Open Shading Language 0.1

Language Specification

(CONFIDENTIAL — draft in progress)

Editor: Larry Gritz
lg@imageworks.com

Date: 16 November, 2009

Contents

1 Introduction

Later there will be some words of wisdom here.

When you see text in this style, it's an annotation. These annotations will not be in the final draft of this document. They are notes to the readers of early drafts, sometimes questions, sometimes guideposts to uncertain areas, sometimes explanations of what is to come but that has not yet been fully fleshed out.

Short annotations will sometimes spring up anywhere, but there's also a full chapter of discussion of design choices (Chapter 10). Opinionated readers should pay particular attention to this chapter, as these big decisions will be the hardest to change later.

2 The Big Picture

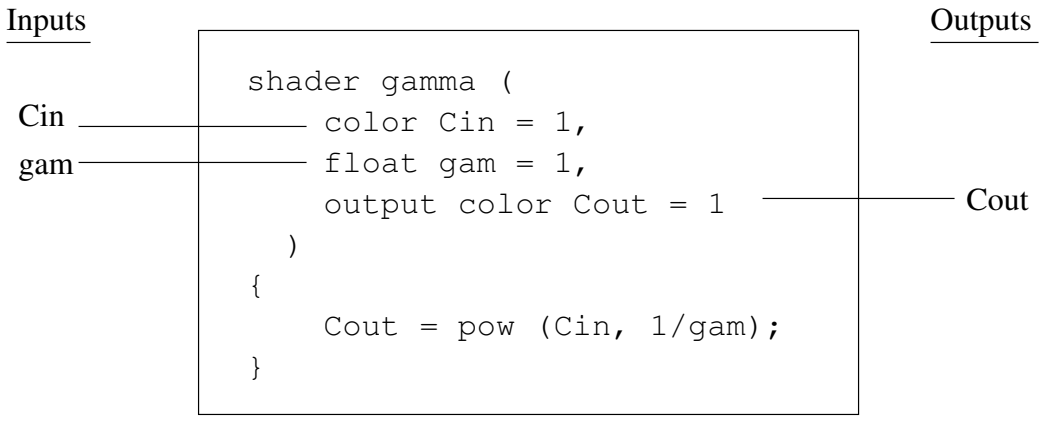
This chapter attempts to lay out the major concepts of Open Shading Language, define key nomenclature, and sketch out how individual shaders fit together in the context of a renderer as a whole.

Other than the background material of this chapter, the rest of this specification deals strictly with the language itself. There will, in the future, be separate (shorter) documents explaining in detail the use of the language compiler, the renderer-side issues, the library and APIs for how a renderer actually causes shaders to be evaluated, and so on.

A shader is code that performs a discrete task

A shader is a program, with inputs and outputs, that performs a specific task when rendering a scene, such as determining the appearance behavior of a material or light. The program code is written in Open Shading Language, the specification of which comprises this document.

For example, here is a simple `gamma` shader that performs simple gamma correction on is `Cin` input, storing the result in its output `Cout`:



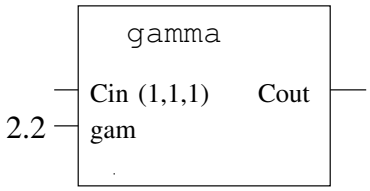
The shader's inputs and outputs are called *shader parameters*. Parameters have default values, specified in the shader code, but may also be given new values by the renderer at runtime.

Shader instances

A particular shader may be used many times in a scene, on different objects or as different layers in a shader group. Each separate use of a shader is called a *shader instance*. Although all

instances of a shader are comprised of the same program code, each instance may override any or all of its default parameter values with its own set of *instance values*.

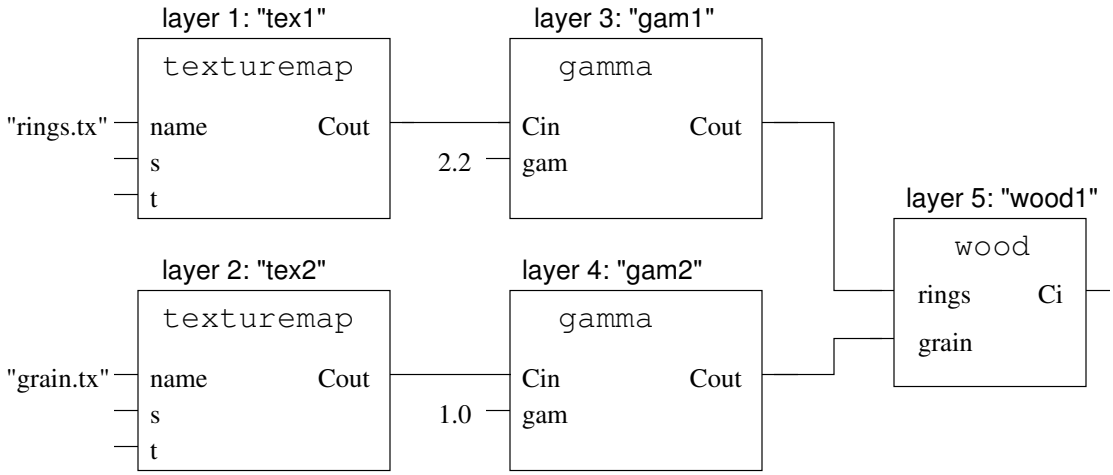
Below is a schematic showing a `gamma` instance with the `gam` parameter overridden with an instance-specific value of `2.2`.



Shader groups and layers

A *shader group* is an ordered sequence of individual shaders called *layers* that are executed in turn. Output parameters of an earlier-executed layer may be *connected* to an input parameter of a later-executed layer. This connected network of layers is sometimes called a *shader network* or a *shader DAG* (directed acyclic graph). Of course, it is fine for the shader group to consist of a single shader layer.

Below is a schematic showing how several shader instances may be connected to form a shader group.



And here is sample pseudo-code shows how the above network may be assembled using an API in the renderer[1]:

```
ShaderGroupBegin ()
Shader ("texturemap",               /* shader name */
        "tex1",                     /* layer name */
        "string name", "rings.tx")  /* instance variable */
Shader ("texturemap", "tex2", "string name", "grain.tx")
```

[1]This document does not dictate a specific renderer API for declaring shader instances, groups, and connections; the code above is just an example of how it might be done.

```
Shader ("gamma", "gam1", "float gam", 2.2)
Shader ("gamma", "gam2", "float gam", 1)
Shader ("wood", "wood1")
ConnectShaders ("tex1",     /* layer name A */
                "Cout",     /* an output parameter of A */
                "gam1",     /* layer name B */
                "Cin")      /* Connect this layer of B to A's Cout */
ConnectShaders ("tex2", "Cout", "gam2", "Cin")
ConnectShaders ("gam1", "Cout", "wood1", "rings")
ConnectShaders ("gam2", "Cout", "wood1", "grain")
ShaderGroupEnd ()
```

Geometric primitives

The *scene* consists of primarily of geometric primitives, light sources, and cameras.

Geometric primitives are shapes such as NURBS, subdivision surfaces, polygons, and curves. The exact set of supported primitives may vary from renderer to renderer.

Each geometric primitive carries around a set of named *primitive variables*. Nearly all shape types will have, among their primitive variables, control point positions that, when interpolated, actually designate the shape. Some shapes will also allow the specification of normals or other shape-specific data. Arbitrary user data may also be attached to a shape as primitive variables. Primitive variables may be interpolated in a variety of ways: one constant value per primitive, one constant value per face, or per-vertex values that are interpolated across faces in various ways.

If a shader input parameter's name and type match the name and type of a primitive variable on the object (and that input parameters is not already explicitly connected to another layer's output), the interpolated primitive variable will override the instance value or default.

Attribute state and shader assignments

Every geometric primitive has a collection of *attributes* (sometimes called the *graphics state*) that includes its transformation matrix, the list of which lights illuminate it, whether it is one-sided or two-sided, shader assignments, etc. There may also be a long list of renderer-specific or user-designated attributes associated with each object. A particular attribute state may be shared among many geometric primitives.

The attribute state also includes shader assignments — the shaders or shader groups for each of several *shader uses*, such as surface shaders that designate how light reflects from each point on the shape, displacement shaders that can add fine detail to the shape on a point-by-point basis, volume shaders that describe how light is scattered within a region of space, and light shaders that describe how light is emitted from a light source. A particular renderer may have additional shader types that it supports.

Shader execution state: parameter binding and global variables

When the body of code of an individual shader is about to execute, all its parameters are *bound* — that is, take on specific values (from connections from other layers, interpolated primitive variables, instance values, or defaults, in that order).

Certain state about the position on the surface where the shading is being run is stored in so-called *global variables*. This includes such useful data as the 3D coordinates of the point being shaded, the surface normal and tangents at that point, etc.

Additionally, the shader may query other information about other elements of the attribute state attached to the primitive, and information about the renderer as a whole (rendering options, etc.).

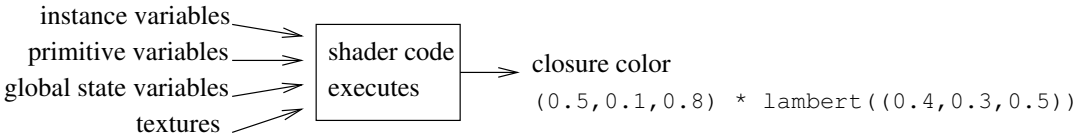
Surface and volume shaders compute closures

Surface shaders (and volume shaders) do not by themselves compute the final color of light eminating from the surface (or along a volume). Rather, the compute a *closure*, which is a symbolic representation describing the appearance of the surface, that may be more fully evaluated later. This is in effect a parameterized formula, in which some inputs have definite numeric values, but others may depend on quantities not yet known (such as the direction from which the surface is being viewed, and the amount of light from each source that is arriving at the surface).

For example, a surface shader may compute its result like this:

```
color paint = texture ("file.tx", u, v);
Ci = paint * lambert (N);
```

In this example, the variable `paint` will take on a specific numeric value (by looking up from a texture map). But the `lambert()` function returns a `closure color`, not a definite numeric `color`. The output variable `Ci` that represents the appearance of the surface is also a `closure color`, whose numeric value is not known yet, except that it will be the product of `paint` and a Lambertian reflectance.



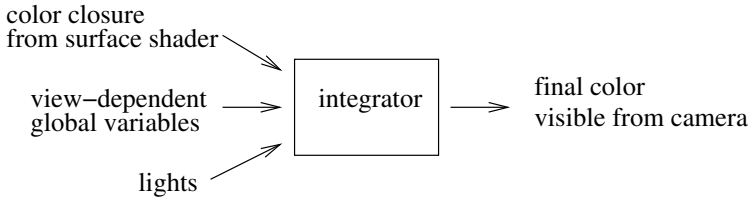
The closures output by surface and volume shaders can do a number of interesting things that a mere number cannot:

- Evaluate: given input and output light directions, compute the proportion of light propagating from input to output.
- Sample: given just an input (or output) direction, choose a scattering direction with a probability distribution that is proportional to the amount of light that will end up going in various directions.
- Integrate: given all lights and a view direction, compute the total amount of light leaving the surface in the view direction.
- Recompute: given changes only to lights (or only to one light), recompute the integrated result without recomputing other lights or any of the calculations that went into assembling constants in the closure (such as texture lookups, noise functions, etc.).

At present, we are assuming that the primitive closure functions (such as `diffuse`, `ward`, `cooktorrance`, etc.) are all built into the renderer, or implemented as renderer plugins. At a later time, possibly in a later draft or maybe not until a truely later version of the spec, we will fully spec it out so that closure primitive functions may be written in Open Shading Language. But I fear that if we do it too soon, we'll screw it up. But, yes, the eventual goal is for you to be able to write these primitive functions in the language itself.

Integrators

The renderer contains a number of *integrators* (selectable via the renderer's API) which will combine the color closures computed by surfaces and volumes with the light sources and view-dependent information, to yield the amount of light visible to the camera.



At present, this document is written as if the integrators are built into the renderer itself (or implemented as renderer plug-ins). At a later time, we may consider making it possible for integrators themselves to be written in Open Shading Language.

Units

You can tell the renderer (through a global option) what units the scene is using for distance and time. Then the shader has a built-in function called `transformu()` that works a lot like `transform()`, but instead of converting between coordinate systems, it converts among units. For example,

```
displacement bumpy (float bumpdist = 1,
                    string bumpunits = "cm")
{
    // convert bumpdist to common units
    float spacing = transformu (bumpunits, "common", bumpdist);
    float n = noise (P / spacing);
    displace (n);
}
```

So you can write a shader to achieve some effect in real world units, and that shader is totally reusable on another show that used different modeling units.

It knows all the standard names like `"cm"`, `"in"`, `"km"`", etc., and can convert among any of those, as well as between named coordinate systems. For example,

```
float x = transformu ("object", "mm", 1);
```

now `x` is the number of millimeters per unit of `"object"` space on that primitive.

3 Lexical structure

3.1 Characters

Source code for Open Shading Language consists of ASCII or UTF-8 characters.

The characters for space, tab, carriage return, and linefeed are collectively referred to as *whitespace*. Whitespace characters delimit identifiers, keywords, or other symbols, but other than that have no syntactic meaning. Multiple whitespace characters in a row are equivalent to a single whitespace character.

Source code may be split into multiple lines, separated by end-of-line markers (carriage return and/or linefeed). Lines may be of any length and end-of-line markers carry no significant difference from other whitespace, except that they terminate `//` comments and delimit preprocessor directives.

3.2 Identifiers

Identifiers are the names of variables, parameters, functions, and shaders. In Open Shading Language, identifiers consist of one or more characters. The first character may be a letter (`A-Z` or `a-z`) or underscore (_), and subsequent characters may be letters, underscore, or numerals (`0-9`). Examples of valid and invalid identifiers are:

```
opacity       // valid
Long_name42   // valid - letters, underscores, numbers are ok
_foo          // valid - ok to start with an unberscore

2smart        // invalid - starts with a numeral
bigbuck$      // invalid - $ is an illegal character
```

3.3 Comments

Comments are text that are for the human reader of programs, and are ignored entirely by the Open Shading Language compiler. Just like in C++, there are two ways to designate comments in Open Shading Language:

1. Any text enclosed by `/*` and `*/` will be considered a comment, even if the comment spans several lines.

```
/* this is a comment */

/* this is also
   a comment, spanning
   several lines */
```

2. Any text following //, up to the end of the current line, will be considered a comment.

```
// This is a comment
a = 3;   // another comment
```

3.4 Keywords and reserved words

There are two sets of names that you may not use as identifiers: keywords and reserved words.

The following are *keywords* that have special meaning in Open Shading Language:

```
break closure color continue do else emit float for if illuminance
illuminate int matrix normal output point public return string struct
trace vector void while
```

The following are *reserved words* that currently have no special meaning in Open Shading Language, but we reserve them for possible future use, or becuase they are confusingly similar to keywords in related programming languages:

```
bool case catch char class const delete default double enum extern
false friend goto inline long new operator private protected short
signed sizeof static switch template this throw true try typedef uniform
union unsigned varying virtual volatile
```

3.5 Preprocessor

Shader source code is passed through a standard C preprocessor as a first step in parsing.

