Open Shading Language

Release 1.14.1

Larry Gritz, editor

May 11, 2024

INTRODUCTION


open
shading
language

CHAPTER

ONE

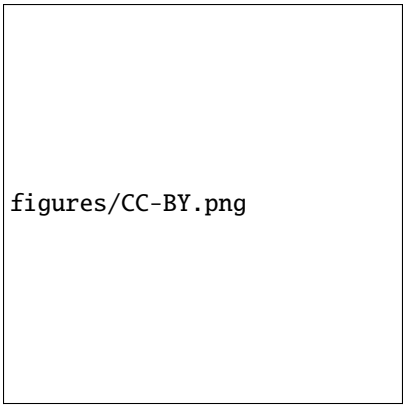
COPYRIGHT NOTICE

The code that implements Open Shading Language is licensed under the BSD 3-clause (also sometimes known as "new BSD" or "modified BSD") license.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This manual and other text documentation about OSL are licensed under the Creative Commons Attribution 4.0 International License.



http://creativecommons.org/licenses/by/4.0/

OSL incorporates code from several other software packages with compatible licenses. Copies of their licenses are reproduced here: https://github.com/AcademySoftwareFoundation/OpenShadingLanguage/blob/master/THIRD-PARTY.md

(chap-introduction=)

CHAPTER

TWO

INTRODUCTION

Welcome to Open Shading Language!

Open Shading Language (OSL) is a small but rich language for programmable shading in advanced renderers and other applications, ideal for describing materials, lights, displacement, and pattern generation.

OSL was developed by Sony Pictures Imageworks for use in its in-house renderer used for feature film animation and visual effects. The language specification was developed with input by other visual effects and animation studios who also wish to use it.

2.1 How OSL is different from other shading languages

OSL has syntax similar to C, as well as other shading languages. However, it is specifically designed for advanced rendering algorithms and has features such as radiance closures, BSDFs, and deferred ray tracing as first-class concepts.

OSL has several unique characteristics not found in other shading languages (certainly not all together). Here are some things you will find are different in OSL compared to other languages:

- **Surface and volume shaders compute radiance closures, not final colors.**

  OSL's surface and volume shaders compute an explicit symbolic description, called a "closure," of the way a surface or volume scatters light, in units of radiance. These radiance closures may be evaluated in particular directions, sampled to find important directions, or saved for later evaluation and re-evaluation. This new approach is ideal for a physically-based renderer that supports ray tracing and global illumination.

  In contrast, other shading languages usually compute just a surface color as visible from a particular direction. These old shaders are "black boxes" that a renderer can do little with but execute to for this once piece of information (for example, there is no effective way to discover from them which directions are important to sample). Furthermore, the physical units of lights and surfaces are often underspecified, making it very difficult to ensure that shaders are behaving in a physically correct manner.

- **Surface and volume shaders do not loop over lights or shoot rays.**

  There are no "light loops" or explicitly traced rays in OSL surface shaders. Instead, surface shaders compute a radiance closure describing how the surface scatters light, and a part of the renderer called an "integrator" evaluates the closures for a particular set of light sources and determines in which directions rays should be traced. Effects that would ordinarily require explicit ray tracing, such as reflection and refraction, are simply part of the radiance closure and look like any other BSDF.

  Advantages of this approach include that integration and sampling may be batched or re-ordered to increase ray coherence; a "ray budget" can be allocated to optimally sample the BSDF; the closures may be used by for

bidirectional ray tracing or Metropolis light transport; and the closures may be rapidly re-evaluated with new lighting without having to re-run the shaders.

- **Surface and light shaders are the same thing.**

  OSL does not have a separate kind of shader for light sources. Lights are simply surfaces that are emissive, and all lights are area lights.

- **Transparency is just another kind of illumination.**

  You don't need to explicitly set transparency/opacity variables in the shader. Transparency is just another way for light to interact with a surface, and is included in the main radiance closure computed by a surface shader.

- **Renderer outputs (AOV's) are specified using "light path expressions."**

  Sometimes it is desirable to output images containing individual lighting components such as specular, diffuse, reflection, individual lights, etc. In other languages, this is usually accomplished by adding a plethora of "output variables" to the shaders that collect these individual quantities.

  OSL shaders need not be cluttered with any code or output variables to accomplish this. Instead, there is a regular-expression-based notation for describing which light paths should contribute to which outputs. This is all done on the renderer side (though supported by the OSL implementation). If you desire a new output, there is no need to modify the shaders at all; you only need to tell the renderer the new light path expression.

- **Shaders are organized into networks.**

  OSL shaders are not monolithic, but rather can be organized into networks of shaders (sometimes called a shader group, graph, or DAG), with named outputs of some nodes being connected to named inputs of other nodes within the network. These connections may be done dynamically at render time, and do not affect compilation of individual shader nodes. Furthermore, the individual nodes are evaluated lazily, only their outputs are "pulled" from the later nodes that depend on them (shader writers may remain blissfully unaware of these details, and write shaders as if everything is evaluated normally).

- **No "uniform" and "varying" keywords in the language.**

  OSL shaders are evaluated in SIMD fashion, executing shaders on many points at once, but there is no need to burden shader writers with declaring which variables need to be uniform or varying.

  In the open source OSL implementation, this is done both automatically and dynamically, meaning that a variable can switch back and forth between uniform and varying, on an instruction-by-instruction basis, depending on what is assigned to it.

- **Arbitrary derivatives without grids or extra shading points.**

  In OSL, you can take derivatives of any computed quantity in a shader, and use arbitrary quantities as texture coordinates and expect correct filtering. This does not require that shaded points be arranged in a rectangular grid, or have any particular connectivity, or that any "extra points" be shaded.

  In the open source OSL implementation, this is possible because derivatives are not computed by finite differences with neighboring points, but rather by "automatic differentiation," computing partial differentials for the variables that lead to derivatives, without any intervention required by the shader writer.

2.2 Acknowledgments

The original designer and project leader of OSL is Larry Gritz. Other early developers of OSL are (in order of joining the project): Cliff Stein, Chris Kulla, Alejandro Conty, Jay Reynolds, Solomon Boulos, Adam Martinez, Brecht Van Lommel.

Additionally, many others have contributed features, bug fixes, and other changes: Steve Agland, Shane Ambler, Martijn Berger, Farchad Bidgolirad, Nicholas Bishop, Stefan Büttner, Matthaus G. Chajdas, Thomas Dinges, Henri Fousse, Syoyo Fujita, Derek Haase, Sven-Hendrik Haase, John Haddon, Daniel Heckenberg, Ronan Keryell, Elvic Liang, Max Liani, Bastien Montagne, Erich Ocean, Mikko Ohtamaa, Alex Schworer, Sergey Sharybin, Stephan Steinbach, Esteban Tovagliari, Alexander von Knorring, Roman Zulak. (Listed alphabetically; if we've left anybody out, please let us know.)

We cannot possibly express sufficient gratitude to the managers at Sony Pictures Imageworks who allowed this project to proceed, supported it wholeheartedly, and permitted us to release the source, especially Rob Bredow, Brian Keeney, Barbara Ford, Rene Limberger, and Erik Strauss.

Huge thanks also go to the crack shading team at SPI, and the brave lookdev TDs and CG supes willing to use OSL on their shows. They served as our guinea pigs, inspiration, testers, and a fantastic source of feedback. Thank you, and we hope we've been responsive to your needs.

OSL was not developed in isolation. We owe a debt to the individuals and studios who patiently read early drafts of the language specification and gave us very helpful feedback and additional ideas, and especially to those at other companies who have taken the risk of incorporating OSL into their products and pipelines.

The open source OSL implementation incorporates or depends upon several other open source packages:

- **OpenImageIO** Copyright Contributors to OpenImageIO. http://openimageio.org
- **Imath** Copyright Contributors to Imath. http://www.openexr.com
- **LLVM** Copyright 2003-2010 University of Illinois at Urbana-Champaign. http://llvm.org

These other packages are all distributed under licenses that allow them to be used by and distributed with OSL.

(chap-bigpicture=)

CHAPTER

THREE

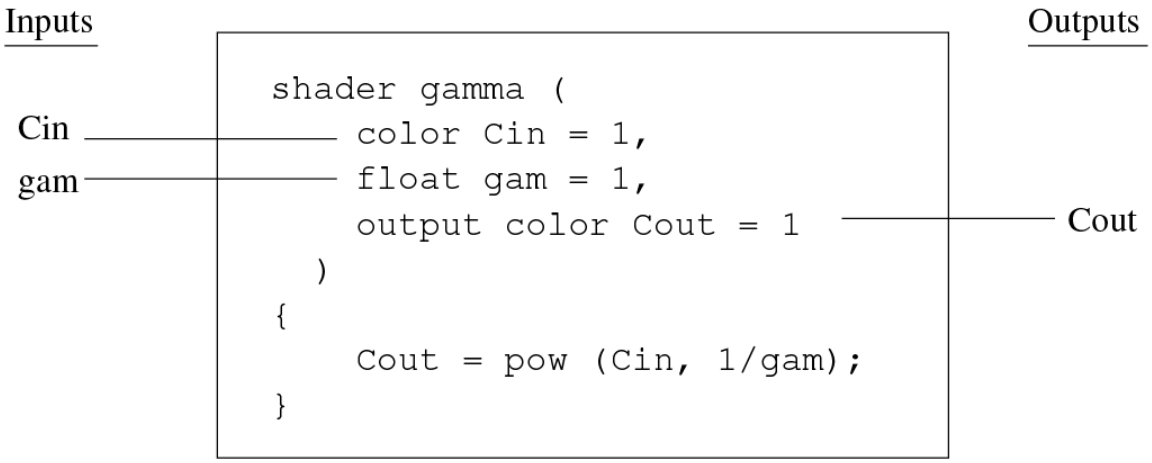
THE BIG PICTURE

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

3.1 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 OSL, 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`:

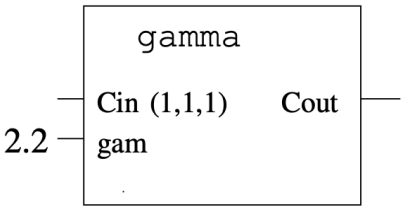


The shader's inputs and outputs are called *shader parameters*.
Parameters have default values, specified in the shader code, but may also be given new values by the renderer at runtime.

