders written for production are surface shaders. Although volume and light shaders are also important, they are more esoteric and less frequently written and so will be covered separately elsewhere in this book.

Listing 7.1 `plastic.sl`: The standard `plastic` shader. Note that we have modified the shader slightly from the RI spec in order to reflect more modern SL syntax and idioms. The line numbers are for reference only and are not part of the shader!

```
1   surface
2   plastic ( float Ka=1, Kd=1, Ks=0.5, roughness = 0.1;
3             color specularcolor = 1;
4           )
5   {
6       /* Simple plastic-like reflection model */
7       normal Nf = faceforward(normalize(N),I);
8       vector V = -normalize(I);
9       Ci = Cs * (Ka*ambient() + Kd*diffuse(Nf))
10            + Ks*specularcolor*specular(Nf,V,roughness);
11      Oi = Os;  Ci *= Oi;
12  }
```

7.1.2 Quick Tour of a Shader

Listing 7.1 is an example surface shader that roughly corresponds to the single built-in shading equation of many renderers. If you are an experienced C programmer, you will immediately pick out several familiar concepts. Shaders look rather like C functions.

Lines 1 and 2 specify the type and name of the shader. By convention, the source code for this shader will probably be stored in a disk file named `plastic.sl`, which is simply the shader name with the extension `.sl`. Lines 2–4 list the parameters to the shader and their default values. These defaults may be overridden by values passed in from the RIB stream. Lines 5–12 are the body of the shader. In lines 7–8, we calculate a forward-facing normal and a normalized "view" vector, which will be needed as arguments to the lighting functions. Lines 9–10 call several built-in functions that return the amount of ambient, diffuse, and specular reflection, scaling each by different weights, and summing them to give the final surface color `Ci`. Because surface shaders must set associated colors and opacities, line 11 sets the final opacity `Oi` simply to the default opacity of the geometric primitive, `Os`, and then multiplies `Ci` by `Oi`, in order to ensure that it represents *associated* color and opacity. Note that several undeclared variables such as `N`, `I`, and `Cs` are used in the shader. These are so-called *global* variables that the renderer precomputes and makes available to the shader.

Most surface shaders end with code identical to lines 7–11 of the example. Their main enhancement is the specialized computations they perform to select a base surface color, rather than simply using the default surface color, `Cs`. Shaders may additionally change the weights of the various lighting functions and might modify `N` and/or `P` for bump or displacement effects.

Table 7.1: Names of built-in data types.

`float`	Scalar floating-point data (numbers)
`point`	Three-dimensional positions, directions, and surface orientations
`vector`	
`normal`	
`color`	Spectral reflectivities and light energy values
`matrix`	4×4 transformation matrices
`string`	Character strings (such as filenames)

7.2 Shading Language Data Types

Shading Language provides several built-in data types for performing computations inside your shader as shown in Table 7.1. Although Shading Language is superficially similar to the C programming language, these data types are not the same as those found in C. Several types are provided that are not found in C because they make it more convenient to manipulate the graphical data that you need to manipulate when writing shaders. Although `float` will be familiar to C programmers, Shading Language has no `double` or `int` types. In addition, SL does not support user-defined structures or pointers of any kind.

7.2.1 Floats

The basic type for scalar numeric values in Shading Language is the `float`. Because SL does not have a separate type for integers, `float` is used in SL in circumstances in which you might use an `int` if you were programming in C. Floating-point constants are constructed the same way as in C. The following are examples of `float` constants: `1`, `2.48`, `-4.3e2`.

7.2.2 Colors

Colors are represented internally by three floating-point components.[1] The components of colors are referent to a particular *color space*. Colors are by default represented as RGB triples (`"rgb"` space). You can assemble a color out of three

[1] Strictly speaking, colors may be represented by more than three components. But since all known RenderMan-compliant renderers use a three-component color model, we won't pretend that you must be general. It's highly unlikely that you'll ever get into trouble by assuming three color components.

Table 7.2: Names of color spaces.

`"rgb"`	The coordinate system that all colors start out in and in which the renderer expects to find colors that are set by your shader (such as `Ci`, `Oi`, and `Cl`).
`"hsv"`	hue, saturation, and value
`"hsl"`	hue, saturation, and lightness
`"YIQ"`	The color space used for the NTSC television standard.
`"xyz"`	CIE *XYZ* coordinates
`"xyY"`	CIE *xyY* coordinates

floats, either representing an RGB triple or some other color space known to the renderer. Following are some examples:

```
color (0, 0, 0)              /* black */
color "rgb" (.75, .5, .5)    /* pinkish */
color "hsv" (.2, .5, .63)    /* specify in "hsv" space */
```

All three of these expressions return colors in `"rgb"` space. Even the third example returns a color in `"rgb"` space—specifically, the RGB value of the color that is equivalent to hue 0.2, saturation 0.5, and value 0.63. In other words, when assembling a color from components given relative to a specific color space in this manner, there is an implied transformation to `"rgb"` space. The most useful color spaces that the renderer knows about are listed in Table 7.2.