Preprocessor directives are designated by a hash mark (#) as the first character on a line, followed by a preprocessor directive name. Whitespace may optionally appear between the hash and the directive name.

Open Shading Language compilers support the full complement of C/C++ preprocessing directives, including:

```
#define
#undef
#if
#ifdef
#ifndef
#elif
#else
#endif
#include
```

4 Gross syntax, shader types, parameters, functions

The overall structure of a shader is as follows:

optional-function-or-struct-declarations

shader-type shader-name (optional-parameters)
{
    statements
}

Note that *statements* may include function or structure definitions, local variable declarations, or public methods, as well as ordinary execution instructions (such as assignments, etc.).

4.1 Shader types

Shader types include the following: `surface`, `displacement`, `light`, `volume`, and generic `shader`. Some operations may only be performed from within certain types of shaders (e.g., one may only call `displace()` or alter `P` in a displacement shader), and some global variables may only be accessed from within certain types of shaders (e.g., `dPdu` is not defined inside a volume shader).

Following are brief descriptions of the basic types of shaders:

`surface` shaders

Surface shaders determine the basic material properties of a surface and how it reacts to light. They are responsible for computing a `closure color` that describes the material, and optionally setting other user-defined output variables. They may not alter the position of the surface.

Surface shaders are written as if they describe the behavior of a single point on the primitive, and the renderer will choose the positions surface at which the shader must be evaluated.

Surface shaders also are used to describe emissive objects, i.e., light sources. OSL does not need a separate shader type to describe lights.

`displacement` shaders

Displacement shaders alter the position and shading normal (or, optionally, just the shading normal) to make a piece of geometry appear deformed, wrinkled, or bumpy. They are the only kind of shader that is allowed to alter a primitive's position.

`volume` shaders

Volume shaders describe how a participating medium (air, smoke, glass, etc.) reacts to light and affects the appearance of objects on the other side of the medium. They are similar to `surface` shaders, except that they may be called from positions that do not lie upon (and are not necessarily associated with) any particular primitive.

`shader` generic shaders

Generic shaders are used for utility code, generic routines that may be called as individual layers in a shader group. Generic shaders need not specify a shader type, and therefore may be reused from inside surface, displacement, or volume shader groups. But as a result, they may not contain any functionality that cannot be performed from inside all shader types (for example, they may not alter `P`, which can only be done from within a displacement shader).

4.2 Shader parameters

An individual shader has (optionally) many *parameters* whose values may be set in a number of ways so that a single shader may have different behaviors or appearances when used on different objects.

4.2.1 Shader parameter syntax

Shader parameters are specified in the shader declaration, in parentheses after the shader's name. This is much like the parameters to a ukrainianfunction (or a function in C or similar languages), except that shader parameters must have an *initializer*, giving a default value for the parameter. Shader parameter default initializers may be expressions (i.e., may be computed rather than restricted to numeric constants), and are evalutated in the order that the parameters are declared, and may include references to previously-declared parameters. Formally, the grammar for a simple parameter declaration looks like this:

type parametername = default-expression

where *type* is one of the data types described in Chapter 5, *parametername* is the name of the parameter, and *default-expression* is a valid expression (see Section 6.2). Multiple parameters are simply separated by parentheses:

type1 parameter1 = expr1 , type2 parameter2 = expr2 , ...

Fixed-length, one-dimensional array parameters are declared as follows:

type parametername [*array-length*] = { *expr0 , expr1 ...* }

where *array-length* is a positive integer constant giving the length of the array, and the initializer is a series of initializing expressions listed between curly braces. The first initializing expression provides the initializer for the first element of the array, the second expression provides the initializer for the second element of the array, and so on. If the number of initializing expressions is less than the length of the array, any additional array elements will have undefined values.

Arrays may also be declared without a set length:

type parametername `[ ]` = { *expr0* , *expr1* ... }

where no array length is found between the square brackets. This indicates that the array's length will be determined based on whatever is passed in — a connection from the output of another shader in the group (take on the length of that output), an instance value (take on the length specified by the declaration of the instance value), or a primitive variable (length determined by its declaration on the primitive). If no instance value, primitive value, or connection is supplied, then the number of initializing expressions will determine the length, as well as the default values, of the array.

Structure parameters are also straightforward to declare:

structure-type parametername = { *expr0* , *expr1* ... }

where *structure-type* is the name of a previously-declared `struct` type, and the *expr* initializers correspond to each respective field within the structure. An initializer of appropriate type is required for every field of the structure.

4.2.2 Shader output parameters

Shader parameters are, by default, read-only in the body of the shader. However, special *output parameters* may be altered by execution of the shader. Parameters may be designated outputs by use of the `output` keyword immediately prior to the type declaration of the parameter:

`output` *type parametername* = *expr*

(Output parameters may be arrays and structures, but we will omit spelling out the obvious syntax here.)

Output parameters may be connected to inputs of later-run shader layers in the shader group, may be queried by later-run shaders in the group via message passing (i.e., `getmessage()` calls), or used by the renderer as an output image channel (in a manner described through the renderer's API).

4.2.3 Shader parameter example

Here is an example of a shader declaration, with several parameters:

```
surface wood (
           /* Simple params with constant initializers */
               float Kd = 0.5,
               color woodcolor = color (.7, .5, .3),
               string texturename = "wood.tx",
           /* Computed from an earlier parameter */
```

```
                color ringcolor = 0.25 * woodcolor,
            /* Fixed-length array */
                color paintcolors[3] = { color(0,.25,0.7), color(1,1,1),
                                         color(0.75,0.5,0.2) },
            /* variable-length array */
                int pattern[] = { 2, 4, 2, 1 },
            /* output parameter */
                output color Cunlit = 0
              )
{
   ...
}
```

4.2.4 How shader parameters get their values

Shader parameters get their values in the following manner, in order of decreasing priority:

- If the parameter has been designated by the renderer to be connected to an output parameter of a previously-executed shader layer within the shader group, that is the value it will get.
- If the parameter matches the name and type of a per-primitive, per-face, or per-vertex *primitive variable* on the particular piece of geometry being shaded, the paramter's value will be computed by interpolating the primitive variable for each position that must be shaded.
- If there is no connection or primitive variable, the parameter may will take on an *instance value*, if that parameter was given an explicit per-instance value at the time that the renderer referenced the shader (associating it with an object or set of objects).
- If none of these overrides is present, the parameter's value will be determined by executing the parameter initialization code in the shader.

This triage is performed per parameter, in order of declaration. So, for example, in the code sample above where the default value for `ringcolor` is a scaled version of `woodcolor`, this relationship would hold whether `woodcolor` was the default, an instance value, an interpolated primitive value, or was connected to another layer's output. Unless `ringcolor` itself was given an instance, primitive, or connection value, in which case that's what would be used.

4.3 Shader metadata

A shader may optionally include *metadata* (data *about* the shader, as opposed to data *used by* the shader). Metadata may be used to annotate the shader or any of its individual parameters with additional hints or information that will be compiled into the shader and may be queried by applications. A common use of metadata is to specify user interface hints about shader parameters — for example, that a particular parameter should only take on integer values, should have an on/off checkbox, is intended to be a filename, etc.

Metadata is specified inside double brackets `[[` and `]]` enclosing a comma-separated list of metadata items. Each metadatum looks like a parameter declaration — having a data type, name, and initializer. However, metadata may only be simple types (not arrays or closures) and their value initializers must be numeric or string constants (not computed expression).

Metadata about the shader as a whole is placed between the shader name and the parameter list. Metadata about shader parameters are placed immediately after the parameter's initializing expression, but before the comma or closing parentheses that terminates the parameter description.

Below is an example shader declaration showing the use of shader and parameter metadata:

```
surface wood
            [[ string description = "Realistic wood shader" ]]
    (
       float Kd = 0.5
           [[ string description = "Diffuse reflectivity",
              float UImin = 0, float UImax = 1 ]] ,
       color woodcolor = color (.7, .5, .3)
           [[ string description = "Base color of the wood" ]],
       color ringcolor = 0.25 * woodcolor
           [[ string description = "Color of the dark rings" ]],
       string texturename = "wood.tx"
           [[ string description = "Texture map for the grain",
              string UItype = "filename" ]]
    )
{
   ...
}
```

The metadata are not semantically meaningful; that is, the metadata does not affect the actual execution of the shader. Most metadata exist only to be embedded in the compiled shader and able to be queried by other applications, such as to construct user interfaces for shader assignment that allow usage tips, appropriate kinds of widgets for setting each parameter, etc.

The choice of metadata and their meaning is completely up to the shader writer and/or modeling system. However, we propose some conventions below. These conventions are not intended to be comprehensive, nor to meet all your needs — merely to establish a common nomenclature for the most common metadata uses.

`string description`

Describes the purpose and use of the shader or parameter.

`string URL`

Provides a URL for full documentation of the shader or parameter.

`string units`

NEW!

Gives the assumed units, if any, for the parameter (e.g., `"cm"`, `"sec"`, `"degrees"`). The compiler or renderer may issue a warning if it detects that this assumption is being violated (for example, the compiler can warn if a `"degrees"` variable is passed as the argument to `cos`).

NEW!

`int normalized`

For a `vector` or `normal`, a nonzero value is a a hint that the shader expects the vector to be unit length. The renderer may use this hint to print a warning if the parameter or value is not of unit length, and the compiler may warn if a `"normalized"` variable is assigned a non-unit value.

`string UIlabel`

A short label to be displayed in the UI for this parameter. If not present, the parameter name itself should be used as the widget label.

`float UImin`
`float UImax`

Minimum and maximum values for a parameter that should never be exceeded. If either the minimum or maximum is not present, valid values for the parameter are assumed to be unbounded in one or both directions. (For an `int` paramter, the `UImin` and/or `UImax` should also be declared as an `int`.)

`float UIsoftmin`
`float UIsoftmax`

"Soft" minimum and maximum value for a `float` parameter, that is, a default range for presentation in the UI, but which may be overridden by the user. (For an `int` paramter, the `UIsoftmin` and/or `UIsoftmax` should also be declared as an `int`.)

`float UIstep`

The suggested step size for incrementing or decrementing the value (within the appropriate min/max range).

`string UIenabler`

The name of another parameter that, only if nonzero (or the non-empty string) unlocks adjustment of this parameter. For example, a parameter called `"Kr"` (reflectivity) may be en enabler for the `"reflectionblur"` paramter; zero reflectivity would gray-out the blur controls, which would be ignored if there were no reflection.

`float UIseparator`

If nonzero, hints that the application's GUI should draw a visual separator immediately before this parameter.

`string UIgroupbegin`
`string UIgroupend`

Denote the first and last parameters of a set of related paramters, as a hint that an application may wish to label or visually separate a collection of paramters. The string value should be the name or brief description of the group.

`string UItype`

Provides a hint about exactly what kind of UI widget is appropriate for the given paramter (if not the obvious default, such as a slider for a `float` paramter, or a color picker for a `color` parameter). Recommended values include:

`"bool"` Indicates that a `float` or `int` parameter should only take on a 0 or 1 value, and therefore a checkbox may be a more appropriate UI widget.

`"string"` Indicates that the appropriate UI widget is a text input box (default for string parameters). A smart UI may also allow a file selection dialog, since most strings (but not all) are used for texture names.

`"texture"` Indicates a string variable whose value will be used as the name of a texture map. An appropriate GUI widget might be a file selection dialog.

`"environment"` Indicates a string variable whose value may either be an environment map or the name of a geometry set. An appropriate GUI widget might be a combination file selection and pull-down menu of known geometry set names.

`"shadow"` Indicates a string variable whose value may either be a shadow map or the name of a geometry set.

`"geometryset"` Indicates a string variable whose value must be a geometry set. An appropriate GUI widget may be a pull-down menu of known geometry sets.

`"coordsys"` Indicates a string variable whose value will be used as a coordinate system name. An appropriate GUI widget may be a pull-down menu of known coordinate systems.

`"enum:label0,label1,..."` Indicates that the appropriate UI widget is a pulldown menu that selects among the strings `"label0"`, `"label1"`, and so on. If the shader parameter is a `string`, the selcted label should be passed as the shader parameter value. If the shader parameter is an `int`, the selected label should indicate that the value passed should be 0, 1, 2, ..., the index of the selected label. Labels may be associated with `float` or `int` values other than their indices using the following syntax:

```
float frequency = 0.5
    [[ string UItype = "enum:low=0.2,medium=0.5,high=0.9" ]]
int pattern = 0
    [[ string UItype = "enum:oak=0,elm=1,walnut=2" ]]
```

The use of metadata is entirely optional on the part of the shader writer, and any application that queries shader metadata is free to honor or ignore any metadata it finds.

4.4 Functions

You may define functions much like in C or C++.

return-type function-name (*optional-parameters*)
{
    statements
}

Parameters to functions are similar to shader parameters, except that they do not permit initializers. A function call must pass values for all formal parameters. Function parameters in Open Shading Language are all *passed by reference*, and are read-only within the body of the function unless they are also designated as `output` (in the same manner as output shader parameters).

Like for shaders, statements inside functions may be actual executions (assignments, function call, etc.), local variable declarations (visible only from within the body of the function), or local function declarations (callable only from within the body of the function).

The return type may be any simple data type, a `struct`, or a `closure`. Functions may not return arrays. The return type may be `void`, indicating that the function does not return a value (and should not contain a `return` statement). A `return` statement inside the body of the function will halt execution of the fuction at that point, and designates the value that will be returned (if not a `void` function).

Functions may be *polymorphic*. That is, multiple functions may be defined to have the same name, as long as they have differently-typed parameters, so that when the function is called the list of arguments can disambiguate which version of the function is desired.

4.5 Public methods

Ordinary (non-public) functions inside a shader may be called only from within the shader; they do not generate entry points that the renderer is aware of.

A *public method* is a function that may be directly called by the renderer. Only top-level local functions of a shader — that is, declared within the braces that define the local scope of the shader, but not within any other such function — may be public methods. A function may be designated a public method by using the `public` keyword immediately before the function declaration:

shader-type shader-name (params)
{
    `public` return-type function-name (optional-parameters)
    {
        statements
    }

    ...
}

A given renderer will publish a list of public methods (names, arguments expected, and return value) that has particular meaning for that renderer. For example, a renderer may honor a public method

```
public float maxdisplacement ()
```

that computes and returns the maximum distance that a displacement shader will move any surface points.

At some later point, this spec will recommend several "standard" public methods that should be honored by most renderers.

5 Data types

Open Shading Language provides several built-in simple data types for performing computations inside your shader:

`int`	Integer data
`float`	Scalar floating-point data (numbers)

`point`
`vector`
`normal`
	Three-dimensional positions, directions, and surface orientations

`color`	Spectral reflectivities and light energy values
`matrix`	4×4 transformation matrices
`string`	Character strings (such as filenames)
`void`	Indicates functions that do not return a value

In addition, you may create arrays and structures (much like C), and Open Shading Language has a new type of data structure called a *closure*.

The remainder of this chapter will describe the simple and aggregate data types available in Open Shading Language.