3.2 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*.

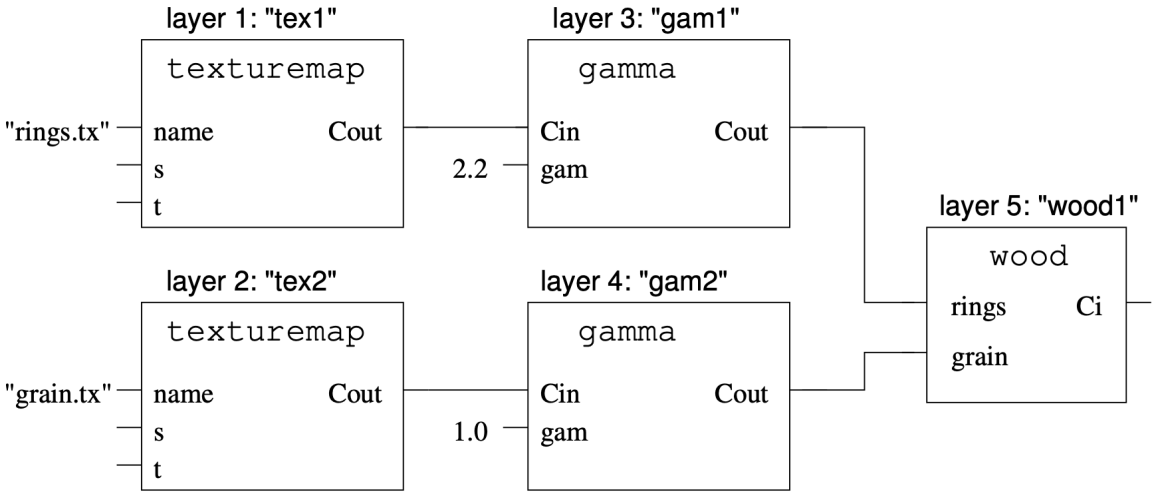
Below is a schematic showing a `gamma` instance with the `gam` parameter overridden with an instance-specific value of `2.2`.



3.3 Shader groups, layers, and connections

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")
Shader ("gamma", "gam1", "float gam", 2.2)
Shader ("gamma", "gam2", "float gam", 1)
```

(continues on next page)

[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.

(continued from previous page)

```
    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 ()
```

Or, expressed as serialized text (as detailed in Chapter *Describing shader groups*:

```
param string name "rings.tx" ;
shader "texturemap" "tex1" ;
param string name "grain.tx" ;
shader "texturemap" "tex2" ;
param float gam 2.2 ;
shader "gamma" "gam1" ;
param float gam 1.0 ;
shader "gamma" "gam2" ;
shader "wood" "wood1" ;
connect tex1.Cout gam1.Cin ;
connect tex2.Cout gam2.Cin ;
connect gam1.Cout wood1.rings ;
connect gam2.Cout wood1.grain ;
```

The rules for which data types may be connected are generally the same as the rules determining which variables may be assigned to each other in OSL source code:

- `source` and `dest` are the same data type.
- `source` and `dest` are both *triples* (`color`, `point`, `vector`, or `normal`), even if they are not the same kind of triple.
- `source` is an `int` and `dest` is a `float`.
- `source` is a `float` or `int` and `dest` is a *triple* (the scalar value will be replicated for all three components of the triple).
- `source` is a single component of an aggregate type (e.g. one channel of a `color`) and `dest` is a `float` (or vice versa).

3.4 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* (also sometimes called *interpolated values* or *user data*). 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.

3.5 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 or emits from each point on a shape, displacement shaders that can add fine detail to the shape on a point-by-point basis, and volume shaders that describe how light is scattered within a region of space. A particular renderer may have additional shader types that it supports.

3.6 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.).

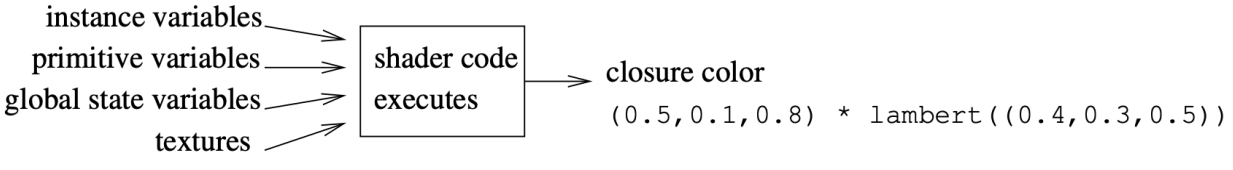
3.7 Surface and volume shaders compute closures

Surface shaders (and volume shaders) do not by themselves compute the final color of light emanating from the surface (or along a volume). Rather, they 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 * diffuse (N);
```

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

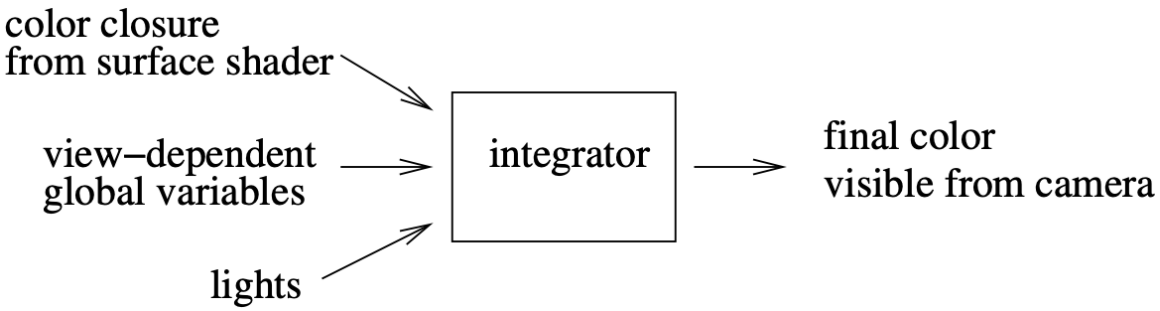


The closures output by surface and volume shaders can do a number of interesting things that a mere number cannot:

- 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.).

3.8 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.



3.9 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.

CHAPTER

FOUR

LEXICAL STRUCTURE

4.1 Characters

Source code for OSL 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.

4.2 Identifiers

Identifiers are the names of variables, parameters, functions, and shaders. In OSL, 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 underscore

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

4.3 Comments

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

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
```

4.4 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 OSL:

```
and break closure color continue do else emit float for if illuminance
illuminate int matrix normal not or output point public return string
struct vector void while
```

The following are *reserved words* that currently have no special meaning in OSL, but we reserve them for possible future use, or because 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
```

4.5 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.

OSL compilers support the full complement of C/C++ preprocessing directives, including:

```
#define
#undef
#if
#ifdef
#ifndef
#elif
#else
#endif
#include
#pragma error "message"
#pragma once
#pragma osl ...
#pragma warning "message"
```

Additionally, the following preprocessor symbols will already be defined by the compiler:

`OSL_VERSION_MAJOR`	Major version (e.g., 1)
`OSL_VERSION_MINOR`	Minor version (e.g., 9)
`OSL_VERSION_PATCH`	Patch version (e.g., 3)
`OSL_VERSION`	Combined version number = `10000*major + 100*minor + patch` (e.g., 10903 for version 1.9.3)

CHAPTER

FIVE

GROSS SYNTAX, SHADER TYPES, PARAMETERS

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.).

5.1 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 `color closure` 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).

5.2 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.

5.2.1 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 an OSL function (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 evaluated 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 *Data types*, *parametername* is the name of the parameter, and *default-expression* is a valid expression (see Section *Expressions*). Multiple parameters are simply separated by commas:

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.

5.2.2 Shader output parameters

Shader parameters are, by default, read-only in the body of the shader. However, special \emph{output parameters} may be altered by execution of the shader. Parameters may be designated outputs by use of the {\cf 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).

5.2.3 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
             )
{
   ...
}
```

5.2.4 How shader parameters get their values

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

1. 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.
2. 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 parameter's value will be computed by interpolating the primitive variable for each position that must be shaded.
3. 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).
4. 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 `ringcolor1 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.

5.3 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 or arrays of simple types (not structs or closures) and their value initializers must be numeric or string constants (not computed expressions).

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 help = "Realistic wood shader" ]]
    (
       float Kd = 0.5
           [[ string help = "Diffuse reflectivity",
              float min = 0, float max = 1 ]] ,
       color woodcolor = color (.7, .5, .3)
           [[ string help = "Base color of the wood" ]],
       color ringcolor = 0.25 * woodcolor
           [[ string help = "Color of the dark rings" ]],
       string texturename = "wood.tx"
           [[ string help = "Texture map for the grain",
              string widget = "filename" ]],
       int pattern = 0
           [[ string widget = "mapper",
              string options = "oak:0|elm:1|walnut:2" ]]
    )
{
   ...
}
```

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.

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.

string label
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.

string help
Help text that describes the purpose and use of the shader or parameter.

string page
Helps to group related widgets by ``page."

string widget
The type of widget that should be used to adjust this parameter. Suggested widget types:

"number"
Provide a slider and/or numeric input. This is the default widget type for `float` or `int` parameters. Numeric inputs also may be influenced by the following metadata: `min`, `max`, `sensitivity`, `digits`, `slider`, `slidermin`, `slidermax`, `slidercenter`, `sliderexponent`.

"string"
Provide a text entry widget. This is the default widget type for {\cf string} parameters.

"boolean"
Provide a pop-up menu with "Yes" and "No" options. Works on strings or numbers. With strings, "Yes" and "No" values are used, with numbers, 0 and 1 are used.

"checkBox"
A boolean widget displayed as a checkbox. Works on strings or numbers. With strings, "Yes" and "No" values are used, with numbers, 0 and 1 are used.

"popup"
A pop-up menu of literal choices. This widget further requires parameter metadata `options` (a string listing the supported menu items, delimited by the `|` character), and optionally `editable` (an integer, which if nonzero means the widget should allow the text field should be directly editable). For example:

```
string wrap = "default"
    [[ string widget = "popup",
       string options = "default|black|clamp|periodic|mirror" ]]
```

"mapper"
A pop-up with associative choices (an enumerated type, if the values are integers). This widget further requires parameter metadata `options`, a `|`-delimited string with "key:value" pairs. For example:

```
int pattern = 0
    [[ string widget = "mapper",
       string options = "oak:0|elm:1|walnut:2" ]]
```

"filename"
A file selection dialog.