Colors can have their individual components examined and set using the `comp` and `setcomp` functions, respectively. Some color calculations are easier to express in some color space other than `"rgb"`. For example, desaturating a color is more easily done in `"hsv"` space. Colors can be explicitly transformed from one color space to another color space using `ctransform` (see Section 7.5 for more details). Note, however, that Shading Language does not keep track of which color variables are in which color spaces. It is the responsibility of the SL programmer to track this and ensure that by the end of the shader, `Ci` and `Oi` are in the standard `"rgb"` space.

7.2.3 Points, Vectors, Normals

Points, vectors, and normals are similar data types with identical structures but subtly different semantics. We will frequently refer to them collectively as the "point-like" data types when making statements that apply to all three types.

A `point` is a position in 3D space. A `vector` has a length and direction but does not exist in a particular location. A `normal` is a special type of vector that is *perpendicular* to a surface and thus describes the surface's orientation. Such a perpendicular vector uses different transformation rules from ordinary vectors, as we will discuss in this section. These three types are illustrated in Figure 7.1.

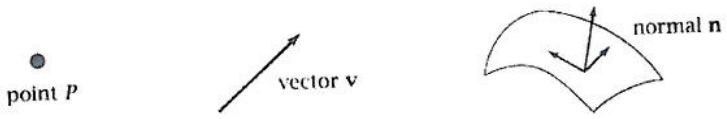

Figure 7.1 Points, vectors, and normals are all comprised of three floats but represent different entities—positions, directions, and surface orientations.

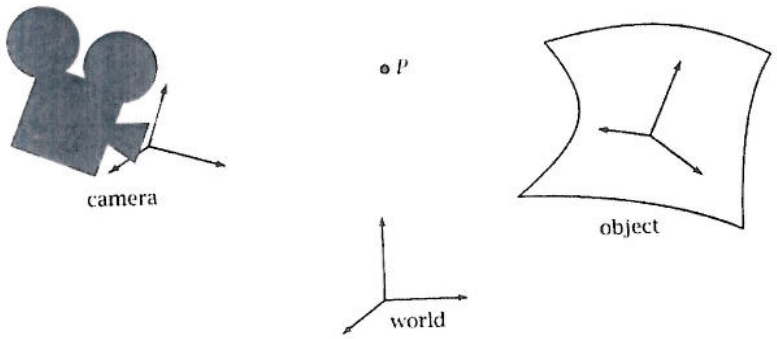

Figure 7.2 A point may be measured relative to a variety of coordinate systems.

All of these point-like types are internally represented by three floating-point numbers that uniquely describe a position or direction relative to the three axes of some coordinate system.

As shown in Figure 7.2, there may be many different coordinate systems that the renderer knows about ("world" space and a local "object" space, for example, were discussed in Chapter 3; others will be detailed later). Obviously, a particular point in 3D can be represented by many different sets of three floating-point numbers—one for each coordinate system. So which one of these spaces is the one against which your points and vectors are measured?

All points, vectors, and normals are described relative to some coordinate system. All data provided to a shader (surface information, graphics state, parameters, and vertex data) are relative to one particular coordinate system that we call the "current" coordinate system. The "current" coordinate system is one that is convenient for the renderer's shading calculations.

You can "assemble" a point-like type out of three floats using a constructor:

```
point (0, 2.3, 1)
vector (a, b, c)
normal (0, 0, 1)
```

These expressions are interpreted as a point, vector, and normal whose three components are the floats given, relative to "current" space. For those times when you really need to access or set these three numbers, SL provides the routines xcomp, ycomp, zcomp, comp, setxcomp, setycomp, setzcomp, setcomp (see Section 7.5).

As with colors, you may also specify the coordinates relative to some other coordinate system:

```
Q = point "object" (0, 0, 0);
```

This example assigns to Q the point at the origin of "object" space. However, this statement does *not* set the components of Q to (0,0,0)! Rather, Q will contain the "current" space coordinates of the point that is at the same location as the origin of "object" space. In other words, the point constructor that specifies a space name implicitly specifies a transformation to "current" space. This type of constructor also can be used for vectors and normals.

The choice of "current" space is implementation dependent. For *PRMan*, "current" space is the same as "camera" space; and in *BMRT*, "current" space is the same as "world" space. Other renderers may be different, so it's important not to depend on "current" space being any particular coordinate system.