5.1 `int`

The basic type for discrete numeric values is `int`. The size of the `int` type is renderer-dependent, but is guaranteed to be at least 32 bits.

Integer constants are constructed the same way as in C. The following are examples of `int` constants: `1`, `-32`, etc.

Unlike C, no unsigned, bool, char, short, or long types are supplied. This is to simplify the process of writing shaders (as well as implementing shading systems).

The following operators may be used with `int` values (in order of decreasing precedence, with each box holding operators of the same precedence):

operation	result	
`int` **`++`**	`int`	post-increment by 1
`int` **`--`**	`int`	post-decrement by 1
`++` `int`	`int`	pre-increment by 1
`--` `int`	`int`	pre-decrement by 1
`-` `int`	`int`	unary negation
`~` `int`	`int`	bitwise complement (1 and 0 bits flipped)
`!` `int`	`int`	boolean 'not' (1 if operand is zero, otherwise 0)
`int` **`*`** `int`	`int`	multiplication
`int` **`/`** `int`	`int`	division
`int` **`%`** `int`	`int`	modulus
`int` **`+`** `int`	`int`	addition
`int` **`-`** `int`	`int`	subtraction
`int` **`<<`** `int`	`int`	shift left
`int` **`>>`** `int`	`int`	shift right
`int` **`<`** `int`	`int`	1 if the first value is less than the second, else 0
`int` **`<=`** `int`	`int`	1 if the first value is less or equal to the second, else 0
`int` **`>`** `int`	`int`	1 if the first value is greater than the second, else 0
`int` **`>=`** `int`	`int`	1 if the first value is greater than or equal to the second, else 0
`int` **`==`** `int`	`int`	1 if the two values are equal, else 0
`int` **`!=`** `int`	`int`	1 if the two values are different, else 0
`int` **`&`** `int`	`int`	bitwise and
`int` **`^`** `int`	`int`	bitwise exclusive or
`int` **`|`** `int`	`int`	bitwise or
`int` **`&&`** `int`	`int`	boolean and (1 if both operands are nonzero, otherwise 0)
`int` **`||`** `int`	`int`	boolean or (1 if either operand is nonzero, otherwise 0)

5.2 `float`

The basic type for scalar floating-point numeric values is `float`. The size of the `float` type is renderer-dependent, but is guaranteed to be at least IEEE 32-bit float (the standard C `float` data type). Individual renderer implementations may choose to implement `float` with even more precision (such as using the C `double` as the underlying representation).

Floating-point constants are constructed the same way as in C. The following are examples of `float` constants: `1.0`, `2.48`, `-4.3e2`.

An `int` may be used in place of a `float` when used with any valid `float` operator. In such cases, the `int` will be promoted to a `float` and the resulting expression will be `float`. An `int` may also be passed to a function that expects a `float` parameters, with the `int` automatically promoted to `float`.

The following operators may be used with `float` values (in order of decreasing precedence, with each box holding operators of the same precedence):

operation	result	
`float` **`++`**	`float`	post-increment by 1
`float` **`--`**	`float`	post-decrement by 1
`++` `float`	`float`	pre-increment by 1
`--` `float`	`float`	pre-decrement by 1
`-` `float`	`float`	unary negation
`float` **`*`** `float`	`float`	multiplication
`float` **`/`** `float`	`float`	division
`float` **`+`** `float`	`float`	addition
`float` **`-`** `float`	`float`	subtraction
`float` **`<`** `float`	`int`	1 if the first value is less than the second, else 0
`float` **`<=`** `float`	`int`	1 if the first value is less or equal to the second, else 0
`float` **`>`** `float`	`int`	1 if the first value is greater than the second, else 0
`float` **`>=`** `float`	`int`	1 if the first value is greater than or equal to the second, else 0
`float` **`==`** `float`	`int`	1 if the two values are equal, else 0
`float` **`!=`** `float`	`int`	1 if the two values are different, else 0

5.3 color

The `color` type is used to represent 3-component (RGB) spectral reflectivities and light energies. You can assemble a color out of three floats, either representing an RGB triple or some other color space known to the renderer, as well as from a single float (replicated for all three channels). Following are some examples:

```
color (0, 0, 0)              // black
color ("rgb", .75, .5, .5)   // pinkish
color ("hsv", .2, .5, .63)   // specify in "hsv" space
color (0.5)                  // same as color (0.5, 0.5, 0.5)
```

All these expressions above 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. Table 5.1 lists the built-in color spaces.

Colors may be assigned another color or a `float` value (which sets all three components to the value). For example:

```
color C;
C = color (0, 0.3, 0.3);
C = 0.5;                    // same as C = color (0.5, 0.5, 0.5)
```

Table 5.1: 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.
`"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.

Colors can have their individual components examined and set using the `[]` array access notation. For example:

```
color C;
float g = C[1];   // get the green component
C[0] = 0.5;       // set the red component
```

Components 0, 1, and 2 are red, green, and blue, respectively. It is an error to access a color component with an index outside the [0...2] range.

The following operators may be used with `color` values (in order of decreasing precedence, with each box holding operators of the same precedence):

operation	result	
`color [ int]`	`float`	component access
`- color`	`color`	unary negation
`color * color`	`color`	component-wise multiplication
`color * float`	`color`	scaling
`float * color`	`color`	scaling
`color / color`	`color`	component-wise division
`color / float`	`color`	scaling
`float / color`	`color`	scaling
`color + color`	`color`	component-wise addition
`color - color`	`color`	component-wise subtraction
`color == color`	`int`	1 if the two values are equal, else 0
`color != color`	`int`	1 if the two values are different, else 0

All of the binary operators may combine a scalar value (`float` or `int`) with a `color`, treating the scalar if it were a `color` with three identical components.

5.4 Point-like types: `point`, `vector`, `normal`

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 than ordinary vectors, as we will describe below.

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.

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 `"common"` coordinate system. The `"common"` 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 `"common"` space .

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 `"common"` 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 `"common"` space. This type of constructor also can be used for vectors and normals.

The choice of `"common"` space is renderer-dependent, though will usually be equivalent to either `"camera"` space or `"world"` space.

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 a built-in `transform` function that allows you to transform points, vectors, and normals among different coordinate systems (see Section 7.2). Note, however, that Open Shading Language does not keep track of which point variables are in which coordinate systems. It is the responsibility of the shader programmer to keep track of this and ensure that, for example, lighting computations are performed using quantities in `"common"` space.

Several coordinate systems are predefined by name, listed in Table 5.2. Additionally, a renderer will probably allow for additional coordinate systems to be named in the scene description, and these names may also be referenced inside your shader to designate transformations.

Point types can have their individual components examined and set using the `[]` array access notation. For example:

```
point P;
float y = P[1];   // get the y component
P[0] = 0.5;       // set the x component
```

Table 5.2: Names of predeclared geometric spaces.

`"common"`	The coordinate system that all spatial values start out in and the one in which all lighting calculations are carried out. Note that the choice of `"common"` 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 local coordinate system active at the time that the shader was instanced.
`"world"`	The world coordinate system designated in the scene.
`"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 coordinate system of the camera's image plane (after perspective transformation, if any). Coordinate (0,0) of `"screen"` space is looking along the *z*-axis of `"camera"` space.
`"raster"`	2D pixel coordinates, with (0,0) as the upper-left corner of the image and (xres, yres) as the lower-right corner.
`"NDC"`	2D 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.

Components 0, 1, and 2 are *x*, *y*, and *z*, respectively. It is an error to access a point component with an index outside the [0...2] range.

The following operators may be used with point-like values (in order of decreasing precedence, with each box holding operators of the same precedence):

operation	result	
ptype `[ int]`	`float`	component access
− *ptype*	`vector`	component-wise unary negation
ptype ***** *ptype*	*ptype*	component-wise multiplication
`float` ***** *ptype*	*ptype*	scaling of all components
ptype ***** `float`	*ptype*	scaling of all components
ptype **/** *ptype*	*ptype*	component-wise division
ptype **/** `float`	*ptype*	division of all components
`float` **/** *ptype*	*ptype*	division by all components
ptype **+** *ptype*	*ptype*	component-wise addition
ptype **−** *ptype*	`vector`	component-wise subtraction
ptype **==** *ptype*	`int`	1 if the two values are equal, else 0
ptype **!=** *ptype*	`int`	1 if the two values are different, else 0

The generic *ptype* is listed in places where any of `point`, `vector`, or `normal` may be used.

All of the binary operators may combine a scalar value (`float` or `int`) with a point-like type, treating the scalar if it were point-like with three identical components.

5.5 matrix

Open 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).

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 "common"
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 $m_{0,0}...m_{3,3}$. Note that although we have used `"shader"` and `"world"` space in our examples, any named space is acceptable.

A matrix may also be constructed from the names of two coordinate systems, yielding the matrix that transforms coordinates from the first named space to the second named space:

```
matrix m = matrix ("object", "world");
```

The example returns the *object-to-world* transformation matrix.

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.

Individual compoents of a matrix variable may be set or accessed using array notation, for example,

```
matrix M;
float x = M[row][col];
M[row][col] = 1;
```

Valid component indices are integers on [0...3]. It is an error to access a matrix component with either a row or column outside this range.

The following operators may be used with matrices (in order of decreasing precedence, with each box holding operators of the same precedence):

operation	result	
`matrix [ int ] [ int ]`	`float`	component access (row, column)
`- matrix`	`matrix`	unary negation
`matrix * matrix`	`matrix`	matrix multiplication
`matrix * float`	`matrix`	component-wise scaling
`float * matrix`	`matrix`	component-wise scaling
`matrix / matrix`	`matrix`	multiply the first matrix by the *inverse* of the second
`matrix / float`	`matrix`	component-wise division
`float / matrix`	`matrix`	multiply the `float` by the *inverse* of the matrix
`matrix == matrix`	`int`	1 if the two values are equal, else 0
`matrix != matrix`	`int`	1 if the two values are different, else 0

5.6 `string`

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 compared using == and !=.

String constants are denoted by surrounding the characters with double quotes, as in `"I am a string literal"`. As in C programs, string literals may contain escape sequences such as `\n` (newline), `\r` (carriage return), `\t` (tab), `\"` (double quote), `\\` (backslash).

Two quote-quoted string literals that are separated only by whitespace (spaces, tabs, or newlines) will be automatically concatenated into a single string literal. In other words,

```
"foo"  "bar"
```

is exactly equivalent to `"foobar"`.

5.7 `void`

The `void` type is used to designate a function that does not return a value. No variable may have type `void`.

5.8 Arrays

Arrays of any of the basic types are supported, provided that they are 1D and statically sized, using the usual syntax for C-like languages:

```
float d[10];                        // Declare an uninitilized array
float c[3] = { 0.1, 0.2, 3.14 };    // Initialize the array

float f = c[1];                     // Access one element
```

The built-in function `arraylength()` returns the number of elements in an array. For example:

```
float c[3];
int clen = arraylength(c);        // should return 3
```

There are two circumstances when arrays do not need to have a declared length — an array parameter to a function, and a shader parameter that is an array. This is indicated by empty array brackets, as shown in the following example:

```
float sum (float x[])
{
    float s = 0;
    for (int i = 0;  i < arraylength(x);  ++i)
        s += x[i];
    return s;
}
```

5.9 Structures

Structures are used to group several fields of potentially different types into a single object that can be referred to by name. You may then use the structure type name to declare structure variables as you would for any of the built-in types. Structure elements are accessed using the 'dot' operator. The syntax for declaring and using structures is similar to C or C++:

```
struct ray {                    // Define a structure type
    point pos;
    vector dir;
};

ray r;                          // Declare a structure
r.pos = point (1, 0, 0);        // Assign to one field
point p = r.pos;                // Read from a structure field
```

It is permitted to have a structure field that is an array, as well as to have an array of structures. But it is not permitted for one structure to have a field that is another structure. For example:

```
struct A {
    color a;
    float b[4];       // struct may contain an array
};

A d[5];               // Array of structures
color e = d[0].a;     // Field of one element of array of struct
d[2].b[4] = 0.25;     // Element of a field of a struct in an array
```

Is this restriction (no "structure recursion") acceptable? It certainly makes the implementation of the language easier.

?

5.10 Closures

A `closure` is an expression or function call that will be stored, along with necessary contextual information, to be evaluated at a later time.

In general, the type "`closure` *gentype*" behaves exactly like a *gentype*, except that its numeric values may not be examined or used for the duration of the shader's execution. For example, a `closure color` behaves mostly like a color — you can multiply it by a scalar, assign it to a `closure color` variable, etc. — but you may not assign it to an ordinary `color` or examine its individual component's numeric values.

It is legal to assign `0` to a closure, which is understood to mean setting it to a *null closure* (even though in all other circumstances, assigning a `float` to a `closure` would not be allowed).

At present, the only type of `closure` supported by Open Shading Language is the `closure color`, and the only allowed operations are those that let you form a linear combination of `closure color`'s. Additional closure types and operations are reserved for future use.

Allowable operations on `closure color`'s include:

operation	result	
− `closure color`	`closure color`	unary negation
`color` ***** `closure color`	`closure color`	component-wise scaling
`closure color` ***** `color`	`closure color`	component-wise scaling
`float` ***** `closure color`	`closure color`	scaling
`closure color` ***** `float`	`closure color`	scaling
`closure color` **+** `closure color`	`closure color`	component-wise addition

6 Language Syntax

The body of a shader is a sequence of individual *statements*. This chapter describes the types of statements and control-flow patterns in Open Shading Language.

Statements in Open Shading Language include the following types of constructs:

- Scoped statements.
- Variable declarations.
- Expressions.
- Assignments.
- Control flow: `if`, `else`, `while`, `do`, `for`, `break`, `continue`
- Function declarations.

Scoping

Any place where it is legal to have a statement, it is legal to have multiple statements enclosed by curly braces { }. This is called a *scope*. Any variables or functions declared declared within a scope are only visible within that scope, and only may be used after their declaration. Variables or functions that are referenced will always resolve to the matching name in the innermost scope relative to its use. For example

```
float a = 1;      // Call this the "outer" 'a'
float b = 2;
{
    float a = 3;  // Call this the "inner" 'a'
    float c = 1;
    b = a;        // b gets 3, because a is resolved to the inner scope
}
b += c;           // ERROR -- c was only in the inner scope
```

6.1 Variable declarations and assignments

6.1.1 Variable declarations

The syntax for declaring a variable in Open Shading Language is:

type name
type name = *value*

where

- *type* is one of the basic data types, described earlier.
- *name* is the name of the variable you are declaring.
- If you wish to initialize your variable an initial value, you may immediately assign it a *value*, which may be any valid expression.

You may declare several variables of the same type in a single declaration by separating multiple variable names by commas:

type name1 , *name2* ...
type name1 [= *value1*] , *name2* [= *value2*] ...