"null"
A hidden widget.

float min, float max, int min, int max
The minimum and/or maximum value that the parameter may take on.

float sensitivity, int sensitivity
The precision or step size for incrementing or decrementing the value (within the appropriate min/max range).

int digits
The number of digits to show (-1 for full precision).

int slider
If nonzero, enables display of a slider sub-widget. This also respects the following additional metadata that control the slider specifically: `slidermin` (minimum value for the slider, `slidermax` (maximum value for the slider), `slidercenter` (origin value for the slider), `sliderexponent` (nonlinear slider options).

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

string units
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`).

CHAPTER

SIX

DATA TYPES

OSL provides several built-in simple data types for performing computations inside your shader:

Type	Explanation
`int`	Integer data
`float`	Scalar floating-point data (numbers)
`point`, `vector`, `normal`	Three-dimensional positions, directions, and face 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 OSL 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 OSL.

6.1 `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`, `-2`, etc. Integer constants may be specified as hexadecimal, for example: `0x01cf`.

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	Explanation
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)

continues on next page

Table 1 – continued from previous page

Operation	Result	Explanation
! *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)

Note that the `not`, `and`, and `or` keywords are synonyms for `!`, `&&`, and `||`, respectively.

6.2 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	Explanation
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

6.3 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. The following table lists the built-in 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 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)
```

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.

Color variables may also have their components referenced using "named components" that look like accessing structure fields named `r`, `g`, and `b`, as synonyms for `[0]`, `[1]`, and `[2]`, respectively:

```
float green = C.g;   // get the green component
C.r = 0.5;           // set the red component
```

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	Explanation
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.

6.4 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 "common" space. For example, it is much more convenient to apply a "solid texture" to a moving object in its "object" space than in "common" space. For these reasons, OSL provides a built-in `transform()` function that allows you to transform points, vectors, and normals among different coordinate systems (see Section *Geometric functions*). Note, however, that OSL 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 the following table. 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.

"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.

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
```

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.

Point-like variables may also have their components referenced using "named components" that look like accessing structure fields named `x`, `y`, and `z`, as synonyms for `[0]`, `[1]`, and `[2]`, respectively:

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

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	Explanation
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.

6.5 matrix

OSL 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 "shader" space to "common" space. Transforming points by this matrix is identical to calling `transform("shader", "common", ...)`. 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 *Standard Library Functions*.

Individual components 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	Explanation
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

6.6 `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"`.

6.7 `void`

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

6.8 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 uninitialized 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;
}
```

It is allowed in OSL to copy an entire array at once using the = operator, provided that the arrays contain elements of the same type and that the destination array is at least as long as the source array. For example:

```
float array[4], anotherarray[4];
...
anotherarray = array;
```

6.9 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 RGBA {                     // Define a structure type
    color rgb;
    float alpha;
};

RGBA col;                         // Declare a structure
r.rgb = color (1, 0, 0);          // Assign to one field
color c = r.rgb;                  // Read from a structure field

RGBA b = { color(.1,.2,.3), 1 }; // Member-by-member initialization
```

You can use "constructor expressions" for a your struct types much like you can construct built-in types like `color` or `point`:

struct_name `(` *first_member_value*, ... `}`

For example,

```
RGBA c = RGBA(col,alpha);        // Constructor syntax

RGBA add (RGBA a, RGBA b)
{
    return RGBA (a.rgb+b.rgb, a.a+b.a);   // return expression
}

// pass constructor expression as a parameter:
RGBA d = add (c, RGBA(color(.3,.4,.5), 0));
```

You may also use the *compound initializer list* syntax to construct a type when it can be deduced from context which compound type is required. For example, this is equivalent to the preceding example:

```
RGBA c = {col,alpha};    // deduce by what is being assigned to

RGBA add (RGBA a, RGBA b)
{
    return { a.rgb+b.rgb, a.a+b.a }; // deduce by func return type
}

RGBA d = add (c, {{.3,.4,.5}, 0}); // deduce by expected arg type
```

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 currently permitted to "nest" arrays (that is, to have an array of structs which contain members that are arrays).

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

RGBA vals[4];         // Ok: Array of structures
vals[0].a = 0;

A d[5];               // NO: Array of structures that contain arrays
```

6.10 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 `color closure` behaves mostly like a color — you can multiply it by a scalar, assign it to a `color closure` 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 OSL is the `color closure`, and the only allowed operations are those that let you form a linear combination of `color closure`'s. Additional closure types and operations are reserved for future use.

Allowable operations on `color closure` include:

Operation	Result	Explanation
- *colorclosure*	`color closure`	unary negation
color * *colorclosure*	`color closure`	component-wise scaling
colorclosure * *color*	`color closure`	component-wise scaling
float * *colorclosure*	`color closure`	scaling
colorclosure * *float*	`color closure`	scaling
colorclosure + *colorclosure*	`color closure`	component-wise addition

CHAPTER

SEVEN

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 OSL.

Statements in OSL include the following types of constructs:

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

7.1 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 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
```

7.2 Variable declarations and assignments

7.2.1 Variable declarations

The syntax for declaring a variable in OSL 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
```

7.2.2 Arrays

Arrays are also supported, declared as follows:

type variablename [*arraylen*]

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

Array variables in OSL 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 uninitialized array
float c[3] = { 0.1, 0.2, 3.14 };  // Initialize the array
```

7.2.3 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:

\spc `struct` *structname* {\cf {}

\spc\spc *type1 fieldname1* ;

\spc\spc ...

\spc\spc *typeN fieldnameN* ;

\spc) ;

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

\spc *structname variablename* ;

\spc *structname variablename* {\cf = { } *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:

\spc *variablename*{\cf .} *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~\ref{sec:types:struct} for more information on using `struct`.

7.3 Expressions

The expressions available in OSL 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:

Operator	Meaning
varref `++`	post-increment
varref `--`	post-decrement
`++` *varref*	pre-increment
`--` *varref*	pre-decrement

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

- Unary and binary arithmetic operators on other expressions:

Operator	Meaning
- *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 multi-component 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 *Data types*.

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

Operator	Meaning
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*

Note that the `not`, `and`, and `or` keywords are synonyms for !, &&, and ||, respectively.

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` :

Operator	Meaning
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 *Data types*.

- 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 *Data types*, where the section describing each data type describes the full complement of operators that may be used with the type. Operator precedence in OSL is identical to that of C.

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

Conditionals in OSL just like in C or C++:

\begin{tabbing} \hspace{0.5in} = \hspace{0.3in} = \kill > {\cf if (} *condition* {\cf)} \ > > *truestatement*
\end{tabbing}

and

\begin{tabbing} \hspace{0.5in} = \hspace{0.3in} = \kill > {\cf if (} *condition* {\cf)} \ > > *truestatement* \ > {\cf else}
\ > > *falsestatement*
\end{tabbing}

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 considered "true" if it is a nonempty string, "false" if it is the empty string ("").
- A closure, which is considered "true" if it's empty (not assigned, or initialized with {\cf =0}), and "false" if anything else has been assigned to it.
- 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* `)` *statements*

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-statements

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

7.5 Functions

7.5.1 Function definitions

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 OSL 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 function at that point, and designates the value that will be returned (if not a `void` function).

Functions may be *overloaded*. 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 select which version of the function is desired. When there are multiple potential matches, function versions whose argument types match exactly are favored, followed by those that match with type coercion (for example, passing an `int` when the function expects a `float`, or passing a `float` when the function expects a `color`), and finally by trying to match the return value to the type of variable the result is assigned to.

7.5.2 Function calls

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 OSL, 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.

Certain functions allow optional arguments to be passed.
Optional arguments are key-value pairs with the key passed as an argument and the associated value passed as the subsequent argument:

functionname (*arg1* , ... , *argn* , *"optionalkey"* , *optionalvalue* , ...)

7.5.3 Operator overloading

OSL permits \emph{operator overloading}, which is the practice of providing a function that will be called when you use an operator like + or *. This is especially handy when you use `struct` to define mathematical types and wish for the usual math operators to work with them. Here is a typical example, which also shows the special naming convention that allows operator overloading:

```
struct vector4 {
    float x, y, z, w;
};

vector4 __operator__add__ (vector4 a, vector4 b) {
    return vector4 (a.x+b.x, a.y+b.y, a.z+b.z, a.w+b.w);
}

shader test ()
{
    vector4 a = vector4 (.2, .3, .4, .5);
    vector4 b = vector4 (1, 2, 3, 4);

    vector4 c = a + b;   // Will call __operator__add__(vector4,vector4)
    printf ("a+b = %g %g %g %g\n", c.x, c.y, c.z, c.w);
}
```

The full list of these special function names is as follows (in order of decreasing operator precedence):

Operator	Overload function name	notes
-	__operator__neg__	unary negation
~	__operator__compl__	unary bitwise
!	__operator__not__	unary boolean 'not'
*	__operator__mul__	
/	__operator__div__	
\%	__operator__mod__	
+	__operator__add__	
-	__operator__sub__	
<<	__operator__shl__	
>>	__operator__shr__	
<	__operator__lt__	
<=	__operator__le__	
>	__operator__gt__	
>=	__operator__ge__	
==	__operator__eq__	
!=	__operator__ne__	
\&	__operator__bitand__	
^	__operator__xor__	
|	__operator__bitor__	

7.6 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 the following table:

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 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`.
`point Ps`	Position at which the light is being queried (currently only used for light attenuation shaders)
`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 radiance — a closure representing the color of the light leaving the surface from `P` in the direction `-I`.

Accessibility of variables by shader type:

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

CHAPTER

EIGHT

STANDARD LIBRARY FUNCTIONS

8.1 Basic math functions

8.1.1 Mathematical constants

OSL defines several mathematical constants:

Constant	Value
`M_PI`	π
`M_PI_2`	$\pi/2$
`M_PI_4`	$\pi/4$
`M_2_PI`	$2/\pi$
`M_2PI`	2π
`M_4PI`	4π
`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`	$\sqrt{1/2}$

8.1.2 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, or tangent of `x` (measured in radians).

void sincos (***type*** **x**, **output** ***type*** **sinval**, **output** ***type*** **cosval**)
Computes both the sine and cosine of x (measured in radians). If both are needed, this function is less expensive than calling `sin()` and `cos()` separately.