Some computations may be easier in a coordinate system other than "current" space. For example, it is much more convenient to apply a "solid texture" to a moving object in its "object" space than in "current" space. For these reasons, SL provides built-in functions that allow you to transform points among different coordinate systems. The built-in functions transform, vtransform, and ntransform can be used to transform points, vectors, and normals, respectively, from one coordinate system to another (see Section 7.5). Note, however, that Shading Language does not keep track of which point variables are in which coordinate systems. It is the responsibility of the SL programmer to keep track of this and ensure that, for example, lighting computations are performed using quantities in "current" space.

Be very careful to use the right transformation routines for the right point-like types. As described in Chapter 2, points, direction vectors, and surface normals all transform in subtly different ways. Transforming with the wrong matrix math will introduce subtle and difficult-to-fix errors in your code. Therefore, it is important to always use transform for points, vtransform for vectors, and ntransform for normals.

Several coordinate systems are predefined by name in the definition of Shading Language. Table 7.3 summarizes some of the most useful ones. The RIB statement CoordinateSystem (or C API RiCoordinateSystem) may be used to give additional names to user-defined coordinate systems. These names may also be referenced inside your shader to designate transformations.

Table 7.3: Names of predeclared geometric spaces.

`"current"`	The coordinate system that all points start out in and the one in which all lighting calculations are carried out. Note that the choice of `"current"` space may be different on each renderer.
`"object"`	The local coordinate system of the graphics primitive (sphere, patch, etc.) that we are shading.
`"shader"`	The coordinate system active at the time that the shader was declared (by the `Surface`, `Displacement`, or `LightSource` statement).
`"world"`	The coordinate system active at `WorldBegin`.
`"camera"`	The coordinate system with its origin at the center of the camera lens, x-axis pointing right, y-axis pointing up, and z-axis pointing into the screen.
`"screen"`	The *perspective-corrected* coordinate system of the camera's image plane. Coordinate (0,0) in `"screen"` space is looking along the z-axis of `"camera"` space.
`"raster"`	The 2D projected space of the final output image, with units of pixels. Coordinate (0,0) in `"raster"` space is the upper-left corner of the image, with x and y increasing to the right and down, respectively.
`"NDC"`	Normalized device coordinates—like raster space but normalized so that x and y both run from 0 to 1 across the whole image, with (0,0) being at the upper left of the image, and (1,1) being at the lower right (regardless of the actual aspect ratio).

7.2.4 Matrices

Shading Language has a `matrix` type that represents the transformation matrix required to transform points and vectors between one coordinate system and another. Matrices are represented internally by 16 floats (a 4×4 homogeneous transformation matrix). Beware if you declare a `matrix` of storage class `varying`. That's going to be a lot of data!

A `matrix` can be constructed from a single float or 16 floats. For example:

```
matrix zero = 0;   /* makes a matrix with all 0 components */
matrix ident = 1;  /* makes the identity matrix */

/* Construct a matrix from 16 floats */
matrix m = matrix (m00, m01, m02, m03, m10, m11, m12, m13,
                   m20, m21, m22, m23, m30, m31, m32, m33);
```

Assigning a single floating-point number x to a matrix will result in a matrix with diagonal components all being x and other components being zero (i.e., x times the

identity matrix). Constructing a matrix with 16 floats will create the matrix whose components are those floats, in row-major order.

Similar to point-like types, a `matrix` may be constructed in reference to a named space:

```
/* Construct matrices relative to something other than "current" */
matrix q = matrix "shader" 1;
matrix m = matrix "world" (m00, m01, m02, m03, m10, m11, m12, m13,
                           m20, m21, m22, m23, m30, m31, m32, m33);
```

The first form creates the matrix that transforms points from `"current"` space to `"shader"` space. Transforming points by this matrix is identical to calling `transform("shader",...)`. The second form prepends the current-to-world transformation matrix onto the 4×4 matrix with components $m_{0,0} \ldots m_{3,3}$. Note that although we have used `"shader"` and `"world"` space in our examples, any named space is acceptable.

Matrix variables can be tested for equality and inequality with the `==` and `!=` Boolean operators. Also, the `*` operator between matrices denotes matrix multiplication, while `m1 / m2` denotes multiplying `m1` by the inverse of matrix `m2`. Thus, a matrix can be inverted by writing `1/m`. In addition, some functions will accept matrix variables as arguments, as described in Section 7.5.