Some examples of variable declarations are

```
float a;           // Declare; current value is undefined
float b = 1;       // Declare and assign a constant initializer
float c = a*b;     // Computed initializer
float d, e = 2, f; // Declare several variables of the same type
```

6.1.2 Arrays

Arrays are also supported, declared as follows:

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

Array variables in Open Shading Language must have a constant length (though function parameters and shader parameters may have undetermined length). Some examples of array variable declarations are:

```
float d[10];                        // Declare an uninitilized array
float c[3] = { 0.1, 0.2, 3.14 };    // Initialize the array
```

6.1.3 Structures

Structures are used to group several fields of potentially different types into a single object that can be referred to by name. The syntax for declaring a structure type is:

`struct` *structname* {
    type1 fieldname1 ;
    ...
    typeN fieldnameN ;

} ;

You may then use the structure type name to declare structure variables as you would for any of the built-in types:

structname variablename ;
structname variablename = { *initializer1* , ... *initializerN* } ;

If initializers are supplied, each field of the structure will be initialized with the initializer in the corresponding position, which is expected to be of the appropriate type.

Structure elements are accessed in the same way as other C-like languages, using the 'dot' operator:

variablename . *fieldname*

Examples of declaration and use of structures:

```
struct ray {
    point pos;
    vector dir;
};

ray r;   // Declare a structure
ray s = { point(0,0,0), vector(0,0,1) };  // declare and initialize
r.pos = point (1, 0, 0);  // Assign to one field
```

It is permitted to have a structure field that is an array, as well as to have an array of structures. But it is not permitted for one structure to have a field that is another structure.

Please refer to Section 5.9 for more information on using `struct`.

6.2 Expressions

The expressions available in Open Shading Language include the following:

- Constants: integer (e.g., `1`, `42`), floating-point (e.g. `1.0`, `3`, `-2.35e4`), or string literals (e.g., `"hello"`)
- point, vector, normal, or matrix constructors, for example:

  ```
  color (1, 0.75, 0.5)
  point ("object", 1, 2, 3)
  ```

  If all the arguments to a constructor are themselves constants, the constructed point is treated like a constant and has no runtime cost. That is, `color(1,2,3)` is treated as a single constant entity, not assembled bit by bit at runtime.

- Variable or parameter references

- An individual element of an array (using `[ ]`)
- An individual component of a `color`, `point`, `vector`, `normal` (using `[ ]`), or of a `matrix` (using `[][]`)
- prefix and postfix increment and decrement operators:

varref `++`	(post-increment)
varref `--`	(post-decrement)
`++` *varref*	(pre-increment)
`--` *varref*	(pre-decrement)

  The post-increment and post-decrement (e.g., `a++`) returns the old value, then increments or decrements the variable; the pre-increment and pre-decrement (`++a`) will first increment or decrement the variable, then return the new value.

- Unary and binary arithmetic operators on other expressions:

`-` *expr*	(negation)
`~` *expr*	(bitwise complement)
expr `*` *expr*	(multiplication)
expr `/` *expr*	(division)
expr `+` *expr*	(addition)
expr `-` *expr*	(subtraction)
expr `%` *expr*	(integer modulus)
expr `<<` *expr*	(integer shift left)
expr `>>` *expr*	(integer shift right)
expr `&` *expr*	(bitwise and)
expr `|` *expr*	(bitwise or)
expr `^` *expr*	(bitwise exclusive or)

  The operators `+`, `-`, `*`, `/`, and the unary `-` (negation) may be used on most of the numeric types. For multicomponent types (`color`, `point`, `vector`, `normal`, `matrix`), these operators combine their arguments on a component-by-component basis. 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.

  The integer and bit-wise operators `%`, `<<`, `>>`, `&`, `|`, `^`, and `~` may only be used with expressions of type `int`.

  For details on which operators are allowed, please consult the operator tables for each individual type in Chapter 5.

- Relational operators (all lower precedence than the arithmetic operators):

expr == *expr*	(equal to)
expr != *expr*	(not equal to)
expr < *expr*	(less then)
expr <= *expr*	(less than or equal to)
expr > *expr*	(greater than)
expr >= *expr*	(greater than or equal)

The == and != operators may be performed between any two values of equal type, and are performed component-by-component for multi-component types. The <, <=, >, >= may not be used to compare multi-component types.

An `int` expression may be compared to a `float` (and is treated as if they are both `float`). A `float` expression may be compared to a multi-component type (and is treated as a multi-component type as if constructed from a single float).

Relation comparisons produce Boolean (true/false) values. These are implemented as `int` values, 0 if false and 1 if true.

- Logical unary and binary operators:

  ! *expr*

  expr1 && *expr2*

  expr1 || *expr2*

  For the logical operators, numeric expressions (`int` or `float`) are considered *true* if nonzero, *false* if zero. Multi-component types (such as `color`) are considered *true* any component is nonzero, *false* all components are zero. Strings are considered *true* if they are nonempty, *false* if they are the empty string (`""`).

- another expression enclosed in parentheses: `( )`. Parentheses may be used to guarantee associativity of operations.

- Type casts, specified either by having the type name in parentheses in front of the value to cast (C-style typecasts) or the type name called as a constructor (C++-style type constructors):

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

vector (P)          /* Means the same thing */
point (f)
color (P)
```

  The three-component types (`color`, `point`, `vector`, `normal`) 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.

- function calls
- assignment expressions: same thing as `var = var OP expr`:

var = *expr*	(assign)
var += *expr*	(add)
var −= *expr*	(subtract)
var *= *expr*	(multiply)
var /= *expr*	(divide)
int-var &= *int-expr*	(bitwise and)
int-var |= *int-expr*	(bitwise or)
int-var ^= *int-expr*	(bitwise exclusive or)
int-var <<= *int-expr*	(integer shift left)
int-var >>= *int-expr*	(integer shift right)

  Note that the integer and bit-wise operators are only allowed with `int` variables and expressions. In general, `var OP= expr` is allowed only if `var = var OP expr` is allowed, and means exactly the same thing. Please consult the operator tables for each individual type in Chapter 5.

- ternary operator, just like C:

  condition `?` *expr1* `:` *expr2*

  This expression takes on the value of *expr1* if *condition* is true (nonzero), or *expr2* if *condition* is false (zero).

Please refer to Chapter 5, where the section describing each data type describes the full complement of operators that may be used with the type. Operator precedence in Open Shading Language is identical to that of C.

6.3 Control flow: `if, while, do, for`

Conditionals in Open Shading Language just like in C or 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) {
    x = s;
    y = 1;
} else {
    x = s+t;
}
```

The *condition* may be any valid expression, including:

- The result of any comparison operator (such as <, ==, etc.).
- Any numeric expression (`int`, `color`, `point`, `vector`, `normal`, `matrix`), which is considered "true" if nonzero and "false" if zero.
- Any string expression, which is considerd "true" if it is a nonempty string, "false" if it is the empty string (`""`).
- A logical combination of expressions using the operators `!` (not), `&&` (logical "and"), or `||` (logical "or"). Note that `&&` and `||` *short circuit* as in C, i.e. `A && B` will only evaluate B if A is true, and `A || B` will only evaluate B if A is false.

Repeated execution of statements for as long as a condition is true is possible with a `while` statement:

`while (` *condition* `)`
    statement

Or the test may happen after the body of the loop, with a `do/while` loop:

`do`
    statement
`while (` *condition* `);`

Also, `for` loops are also allowed:

`for (` *initialization-statement* `;` *condition* `;` *iteration-statement* `)`
    body

As in C++, a `for` loop's initialization may contain variable declarations and initializations, which are scoped locally to `for` loop itself. For example,

```
for (int i = 0;  i < 3;  ++i) {
    ...
}
```

As with `if` statements, loop conditions may be relations or numerical quantities (which are considered "true" if nonzero, "false" if zero), or strings (considered "true" if nonempty, "false" if the empty string `""`).

Inside the body of a loop, the `break` statement terminates the loop altogether, and the `continue` statement skip to the end of the body and proceeds to the next iteration of the loop.

6.4 Functions

Function calls are very similar to C and related programming languages:

functionname (*arg1* , ... , *argn*)

If the function returns a value (not `void`), you may use its value as an expression. It is fine to completely ignore the value of even a non-`void` function.

In Open Shading Language, all arguments are passed by reference. This generally will not be noticeably different from C-style "pass by value" semantics, except if you pass the same variable as two separate arguments to a function that modifies an argument's value.

Function definitions are described in detail in Section 4.4.

6.5 Global variables

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 shaders are listed in Table 6.1.

Variable	Description
`point` **`P`**	Position of the point you are shading. In a displacement shader, changing this variable displaces the surface.
`vector` **`I`**	The *incident* ray direction, pointing from the viewing position to the shading position `P`.
`normal` **`N`**	The surface "Shading" normal of the surface 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.True geometric normal of the surface at `P`.
`float` **`u`**, **`v`**	The 2D parametric coordinates of `P` (on the particular geometric primitive you are shading).
`vector` **`dPdu`**, **`dPdv`**	Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ tangent to the surface at `P`.
`float` **`time`**	Current shutter time for the point being shaded.
`float` **`dtime`**	The amount of time covered by this shading sample.
`vector` **`dPdtime`**	How the surface position `P` is moving per unit time.
`closure color` **`Ci`**	Incident radience — a closure representing the color of the light leaving the surface from `P` in the direction `-I`.

Table 6.1: Global variables available inside shaders.

Variable	surface	displacement	volume
`P`	R	RW	R
`I`	R		R
`N`	RW	RW	
`Ng`	R	R	
`dPdu`	R	R	
`dPdv`	R	R	
`u, v`	R	R	R
`time`	R	R	R
`dtime`	R	R	R
`dPdtime`	R	R	R
`Ci`	RW		RW

Table 6.2: Accessibility of variables by shader type

7 Standard Library Functions

7.1 Basic math functions

7.1.1 Mathematical constants

Open Shading Language defines several mathematical constants:

`M_PI`	π
`M_PI_2`	$\pi/2$
`M_PI_4`	$\pi/4$
`M_2_PI`	$2/\pi$
`M_2_SQRTPI`	$2/\sqrt{\pi}$
`M_E`	e
`M_LN2`	$\ln 2$
`M_LN10`	$\ln 10$
`M_LOG2E`	$\log_2 e$
`M_LOG10E`	$\log_{10} e$
`M_SQRT2`	$\sqrt{2}$
`M_SQRT1_2`	$\sqrt{1/2}$

7.1.2 Mathematical functions

Most of these functions operate on a generic *type* that my be any of `float`, `color`, `point`, `vector`, or `normal`. For `color` and `point`-like types, the computations are performed component-by-component (separately for x, y, and z).

type **radians** (*type* deg)
type **degrees** (*type* rad)

Convert degrees to radians or radians to degrees.

type **cos** (*type* x)
type **sin** (*type* x)
type **tan** (*type* x)

Computes the cosine, sine, and tangent of x (measured in radians).

type **acos** (*type* x)
type **asin** (*type* y)
type **atan** (*type* y_over_x)
type **atan2** (*type* y, *type* x)

Compute the principal value of the arc cosine, arc sine, and arc For `acos()` and `asin()`, the value of the argument will first be clamped to $[-1,1]$ to avoid invalid domain.

For `acos()`, the result will always be in the range of $[0,\pi]$, and for `asin()` and `atan()`, the result will always be in the range of $[-\pi/2,\pi/2]$. For `atan2()`, the signs of both arguments are used to determine the quadrant of the return value.

type **cosh** (*type* x)
type **sinh** (*type* x)
type **tanh** (*type* x)

Computes the hyperbolic cosine, sine, and tangent of x (measured in radians).

type **pow** (*type* x, *type* y)
type **pow** (*type* x, float y)

Computes x^y. This function will return 0 for "undefined" operations, such as `pow(-1,0.5)`.

type **exp** (*type* x)
type **exp2** (*type* x)
type **expm1** (*type* x)

Computes e^x, 2^x, and $e^x - 1$, respectively. Note that `expm1(x)` is accurate even for very small values of x.

type **log** (*type* x)
type **log2** (*type* x)
type **log10** (*type* x)
type **log** (*type* x, float b)

Computes the logarithm of x in base e, 2, 10, or arbitrary base b, respectively.

type **sqrt** (*type* x)
type **inversesqrt** (*type* x)

Computes $\sqrt{x}$ and $1/\sqrt{x}$. Returns 0 if $x < 0$.

float **hypot** (float x, float y)
float **hypot** (float x, float y, float z)

Computes $\sqrt{x^2+y^2}$ and $\sqrt{x^2+y^2+z^z}$, respectively.

type **abs** (*type* x)
type **fabs** (*type* x)

Absolute value of x. (The two functions are synonyms.)

Should we fully return to C conventions, with abs() being for integers and fabs being for float? Or should abs() simply be overloaded, like you'd do in C++ if we were starting from scratch today?

type **sign** (*type* x)

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

type **floor** (float x)
type **ceil** (*type* x)
type **round** (*type* x)
type **trunc** (*type* x)

Various rouinding methods: `floor` returns the highest integer less than or equal to x; `ceil` returns the lowest integer greater than or equal to x; `round` returns the the closest integer to x, in either direction; and `trunc` returns the integer part of x (equivalent to `floor` if $x > 0$ and `ceil` if $x < 0$).

type **mod** (*type* a, *type* b)
type **fmod** (*type* a, *type* b)

Computes $a - b * \text{floor}(a/b)$. The two functions are synonyms.

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

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

The `mix` function returns a linear blending : $x * (1 - \alpha) + y * (\alpha)$

int **isnan** (float x)
int **isinf** (float x)
int **isfinite** (float x)

The `isnan()` function returns 1 if x is a not-a-number (NaN) value, 0 otherwise. The `isinf()` function returns 1 if x is an infinite (Inf or –Inf) value, 0 otherwise. The `isfinite()` function returns 1 if x is an ordinary number (neither infinite nor NaN), 0 otherwise.

```
float erf (float x)
float erfc (float x)
```

The `erf()` function returns the error function $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$. The `erfc` returns the complementary error function `1-erf(x)` (useful in maintaining precision for large values of x).

7.2 Geometric functions

```
ptype ptype (float f)
ptype ptype (float x, float y, float z)
```

Constructs a point-like value (*ptype* may be any of `point`, `vector`, or `normal`) from individual `float` values. If constructed from a single `float`, the value will be replicated for x, y, and z.

```
ptype ptype (string space, f)
ptype ptype (string space, float x, float y, float z)
```

Constructs a point-like value (*ptype* may be any of `point`, `vector`, or `normal`) from individual `float` coordinates, relative to the named coordinate system. In other words,

```
point (space, x, y, z)
```

is equivalent to

```
transform (space, "common", point(x,y,z))
```

(And similarly for `vector`/`normal`.)

```
float dot (vector A, vector B)
```

Returns the inner product of the two vectors (or normals), i.e., $A \cdot B = A_x B_x + A_y B_y + A_z C_z$.

```
vector cross (vector A, vector B)
```

Returns the cross product of two vectors (or normals), i.e., $A \times B$.

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

Returns the length of a vector or normal.

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

Returns the distance between two points.

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

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

In RMan this was called 'ptlined'. Do we like that better, or 'distance'?

?