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 **logb** (***type*** **x**)
Returns the exponent of x, as a floating-point number.

type **sqrt** (***type*** **x**) ***type*** **inversesqrt** (***type*** **x**)
Computes $\sqrt{x}$ and $1/\sqrt{x}$. Returns 0 if $x < 0$.

type **cbrt** (***type*** **x**)
Computes $\sqrt[3]{x}$. The sign of the return value will match x.

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.)

type **sign** (***type*** **x**)
Returns 1 if $x > 0$, -1 if $x < 0$, 0 if $x = 0$.

type **floor** (***type*** **x**) ***type*** **ceil** (***type*** **x**) ***type*** **round** (***type*** **x**) ***type*** **trunc** (***type*** **x**)
Various rounding methods: `floor` returns the largest integer less than or equal to x; `ceil` returns the smallest integer greater than or equal to x; `round` returns the closest integer to x, in either direction (rounding away from 0 in cases where x is exactly half way between integers); and `trunc` returns the integral part of x (equivalent to `floor` if $x > 0$ and `ceil` if $x < 0$).

type **fmod** (***type*** **a**, ***type*** **b**) ***type*** **mod** (***type*** **a**, ***type*** **b**)
The `fmod()` function returns the floating-point remainder of a/b, i.e., is the floating-point equivalent of the integer `%` operator. It is nearly identical to the C or C++ `fmod` function, except that in OSL, `fmod(a,0)` returns `0`, rather than `NaN`. Note that if $a < 0$, the return value will be negative.

The `mod()` function returns $a - b * \text{floor}(a/b)$, which will always be a positive number or zero.

As an example, `fmod(-0.25,1.0) = -0.25`, but `mod(-0.25,1.0) = 0.75`. For positive `a` they return the same value.

For both functions, the *type* may be any of `float`, `point`, `vector`, `normal`, or `color`.

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: $x * (1 - \alpha) + y * (\alpha)$

type **select** (***type*** **x**, ***type*** **y**, ***type*** **cond**) ***type*** **select** (***type*** **x**, ***type*** **y, float cond**) ***type*** **select** (***type*** **x**, ***type*** **y, int cond**)

The `select` function returns `x` if `cond` is zero, or `y` if `cond` is nonzero. This is roughly equivalent to `(cond ? y : x)`, except that if `cond` is a component-based type (such as `color`), the selection happens on a component-by-component basis. It is presumed that the underlying implementation is not a true conditional and will not incur any branching penalty.

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

8.2 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., $A \cdot B = A_x B_x + A_y B_y + A_z B_z$.

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`.

vector normalize (vector V) normal 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 OSL:

```
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 OSL 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) ? vector(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) point rotate (point Q, float angle, vector axis)

Returns the point computed by rotating point `Q` by `angle` radians about the axis that passes from point `P0` to `P1`, or about the `axis` vector centered on the origin.

***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.

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.

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.

8.3 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 blackbody (float temperatureK)
The `blackbody()` function returns the blackbody emission (the incandescent glow of warm bodies) expected from a material of the given temperature in Kelvin, in units of W/m^2. Note that `emission()` has units of radiance, so will require a scaling factor of $1/\pi$ on surfaces, and $1/4\pi$ on volumes to convert to $W/m^2/sr$.

color wavelength_color (float wavelength_nm)
Returns an RGB color corresponding as closely as possible to the perceived color of a pure spectral color of the given wavelength (in nm).

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.

8.4 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. If the coordinate system name is unknown, it will be assumed to be the identity matrix.

Note that `matrix (space, 1)` returns the *space*-to-common transformation matrix. If the coordinate system name is unknown, it will be assumed to be the identity matrix.

matrix matrix (string fromspace, string tospace)
Constructs a `matrix` that can be used to transform coordinates from `fromspace` to `tospace`. If either of the coordinate system names are unknown, they will be assumed to be the identity matrix.

int getmatrix (string fromspace, string tospace, output matrix M)
Sets `M` to the `matrix` that transforms coordinates from `fromspace` to `tospace`. Return 1 upon success, or 0 if either of the coordinate system names are unknown (in which case `M` will not be modified). This is very similar to the `matrix(from,to)` constructor, except that `getmatrix()` allows the shader to gracefully handle unknown coordinate system names.

float determinant (matrix M)
Computes the determinant of matrix `M`.

matrix transpose (matrix M)
Computes the transpose of matrix `M`.

8.5 Pattern generation

float step (float edge, float x) *type* step (*type* edge, *type* x)
Returns 0 if $x < edge$ and 1 if $x \geq edge$.

The *type* may be any of 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).

float linearstep (float edge0, float edge1, float x) *type* linearstep (*type* edge0, *type* edge1, *type* x)
Returns 0 if `x` $\leq$ `edge0`, and 1 if `x` $\geq$ `edge1`, and performs a linear interpolation between 0 and 1 when `edge0` $<$ `x` $<$ `edge1`. This is equivalent to `step(edge0, x)` when `edge0 == edge1`. For `color` and `point`-like types, the computations are performed component-by-component (separately for x, y, and z).

float smoothstep (float edge0, float edge1, float x) *type* smoothstep (*type* edge0, *type* edge1, *type* 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.

The *type* may be any of of `float`, `color`, `point`, `vector`, or `normal`. For `color` and `point`-like types, the computations are performed component-by-component.

float smooth_linearstep (float edge0, float edge1, float x, float eps) ***type*** **smooth_linearstep (*type* edge0, *type* edge1, *type* x, *type* eps)**

This function is strictly linear between `edge0 + eps` and `edge1 - eps` but smoothly ramps to 0 between `edge0 - eps` and `edge0 + eps` and smoothly ramps to 1 between `edge1 - eps` and `edge1 + eps`. It is 0 when `x` $\leq$ `edge0-eps`, and 1 if `x` $\geq$ `edge1 + eps`, and performs a linear interpolation between 0 and 1 when `edge0` < x < `edge1`. For `color` and `point`-like types, the computations are performed component-by-component.

type **noise (string noisetype, float u, ...)** ***type*** **noise (string noisetype, float u, float v, ...)** ***type*** **noise (string noisetype, point p, ...)** ***type*** **noise (string noisetype, 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`), with a return value of either 1D (`float`) or 3D (`color`, `point`, `vector`, or `normal`).

The `noisename` specifies which of a variety of possible noise functions will be used:

"perlin", "snoise"
: A signed Perlin-like gradient noise with an output range of $[-1, 1]$, approximate average value of 0, and is exactly 0 at integer lattice points. This is equivalent to the `snoise()` function.

"uperlin", "noise"
: An unsigned Perlin-like gradient noise with an output range of $(0, 1)$, approximate average value of 0.5, and is exactly 0.5 at integer lattice points. This is equivalent to the `noise()` function (the one that doesn't take a name string).

"cell"
: A discrete function that is constant on $[i, i + 1)$ for all integers i (i.e., `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 $[0, 1]$.

"hash"
: A function that returns a different, uncorrelated (but deterministic and repeatable) value at every real input coordinate. The range is $[0, 1]$ its large-scale average is 0.5, and its values are evenly distributed over $[0, 1]$.

"simplex"
: A signed simplex noise with an output range of $[-1, 1]$, approximate average value of 0.

"usimplex"
: An unsigned simplex noise with an output range of $[0, 1]$, approximate average value of 0.5.

"gabor"
: A band-limited, filtered, sparse convolution noise based on the Gabor impulse function (see Lagae et al., SIGGRAPH 2012). Our Gabor noise is designed to have somewhat similar frequency content and range as Perlin noise (range $[-1, 1]$, approximately large-scale average of 0). It is significantly more expensive than Perlin noise, but its advantage is that it correctly filters automatically based on the input derivatives. Gabor noise allows several optional parameters to the `noise()` call:

  "anisotropic", ***int*** **"direction",** ***vector***
  : If `anisotropic` is 0 (the default), Gabor noise will be isotropic. If `anisotropic` is 1, the Gabor noise will be anisotropic with the 3D frequency given by the `direction` vector (which defaults to `(1,0,0)`). If `anisotropic` is 2, a hybrid mode will be used which is anisotropic along the `direction` vector, but radially isotropic perpendicular to that vector. The `direction` vector is not used if `anisotropic` is 0.

  "bandwidth", ***float***
  : Controls the bandwidth for Gabor noise. The default is 1.0.

  "impulses", ***float***
  : Controls the number of impulses per cell for Gabor noise. The default is 16.

"do_filter", *int*
: If do_filter is 0, no filtering/antialiasing will be performed. The default is 1 (yes, do filtering). There is probably no good reason to ever turn off the filtering, it is primarily to test that the filtering is working properly.

Note that some of the noise varieties have an output range of $[-1, 1]$ but others have range $[0, 1]$; some may automatically antialias their output (based on the derivatives of the lookup coordinates) and others may not, and various other properties may differ. The user should be aware of which noise varieties are useful in various circumstances.

A particular renderer's implementation of OSL may supply additional noise varieties not described here.

The noise() functions take optional arguments after their coordinates, passed as token/value pairs (similarly to optional texture arguments). Generally, such arguments are specific to the type of noise, and are ignored for noise types that don't understand them.

***type* pnoise (string noisetype, float u, float uperiod)** ***type* pnoise (string noisetype, float u, float v, float uperiod, float vperiod)** ***type* pnoise (string noisetype, point p, point pperiod)** ***type* pnoise (string noisetype, point p, float t, point pperiod, float tperiod)**
: Periodic version of noise(), in which the domain wraps with the given period(s). Generally, only integer-valued periods are supported.

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

***type* snoise (float u)** ***type* snoise (float u, float v)** ***type* snoise (point p)** ***type* snoise (point p, float t)**
: The old noise(...coords...) function is equivalent to noise("uperlin",...coords...) and snoise(...coords...) is equivalent to noise("perlin",...coords...).

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 old pnoise(...coords...) function is equivalent to pnoise("uperlin",...coords...) and psnoise(...coords...) is equivalent to pnoise("perlin",...coords...).

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

: The old cellnoise(...coords...) function is equivalent to noise("cell",...coords...).

***type* hashnoise (float u)** ***type* hashnoise (float u, float v)** ***type* hashnoise (point p)** ***type* hashnoise (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 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).

int hash (float u) int hash (float u, float v) int hash (point p) int hash (point p, float t) int hash (int i)
: Returns a deterministic, repeatable integer hash of the 1-, 2-, 3-, or 4-D coordinates.

type* spline (string basis, float x,** ***type y_0**,** ***type*** y_1**, ...** ***type*** y_{n-1}**)** ***type* spline (string basis, float x,** ***type* y[])** ***type* spline (string basis, float x, int nknots,** ***type* y[])**
: As x varies from 0 to 1, spline returns the value of a cubic interpolation of uniformly-spaced knots $y_0 \ldots y_{n-1}$, or $y[0] \ldots y[n-1]$ for the array version of the call (where n is the length of the array), or $y[0] \ldots y[nknots-1]$ for the version that explicitly specifies the number of knots (which may be less than the full array length). 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 parameter, which may be any of: "catmull-rom", "bezier", "bspline", "hermite", "linear", or "constant". Some basis types require particular numbers of knot

values – Bezier splines require $3n + 1$ values, Hermite splines require $2n + 2$ values, and all of Catmull-Rom, linear, and constant requires $3 + n$, where in all cases, $n \geq 1$ is the number of spline segments.

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}, and `"constant"` splines ignore the first and the two last data values.

float splineinverse (string basis, float v, float y_0, ... float y_{n-1}) float splineinverse (string basis, float v, float y[]) float splineinverse (string basis, float v, int nknots, 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 specify a monotonic (only increasing or only decreasing) set of values.

Note that the combination of `spline()` and `splineinverse()` makes it possible to compute a full spline-with-nonuniform-abscissae:

```
float v = splineinverse (basis, x, nknots, abscissa);
result = spline (basis, v, nknots, value);
```

8.6 Derivatives and area operators

float Dx (float a), Dy (float a), Dz (float a) vector Dx (point a), Dy (point a), Dz (point a) vector Dx (vector a), Dy (vector a), Dz (vector a) color Dx (color a), Dy (color a), Dz (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.

The `Dz` function is only meaningful for volumetric shading, and is expected to return `0` in other contexts. It is also possible that particular OSL implementations may only return "correct" `Dz` values for particular inputs (such as `P`).

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)

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 `filterwidth()`).

8.7 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 "common" 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.

8.8 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 a format string similar to `printf` in the C language. The `%d`, `%i`, `%o`, and `%x` arguments expect an `int` argument. The `%f`, `%g`, and `%e` 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). The `%s` expects a `string` or `closure` argument.

All of the substitution 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 `string`.

void error (string fmt, ...) void warning (string fmt, ...)
The `error()` and `warning()` functions work similarly to `printf`, but the results will be printed as a renderer error or warning 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 concatenated 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.

int stoi (string str)
Convert/decode the initial part of `str` to an `int` representation. Base 10 is assumed. The return value will be `0` if the string doesn't appear to hold valid representation of the destination type.

float stof (string str)
Convert/decode the initial part of `str` to a `float` representation. The return value will be `0` if the string doesn't appear to hold valid representation of the destination type.

string substr (string str, output string results[], string sep, int maxsplit) string substr (string str, output string results[], string sep) string substr (string str, output string results[])
Fills the `result` array with the words in the string `str`, using `sep` as the delimiter string. If `maxsplit` is supplied, at most `maxsplit` splits are done. If `sep` is "" (or if not supplied), any whitespace string is a separator. The value returned is the number of elements (separated strings) written to the `results` array.

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 getchar (string s, int n)
Returns the numeric value of the n^{th} character of the string, or `0` if `N` does not index a valid character of the string.

int hash (string s)
Returns a deterministic, repeatable hash of the string.

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:

```
r = regex_search ("foobar.baz", "bar");    //  = 1
r = 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.

8.9 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`, `dsdy` 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 key-value arguments (see Section *Function calls*) 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).

"subimage", *int* "subimage", *string*

Specify the subimage (by numerical index, or name) of the subimage within a multi-image texture file (default: subimage 0).

"fill", *float*

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

"missingcolor", *color*, "missingalpha", *float*

If present, supplies a missing color (and alpha value) that will be used for missing or broken textures – *instead of* treating it as an error. If you want a missing or broken texture to be reported as an error, you must not supply the optional `"missingcolor"` parameter.

"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()`.

"errormessage", ***stringvariable***
If this option is supplied, any error messages generated by the texture system will be stored in the specified variable rather than issuing error calls to the renderer, thus leaving it up to the shader to handle any errors. The error message stored will be "" if no error occurred.

"interp", ***string***
Overrides the texture interpolation method: "smartcubic" (the default), "cubic", "linear", or "closest".

type **texture3d (string filename, point p, ...*params...*)** ***type*** **texture3d (string filename, point p, vector dpdx, vector dpdy, vector dpdz, ...*params...*)**
Perform a 3D lookup of a volume texture, indexed by 3D coordinate p, antialiased over a region defined by the differentials dpdx, dpdy, and dpdz (which are computed automatically from p, 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 p coordinate (and dpdx, dpdy, and dpdz derivatives, if supplied) are assumed to be in "common" space and will be automatically transformed into volume local coordinates, if such a transormation is specified in the volume file itself.

The 3D lookup coordinate may be followed by optional token/value arguments that control the behavior of texture3d():

"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, height, and depth of the texture).

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

"width", ***float***
Scale (multiply) the size of the filter as defined by the differentials (or implicitly by the differentials of s, t, and r). 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, t, and r directions by using the "swidth", "twidth", and "rwidth" 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, t, and r directions by using the "swrap", "twrap", and "rwrap" parameters, respectively.

"firstchannel", ***int***
The first channel to look up from the texture map (default: 0).

"subimage", ***int*** **"subimage",** ***string***
Specify the subimage (by numerical index, or name) of the subimage within a multi-image texture file (default: subimage 0).

"fill", ***float***
The value to return for any channels that are requested, but not present in the texture file (default: 0).

"missingcolor", ***color*****, "missingalpha",** ***float***
If present, supplies a missing color (and alpha value) that will be used for missing or broken textures – *instead of* treating it as an error. If you want a missing or broken texture to be reported as an error, you must not supply the optional "missingcolor" parameter.

"time", *float*
A time value to use if the volume texture specifies a time-varying local transformation (default: 0).

"alpha", *floatvariable*
The alpha channel (presumed to be the next channel following the channels returned by the `texture3d()` call) will be stored in the variable specified. This allows for RGBA lookups in a single call to `texture3d()`.

"errormessage", *'stringvariable'*
If this option is supplied, any error messages generated by the texture system will be stored in the specified variable rather than issuing error calls to the renderer, thus leaving it up to the shader to handle any errors. The error message stored will be "" if no error occurred.

type **environment (string filename, vector R, ...*params...*) *type* environment (string filename, vector R, vector dRdx, vector dRdy, ...*params...*)**
Perform an environment map lookup of an image file, indexed by direction `R`, antialiased over a region defined by the differentials `dRdx`, `dRdy` (which are computed automatically from `R`, 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 lookup direction (and optional derivatives) may be followed by optional token/value arguments that control the behavior of `environment()`:

"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.

"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 requested, but not present in the texture file (default: 0).

"missingcolor", *color*, **"missingalpha",** *float*
If present, supplies a missing color (and alpha value) that will be used for missing or broken textures – *instead of* treating it as an error. If you want a missing or broken texture to be reported as an error, you must not supply the optional "missingcolor" parameter.

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

"errormessage", *stringvariable*
If this option is supplied, any error messages generated by the texture system will be stored in the specified variable rather than issuing error calls to the renderer, thus leaving it up to the shader to handle any errors. The error message stored will be "" if no error occurred.

int gettextureinfo (string texturename, string paramname, output *type* destination) int gettextureinfo (string texturename, float s, float t, 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 `gettextureinfo()` will return `0`.

The version of `gettextureinfo()` that takes `s` and `t` parameters retrieves information about the texture file that will be used for those texture coordinates. This can be useful for UDIM textures that may use different texture files for different regions, based on the corodinates. For regular, non-UDIM textures, the coordinates, if supplied, will be ignored. When UDIM textures are queried without coordinates supplied, it will succeed and return the texture info only if that parameter is found and has the same value in all files comprising the UDIM set. (Note: the version with coordinates was added in OSL 1.12.)

Valid parameters recognized are listed below:

Name	Type	Description
`"exists"`	`int`	Result is 1 if the file exists and is an texture format that OSL can read, or 0 if the file does not exist, or could not be properly read as a texture. Note that unlike all other queries, this query will "succeed" (return 1) if the file does not exist.
`"resolution"`	`int[2]`	The resolution (x and y) of the highest MIPmap level stored in the texture map.
`"resolution"`	`int[3]`	The resolution (x, y, and z) of the highest MIPmap level stored in the 3D texture map. If it isn't a volumetric texture, the third component (`z` resolution) will be 1.
`"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"`.
`"subimages"`	`int`	Returns the number of subimages in the texture file.
`"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.
`"datawindow"`	`int[4]` or `int[6]`	Returns the pixel data window of the image. The argument is an `int` array either of length 4 or 6, in which will be placed the (xmin, ymin, xmax, ymax) or (xmin, ymin, zmin, xmax, ymax, zmax), respectively. (N.B. the `z` values may be useful for 3D/volumetric images; for 2D images they will be 0).
`"displaywindow"`	`int[4]` or `int[6]`	Returns the display (a.k.a. full) window of the image. The argument is an `int` array either of length 4 or 6, in which will be placed the (xmin, ymin, xmax, ymax) or (xmin, ymin, zmin, xmax, ymax, zmax), respectively. (N.B. the `z` values may be useful for 3D/volumetric images; for 2D images they will be 0).
`"worldtocamera"`	`matrix`	If the texture is a rendered image, retrieves the world-to-camera 3D transformation matrix that was used when it was created.
`"worldtoscreen"`	`matrix`	If the texture is a rendered image, retrieves the matrix that projected points from world space into a 2D screen coordinate system where x and y range from -1 to $+1$.
`"averagecolor"`	`color`	Retrieves the average color (first three channels) of the texture.
`"averagealpha"`	`float`	Retrieves the average alpha (the channel with `"A"` name) of the texture.
anything else	*any*	Searches for matching name and type in the metadata or other header information of the texture file.

int pointcloud_search (string ptcname, point pos, float radius, int maxpoints, [int sort,] string attr, Type data[], ..., string attrN, Type dataN[])

Search the named point cloud for the `maxpoints` closest points to `pos` within the given `radius`, returning the values of any named attributes of those points in the the given `data` arrays. If the optional `sort` parameter is present and is nonzero, the ordering of the points found will be sorted by distance from `pos`, from closest to farthest; otherwise, the results are guaranteed to be the `maxpoints` closest to `pos`, but not necessarily sorted by distance (this may be faster for some implementations than when sorted results are required). The return value is

the number of points returned, ranging from 0 (nothing found in the neighborhood) to the lesser of `maxpoints` and the actual lengths of the arrays (the arrays will never be written beyond their actual length).