7.2.5 Strings

The `string` type may hold character strings. The main application of strings is to provide the names of files where textures may be found. Strings can be checked for equality and can be manipulated with the `format()` and `concat()` functions. String constants are denoted by surrounding the characters with double quotes, as in `"I am a string literal"`. Strings in Shading Language may be `uniform` only.

7.3 Shading Language Variables

There are three kinds of variables in Shading Language: global variables, local variables, and shader parameters. These correspond pretty much exactly to the globals, locals, and parameters of a subroutine in a language like C. The one difference is that variables in Shading Language not only have a data type (`float`, `point`, etc.) but also a designated *storage class*. The storage class can be either `uniform` or `varying` (except for strings, which can only be `uniform`). Variables that are declared as `uniform` have the same value everywhere on the surface. (Note that you may assign to, or change the value of, `uniform` variables. Do not confuse `uniform` with the concept of "read-only.") Variables that are declared as `varying` may take on different values at different surface positions.

Table 7.4: Global variables available inside surface and displacement shaders. Variables are read-only except where noted.

`point P`	Position of the point you are shading. Changing this variable displaces the surface.
`normal N`	The surface shading normal (orientation) at `P`. Changing `N` yields bump mapping.
`normal Ng`	The true surface normal at `P`. This can differ from `N`; `N` can be overridden in various ways including bump mapping and user-provided vertex normals, but `Ng` is always the true surface normal of the facet you are shading.
`vector I`	The *incident* vector, pointing from the viewing position to the shading position `P`.
`color Cs, Os`	The default surface color and opacity, respectively.
`float u, v`	The 2D parametric coordinates of `P` (on the particular geometric primitive you are shading).
`float s, t`	The 2D texturing coordinates of `P`. These values can default to `u`, `v`, but a number of mechanisms can override the original values.
`vector dPdu` `vector dPdv`	The partial derivatives (i.e., tangents) of the surface at `P`.
`time`	The time of the current shading sample.
`float du, dv`	An estimate of the amount that the surface parameters `u` and `v` change from sample to sample.
`vector L` `color Cl`	These variables contain the information coming from the lights and may be accessed from inside `illuminance` loops only.
`color Ci, Oi`	The final surface color and opacity of the surface at `P`. Setting these two variables is the primary goal of a surface shader.

7.3.1 Global Variables

So-called *global variables* (sometimes called *graphics state variables*) contain the basic information that the renderer knows about the point being shaded, such as position, surface orientation, and default surface color. You need not declare these variables; they are simply available by default in your shader. Global variables available in surface shaders are listed in Table 7.4.

7.3.2 Local Variables

Local variables are those that you, the shader writer, declare for your own use. They are analogous to local variables in C or any other general-purpose programming language.

The syntax for declaring a variable in Shading Language is (items in brackets are optional)

[class] type variablename [= initializer]

where

- the optional *class* specifies one of `uniform` or `varying`. If *class* is not specified, it defaults to `varying` for local variables.
- *type* is one of the basic data types, described earlier.
- *variablename* is the name of the variable you are declaring.
- if you wish to give your variable an initial value, you may do so by assigning an *initializer*.

Recent renderers also support arrays, declared as follows:

class type variablename [arraylen] = { init0, init1 . . . }

Arrays in Shading Language must have a constant length; they may not be dynamically sized. Also, only 1D arrays are allowed. Other than that, however, the syntax of array usage in Shading Language is largely similar to C. Some examples of variable declarations are

```
float a;           /* Declare; current value is undefined */
uniform float b;   /* Explicitly declare b as uniform */
float c = 1;       /* Declare and assign */
float d = b*a;     /* Another declaration and assignment */
float e[10];       /* The variable e is an array */
```

When you declare local variables, you will generally want them to be `varying`. But be on the lookout for variables that take on the same value everywhere on the surface (for example, loop control variables) because declaring them as `uniform` may allow some renderers to take shortcuts that allow your shaders to execute more quickly and use less memory. (*PRMan* is a renderer for which `uniform` variables take much less memory and experience much faster computation.)

7.3.3 Shader Parameters

Parameters to your shader allow you to write shaders that are capable of simulating a family of related surfaces. For example, if you are writing a shader for a wood surface, you may wish to use parameters to specify such things as the grain color and ring spacing, rather than having them "hard-coded" in the body of the shader.

Parameterizing your shader not only allows you to reuse the shader for a different object later but also allows you to write the shader without knowing the value of the parameter. This is particularly useful if you are working in a production environment where an art director is likely to change his or her mind about the details of an object's appearance after you have written the shader. In this case, it is much easier to change the parameter value of the shader than to return to the source code and try to make deeper changes. For this reason, we strongly encourage parameterizing

your shader to the greatest degree possible, eliminating nearly all hard-coded constants from your shader code, if possible. Well-written shaders for "hero" objects often have dozens or even hundreds of parameters.