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

If `dot (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 Open Shading Language:

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

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

`vector` **`reflect`** `(vector I, vector 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, vector 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. The result is not necessarily normalized and a zero-length vector is returned in the case of total internal reflection. For reference, here is the equivalent Open Shading Language of the implementation:

```
vector refract (vector I, vector N, float eta)
{
    float IdotN = dot (I, N);
    float k = 1 - eta*eta * (1 - IdotN*IdotN);
    return (k < 0) ? point(0,0,0) : (eta*I - N * (eta*IdotN + sqrt(k)));
}
```

```
void fresnel (vector I, normal N, float eta,
              output float Kr, output float Kt,
              output vector R, output vector T);
```

According to Snell's law and the Fresnel equations, `fresnel` computes the reflection and transmission direction vectors `R` and `T`, respectively, as well as the scaling factors for reflected and transmitted light, `Kr` and `Kt`. The `I` parameter is the normalized incident ray, `N` is the normalized surface normal, and `eta` is the ratio of refractive index of the medium containing `I` to that on the opposite side of the surface.

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

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

```
ptype transform (string tospace, ptype p)
ptype transform (string fromspace, string tospace, ptype p)
ptype transform (matrix Mto, ptype p)
```

Transform a `point`, `vector`, or `normal` (depending on the type of the *ptype p* argument) from the coordinate system named by *fromspace* to the one named by *tospace*. If *fromspace* is not supplied, *p* is assumed to be in `"common"` space coordinates, so the transformation will be from `"common"` space to *tospace*. A 4×4 matrix may be passed directly rather than specifying coordinate systems by name.

Depending on the type of the passed point *p*, different transformation semantics will be used. A `point` will transform as a position, a `vector` as a direction without regard to positioning, and a `normal` will transform subtly differently than a `vector` in order to preserve orthogonality to the surface under nonlinear scaling.[1]

```
float transformu (string tounits, float x)
float transformu (string fromunits, string tounits, float x)
```

Transform a measurement from *fromunits* to *tounits*. If *fromunits* is not supplied, *x* will be assumed to be in `"common"` space units.

For length conversions, unit names may be any of: `"mm"`, `"cm"`, `"m"`, `"km"`, `"in"`, `"ft"`, `"mi"`, or the name of any coordinate system, including `"common"`, `"world"`, `"shader"`, or any other named coordinate system that the renderer knows about.

For time conversions, units may be any of: `"s"`, `"frames"`, or `"common"` (which indicates whatever timing units the renderer is using).

It is only valid to convert length units to other length units, or time units to other time units. Attempts to convert length to time or vice versa will result in an error. Don't even think about trying to convert monetary units to time.

[1] Technically, what happens is this: The *from* and *to* spaces determine a 4×4 matrix. A `point` (x,y,z) will transform the 4-vector $(x,y,z,1)$ by the matrix; a `vector` will transform $(x,y,z,0)$ by the matrix; a `normal` will transform $(x,y,z,0)$ by the inverse of the transpose of the matrix.

7.3 Color functions

```
color color (float f)
color color (float r, float g, float b)
```

Constructs a `color` from individual `float` values. If constructed from a single `float`, the value will be replicated for *r*, *g*, and *b*.

```
color color (string colorspace, f)
color color (string colorspace, float r, float g, float b)
```

Constructs an RGB `color` that is equivalent to the individual `float` values in a named color space. In other words,

```
color (colorspace, r, g, b)
```

is equivalent to

```
transformc (colorspace, "rgb", color(r, g, b))
```

```
float luminance (color rgb)
```

Returns the linear luminance of the color *rgb*, which is implemented per the ITU-R standard as $0.2126R + 0.7152G + 0.0722B$.

```
color transformc (string fromspace, string tospace, color Cfrom)
color transformc (string tospace, color Cfrom)
```

Transforms color *Cfrom* from color space *fromspace* to color space *tospace*. If *fromspace* is not supplied, it is assumed to be transforming from `"rgb"` space.

Should this be renamed 'ctransform' to match RMan?

?

7.4 Matrix functions

```
matrix matrix (float m00, float m01, float m02, float m03,
               float m10, float m11, float m12, float m13,
               float m20, float m21, float m22, float m23,
               float m30, float m31, float m32, float m33)
```

Constructs a `matrix` from 16 individual `float` values, in row-major order.

```
matrix matrix (float f)
```

Constructs a `matrix` with f in all diagonal components, 0 in all other components. In other words, `matrix(1)` is the identity matrix, and `matrix(f)` is `f*matrix(1)`.

```
matrix matrix (string fromspace, float m00, ..., float m33)
matrix matrix (string fromspace, float f)
```

Constructs a `matrix` relative to the named space, multiplying it by the `space`-to-`common` transformation matrix.

Note that `matrix (space, 1)` returns the *space*-to-`common` transformation matrix.

```
matrix matrix (string fromspace, string tospace)
```

Constructs a `matrix` that can be used to transform coordinates from *fromspace* to *tospace*.

```
float determinant (matrix M)
```

Computes the determinant of matrix M.

```
matrix transpose (matrix M)
```

Computes the transpose of matrix M.

7.5 Pattern generation

```
float step (float edge, float x)
```

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

```
float smoothstep (float edge0, float edge1, float 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.

```
type noise (float u)
type noise (float u, float v)
type noise (point p)
type noise (point p, float t)
```

Returns a continuous, pseudo-random (but repeatable) scalar field defined on a domain of dimension 1 (`float`), 2 (2 `float`'s), 3 (`point`), or 4 (`point` and `float`).

The range of `noise()` is $[0,1]$, its large-scale average is 0.5, it is fairly isotropic and non-periodic (or at least has an extremely large period), and mostly band-limited to frequences between 0.5 and 1.0. This makes it ideal to use as a basis function for pattern generation.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float noise()` function.

```
type snoise (float u)
type snoise (float u, float v)
type snoise (point p)
type snoise (point p, float t)
```

The `snoise()` function (*signed* noise) is just like `noise()`, but scaled so that its range is $[-1,1]$ and its large-scale average is 0.

```
type pnoise (float u, float uperiod)
type pnoise (float u, float v, float uperiod, float vperiod)
type pnoise (point p, point pperiod)
type pnoise (point p, float t, point pperiod, float tperiod)
type psnoise (float u, float uperiod)
type psnoise (float u, float v, float uperiod, float vperiod)
type psnoise (point p, point pperiod)
type psnoise (point p, float t, point pperiod, float tperiod)
```

The `pnoise()` function is just like `noise()`, but is periodic with the given period. In other words, `pnoise(x,period) == pnoise(x+period,period)`. Periods are only meaningful if they are positive integers, so actually the period is rounded down to the nearest integer, and clamped to a minimum of 1.

The `psnoise()` function a signed version of `pnoise()`, i.e., identical to `pnoise()` but having a range of $[-1,1]$ and average value of 0.

```
type cellnoise (float u)
type cellnoise (float u, float v)
type cellnoise (point p)
type cellnoise (point p, float t)
```

Returns a discrete pseudo-random (but repeatable) scalar field defined on a domain of dimension 1 (`float`), 2 (2 `float`'s), 3 (`point`), or 4 (`point` and `float`).

The `cellnoise()` function is constant on $[i,i+1)$ for all integers i, in other words, `cellnoise(x) == cellnoise(floor(x))`, but has a different and uncorrelated value at every integer. The range is $[0,1]$, its large-scale average is 0.5, and its values are evenly distributed over $[0,1]$.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float cellnoise()` function.

```
type hash (float u)
type hash (float u, float v)
type hash (point p)
type hash (point p, float t)
```

Returns a deterministic, repeatable *hash* of the 1-, 2-, 3-, or 4-D coordinates. The return values will be evenly distributed on $[0,1]$ and be be completely repeatable when passed the same coordinates again, yet will be uncorrellated to hashes of any other positions (including nearby points). This is like having a random value indexed spatially, but that will be repeatable from frame to frame of an animation (provided its input is *precisely* identical).

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float hash` function.

type **random** ()

Returns a pseudorandom value uniformly distributed on $[0,1]$. Use with extreme caution — the results are not necessarily repeatable from frame to frame.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float random()` function.

type **spline** (string basis, float x, *type* y_0, *type* y_1, ... *type* y_{n-1})
type **spline** (string basis, float x, *type* y[])

As x varies from 0 to 1, `spline` returns the value of a cubic interpolation of uniformly-spaced knots $y_0 ... y_{n-1}$, or $y[0] ... y[n-1]$ for the array version of the call (where n is the length of the array). The input value x will be clamped to lie on $[0,1]$. The *type* may be any of `float`, `color`, `point`, `vector`, or `normal`; for multi-component types (e.g. `color`), each component will be interpolated separately.

The type of interpolation is specified by the *basis* name, *basis* parameter, which may be any of: `"catmull-rom"`, `"bezier"`, `"bspline"`, `"hermite"`, or `"linear"`. Some basis types require particular numbers of knot values – Bezier splines require $4n+3$ values, Hermite splines require $4n+2$ values, Catmull-Rom and linear splines may use any number of values $n \geq 4$. To maintain consistency with the other spline types, linear splines will ignore the first and last data value, interpolating piecewise-linearly between y_1 and y_{n-2}.

float **inversespline** (string basis, float v, float y_0, ... float y_{n-1})
float **inversespline** (string basis, float v, float y[])

Computes the *inverse* of the `spline()` function — i.e., returns the value x for which `spline (basis, x, y...)` would return value v. Results are undefined if the knots do not specifiy a monotonic (only increasing or only decreasing) spline.

7.6 Derivatives and area operators

```
float Dx (float a), Dy (float a)
vector Dx (point a), Dy (point a)
vector Dx (vector a), Dy (vector a)
color Dx (color a), Dy (color a)
```

Compute an approximation to the partial derivatives of a with respect to each of two principal directions, $\partial a/\partial x$ and $\partial a/\partial y$. Depending on the renderer implementation, those directions may be aligned to the image plane, on the surface of the object, or something else.

```
float filterwidth (float x)
vector filterwidth (point x)
vector filterwidth (vector x)
```

Compute differentials of the argument x, i.e., the approximate change in x between adjacent shading samples.

```
float area (point p)
```

Returns the differential area of position p corresponding to this shading sample. If p is the actual surface position `P`, then **`area(P)`** will return the surface area of the section of the surface that is "covered" by this shading sample.

```
vector calculatenormal (point p)
```

Returns a vector perpendicular to the surface that is defined by point p (as p is computed at all points on the currently-shading surface), taking into account surface orientation.

```
float aastep (float edge, float s)
float aastep (float edge, float s, float ds)
float aastep (float edge, float s, float dedge, float ds)
float aastep (float edge, float s, string filter)
float aastep (float edge, float s, float ds, string filter)
float aastep (float edge, float s, float dedge, float ds, string filter)
```

Computes an antialiased step function, similar to `step(edge,s)` but filtering the edge to take into account how rapidly `s` and `edge` are changing over the surface. If the differentials `ds` and/or `dedge` are not passed explicitly, they will be automatically computed (using `aastep()`).

The optional `filter` parameter specifies which filter should be used: `"catmull-rom"`, `"box"`, `"triangle"`, or `"gaussian"`. If no `filter` parameter is supplied, a Catmull-Rom filter will be used.

This is similar to RenderMan's "filterstep", but not quite. Is it better to call it "filterstep" even

if it doesn't exactly match, or should we stick to the changed name to match the changed functionality?

7.7 Displacement functions

```
void displace (float amp)
void displace (string space, float amp)
void displace (vector offset)
```

Displace the surface in the direction of the shading normal `N` by *amp* units as measured in the named *space* (or `"current"` space if none is specified). Alternately, the surface may be moved by a fully general *offset*, which does not need to be in the direction of the surface normal.

In either case, this function both displaces the surface and adjusts the shading normal `N` to be the new surface normal of the displaced surface (properly handling both continuously smooth surfaces as well as interpolated normals on faceted geometry, without introducing faceting artifacts).

```
void bump (float amp)
void bump (string space, float amp)
void bump (vector offset)
```

Adjust the shading normal `N` to be the surface normal as if the surface had been displaced by the given amount (see the `displace()` function description), but without actually moving the surface positions.

7.8 String functions

```
void printf (string fmt, ...)
```

Much as in C, `printf()` takes a format string `fmt` and an argument list, and prints the resulting formatted string to the console.

Where the `fmt` contains the characters `%d`, `%i`, `%f`, `%g`, and `%s`, `printf` will substitute arguments, in order. The `%d` and `%i` arguments expect an `int` argument; `%f` and `%g` expect a `float`, `color`, point-like, or `matrix` argument (for multi-component types such as `color`, the format will be applied to each of the components); and `%s` expects a `string` argument. In addition, `%d` and `%i` will also print `float`'s, truncating and printing them as if they were integers.

All of the substition commands follow the usual C/C++ formatting rules, so format commands such as `"%6.2f"`, etc., should work as expected.

`string` **`format`** `(string fmt, ...)`

The `format` function works similarly to `printf`, except that instead of printing the results, it returns the formatted text as a .

`void` **`error`** `(string fmt, ...)`

The `error()` function works similarly to `printf`, but the results will be printed as a renderer error message, possibly including information about the name of the shader and the object being shaded, and other diagnostic information.

`void` **`fprintf`** `(string filename, string fmt, ...)`

The `fprintf()` function works similarly to `printf`, but rather than printing to the default text output stream, the results will be concatentated onto the end of the text file named by `filename`.

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

Concatenates a list of strings, returning the aggregate string.

`int` **`strlen`** `(string s)`

Return the number of characters in string **`s`**.

`int` **`startswith`** `(string s, string prefix)`

Return 1 if string `s` begins with the substring `prefix`, otherwise return 0.

`int` **`endswith`** `(string s, string suffix)`

Return 1 if string `s` ends with the substring `suffix`, otherwise return 0.

`string` **`substr`** `(string s, int start, int length)`
`string` **`substr`** `(string s, int start)`

Return at most *length* characters from *s*, starting with the character indexed by *start* (beginning with 0). If *length* is omitted, return the rest of *s*, starting with *start*. If *start* is negative, it counts backwards from the end of the string (for example, `substr(s,-1)` returns just the last character of *s*).

`int` **`regex_search`** `(string subject, string regex)`
`int` **`regex_search`** `(string subject, int results[], string regex)`

Returns 1 if any substring of *subject* matches a standard POSIX regular expression *regex*, 0 if it does not.

In the form that also supplies a `results` array, when a match is found, the array will be filled in as follows:

`results[0]`	the character index of the start of the sequence that matched the regular expression.
`results[1]`	the character index of the end (i.e., one past the last matching character) of the sequence that matched the regular expression.
`results[`$2i$`]`	the character index of the start of the sequence that matched sub-expression i of the regular expression.
`results[`$2i+1$`]`	the character index of the end (i.e., one past the last matching character) of the sequence that matched sub-expression i of the regular expression.

Sub-expressions are denoted by surrounding them in parentheses. in the regular expression.

A few examples illustrate regular expression searching:

```
regex_search ("foobar.baz", "bar")    = 1
regex_search ("foobar.baz", "bark")   = 0

int match[2];
regex_search ("foobar.baz", match, "[Oo]{2}") = 1
                                  (match[0] == 1, match[1] == 3)
substr ("foobar.baz", match[0], match[1]-match[0]) = "oo"

int match[6];
regex_search ("foobar.baz", match, "(f[Oo]{2}).*(.az)") = 1
substr ("foobar.baz", match[0], match[1]-match[0]) = "foobar.baz"
substr ("foobar.baz", match[2], match[3]-match[2]) = "foo"
substr ("foobar.baz", match[4], match[5]-match[4]) = "baz"
```