These attribute names are reserved:

Name	Type	Description
"position"	`point`	The position of each point
"distance"	`float`	The distance between the point and `pos`
"index"	`int`	The point's unique index within the cloud

Note that the named point cloud will be created, if it does not yet exist in memory, and that it will be initialized by reading a point cloud from disk, if there is one matching the name.

Generally, the element type of the data arrays must match exactly the type of the point data attribute, or else you will get a runtime error. But there are two exceptions: (1) "triple" types (`color`, `point`, `vector`, `normal`) are considered interchangeable; and (2) it is legal to retrieve `float` arrays (e.g., a point cloud attribute that is `float[4]`) into a regular array of `float`, and the results will simply be concatenated into the larger array (which must still be big enough, in total, to hold `maxpoints` of the data type in the file).

Example:

```
float r = 3.0;
point pos[10];
color col[10];
int n = pointcloud_search ("particles.ptc", P, r, 10,
                           "position", pos, "color", col);
printf ("Found %d particles within radius %f of (%p)\n", r, P);
for (int i = 0;  i < n;  ++i)
    printf ("  position (%f) -> color (%g)\n", pos[i], col[i]);
```

int pointcloud_get (string ptcname, int indices[], int count, string attr, *type* data[])

Given a point cloud and a list of points `indices[0..count-1]`, store the attribute named by `attr` for each point, respectively, in `data[0..count-1]`. Return 1 if successful, 0 for failure, which could include the attribute not matching the type of `data`, invalid indices, or an unknown point cloud file.

This can be used in conjunction with `pointcloud_search()`, as in the following example:

```
float r = 3.0;
int indices[10];
int n = pointcloud_search ("particles.ptc", P, r, 10,
                           "index", indices);
float temp[10];         // presumed to be "float" attribute
float quaternions[40];  // presumed to be "float[4]" attribute
int ok = pointcloud_get ("particles.ptc", indices, n,
                         "temperature", temp,
                         "quat", quaternions);
```

As with `pointcloud_search`, the element type of the data array must either be equivalent to the point cloud attribute being retrieved, or else when retrieving `float` arrays (e.g., a point cloud attribute that is `float[4]`) into a regular array of `float`, and the results will simply be concatenated into the larger array (which must still be big enough, in total, to hold `maxpoints` of the data type in the file).

int pointcloud_write (string ptcname, point pos, string attr1, *type* data1, ...)

Save the tuple (`attr1`, `data1`, ..., `attrN`, `dataN`) at position `pos` in a named point cloud. The point cloud will be saved when the frame is finished computing. Return 1 if successful, 0 for failure, which could include the attributes not matching names or types at different positions in the point cloud.

Example:

```
color C = ...;
int ok = pointcloud_write ("particles.ptc", P, "normal", N, "color", C);
```

8.10 Material Closures

For `closure color` functions, the return "value" is symbolic and may be passed to an output variable or assigned to `Ci`, to be evaluated at a later time convenient to the renderer in order to compute the exitant radiance in the direction `-I`. But the shader itself cannot examine the numeric values of the `closure color`.

The intent of this specification is to give a minimal but useful set of material closures that you can expect any renderer implementation to provide. Individual renderers may supply additional closures that are specific to the workings of that renderer. Additionally, individual renderers may allow additional parameters or controls on the standard closures, passed as token/value pairs following the required arguments (much like the optional arguments to the `texture()` function). Consult the documentation for your specific renderer for details.

OSL's standard material closures are by synchronized to match the names and properties of the physically-based shading nodes of MaterialX v1.38 (https://www.materialx.org/).

8.10.1 Surface BSDF closures

closure color oren_nayar_diffuse_bsdf (normal N, color albedo, float roughness)
Constructs a diffuse reflection BSDF based on the Oren-Nayar reflectance model.

Parameters include:

N
Normal vector of the surface point being shaded.

albedo
Surface albedo.

roughness
Surface roughness [0,1]. A value of 0.0 gives Lambertian reflectance.

The Oren-Nayar reflection model is described in M. Oren and S. K. Nayar, "Generalization of Lambert's Reflectance Model," Proceedings of SIGGRAPH 1994, pp.239-246 (July, 1994).

closure color burley_diffuse_bsdf (normal N, color albedo, float roughness)
Constructs a diffuse reflection BSDF based on the corresponding component of the Disney Principled shading model.

Parameters include:

N
Normal vector of the surface point being shaded.

albedo
Surface albedo.

roughness
Surface roughness [0,1]. A value of 0.0 gives Lambertian reflectance.

closure color dielectric_bsdf (normal N, vector U, color reflection_tint, color transmission_tint, float roughness_x, float roughness_y, float ior, string distribution)
Constructs a reflection and/or transmission BSDF based on a microfacet reflectance model and a Fresnel curve for dielectrics. The two tint parameters control the contribution of each reflection/transmission lobe. The tints

should remain 100% white for a physically correct dielectric, but can be tweaked for artistic control or set to 0.0 for disabling a lobe.

The closure may be vertically layered over a base BSDF for the surface beneath the dielectric layer. This is done using the `layer()` closure. By chaining multiple `dielectric_bsdf` closures you can describe a surface with multiple specular lobes. If transmission is enabled (`transmission_tint` > 0.0) the closure may be layered over a VDF closure describing the surface interior to handle absorption and scattering inside the medium.

Parameters include:

N
Normal vector of the surface point being shaded.

U
Tangent vector of the surface point being shaded.

reflection_tint
Weight per color channel for the reflection lobe. Should be (1,1,1) for a physically-correct dielectric surface, but can be tweaked for artistic control. Set to (0,0,0) to disable reflection.

transmission_tint
Weight per color channel for the transmission lobe. Should be (1,1,1) for a physically-correct dielectric surface, but can be tweaked for artistic control. Set to (0,0,0) to disable transmission.

roughness_x
Surface roughness in the U direction with a perceptually linear response over its range.

roughness_y
Surface roughness in the V direction with a perceptually linear response over its range.

ior
Refraction index.

distribution
Microfacet distribution. An implementation is expected to support the following distributions: `"ggx"`

thinfilm_thickness
Optional float parameter for thickness of an iridescent thin film layer on top of this BSDF. Given in nanometers.

thinfilm_ior
Optional float parameter for refraction index of the thin film layer.

closure color conductor_bsdf (normal N, vector U, float roughness_x, float roughness_y, color ior, color extinction, string distribution)
Constructs a reflection BSDF based on a microfacet reflectance model. Uses a Fresnel curve with complex refraction index for conductors/metals. If an artistic parametrization is preferred the `artistic_ior()` utility function can be used to convert from artistic to physical parameters.

Parameters include:

N
Normal vector of the surface point being shaded.

U
Tangent vector of the surface point being shaded.

roughness_x
Surface roughness in the U direction with a perceptually linear response over its range.

roughness_y
Surface roughness in the V direction with a perceptually linear response over its range.

ior
Refraction index.

extinction
Extinction coefficient.

distribution
Microfacet distribution. An implementation is expected to support the following distributions: "ggx"

thinfilm_thickness
Optional float parameter for thickness of an iridescent thin film layer on top of this BSDF. Given in nanometers.

thinfilm_ior
Optional float parameter for refraction index of the thin film layer.

closure color generalized_schlick_bsdf (normal N, vector U, color reflection_tint, color transmission_tint, float roughness_x, float roughness_y, color f0, color f90, float exponent, string distribution)
Constructs a reflection and/or transmission BSDF based on a microfacet reflectance model and a generalized Schlick Fresnel curve. The two tint parameters control the contribution of each reflection/transmission lobe.

The closure may be vertically layered over a base BSDF for the surface beneath the dielectric layer. This is done using the layer() closure. By chaining multiple `dielectric_bsdf` closures you can describe a surface with multiple specular lobes. If transmission is enabled (`transmission_tint` > 0.0) the closure may be layered over a VDF closure describing the surface interior to handle absorption and scattering inside the medium.

Parameters include:

N
Normal vector of the surface point being shaded.

U
Tangent vector of the surface point being shaded.

reflection_tint
Weight per color channel for the reflection lobe. Set to (0,0,0) to disable reflection.

transmission_tint
Weight per color channel for the transmission lobe. Set to (0,0,0) to disable transmission.

roughness_x
Surface roughness in the U direction with a perceptually linear response over its range.

roughness_y
Surface roughness in the V direction with a perceptually linear response over its range.

f0
Reflectivity per color channel at facing angles.

f90
Reflectivity per color channel at grazing angles.

exponent
Variable exponent for the Schlick Fresnel curve, the default value should be 5.

distribution
Microfacet distribution. An implementation is expected to support the following distributions: "ggx"

thinfilm_thickness
Optional float parameter for thickness of an iridescent thin film layer on top of this BSDF. Given in nanometers.

thinfilm_ior
Optional float parameter for refraction index of the thin film layer.

closure color translucent_bsdf (normal N, color albedo)
Constructs a translucent (diffuse transmission) BSDF based on the Lambert reflectance model.

Parameters include:

N
Normal vector of the surface point being shaded.

albedo
Surface albedo.

roughness
Surface roughness [0,1]. A value of 0.0 gives Lambertian reflectance.

closure color transparent_bsdf ( )
Constructs a closure that represents straight transmission through a surface.

closure color subsurface_bssrdf ( )
Constructs a BSSRDF for subsurface scattering within a homogeneous medium.