Here is an example partial shader, showing several parameters being declared:

```
surface pitted ( float Ka=1, Kd=1, Ks=0.5;
                 float angle = radians(30);
                 color splotcolor = 0;
                 color stripecolor = color (.5, .5, .75);
                 string texturename = "";
                 string dispmapname = "mydisp.tx";
                 vector up = vector "shader" (0,0,1);
                 varying point Pref = point (0,0,0);
               )
{
  :
}
```

Note the similarity to a function declaration in C. The syntax for parameter declarations is like that for ordinary local variable declarations, except that shader parameters *must* be declared with default values assigned to them. If a storage class is not specifed, it defaults to `uniform` for shader parameters.

In the RIB file, you'll find something like

```
Declare "Kd" "float"
Declare "stripecolor" "color"
Surface "pitted" "Kd" [0.8] "stripecolor" [.2 .3 .8]
Sphere 1 -1 1 360
```

The `Surface` line specifies that the given shader should become part of the attribute state and hence be attached to any subsequent geometric primitives. That line not only specifies the name of the shader to use but also overrides two of its parameter values: `Kd` and `stripecolor`. Notice that prior to their use, those parameters are declared so that the renderer will know their types.

7.3.4 Declarations and Scoping

It is worth noting that a local variable declaration is just an ordinary program statement; a declaration may be placed anywhere that a statement would be allowed. In particular, it is not necessary to sharply divide your shader such that all variables are declared, then all statements are listed with no further variable declarations. Rather, you may freely mix variable declarations and other statements as long as all variables are declared prior (textually) to their first use. This is largely a stylistic choice, but many programmers feel that declaring variables near their first use can make code more readable.

Declarations may be scoped, as in C, by enclosing a group of statements inside curly braces. For example,

```
float x = 2;                /* outer declaration */
{
    float x = 1;            /* inner declaration */
    printf("%f\n", x);
}
    printf ("%f/n", x);
```

In this code fragment, the first `printf` statement will produce `1`, but the second `printf` will produce `2`. In other words, the variable `x` declared in the *inner* scope hides the similarly named but nevertheless separate variable `x` declared in the *outer* scope.

7.4 Statements and Control Flow

The body of a shader is a sequence of individual *statements*. This section briefly explains the major types of statements and control-flow patterns in Shading Language.

7.4.1 Expressions

The expressions available in Shading Language include the following:

- constants: floating point (e.g., `1.0`, `3`, `-2.35e4`), string literals (e.g., `"hello"`), and the named constant `PI`
- point, vector, normal, or matrix constructors, for example: `point "world" (1,2,3)`
- variable references
- unary and binary operators on other expressions, for example:

- expr	(negation)
expr + expr	(addition)
*expr * expr*	(multiplication)
expr - expr	(subtraction)
expr / expr	(division)
expr ^ expr	(vector cross product)
expr . expr	(vector dot product)

The operators `+`, `-`, `*`, `/`, and the unary `-` (negation) may be used on any of the numeric types. For multicomponent types (colors, vectors, matrices), these operators combine their arguments on a component-by-component basis.

The ^ and . operators only work for vectors and normals and represent cross product and dot product, respectively.[2]

The only operators that may be applied to the `matrix` type are * and /, which respectively denote matrix-matrix multiplication and matrix multiplication by the inverse of another matrix.

- type casts, specified by simply having the type name in front of the value to cast:

```
vector P        /* cast a point to a vector */
point f         /* cast a float to a point */
color P         /* cast a point to a color! */
```

The three-component types (`point`, `vector`, `normal`, `color`) may be cast to other three-component types. A `float` may be cast to any of the three-component types (by placing the float in all three components) or to a `matrix` (which makes a matrix with all diagonal components being the `float`). Obviously, there are some type casts that are not allowed because they make no sense, like casting a `point` to a `float` or casting a `string` to a numerical type.

- ternary operator, just like C: *condition* ? *expr1* : *expr2*
- function calls

7.4.2 Assignments

Assignments are nearly identical to those found in the C language:

variable = *expression* ;

arrayvariable[*expression*] = *expression* ;

Also, just as in C, you may combine assignment and certain arithmetic operations using the +=, -=, *=, and /= operators. Examples of declarations and assignments follow:

```
a = b;          /* Assign b's value to a */
d += 2;         /* Add 2 to d */
e[5] = a;       /* Store a's value in element 5 of c */
c = e[2];       /* Reference an array element */
```

Unlike C, Shading Language *does not* have any of the following operators: integer modulus (%), bit-wise operators or assignments (&, |, ^, &=, |=, ^=), pre- and postincrement and decrement (++ and --).