```
int regex_match (string subject, string regex)
int regex_match (string subject, int results[], string regex)
```

Identical to `regex_search`, except that it must match the *whole* `subject` string, not merely a substring.

Should we look at Pystring and add useful things here?

7.9 Texture

```
type texture (string filename, float s, float t, ...params...)
type texture (string filename, float s, float t,
              float dsdx, float dtdx, float dsdy, float dtdy, ...params...)
```

Perform a texture lookup of an image file, indexed by 2D coordinates (s,t), antialiased over a region defined by the differentials *dsdx, dtdx, dsty* and *dtdy* (which are computed automatically from s and t, if not supplied). Whether the results are assigned to a `float`

or a `color` (or type cast to one of those) determines whether the texture lookup is a single channel or three channels.

The 2D lookup coordinate(s) may be followed by optional token/value arguments that control the behavior of `texture()`:

`"blur", <float>`

Additional blur when looking up the texture value (default: 0). The blur amount is relative to the size of the texture (i.e., 0.1 blurs by a kernel that is 10% of the full width and height of the texture).

The blur may be specified separately in the *s* and *t* directions by using the `"sblur"` and `"tblur"` parameters, respectively.

`"width", <float>`

Scale (multiply) the size of the filter as defined by the differentials (or implicitly by the differentials of *s* and *t*). The default is `1`, meaning that no special scaling is performed. A width of `0` would effectively turn off texture filtering entirely.

The width value may be specified separately in the *s* and *t* directions by using the `"swidth"` and `"twidth"` parameters, respectively.

`"wrap", <string>`

Specifies how the texture *wraps* coordinates outside the $[0,1]$ range. Supported wrap modes include: `"black"`, `"periodic"`, `"clamp"`, `"mirror"`, and `"default"` (which is the default). A value of `"default"` indicates that the renderer should use any wrap modes specified in the texture file itself (a non-`"default"` value overrides any wrap mode specified by the file).

The wrap modes may be specified separately in the *s* and *t* directions by using the `"swrap"` and `"twrap"` parameters, respectively.

`"firstchannel", <int>`

The first channel to look up from the texture map (default: 0).

`"fill", <float>`

The value to return for any channels that are not present in the texture file (default: 0).

`"alpha", <floatvariable>`

The alpha channel (presumed to be the next channel following the channels returned by the `texture()` call) will be stored in the variable specified. This allows for RGBA lookups in a single call to `texture()`.

```
int gettextureinfo (string texturename, string paramname,
                    output type destination)
```

Retrieves a parameter from a named texture file. If the file is found, and has a parameter that matches the name and type specified, its value will be stored in *destination* and `gettextureinfo` will return `1`. If the file is not found, or doesn't have a matching parameter (including if the type does not match), *destination* will not be modified and `getattribute` will return `0`. Valid parameters recognized are listed below:

Name	Type	Description
`"resolution"`	`int[2]`	The resolution of the highest MIPmap level stored in the texture map.
`"channels"`	`int`	The number of channels in the texture map.
`"type"`	`string`	Returns the semantic type of the texture, one of: `"Plain Texture"`, `"Shadow"`, `"Environment"`, `"Volume Texture"`.
`"textureformat"`	`string`	Returns the texture format, one of: `"Plain Texture"`, `"Shadow"`, `"CubeFace Shadow"`, `"Volume Shadow"`, `"CubeFace Environment"`, `"LatLong Environment"`, `"Volume Texture"`. Note that this differs from `"type"` in that it specifically distinguishes between the different types of shadows and environment maps.
`"viewingmatrix"`	`matrix`	(Shadow maps only) The matrix that transforms points from `"common"` space to the `"camera"` space from which the texture was created.
`"projectionmatrix"`	`matrix`	(Shadow maps only) The matrix that transforms points from `"common"` space to a 2D coordinate system where x and y range from -1 to 1.
anything else	*any*	Searches for matching name and type in the metadata or other header information of the texture file.

7.10 Light and Shadows

`closure color` **`diffuse`** `(normal N,` *...params...*`)`

Returns a `closure color` that represents the Lambertian diffuse reflectance of a smooth surface,

$$\int_{\Omega} Cl(P,\omega)\max(0,N\cdot\omega)d\omega$$

where N is the unit-length forward-facing surface normal at `P`, Ω is the set of all outgoing directions in the hemisphere surrounding N, and $Cl(P,\omega)$ is the incident radiance at `P` coming from the direction $-\omega$.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all `closure color`'s, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction `-I`. But aside from the fact that the shader cannot examine the numeric values of the `closure color`, you may program *as if* `diffuse()` was implemented as follows:

```
color C = 0;
normal Nf = faceforward (normalize(N), I);
for all lights within the hemisphere defined by (P, Nf, PI/2) {
    /* L is the direction of light i, Cl is its incoming radiance */
    C += Cl * dot (normalize(L), Nf);
}
return C;
```

`closure color` **`orennayar`** `(normal N, float roughness,` *...params...*`)`

Returns a `closure color` that represents the diffuse reflectance of a rough surface, implementing the Oren-Nayar reflectance formula. The *roughness* parameter indicates how smooth or rough the microstructure of the material is, with 0 being perfectly smooth and giving an appearance identical to `lambert()`.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all `closure color`'s, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction `-I`. But aside from the fact that the shader cannot examine the numeric values of the `closure color`, you may program *as if* `orennayar()` was implemented as follows:

```
normal Nf = faceforward (normalize(N), I);
vector V = -normalize(I);
float sigma2 = roughness * roughness;
float A = 1 - 0.5 * sigma2 / (sigma2 + 0.33);
```

```
float B = 0.45 * sigma2 / (sigma2 + 0.09);
float  theta_r = acos (dot (V, Nf));         // Angle between V and N
vector V_perp_N = normalize(V-Nf*dot(V,Nf)); // Part of V perpendicular to N
color C = 0;
for all lights within the hemisphere defined by (P, Nf, PI/2) {
    /* L is the direction of light i, Cl is its incoming radiance */
    vector LN = normalize(L);
    float cos_theta_i = dot(LN, N);
    float cos_phi_diff = dot (V_perp_N, normalize(LN - Nf*cos_theta_i));
    float theta_i = acos (cos_theta_i);
    float alpha = max (theta_i, theta_r);
    float beta = min (theta_i, theta_r);
    C += Cl * cos_theta_i *
            (A + B * max(0,cos_phi_diff) * sin(alpha) * tan(beta));
}
return C;
```

`closure color` **`phong`** `(normal N, float exponent, ...`*params*`...)`

Returns a `closure color` that represents specular reflectance of the surface using the Phong BRDF. The *exponent* parameter indicates how smooth or rough the material is.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all `closure color`'s, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction `-I`. But aside from the fact that the shader cannot examine the numeric values of the `closure color`, you may program *as if* `phong()` was implemented as follows:

```
code eventually will go here
```

`closure color` **`cooktorrance`** `(normal N, float roughness, ...`*params*`...)`

Returns a `closure color` that represents specular reflectance of the surface using the Cook-Torrence BRDF. The *roughness* parameter indicates how smooth or rough the microstructure of the material is.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all `closure color`'s, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction `-I`. But aside from the fact that the shader cannot examine the numeric values of the `closure color`, you may program *as if* `cooktorrance()` was implemented as follows:

```
code eventually will go here
```

`closure color` **`ward`** `(normal N, vector T,`
`float xrough, float yrough,` *...params...*`)`

Returns a `closure color` that represents the anisotropic specular reflectance of the surface at `P`. The `N` and `T` vectors, both presumed to be unit-length, are the surface normal and tangent, used to establish a local coordinate system for the anisotropic effects. The *xrough* and *yrough* specify the amount of roughness in the `T` and `T` $\times$ `N` directions, respectively.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all `closure color`'s, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction `-I`. But aside from the fact that the shader cannot examine the numeric values of the `closure color`, you may program *as if* `ward()` was implemented as follows:

```
float sqr (float x) { return x*x; }

normal Nf = faceforward (normalize(N), I);
vector V = -normalize(I);
float cos_theta_r = clamp (dot(Nf,V), 0.0001, 1);
vector X = T / xroughness;
vector Y = cross(T,N) / yroughness;
color C = 0;
for all lights within the hemisphere defined by (P, Nf, PI/2) {
    /* L is the direction of light i, Cl is its incoming radiance */
    vector LN = normalize (L);
    float cos_theta_i = dot (LN,Nf);
    if (cos_theta_i > 0.0) {
        vector H = normalize (V + LN);
        float rho = exp (-2 * (sqr(dot(X,H)) +
                               sqr(dot(Y,H))) / (1 + dot(H,N)))
                    / sqrt (cos_theta_i * cos_theta_r);
        C += Cl * cos_theta_i * rho;
    }
}
return C / (4 * xroughness * yroughness);
```

`closure color` **`microfacet_beckmann`** `(normal N, float roughness, float eta,` *...params...*`)`

Returns a `closure color` that represents specular reflectance of the surface using a Beckmann microfacet BRDF (as described in Walter et al., "Microfacet Models for Refraction through Rough Surfaces," Eurographics Symposium on Rendering, 2007). The `roughness` controls how rough the microstructure of the surface is. The `eta` parameter is the index of refraction of the material.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

`closure color` **`microfacet_ggx`** `(normal N, float roughness, float eta,` *...params...*`)`

Returns a `closure color` that represents specular reflectance of the surface using the "GGX" microfacet BRDF described in Walter et al., "Microfacet Models for Refraction through Rough Surfaces," Eurographics Symposium on Rendering, 2007. The `roughness` controls how rough the microstructure of the surface is. The `eta` parameter is the index of refraction of the material.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

`closure color closure color` **`reflection`** `(normal N, float eta,` *...params...*`)`

Returns a `closure color` that represents sharp mirror-like reflection from the surface. The reflection direction will be automatically computed based on the incident angle. The `eta` parameter is the index of refraction of the material.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all `closure color`'s, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction `-I`. But aside from the fact that the shader cannot examine the numeric values of the `closure color`, you may program *as if* `reflection()` was implemented as follows:

```
vector R = reflect (I, N);
return trace (R);
```

`closure color` **`refraction`** `(normal N, float eta,` *...params...*`)`

Returns a `closure color` that represents sharp glass-like refraction of objects "behind" the surface. The *eta* parameter is the ratio of the index of refraction of the medium on the "inside" of the surface divided by the index of refration of the medium on the "outside" of the surface. The "outside" direction is the one specified by `N`.

The refraction direction will be automatically computed based on the incident angle and *eta*, and the radiance returned will be automatically scaled by the Fresnel factor for dielectrics.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all `closure color`'s, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the

direction -I. But aside from the fact that the shader cannot examine the numeric values of the closure color, you may program *as if* refraction() was implemented as follows:

```
float Kr, Kt;
vector R, T;
fresnel (I, N, eta, Kr, Kt, R, T);
return Kt * trace (T);
```

closure color **dielectric** (normal N, float eta, ...*params...*)

Returns a closure color that represents both the reflection and refraction from delectric materials such as glass or transparent plastic. The *eta* parameter is the ratio of the index of refraction of the medium on the "inside" of the surface divided by the index of refration of the medium on the "outside" of the surface. The "outside" direction is the one specified by N.

The reflection and refraction directions will be automatically computed based on the incident angle and *eta*, and the radiance returned will be automatically scaled by the Fresnel factors for dielectrics.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all closure color's, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction -I. But aside from the fact that the shader cannot examine the numeric values of the closure color, you may program *as if* dielectric() was implemented as follows:

```
float Kr, Kt;
vector R, T;
fresnel (I, N, eta, Kr, Kt, R, T);
return Kr * trace (R) + Kt * trace (T);
```

closure color **transparent** (...*params...*)

Returns a closure color that shows the light *behind* the surface without any refractive bending of the light directions.

Additional optional arguments may be passed as token/value pairs:

None currently supported.

Like all closure color's, the return "value" is symbolic and is may be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction -I. But aside from the fact that the shader cannot examine the numeric values of the closure color, you may program *as if* transparent() was implemented as follows:

```
return trace (I);
```

`closure color` **`translucence`** (*...params...*)

Returns a `closure color` that represents the Lambertian diffuse translucence of a smooth surface, which is much like `diffuse()` except that it gathers light from the *far* side of the surface.

Is a general term such as "translucence" too presumptuous for a simple backfacing lambert? Should we instead call this `backdiffuse`, `backlambert`, or something more specific? Since probably later we will add one or more physically-accurate, translucence function.

`closure color` **`subsurface`** (*...params...*)

Returns a `closure color` that represents the amount of *subsurface scattering* exhibited by the surface.

We'll determine later what the arguments to `subsurface` need to be. FIXME

`closure color` **`emission`** (*...params...*)
`closure color` **`emission`** (`vector axis, float angle,` *...params...*)

Returns a `closure color` that represents a glowing/emissive surface. By default, light is emitted in a full hemisphere centered around the surface normal. Optionally, *axis* and *angle* parameters may explicitly specify the emission cone (*axis* = `N` and *angle* = `PI/2` indicate the full hemisphere around `N`).

The emission closure returns a radiance value (e.g., $\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}$) that, integrated over all angles, yields a power density of 1.0 (e.g., $\mathrm{W} \cdot \mathrm{m}^{-2}$). This may be multiplied by a scalar to adjust the desired power density of your light.

Additional optional arguments may be passed as token/value pairs:

`"penumbra", <float>`

Specifies a transition angle for light to fall off. If α is the angle between the axis and an illumination direction, then light is emitted evenly in all diretions for which $\alpha \leq \mathit{angle} - \mathit{penumbra}$, no light is emitted in directions for which $\alpha > \mathit{angle}$, and the illumination falls gradually for $\mathit{angle} - \mathit{penumbra} < \alpha \leq \mathit{angle}$. If *penumbra* is not supplied, it defaults to 0, indicating a hard cutoff at the cone boundary.

7.11 Renderer state and message passing

`int` **`getattribute`** `(string name, output` *`type`* `destination)`
`int` **`getattribute`** `(string name, int arrayindex, output` *`type`* `destination)`

Searches for the renderer attribute with the given *name* (first searching per-object attributes on the current object, then if not found searching scene-wide attributes). If found and it matches the type of *destination*, the attribute's value will be stored in *destination* and `getattribute` will return `1`. If not found, or the type does not match, *destination* will not be modified and `getattribute` will return `0`.

The second form of this function, with the `arrayindex` parameter, retrieves the individual indexed element of the named array. In this case, *name* must be an array attribute, the type of *destination* must be the type of the array element (not the type of the whole array), and the value of *arrayindex* must be a valid index given the array's size.

`void` **`setmessage`** `(string name, output` *`type`* `value)`

Store a *name*/*value* pair in an area where it can later be retrieved by other shaders attached to the same objects. If there is already a message with the same name attached to this shader invocation, it will be replaced by the new value.

`int` **`getmessage`** `(string source, string name, output` *`type`* `destination)`

Retrieve a message from another shader attached to the same object. Specifically, it will try retrieve the value associated with the given *name* from a specific *source*. If the source has a value with the given name, and whose type matches that of *destination*, the value will be stored in *destination* and `getmessage()` will return `1`. If no message is found that matches both the name and type, *destination* will be unchanged and `getmessage()` will return `0`.

The *source* designates from where the message should be retrieved, and may have any of the following values:

`"surface", "displacement", "volume"`

Retrieve the message from the named shader type present on the object being shaded.

`"parent"`

Retrieve the message from the shader (potentially on another object) that spawned the ray that caused the current shader to be executed.

`"trace"`

Valid only if `getmessage()` is called within a `trace` loop, retrieves the message from the object hit by the ray. Note also that `getmessage("trace",...)` may be used to get information about the individual ray sample itself.

`"primitive"`

Retrieve the value by interpolating the named user variable attached to the geometric primitive that is being shaded (regardless of whether it is a parameter to the shader).

First, `getmessage()` will search the source's message list as set by `setmessage()` when that shader ran. If not found, then the source's shader parameters will be searched. If multiple shaders of the same type are present, they will be searched starting with the last one executed, and moving backwards in execution order (in other words, later-executed shaders in the network effectively take precedence over parameteters or messages from earlier-executed shaders).

`int` **`raylevel`** `()`

Returns the *ray level* of the current shading sample — `0` if the surface is viewed directly by the camera, `1` if it is a first-level reflection or refraction, `2` if it is a reflection/refraction visible in a reflection/refraction, etc.

`int` **`isshadowray`** `()`

Returns `1` if the shader is executing for the sole purpose of determining the object's opacity (typically for a shadow), in which case it may be prudent for the shader to avoid computations that are only needed to determine the surface color or other outputs. This function returns `0` (indicating that the surface should be computing surface color, opacity, and all its other outputs).

`int` **`isindirectray`** `()`

Returns `1` if the shader is executing for the sole purpose of determining the object's appearance for an indirect (or global illumination) sample and therefore will not be seen distinctly in the final image, in which case it may be prudent for the shader to take shortcuts or avoid computing fine detail. This function returns `0` if it is being run for ordinary visibility rays, in which case the results may be clearly seen and all details should be computed.

7.12 Miscellaneous

`int` **`arraylength`** `(`*`type`* `A[])`

Returns the length of the referenced array, which may be of any type.

`void` **`exit`** `()`

Exits the shader without further execution. Within the main body of a shader, this is equivalent to calling `return`, but inside a function, `exit()` will exit the entire shader, whereas `return` would only exit the enclosing function.

8 Formal Language Grammar

This section gives the complete syntax of Open Shading Language. Syntactic structures that have a name ending in "-opt" are optional. Structures surrounded by curly braces { } may be repeated 0 or more times. Text in `typewriter` face indicates literal text. The ε character is used to indicate that it is acceptable for there to be nothing (empty, no token).

Lexical elements

⟨*digit*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨*digit-sequence*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }

⟨*integer*⟩ ::= ⟨*sign*⟩ ⟨*digit-sequence*⟩

⟨*floating-point*⟩ ::= ⟨*digit-sequence*⟩ ⟨*decimal-part-opt*⟩ ⟨*exponent-opt*⟩
| ⟨*decimal-part*⟩ ⟨*exponent-opt*⟩

⟨*decimal-part*⟩ ::= '`.`' { ⟨*digit*⟩ }

⟨*exponent*⟩ ::= '`e`' ⟨*sign*⟩ ⟨*digit-sequence*⟩

⟨*sign*⟩ ::= '`-`' | '`+`' | ε

⟨*number*⟩ ::= ⟨*integer*⟩
| ⟨*floating-point*⟩

⟨*char-sequence*⟩ ::= { ⟨*any-char*⟩ }

⟨*stringliteral*⟩ ::= '`"`' ⟨*char-sequence*⟩ '`"`'

⟨*identifier*⟩ ::= ⟨*letter-or-underscore*⟩ { ⟨*letter-or-underscore-or-digit*⟩ }

Overall structure

⟨*shader-file*⟩ ::= { ⟨*global-declaration*⟩ }

⟨*global-declaration*⟩ ::= ⟨*function-declaration*⟩
| ⟨*struct-declaration*⟩
| ⟨*shader-declaration*⟩

⟨*shader-declaration*⟩ ::=
⟨*shadertype*⟩ ⟨*identifier*⟩ ⟨*metadata-block-opt*⟩ `(` ⟨*shader-formal-params-opt*⟩ `)` `{`
⟨*statement-list*⟩ `}`

⟨*shadertype*⟩ ::= `displacement` | `imager` | `integrator` | `shader` | `surface` | `volume`

⟨*shader-formal-params*⟩ ::= ⟨*shader-formal-param*⟩ { `,` ⟨*shader-formal-param*⟩ }

⟨*shader-formal-param*⟩ ::= ⟨*outputspec*⟩ ⟨*typespec*⟩ ⟨*identifier*⟩ ⟨*initializer*⟩ ⟨*metadata-block-opt*⟩
| ⟨*outputspec*⟩ ⟨*typespec*⟩ ⟨*identifier*⟩ ⟨*arrayspec*⟩ ⟨*array-initializer*⟩ ⟨*metadata-block-opt*⟩

⟨*metadata-block*⟩ ::= `[[` ⟨*metadata*⟩ { `,` ⟨*metadata*⟩ } `]]`

⟨*metadata*⟩ ::= ⟨*simple-typespec*⟩ ⟨*identifier*⟩ ⟨*initializer*⟩

Declarations

⟨*function-declaration*⟩ ::=
⟨*typespec*⟩ ⟨*identifier*⟩ `(` ⟨*function-formal-params-opt*⟩ `)` `{` ⟨*statement-list*⟩ `}`

⟨*function-formal-params*⟩ ::= ⟨*function-formal-param*⟩ { `,` ⟨*function-formal-param*⟩ }

⟨*function-formal-param*⟩ ::= ⟨*outputspec*⟩ ⟨*typespec*⟩ ⟨*identifier*⟩ ⟨*arrayspec-opt*⟩

⟨*outputspec*⟩ ::= `output` | ε

⟨*struct-declaration*⟩ ::= `struct` ⟨*identifier*⟩ `{` ⟨*field-declarations*⟩ `}` `;`

⟨*field-declarations*⟩ ::= ⟨*field-declaration*⟩ { ⟨*field-declaration*⟩ }

⟨*field-declaration*⟩ ::= ⟨*typespec*⟩ ⟨*identifier*⟩ `;`
| ⟨*typespec*⟩ ⟨*identifier*⟩ ⟨*arrayspec*⟩ `;`

⟨*local-declaration*⟩ ::= ⟨*function-declaration*⟩
| ⟨*variable-declaration*⟩

⟨*arrayspec*⟩ ::= `[` ⟨*integer*⟩ `]`

⟨*variable-declaration*⟩ ::= ⟨*typespec*⟩ ⟨*def-expressions*⟩ `;`

⟨*def-expressions*⟩ ::= ⟨*def-expression*⟩ { `,` ⟨*def-expression*⟩ }

⟨*def-expression*⟩ ::= ⟨*identifier*⟩ ⟨*initializer-opt*⟩
| ⟨*identifier*⟩ ⟨*arrayspec*⟩ ⟨*array-initializer-opt*⟩

⟨*initializer*⟩ ::= = ⟨*expression*⟩

⟨*array-initializer*⟩ ::= = { ⟨*expression-list*⟩ }

⟨*typespec*⟩ ::= ⟨*simple-typename*⟩
| `closure` ⟨*simple-typename*⟩
| ⟨*identifier-structname*⟩

⟨*simple-typename*⟩ ::= `color` | `float` | `matrix` | `normal` | `point` | `string` | `vector` | `void`

Statements

⟨*statement-list*⟩ ::= ⟨*statement*⟩ { ⟨*statement*⟩ }

⟨*statement*⟩ ::= ⟨*expression-opt*⟩ ;
| ⟨*scoped-statements*⟩
| ⟨*local-declaration*⟩
| ⟨*conditional-statement*⟩
| ⟨*loop-statement*⟩
| ⟨*loopmod-statement*⟩
| ⟨*return-statement*⟩

⟨*scoped-statements*⟩ ::= { ⟨*statement-list-opt*⟩ }

⟨*conditional-statement*⟩ ::=
`if` (⟨*expression*⟩) ⟨*statement*⟩
| `if` (⟨*expression*⟩) ⟨*statement*⟩ `else` ⟨*statement*⟩

⟨*loop-statement*⟩ ::=
`while` (⟨*expression*⟩) ⟨*statement*⟩
| `do` ⟨*statement*⟩ `while` (⟨*expression*⟩) ;
| `for` (⟨*for-init-statement-opt*⟩ ⟨*expression-opt*⟩ ; ⟨*expression-opt*⟩) ⟨*statement*⟩

⟨*for-init-statement*⟩ ::=
⟨*expression-opt*⟩ ;
| ⟨*variable-declaration*⟩

⟨*loopmod-statement*⟩ ::= `break` ;
| `continue` ;

⟨*return-statement*⟩ ::= `return` ⟨*expression-opt*⟩ ;

Expressions

⟨*expression-list*⟩ ::= ⟨*expression*⟩ { , ⟨*expression*⟩ }

⟨*expression*⟩ ::= ⟨*number*⟩
| ⟨*stringliteral*⟩
| ⟨*type-constructor*⟩
| ⟨*incdec-op*⟩ ⟨*variable-ref*⟩
| ⟨*expression*⟩ ⟨*binary-op*⟩ ⟨*expression*⟩
| ⟨*unary-op*⟩ ⟨*expression*⟩
| `(` ⟨*expression*⟩ `)`
| ⟨*function-call*⟩
| ⟨*assign-expression*⟩
| ⟨*ternary-expression*⟩
| ⟨*typecast-expression*⟩
| ⟨*variable-ref*⟩

⟨*variable-lvalue*⟩ ::= ⟨*identifier*⟩ ⟨*array-deref-opt*⟩ ⟨*component-deref-opt*⟩
| ⟨*variable_lvalue*⟩ `[` ⟨*expression*⟩ `]`
| ⟨*variable_lvalue*⟩ `.` ⟨*identifier*⟩

⟨*variable-ref*⟩ ::= ⟨*identifier*⟩ ⟨*array-deref-opt*⟩

⟨*binary-op*⟩ ::= `*` | `/` | `%`
| `+` | `-`
| `<<` | `>>`
| `<` | `<=` | `>` | `>=`
| `==` | `!=`
| `&`
| `^`
| `|`
| `&&`
| `||`

⟨*unary-op*⟩ ::= `-` | `!` | `~`

⟨*incdec-op*⟩ ::= `++` | `--`

⟨*type-constructor*⟩ ::= ⟨*typespec*⟩ `(` ⟨*expression-list*⟩ `)`

⟨*function-call*⟩ ::= ⟨*identifier*⟩ `(` ⟨*function-args-opt*⟩ `)`

⟨*function-args*⟩ ::= ⟨*expression*⟩ { `,` ⟨*expression*⟩ }

⟨*assign-expression*⟩ ::= ⟨*variable-lvalue*⟩ ⟨*assign-op*⟩ ⟨*expression*⟩

⟨*assign-op*⟩ ::= `=` | `*=` | `/=` | `+=` | `-=` | `&=` | `|=` | `^=` | `<<=` | `>>=`

⟨*ternary-expression*⟩ ::= ⟨*expression*⟩ `?` ⟨*expression*⟩ `:` ⟨*expression*⟩

⟨*typecast-expression*⟩ ::= `(` ⟨*simple-typename*⟩ `)` ⟨*expression*⟩

9 Example Shaders

Sorry, I haven't done this yet. Eventually, this chapter will have several simple (but useful and fully functional) shaders to illustrate how all these concepts fit together.

10 Discussion

In this chapter, which will presumably not be included in the final language specicification document, we discuss some of the design choices we've made, as well as topics for which decisions have not yet been made.

Readers: this is your chance. Any topic that makes it into this chapter really deserves careful thought and feedback.

This specification is a living document. Nothing is permanently set in stone. There will someday be a 2.0 spec. It's easier to add things we left out than to take out things that turned out to be unnecessary things or a bad idea.

10.1 Variables, Functions, and Scoping

10.1.1 Separate namespace for functions?

NEW!

RSL and GSL had separate symbol tables for functions and for all other variables, for example allowing a variable or parameter "foo" and also a function called "foo". This is not for any principled reason, but just a historical accident or arbitrary choice. Once you do something like that, you can never change it, because then you'll break old shaders.

One of the nice things that came from having separate function and variable namespaces is that it's perfectly ok to have a shader parameter that happens to have the same name as a built-in function (for example, surprisingly many RenderMan shaders have a parameter named "diffuse"!). If you've ever tried to duplicate a set of shaders from a 3rd party package (such as making a set of shaders that duplicate all the Maya built-in nodes) you're very happy for separate symbol tables, so that you can pick param names that are the same as the Maya parameter names, without regard to whether their names happen to conflict with your built-in functions, otherwise you need to mangle the symbols somehow.

C, C++, and related languages tend to have a single symbol table for functions and variables. In other words, if you have a function named "foo", you can't have a variable named "foo" in the same scope, and vice versa. Part of the reason for this is that these languages can pass functions as parameters, take their address, etc., so they use a single namespace (though I think they could have separate namespace and still disambiguate based on the use).

So should we have separate function and variable namespaces (similar to RMan, possibly handy to avoid name clashes, but arbitrarily different than C's scoping rules), or single namespace (more "standard" in modern programming language design)?

10.2 Data types

10.2.1 Should `color` be spectral?

This spec is written assuming that `color` *should explicitly be 3-component RGB. Alternately, we could add global* `int ncomps` *that gives the numer of components in a* `color`*, and eliminate any built-in operations and library functions that seem to require or imply a 3-component color model (including type casting between* `color` *and the 3-component* `point`*-like types).*

Spectral color is a nice abstraction. It allows implementations to use a different number of color components if they choose to. It allows shaders to potentially be more physically accurate or perform computions that require more spectral samples.

On the other hand, Open Shading Language is designed for production rendering. Think about the vast number of things that shaders do that assume a 3-channel model: convert back and forth to HSV, tweak the red channel, generate `color noise` *(that's really 3D noise underneath), read 3 channels of texture, read channel #4 and assume it's alpha, etc. There are no scanners (outside the lab) to capture, and no paint programs to generate,* n*-channel textures. No compositing software to handle spectral images. No output media for* n*-channel images. With 3-channel input and 3-channel output, if the only* n*-channel operations are internal to shaders, users could do it with arrays if they really wanted to.*

Abstraction is nice, but so is concreteness and explicitness about what a shader is really doing and how the system really works. If all implementations are likely to be 3-components underneath, why maintain the pretense of n*-components and in the process sacrifice shader clarity and performance by pretending to support an abstraction that doesn't really exist?*

RenderMan was specified using spectral colors, but (1) nobody ever implemented it with ncomps $\neq 3$*, and (2) if you really looked hard at what would be required to do so, you quickly realized that there were all sorts of ways that a 3-component model crept into other aspects of the language, the implementation, the renderer APIs, and the shaders themselves. Actually implementing an* n*-component model would have been hugely disruptive. When I designed and implemented Gelato's shading language (GSL), I decided that, after 15 years of RenderMan never implementing spectral colors nor users requesting it, I should drop the pretense. So GSL explicitly had a 3-component color model. It was easier to implement RGB (for example, certain internals that operated on* `point`*-like types could be reused for* `color` *without embarrassment), and no user ever complainted.*

It seems to me that after 20 years of RSL having n*-colors that nobody has ever used, and 6 years of GSL leaving it out without negative consequence, we can probably assume that we can live without it for several more years, if not forever.*

Does anybody want to argue that we should nonetheless specify the language as n*-component colors? Should we use it for a while and later consider a 2.0 revision that either changes the definition of* `color` *or else adds a new* `spectral` *type that has a varying number of components, for those shaders that wish to operate that way?*

10.2.2 4-channel colors, color/alpha combinations

Some shading languages have 4-channel colors (RGBA). This spec (like RSL and GSL before it) does not.

I strongly disagree with the notion of the basic `color` *type having an embedded alpha. It mixes two concepts that don't belong together. The* `color` *type should be spectral (even if explicitly 3-channel).*

I feel less strongly about having a second type that combines a `color` *and an alpha value (for the sake of argument, let's call it a* `colora`*). The most important benefit of doing so would be to simplify the common operation of reading both color and alpha data from a 4-channel texture, which awkwardly required two separate texture reads in RenderMan's shading language. Gelato's shading language avoided this in the same manner as OSL, namely by having an argument to the texture command that allows you to return an alpha:*