Parameters include:

N
Normal vector of the surface point being shaded.

albedo
Single-scattering albedo of the medium.

transmission_depth
Distance travelled inside the medium by white light before its color becomes transmission_color by Beer's law. Given in scene length units, range [0,infinity). Together with transmission_color this determines the extinction coefficient of the medium.

transmission_color
Desired color resulting from white light transmitted a distance of 'transmission_depth' through the medium. Together with transmission_depth this determines the extinction coefficient of the medium.

anisotropy
Scattering anisotropy [-1,1]. Negative values give backwards scattering, positive values give forward scattering, and 0.0 gives uniform scattering.

closure color sheen_bsdf (normal N, color albedo, float roughness)
Constructs a microfacet BSDF for the back-scattering properties of cloth-like materials. This closure may be vertically layered over a base BSDF, where energy that is not reflected will be transmitted to the base closure.

Parameters include:

N
Normal vector of the surface point being shaded.

albedo
Surface albedo.

roughness
Surface roughness [0,1].

8.10.2 Volumetric material closures

closure color anisotropic_vdf (color albedo, color extinction, float anisotropy)
Constructs a VDF scattering light for a general participating medium, based on the Henyey-Greenstein phase function. Forward, backward and uniform scattering is supported and controlled by the anisotropy input.

Parameters include:

albedo
Single-scattering albedo of the medium.

extinction
Volume extinction coefficient.

anisotropy
Scattering anisotropy [-1,1]. Negative values give backwards scattering, positive values give forward scattering, and 0.0 gives uniform scattering.

closure color medium_vdf (color albedo, float transmission_depth, color transmission_color, float anisotropy, float ior, int priority)
Constructs a VDF for light passing through a dielectric homogeneous medium, such as glass or liquids. The parameters `transmission_depth` and `transmission_color` control the extinction coefficient of the medium in an artist-friendly way. A priority can be set to determine the ordering of overlapping media.

Parameters include:

albedo
Single-scattering albedo of the medium.

transmission_depth
Distance travelled inside the medium by white light before its color becomes transmission_color by Beer's law. Given in scene length units, range [0,infinity). Together with transmission_color this determines the extinction coefficient of the medium.

transmission_color
Desired color resulting from white light transmitted a distance of 'transmission_depth' through the medium. Together with transmission_depth this determines the extinction coefficient of the medium.

anisotropy
Scattering anisotropy [-1,1]. Negative values give backwards scattering, positive values give forward scattering, and 0.0 gives uniform scattering.

ior
Refraction index of the medium.

priority
Priority of this medium (for nested dielectrics).

8.10.3 Light emission closures

closure color uniform_edf (color emittance)
Constructs an EDF emitting light uniformly in all directions. This is used to represent a glowing/emissive material. When called in the context of a surface shader group, it implies that light is emitted in a full hemisphere centered around the surface normal. When called in the context of a volume shader group, it implies that light is emitted evenly in all directions around the point being shaded.

The `emittance` parameter is the amount of emission and has units of radiance (e.g., $\mathrm{W}\cdot\mathrm{sr}^{-1}\cdot\mathrm{m}^{-2}$). This means that a surface directly seen by the camera will directly reproduce the closure weight in the final pixel, regardless of being a surface or a volume.

For an emissive surface, if you divide the return value of `uniform_edf()` by `surfacearea() * M_PI`, then you can easily specify the total emissive power of the light (e.g., W), regardless of its physical size.

8.10.4 Layering and Signaling closures

closure color layer (closure color top, closure color base)
Vertically layer a layerable BSDF such as `dielectric_bsdf`, `generalized_schlick_bsdf` or `sheen_bsdf` over a BSDF or VDF. The implementation is target specific, but a standard way of handling this is by albedo scaling, using `base*(1-reflectance(top)) + top`, where `reflectance()` calculates the directional albedo of a given top BSDF.

closure color holdout ( )
Returns a `closure color` that does not represent any additional light reflection from the surface, but does signal to the renderer that the surface is a *holdout object* (appears transparent in the final output yet hides objects behind it). "Partial holdouts" may be designated by weighting the `holdout()` closure by a weight that is less than 1.0.

closure color debug (string outputname)
Returns a `closure color` that does not represent any additional light reflection from the surface, but does signal to the renderer to add the weight of the closure (which may be a `float` or a `color`) to the named output (i.e., AOV).

8.10.5 Material utility functions

void artistic_ior (color reflectivity, color edge_tint, output color ior, output color extinction)
Converts the artistic parameterization reflectivity and edge_tint to complex IOR values. To be used with the `conductor_bsdf()` closure.

Parameters include:

reflectivity
Reflectivity per color channel at facing angles (r parameter in [OG14])

edge_tint
Color bias for grazing angles (g parameter in [OG14]). NOTE: This is not equal to 'f90' in a Schlick Fresnel parameterization.

ior
Output refraction index.

extinction
Output extinction coefficient.

Reference: [OG14] Ole Gulbrandsen, "Artist Friendly Metallic Fresnel", Journal of Computer Graphics Tools 3(4), 2014. http://jcgt.org/published/0003/04/03/paper.pdf

8.10.6 Deprecated closures

These were described in the original OSL language specification, but beginning with OSL 1.12, these are considered deprecated. Support for them will be removed entirely in OSL 2.0.

Deprecated Surface closures

`closure color diffuse (normal N)`
Returns a `closure color` that represents the Lambertian diffuse reflectance of a smooth surface,

$$\int_\Omega \frac{1}{\pi} \max(0, N \cdot \omega) Cl(P, \omega) d\omega$$

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 $-\omega$.

`closure color phong (normal N, float exponent)`
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 (higher `exponent` values indicate a smoother surface).

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

The Oren-Nayar reflection model is described in M. Oren and S. K. Nayar, "Generalization of Lambert's Reflectance Model," Proceedings of SIGGRAPH 1994, pp.239-246 (July, 1994).

`closure color ward (normal N, vector T, float xrough, float yrough)`
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 tangent (`T`) and bitangent (`N` $\times$ `T`) directions, respectively.

The Ward BRDF is described in Ward, G., "Measuring and Modeling Anisotropic Reflection," Proceedings of SIGGRAPH 1992.

`closure color microfacet (string distribution, normal N, float alpha, float eta, int refract)`
Returns a `closure color` that represents scattering on the surface using some microfacet distribution. A simplified isotropic version of the previous function.

`closure color reflection (normal N, float eta)`
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. The `reflection()` closure behaves as if it were implemented as follows:

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

`closure color refraction (normal N, float eta)`
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. The `refraction()` closure behaves as if it were implemented as follows:

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

closure color transparent ( )
Returns a `closure color` that shows the light *behind* the surface without any refractive bending of the light directions. The `transparent()` closure behaves as if it were implemented as follows:

```
return raytrace (I);
```

closure color translucent ( )
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. The `translucent()` closure behaves as if it were implemented as follows:

```
return diffuse (-N);
```

Deprecated Volumetric closures

closure color isotropic ( )
Returns a `closure color` that represents the scattering of an isotropic volumetric material, scattering light evenly in all directions, regardless of its original direction.

closure color henyey_greenstein (float g)
Returns a `closure color` that represents the directional volumetric scattering by small suspended particles. The `g` parameter is the anisotropy factor, in the range $(-1, 1)$, with positive values indicating predominantly forward-scattering, negative values indicating predominantly back-scattering, and value of $g = 0$ resulting in isotropic scattering.

closure color absorption ( )
Returns a `closure color` that does not represent any additional light scattering, but rather signals to the renderer the absorption represents the scattering of an isotropic volumetric material, scattering light evenly in all directions, regardless of its original direction.

Deprecated Emission closures

closure color emission ( )
Returns a `closure color` that represents a glowing/emissive material. When called in the context of a surface shader group, it implies that light is emitted in a full hemisphere centered around the surface normal. When called in the context of a volume shader group, it implies that light is emitted evenly in all directions around the point being shaded.

The weight of the emission closure has units of radiance (e.g., $\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}$). This means that a surface directly seen by the camera will directly reproduce the closure weight in the final pixel, regardless of being a surface or a volume.

For an emissive surface, if you divide the return value of `emission()` by `surfacearea() * M_PI`, then you can easily specify the total emissive power of the light (e.g., W), regardless of its physical size.

closure color background ( )
Returns a `closure color` that represents the radiance of the "background" infinitely far away in the view direction. The implementation is renderer-specific, but often involves looking up from an HDRI environment map.

8.11 Renderer state and message passing

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

Retrieves a named renderer attribute or the value of an interpolated geometric variable. If an object is explicitly named, that is the only place that will be searched (`"global"` means the global scene-wide attributes). For the forms of the function with no object name, or if the object name is the empty string `""`, the renderer will first search per-object attributes on the current object (or interpolated variables with that name attached to the object), then if not found it will search global scene-wide attributes.

If the attribute is found and can be converted to the type of `destination`, the attribute's value will be stored in `destination` and `getattribute` will return `1`. If not found, or the type cannot be converted, `destination` will not be modified and `getattribute` will return `0`.

The automatic type conversions include those that are allowed by assignment in OSL source code: `int` to `float`, `float` to `int` (truncation), `float` (or `int`) to *triple* (replicating the value), any *triple* to any other *triple*. Additionally, the following conversions which are not allowed by assignment in OSL source code will also be performed by this call: `float` (or `int`) to `float[2]` (replication into both array elements), `float[2]` to *triple* (setting the third component to 0).

The forms of this function that have the the `arrayindex` parameter will retrieve 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.

Tables giving "standardized" names for different kinds of attributes may be found below. All renderers are expected to use the same names for these attributes, but are free to choose any names for additional attributes they wish to make queryable.

Names of standard attributes that may be retrieved:

Name	Type	Description
`"osl:version"`	`int`	Major x 10000 + Minor x 100 + patch.
`"shader:shadername"`	`string`	Name of the shader master.
`"shader:layername"`	`string`	Name of the layer instance.
`"shader:groupname"`	`string`	Name of the shader group.

Names of standard camera attributes that may be retrieved are in the table below. If the `getattribute()` function specifies an `objectname` parameter and it is the name of a valid camera, the value specific to that camera is retrieved. If no specific camera is named, the global or default camera is implied.