[2] Because the `vector` and `normal` type are recent additions to SL (with `point` serving their purposes previously), most SL compilers will allow the vector operations to be performed on `points` but will issue a warning.

7.4.3 Decisions, Decisions

Conditionals in Shading Language work much as in C:

```
if ( condition )
    truestatement
```

and

```
if ( condition )
    truestatement
else
    falsestatement
```

The statements can also be entire blocks, surrounded by curly braces. For example,

```
if (s > 0.5) {
    Ci = s;
    Oi = 1;
} else {
    Ci = s+t;
}
```

In Shading Language, the condition may be one of the following Boolean operators: ==, != (equality and inequality); <, <=, >, >= (less-than, less-than or equal, greater-than, greater-than or equal). Conditions may be combined using the logical operators: && (and), || (or), ! (not).

Unlike C, Shading Language has no implied cast from float to Boolean. In other words, the following *is not legal*:

```
float f = 5;
if (f) {   /* Not legal */
    ...
}
```

A C programmer may instinctively write this code fragment, intending that the conditional will evaluate to true if f is nonzero. But this is not the case in Shading Language. Rather, the shader programmer must write

```
float f = 5;
if (f != 0) {   /* OK */
    ...
}
```

7.4.4 Lather, Rinse, Repeat

Two types of loop constructs work nearly identically to their equivalents in C. Repeated execution of statements for as long as a condition is true is possible with a `while` statement:

```
while ( condition )
    truestatement
```

Also, C-like `for` loops are also allowed:

```
for ( init; condition; loopstatement )
    body
```

As with `if` statements, loop conditions must be relations, not floats. As with C, you may use `break` and `continue` statements to terminate a loop altogether or skip to the next iteration, respectively. As an enhancement over C, the `break` and `continue` statements may take an optional numeric constant, allowing you to efficiently exit from nested loops. For example,

```
for (i = 0;  i < 10;  i += 1) {
    for (j = 0;  j < 5;  j += 1) {
        if (...some condition...)
            continue 2;
    }
    ...
}
```

In this example, the numeral 2 after the `continue` indicates that under the appropriate conditions, execution should skip to the next iteration in the loop involving `i`—that is, the outer loop. If no number indicating a nesting level is given, it is assumed that 1 is intended—that is, that only the current loop should be exited or advanced.

As discussed in Chapter 6, *PRMan* shades entire grids at a time by simulating a virtual SIMD machine. This introduces extra overhead into keeping track of which points are executing inside the body of loops and conditionals that have varying conditions. Be sure to declare your loop control variables (the counters controlling the loop iteration, such as `i` and `j` in the example) as `uniform` whenever possible. Care that the conditions controlling `if`, `while`, and `for` statements are `uniform` can greatly speed up execution of your shaders. In addition, using `varying` variables in the condition of a loop or conditional is asking for trouble, because this can lead to jaggies on your surface and can even produce incorrect results if derivatives are calculated inside the body of the loop or conditional (see Chapter 11 for more details).

7.5 Simple Built-in Functions

Shading Language provides a variety of built-in functions. Many are described in this section. For brevity, functions that are identical to those found in the standard C library are presented with minimal elaboration, as are simple functions that

are adequately explained in both *The RenderMan Companion* and *The RenderMan Interface* 3.1.

This section documents most of the everyday functions you will use for surface shaders but is not intended to be comprehensive. Many built-in functions are covered elsewhere in this book. Functions used for patterns are covered in Chapter 10. Derivatives (`Du()`, `Dv()`, `area()`) are covered in Chapter 11. Lighting and environment mapping functions are covered in Chapter 9.

Note that some functions are *polymorphic*; that is, they can take arguments of several different types. In some cases we use the shorthand `ptype` to indicate a type that could be any of the point-like types `point`, `vector`, or `normal`. (Note that there is no actual type `ptype`—we are just using this as shorthand!)

Angles and Trigonometry

```
float radians (float d)
float degrees (float r)

float sin (float angle)
float cos (float angle)
float tan (float angle)

float asin (float f)
float acos (float f)
float atan (float y, x)
float atan (float y_over_x)
```

Angles, as in C, are assumed to be expressed in radians.

Exponentials, etc.

```
float pow (float x, float y)
float exp (float x)
float log (float x)

float log (float x, base)
```

Arbitrary base logarithm of x.

```
float sqrt (float x)
float inversesqrt (float x)
```

Square root and `1/sqrt`.

Miscellaneous Simple Scalar Functions

```
float abs (float x)
```

Absolute value of x.

`float sign (float x)`

Returns 1 if $x > 0$, -1 if $x < 0$, 0 if $x = 0$.