```
float alphavalue;
color C = texture ("name.tx", s, t, "alpha", alphavalue);
```

This is slightly awkward inasmuch as it does not combine the color and alpha into a single "return value," but at least it avoids two texture calls, as well as avoiding cluttering the language with another type that exists just to make this texture call simpler.

But I can see that others may disagree. Does anybody want to argue for a `colora` *type that explicitly combines color and alpha as a core type?*

10.2.3 Array operations

As written, the only array operations allowed are element access (read and write), and passing an array by reference to a function. This roughly corresponds to C's and C++'s array functionality. Does anybody want to argue for any of the following to be included?

- *Array assignment:* `A = B`*, legal as long as* `A` *and* `B` *are the same type and have the same number of elements length?*
- *Casting to built-in types: Ability to assign an array of length 3 to a* `color`*,* `point`*,* `vector`*, or* `normal`*, and vice versa?*
- *Array arithmetic:* `A + B`*, and so on, defined as element-by-element addition, subtraction, etc., legal as long as the two arrays are the same type and have the same number of elements?*
- *Anything else you feel strongly that arrays should be able to do?*

10.2.4 Recursive structures

We have defined structures as being allowed to consist only of basic types and arrays thereof — not structures of structures. This restriction makes for a simpler grammar and implementation, simplier renderer-side APIs for passing data to shaders, and simpler APIs for libraries that query parameters of compiled shaders. But if people strongly feel like structures-of-structures is an important feature, we can consider generalizing in this way. Or, if we discover later that it's more helpful than we had once thought, we can always add it in future version of the spec.

10.3 Syntax

10.3.1 More C++-like overall syntax

I've considered that rather than

```
surface plastic (float Kd = 0.5, float Ks = 0.5)
{
    Ci = Kd * diffuse() + Ks * specular();
}
```

or its method-like equivalent:

```
surface plastic (float Kd = 0.5, float Ks = 0.5)
{
    public void main () {
        Ci = Kd * diffuse() + Ks * specular();
    }
}
```

that we switch to a syntax that even more closely mimics C++, such as:

```
class plastic : public surface
{
public:
    float Kd = 0.5;
    float Ks = 0.5;

    void main (output color Ci) {
        Ci = Kd * diffuse() + Ks * specular();
    }
}
```

So the idea is that instead of a shader type, you "inherit" from the kind of shader you want; instead of parameters, you have public member variables; and so on.

But this is not really clean. Notice that the assignment of default values to the "parameters" (now public data members) is not really correct C++ syntax. I guess we could move to having a constructor:

```
class plastic : public surface
{
    ...
    plastic () : Kd(0.5), Ks(0.5)
    {
    }
    ...
}
```

But is this really any better? Are we just cluttering things up in a slavish attempt to exactly replicate C++? And we leave behind the nice semantics we had before where one parameter's

default value could actually depend on another parameter's actual value. In a number of ways, OSL is NOT C++, and maybe it makes things more confusing if it looks just like C++ but doesn't act just like it, or if some things look just like C++ but others clearly aren't.

I'm currently tempted to stick to the syntax in this document, and not to exactly replicate C++'s class structure. But I think it's worth opening it to discussion, if others want to argue strongly for it.

10.4 Global variables

10.4.1 Nomenclature

Does anybody have a perference for `dPdu` *and* `dPdv` *(borrowing the RenderMan names, but awkward for primitives without a consistent* u,v *parameterization?), versus something more generic such as* `T` *and* `B` *(tangent and bitangent, without explicitly saying how they are allegedly computed)?*

In this document, I'm tentatively using the RenderMan-inspired `Ci` *and* `Oi` *for output color and opacity. (The "i" is for* incident.*) In Gelato, I used* `C` *and* `opacity`*, which I think was slightly less obtuse (though loses the advantage of being familiar to RenderMan experts). Other names are also up for grabs if people feel strongly.*

10.4.2 Global variables and view-dependence

Should we allow access to `I` *(incident viewing ray) from anyplace except inside the integrator?*

On one hand, it's just an invitation to inadvertently build view-dependence into material descriptions.

On the other hand, view-independence is less important than light-direction-independence, and we can always warn (or optionally error, if a compiler flag is set) if view dependence is detected. It's hard to tell if there are shader effects we will want that can only be done (or be drastically easier) using old fashioned view-dependent code. Perhaps we should allow it, and all due caveats and warnings to those who dare use it instead of the recommended view-independent closures.

I'm leaning toward allowing `I`*, for when it's needed, but warning and making it easy to avoid using it. Anybody want to argue that we should flat-out ban* `I` *and other view-dependencies?*

10.4.3 Normals

If you're not supposed to use `I` *or have explicit view-dependence in the shaders, then what becomes of* `faceforward()`*? Do we assume that* `N` *is already frontfacing by the time the shader runs? Do we add a* `Nf` *that's the faceforward-ed version? Do we let* `N` *be the natural normal and assume models will always be consistently-facing (which has never worked)? Force the renderer to explicitly mark "double-sided" geometry and potentially double-shade so that normals are always consistent for each "side?"*

10.5 Closures and Illumination

10.5.1 Light loops – to be, or not to be?

With closures, can we remove all facilities for an explicit loop over lights (i.e., what RenderMan calls `illuminance`*)? At least, unless/until we allow people to code integrators in the language itself?*

Or does it makes sense to allow explicit light loops, in case somebody wants/needs them, but any shader that uses them loses all ability to cheaply re-evaluate itself under moving cameras or changing lights, and possibly loses the ability to interact with smart global illumination integrators that depend on sampling their BRDF's?

My current inclination is to leave light loops out entirely, and consider adding them later if it turns out that we've made a mistake.

10.5.2 Integrators and closure implementations – in the shading language?

For now, I'm assuming that the integrator itself, as well as the implementations of the "primitive" closure functions, are a hard-coded part of the renderer itself, or implemented as plug-ins in a way that's proprietary to the renderer.

It's very tempting to extend the language itself to be able to express a primitive closure function or an integrator. But I fear that (1) this will be more expensive than we want; and (2) we don't know the best way to do it yet; (3) it will require a variety of syntaxes, functions, and complications in the language that would not otherwise be needed.

So I'm currently leaning toward leaving this out of the language for now, and considering it for a future revision. Comments?

10.6 Transparency, Opacity, and Holdout mattes

Transparency is a tricky subject. There are at least four slightly different concepts at play here: (1) Gathering illumination from the back side of a surface; (2) Proper coherent refraction of objects "behind" the primitive being shaded (usually employing ray tracing); (3) Seeing an undistored image (unrefracted) image of what's behind the primitive, usually employing a "compositing" method; (4) making an object "disappear" to form a "stencil" effect under shader control.

It is clear that #1 and #2 are properly handled with our usual material closures (`translucence` *and* `refraction()`*, respectively).*

The real question is how to handle the kind of compositing opacity of #3 and #4. The "usual" way (in renderers I've written before, anyway), is to have a special variable that holds the opacity of the surface. In RenderMan, this is called `Oi`*, in Gelato it was the less obscure* `opacity`*. It defaults to (1,1,1). The shader is responsible for scaling reflectivities by this amount. The renderer accumulates this for all subpixel samples, filters it, and reduces this to a single* alpha *for each pixel (usually by averaging the channels, or using a luminance transformation).*

There is usually a special case for holdout mattes *— objects that leave a "hole" with* $\alpha = 0$ *in the final image, yet still hide any in-render objects behind it. In RenderMan and Gelato, objects were designated as holdouts via the renderer API, not under shader control (in fact, shaders generally aren't even run on holdout objects). In PRMan and Entropy, this was extended so that*

holdout objects would run their shaders, and the computed opacity (`Oi`) was used to control partial matteness*, allowing shaders to "fade" matteness in, or use a pattern to programmatically define the matte area in the shader. It was still really hard to have full shader control over all aspects of matteing, such as wanting an object to completely block what's behind it in-scene but leave a non-zero and non-one value in the image's* alpha *channel, and it still depended on the object being tagged as "matte" through the renderer API (a clever shader couldn't do it on its own).*

I've under-specified how this should work in this document, presumably falling back to the RenderMan-like behavior as our default position. But I want to throw a few additional options out here for debate:

- *Default / RenderMan way: shader sets `Oi` < 1 for "compositing transparency", renderer auto-composites objects behind the surface (without refraction), accumulated opacity directly translates into alpha channel, holdout mattes designated through renderer API.*
- *Separate opacity/alpha controls: `Oi` controls compositing transparency; a separate `alpha` global variable can be set that becomes the opacity (if not set by the shader, it will be set to the `Oi` result). In other words, a holdout matte could be controlled by the shader with `Oi=1, alpha=0`, that blocks visibility of in-render objects, but puts a hole in the output image's alpha channel.*
- *Similar to the last item, but instead of the confusing `Oi, alpha` nomenclature, use `Oi` and `holdout` (defaulting to 0, i.e., the object is not a holdout object). Partial holdoutness can be set. The renderer handles how the holdoutness affects the image channels and compositing. So a shader would make a holdout matte with `Oi=1, holdout=1`, transparency with `Oi<1, holdout=0`.*
- *No explicit `Oi` at all (!), but rather make a material closure primitive function `transparency()` that the renderer understands. So compositing transparency could be explicit in the lighting function:*

```
opac = ...compute...;
Ci = opac * diffuse() + (1-opac) * transparency();
```

Or even, `refraction()` could serve the same purpose, and the renderer could just know that if the index of refraction is 1, it should use the compositing trick.

Holdout mattes don't fall into this scheme especially well, so there's an orthogonal problem of whether to have a `holdout` variable to set, or a renderer API function.

What do you guys think? I'm kind of leaning toward the third solution — having shaders set `Oi` and `holdout` variables, allowing full control in the shader.

Part I

Appendices

A Glossary

Attribute state. The set of variables that determines all the properties (other than shape) of a *geometric primitive* — such as its local transformation matrix, the surface, displacement, and volume shaders to be used, which light sources illuminate objects that share the attribute state, whether surfaces are one-sided or two-sided, etc. Basically all of the options that determine the behavior and appearance of the primitives, that are not determined by the shape itself or the code of the shaders. A single attribute state may be shared among multiple geometric primitives. Also sometimes called *graphics state*.

Built-in function. A function callable from within a shader, where the implementation of the function is provided by the renderer (as opposed to a function that the shader author writes in Open Shading Language itself).

Closure. A symbolic representation of a function to be called, and values for its parameters, that are packaged up to be be evaluated at a later time to yield a final numeric value.

Connection. A routing of the value of an *output parameter* of one *shader layer* to an *input paramter* of another shader layer within the same *shader group*.

Default parameter value. The initial value of a *shader parameter*, if the renderer does not override it with an *instance value*, an interpolated *primitive variable*, or a *connection* to an output parameter of another *layer* within the *group*. The default value of a shader parameter is explicitly given in the code for that shader, and may either be a constant or a computed expression.

Geometric primitive. A single shape, such as a NURBS patch, a polygon or subdivision mesh, a hair primitive, etc.

Global variables. The set of "built-in" variables describing the common renderer inputs to all shaders (as opposed to shader-specific parameters). These include position (`P`), surface normal (`N`), surface tangents (`dPdu`, `dPdv`), as well as standard radiance output (`Ci`). Different *shader types* support different subsets of the global variables.

Graphics state. See *attribute state*.

Group. See *shader group*.

Input parameter. A read-only *shader parameter* that provides a value to control a shader's behavior. Can also refer to a read-only parameter to a *shader function*.

Instance value. A constant value that overrides a *default parameter value* for a particular *shader instance*. Each instance of a shader may have a completely different set of instance values for its parameters.

Layer. See *shader layer*.

Output parameter. A read/write *shader parameter* allows a shader to provide outputs beyond the *global variables* such as `Ci`. Can also refer to a read/write parameter to a *shader function*, allowing a function to provide more outputs than a simple return value.

Primitive. Usually refers to a *geometric primitive*.

Primitive variable. A named variable, and values, attached to an individual geometric primitive. Primitive variables may have one of serveral *interpolation methods* — such as a single value for the whole primitive, a value for each piece or face of the primitive, or per-vertex values that are smoothly interpolated across the surface.

Public method. A function within a shader that has an entry point that is visible and directly callable by the renderer, as opposed to merely being called from other code within the shader. Public methods must be *top-level* (not defined within other functions) and must be preceeded by the `public` keyword.

Shader. A small self-contained program written in Open Shading Language, used to extend the functionality of a renderer with custom behavior of materials and lights. A particular shader may have multiple *shader instances* within a scene, each of which has ts unique *instance parameters*, transformation, etc.

Shader function. A function written in Open Shading Language that may be called from within a shader.

Shader group. An ordered collection of *shader instances* (individually called the *layers* of a group) that are executed to collectively determine material properties or displacement of a geometric primitive, emission of a light source, or scattering properties of a volume. In addition to executing sequentially, layers within a group may optionally have any of their input parameters explicitly connected to output parameters of other layers within the group in an acyclic manner (thus, sometimes being referred to as a *shader network*).

Shader instance. A particular reference to a *shader*, with a unique set of `instance values`, transformation, and potentially other attributes. Each shader instance is a separate entity, despite their sharing executable code.

Shader network. See *shader group*.

Shader layer. An individual *shader instance* within a *shader group*.

Shader parameter. A named input or output variable of a shader. *Input parameters* provide "knobs" that control the behavior of a shader; *output parameters* additional provide a way for shaders to produce additional output beyond the usual *global variables*.

Shading. The computations within a renderer that implement the behavior and visual appearance of materials and lights.

Index