`float floor (float x)`
`float ceil (float x)`
`float round (float x)`

Return the highest integer less than or equal to x, the lowest integer greater than or equal to x, or the closest integer to x, respectively.

`float mod (float a, b)`

Just like the *fmod* function in C, returns $a - b * \text{floor}(a/b)$.

`float min (`*type* `a, b, ...)`
`float max (`*type* `a, b, ...)`
`float clamp (`*type* `x, minval, maxval)`

The `min` and `max` functions return the minimum or maximum, respectively, of a list of two or more values. The `clamp` function returns

```
min(max(x,minval),maxval)
```

that is, the value x clamped to the specified range. The *type* may be any of `float`, `point`, `vector`, `normal`, or `color`. The variants that operate on colors or point-like objects operate on a component-by-component basis (i.e., separately for x, y, and z).

type `mix (`*type* `x, y; float alpha)`

The `mix` function returns a linear blending of any simple *type* (any of `float`, `point`, `vector`, `normal`, or `color`): $x * (1 - \alpha) + y * (\alpha)$

`float step (float edge, x)`

Returns 0 if $x < edge$ and 1 if $x \geq edge$.

`float smoothstep (float edge0, edge1, x)`

Returns 0 if $x \leq edge0$, and 1 if $x \geq edge1$ and performs a smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is useful in cases where you would want a thresholding function with a smooth transition.

Color Operations

`float comp (color c; float i)`

Returns the ith component of a color.

`void setcomp (output color c; float i, float x)`

Modifies color `c` by setting its ith component to value `x`.

```
color ctransform (string tospacename; color c_rgb)
color ctransform (string fromspacename, tospacename; color c_from)
```

Transform a color from one color space to another. The first form assumes that `c_rgb` is already an "rgb" color and transforms it to another named color space. The second form transforms a color between two named color spaces.

Geometric Functions

```
float xcomp (ptype p)
float ycomp (ptype p)
float zcomp (ptype p)
float comp (ptype p; float i)
```

Return the x, y, z, or simply the ith component of a point-like variable.

```
void setxcomp (output ptype p; float x)
void setycomp (output ptype p; float x)
void setzcomp (output ptype p; float x)
void setcomp (output ptype p; float i, x)
```

Set the x, y, z, or simply the ith component of a point-like type. These routines alter their first argument but do not return any value.

```
float length (vector V)
float length (normal V)
```

Return the length of a vector or normal.

```
float distance (point P0, P1)
```

Returns the distance between two points.

```
float ptlined (point P0, P1, Q)
```

Returns the distance from `Q` to the closest point on the line segment joining `P0` and `P1`.

```
vector normalize (vector V)
vector normalize (normal V)
```

Return a vector in the same direction as V but with length 1—that is, `V / length(V)`.

```
vector faceforward (vector N, I, Nref)
vector faceforward (vector N, I)
```

If `Nref . I` < 0, returns `N`; otherwise, returns `-N`. For the version with only two arguments, `Nref` is implicitly `Ng`, the true surface normal. The point of these routines is to return a version of `N` that faces towards the camera—in the direction "opposite" of `I`.

To further clarify the situation, here is the implementation of faceforward expressed in Shading Language:

```
vector faceforward (vector N, I, Nref)
{
    return (I.Nref > 0) ? -N : N;
}

vector faceforward (vector N, I)
{
    extern normal Ng;
    return faceforward (N, I, Ng);
}
```

`vector reflect (vector I, N)`

For incident vector I and surface orientation N, returns the reflection direction R = I - 2*(N.I)*N. Note that N must be normalized (unit length) for this formula to work properly.

`vector refract (vector I, N; float eta)`

For incident vector I and surface orientation N, returns the refraction direction using Snell's law. The eta parameter is the ratio of the index of refraction of the volume containing I divided by the index of refraction of the volume being entered.

```
point transform (string tospacename; point p_current)
vector vtransform (string tospacename; vector v_current)
normal ntransform (string tospacename; normal n_current)
```

Transform a point, vector, or normal (assumed to be in "current" space) into the tospacename coordinate system.

```
point transform (string fromspacename, tospacename; point pfrom)
vector vtransform (string fromspacename, tospacename; vector vfrom)
normal ntransform (string fromspacename, tospacename; normal nfrom)
```

Transform a point, vector, or normal (assumed to be represented by its "fromspace" coordinates) into the tospacename coordinate system.

```
point transform (matrix tospace; point p_current)
vector vtransform (matrix tospace; vector v_current)
normal ntransform (matrix tospace; normal n_current)

point transform (string fromspacename; matrix tospace; point pfrom)
```

```
vector vtransform (string fromspacename; matrix tospace; vector vfrom)
normal ntransform (string fromspacename; matrix tospace; normal nfrom)
```

These routines work just like the ones that use the space names but instead use transformation matrices to specify the spaces to transform into.

```
point rotate (point Q; float angle; point P0, P1)
```

Returns the point computed by rotating point `Q` by `angle` radians about the axis that passes from point `P0` to `P1`.

String Functions

```
void printf (string template, ...)
string format (string template, ...)
```

Much as in C, `printf` takes a template string and an argument list. Where the format string contains the characters `%f`, `%c`, `%p`, `%m`, and `%s`, `printf` will substitute arguments, in order, from the argument list (assuming that the arguments' types are float, color, point-like, matrix, and string, respectively).

The `format` function, like `printf`, takes a template and an argument list. But `format` returns the assembled, formatted string rather than printing it.

```
string concat (string s1, ..., sN)
```

Concatenates a list of strings, returning the aggregate string.

```
float match (string pattern, subject)
```

Does a string pattern match on `subject`. Returns 1 if the `pattern` exists anywhere within `subject` and 0 if the `pattern` does not exist within `subject`. The `pattern` can be a standard Unix expression. Note that the pattern does not need to start in the first character of the `subject` string, unless the pattern begins with the ^ (beginning of string) character.

Matrix Functions

```
float determinant (matrix m)
```

Returns the determinant of matrix `m`.

```
matrix translate (matrix m; point t)
matrix rotate (matrix m; float angle; vector axis)
matrix scale (uniform matrix m; uniform point t)
```

Return a matrix that is the result of appending simple transformations onto the matrix `m`. In each case, all arguments to these functions must be uniform. These functions are similar to the RIB `Translate`, `Rotate`, and `Scale` commands, except that the rotation angle in `rotate()` is in radians, not in degrees as with the RIB `Rotate`. There are no perspective or skewing functions.

7.6 Writing SL Functions

Even though Shading Language provides many useful functions, you will probably want to write your own, just as you would in any other programming language. Defining your own functions is similar to doing it in C:

```
returntype functionname ( params )
{
    :  do some computations
    :
    return return_value ;
}
```

However, in many ways SL function definitions are not quite like C:

- Only one `return` statement is allowed per function. The exception to this rule is for `void` functions, which have no return type and thus have no `return` statement.
- All function parameters are passed by reference.
- You may not compile functions separately from the body of your shader. The functions must be declared prior to use and in the same compilation pass as the rest of your shader (though you may place them in a separate file and use the `#include` mechanism).[3]

Valid return types for functions are the same as variable declarations: `float`, `color`, `point`, `vector`, `normal`, `matrix`, `string`. You may declare a function as `void`, indicating that it does not return a value. You may not have a function that returns an array.

In C, parameters are passed by value, which means that the function has a private copy that can be modified without affecting the variable specified when the function was called. SL function parameters are passed by *reference*, which means that if you modify it, it will actually change the original variable that was passed. However, any parameters you want to modify must be declared with the `output` keyword, as in the following example:

```
float myfunc (float f;          /* you can't assign to f */
              output float g;) /* but you can assign to g */
```

In the SL compilers of both *PRMan* and *BMRT*, functions are expanded in-line, not compiled separately and called as subroutines. This means that there is no overhead associated with the call sequence. The downside is increased size of compiled code and the lack of support for recursion.

[3] It's possible that other renderers will allow separate compilation, but as of the time of this writing, both *PRMan* and *BMRT* require functions to be compiled at the same time as the shader body.

Shading Language functions obey standard variable lexical scope rules. Functions may be declared outside the scope of the shader itself, as you do in C. By default, SL functions may only access their own local variables and parameters. However, this can be extended by use of an `extern` declaration—global variables may be accessed by functions if they are accessed as `extern` variables. Newer SL compilers also support *local* functions defined inside shaders or other functions—that is, defined anyplace where a local variable might be declared. In the case of local functions, variables declared in the outer lexical scope may be accessed if they are redeclared using the `extern` keyword. Following is an example:

```
float x, y;

float myfunc (float f)
{
    float x;          /* local hides the one in the outer scope */
    extern float y; /* refers to the y in the outer scope */
    extern point P; /* refers to the global P */
    ...
}
```

Further Reading

More formal documentation on the RenderMan Shading Language can be found in the *RenderMan Interface Specification* (Pixar, 1989) as well as Upstill (1990) and Hanrahan and Lawson (1990).

Discussions of color spaces can be found in Foley, van Dam, Feiner, and Hughes (1990), Hall (1989), or Glassner (1995).