Name	Type	Description
"camera:resolution"	int[2]	Image resolution.
"camera:pixelaspect"	float	Pixel aspect ratio.
"camera:projection"	string	Projection type (e.g., "perspective", "orthographic", etc.)
"camera:fov"	float	Field of fiew.
"camera:clip_near"	float	Near clip distance.
"camera:clip_far"	float	Far clip distance.
"camera:clip"	float[2]	Near and far clip distances.
"camera:shutter_open"	float	Shutter open time.
"camera:shutter_close"	float	Shutter close time.
"camera:shutter"	float[2]	Shutter open and close times.
"camera:screen_window"	float[4]	Screen window (xmin, ymin, xmax, ymax).

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 object. If there is already a message with the same name attached to this shader invocation, it will be replaced by the new value. The message `value` may be any basic scalar type, array, or closure, but may not be a `struct`.

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

Retrieve a message from another shader attached to the same object. If a message is found 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`, if supplied, designates from where the message should be retrieved, and may have any of the following values:

"trace"

Retrieves data about the object hit by the last `trace` call made. Data recognized include:

Name	Type	Description
"hit"	int	Zero if the ray hit nothing, 1 if it hit.
"hitdist"	float	The distance to the hit.
"geom:name"	string	The name of the object hit.
other		Retrieves the named global (`P`, `N`, etc.), shader parameter, or set message of the closest object hit (only if it was a shaded ray).

Note that which information may be retrieved depends on whether the ray was traced with the optional `"shade"` parameter indicating whether or not the shader ought to execute on the traced ray. If `"shade"` was 0, you may retrieve "globals" (`P`, `N`, etc.), interpolated vertex variables, shader instance values, or graphics state attributes (object name, etc.). But `"shade"` must be nonzero to correctly retrieve shader output variables or messages that are set by the shader (via `setmessage()`).

float surfacearea ( )

Returns the surface area of the area light geometry being shaded. This is meant to be used in conjunction with `emission()` in order to produce the correct emissive radiance given a user preference for a total wattage for the area light source. The value of this function is not expected to be meaningful for non-light shaders.

int raytype (string name)

Returns 1 if ray being shaded is of the given type, or 0 if the ray is not of that type or if the ray type name is not recognized by the renderer.

The set of ray type names is customizeable for renderers supporting OSL, but is expected to include at a minimum `"camera"`, `"shadow"`, `"diffuse"`, `"glossy"`, `"reflection"`, `"refraction"`. They are not necessarily mutually exclusive, with the exception that camera rays should be of class `"camera"` and no other.

int backfacing ( )
Returns 1 if the surface is being sampled as if "seen" from the back of the surface (or the "inside" of a closed object). Returns 0 if seen from the "front" or the "outside" of a closed object.

int isconnected (*type* parameter)
Returns 1 if the argument is a shader parameter and is connected to an earlier layer in the shader group, 2 if the argument is a shader output parameter connected to a later layer in the shader group, 3 if connected to both earlier and later layers, otherwise returns 0. Remember that function arguments in OSL are always pass-by-reference, so `isconnected()` applied to a function parameter will depend on what was passed in as the actual parameter.

int isconstant (*type* expr)
Returns 1 if the expression can, at runtime (knowing the values of all the shader group's parameter values and connections), be discerned to be reducible to a constant value, otherwise returns 0.

This is primarily a debugging aid for advanced shader writers to verify their assumptions about what expressions can end up being constant-folded by the runtime optimizer.

8.12 Dictionary Lookups

int dict_find (string dictionary, string query) int dict_find (int nodeID, string query)
Find a node in the dictionary by a query. The `dictionary` is either a string containing the actual dictionary text, or the name of a file containing the dictionary. (The system can easily distinguish between them.) XML dictionaries are currently supported, and additional formats may be supported in the future. The query is expressed in "XPath 1.0" syntax (or a reasonable subset therof).

The return value is a *Node ID*, an opaque integer identifier that is the handle of a node within the dictionary data. The value 0 is reserved to mean "query not found" and the value -1 indicates that the dictionary was not a valid syntax (or, if a file, could not be read). If more than one node within the dictionary matched the query, the node ID of the first match is returned, and `dict_next()` may be used to step to the next matching node.

The version that takes a nodeID rather than a dictionary string simply interprets the query as being relative to the node specified by nodeID, as opposed to relative to the root of the dictionary.

All expensive operations (such as reading the dictionary from a file and the initial parsing of the dictionary) are performed only once, and subsequent lookups merely copy data and are thus inexpensive. The `dictionary` string is, therefore, used as a hash into a cached data structure holding the parsed dictionary database. Implementations may also cache individual node lookups or type conversions behind the scenes.

int dict_next (int nodeID)
Return the node ID of the next node that matched the query that returned nodeID, or 0 if nodeID was the last matching node for its query.

int dict_value (int nodeID, string attribname, output *type* value)
Retrieves the named attribute of the given dictionary node, or the value of the node itself if `attribname` is the empty string `""`. If the attribute is found, its value will be stored in `value` and 1 will be returned. If the requested attribute is not found on the node, or if the type of `value` does not appear to match that of the named node, `value` will be unmodified and 0 will be returned.

Type conversions are straightforward: anything may be retrieved as a string; to retrieve as an int or float, the value must parse as a single integer or floating point value; to retrieve as a point, vector, normal, color, or matrix (or any array), the value must parse as the correct number of values, separated by spaces and/or commas.

int trace (point pos, vector dir, ...)
Trace a ray from pos in the direction dir. The ray is traced immediately, and may incur significant expense

compared to rays that might be traced incidentally to evaluating the `Ci` closure. Also, beware that this can be easily abused in such a way as to introduce view-dependence into shaders. The return value is 0 if the ray missed all geometry, 1 if it hit anything within the distance range.

The following optional key-value arguments (see Section *Function calls*) can be passed:

"mindist", *float*
The minimum hit distance (default: 0).

The units of the "`mindist`" and "`maxdist`" are determined by the renderer and are sometimes defined by the dir vector, which can lead to unexpected behavior. So, generally, clearly written and portable shaders should pass a unit length (see Section *Geometric functions*) dir vector.

"mindist", *float*
The maximum hit distance (default: infinite).

"shade", *int*
Defines whether objects hit will be shaded (default: 0).

"traceset", *string*
An optional named set of objects to ray trace (if preceded by a `-` character, it means to exclude that set).

Information about the closest object hit by the ray may be retrieved using

```
getmessage("trace",...)
```

(see Section *Renderer state and message passing*).

The main purpose of this function is to allow shaders to "probe" nearby geometry, for example to apply a projected texture that can be blocked by geometry, apply more "wear" to exposed geometry, or make other ambient occlusion-like effects.

8.13 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.

CHAPTER

NINE

FORMAL LANGUAGE GRAMMAR

This section gives the complete syntax of OSL. 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).

9.1 Lexical elements

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<digit-sequence> ::= <digit> { <digit> }

<hexdigit> ::= <digit> | "a" | "A" | "b" | "B" | "c" | "C" | "d" | "D" | "e" | "E" | "f" | "F"

<hexdigit-sequence> ::= <hexdigit> { <hexdigit> }

<integer> ::= <sign> <digit-sequence> | <sign> "0x" <hexdigit-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> }

9.2 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" | "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> <initializer-list> <metadata-block-opt>

<metadata-block> ::= "[[" <metadata> { "," <metadata> } "]]"

<metadata> ::= <simple-typespec> <identifier> <initializer>

9.3 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> <typed-field-list> ";"

<typed-field-list> ::= <typed-field> { "," <typed-field> }

<typed-field> ::= <identifier> <arrayspec-opt>

<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> <initializer-list-opt>

<initializer> ::= "=" <expression>

<initializer-list> ::= "=" <compound-initializer>

<compound-initializer> ::= "{" <init-expression-list> "}"

<init-expression-list> ::= <init-expression> { "," <init-expression> }

<init-expression> ::= <expression> | <compound-initializer>

<typespec> ::= <simple-typename> | "closure" <simple-typename> | <identifier-structname>

<simple-typename> ::= "color" | "float" | "matrix" | "normal" | "point" | "string" | "vector" | "void"

9.4 Statements

<statement-list> ::= <statement> { <statement> }

<statement> ::= <compound-expression-opt> ";" | <scoped-statements> | <local-declaration> | <conditional-statement> | <loop-statement> | <loopmod-statement> | <return-statement>

<scoped-statements> ::= "{" <statement-list-opt> "}"

<conditional-statement> ::= "if" "(" <compound-expression> ")" <statement> | "if" "(" <compound-expression> ")" <statement> "else" <statement>

<loop-statement> ::= “while” “(” <compound-expression> “)” <statement> | “do” <statement> “while” “(” <compound-expression> “)” “;” | “for” “(” <for-init-statement-opt> <compound-expression-opt> “;” <compound-expression-opt> “)” <statement>

<for-init-statement> ::= <expression-opt> “;” | <variable-declaration>

<loopmod-statement> ::= “break” “;” | “continue” “;”

<return-statement> ::= “return” <expression-opt> “;”

9.5 Expressions

<expression-list> ::= <expression> { “,” <expression> }

<expression> ::= <number> | <stringliteral> | <type-constructor> | <incdec-op> <variable-ref> | <expression> <binary-op> <expression> | <unary-op> <expression> | “(” <compound-expression> “)” | <function-call> | <assign-expression> | <ternary-expression> | <typecast-expression> | <variable-ref> | <compound-initializer>

<compound-expression> ::= <expression> { “,” <expression> }

<variable-lvalue> ::= <identifier> <array-deref-opt> <component-deref-opt> | <variable_lvalue> “[” <expression> “]” | <variable_lvalue> “.” <identifier>

<variable-ref> ::= <identifier> <array-deref-opt>

<array-deref> ::= “[” <expression> “]”

<component-deref> ::= “[” <expression> “]” | “.” <component-field>

<component-field> ::= “x” | “y” | “z” | “r” | “g” | “b”

<binary-op> ::= “*” | “/” | “%” | “+” | “-” | “<<” | “>>” | “<” | “<=” | “>” | “>=” | “==” | “!=” | “&” | “^” | “|” | “&&” | “and” | “||” | “or”

<unary-op> ::= “-” | “~” | “!” | “not”

<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>

CHAPTER

TEN

DESCRIBING SHADER GROUPS

Below, we propose a simple grammar giving a standard way to serialize (i.e., express as text) a full shader group, including instance values and connectivity of the shader layers. There are only three statements/operations: set an instance value, make a shader instance, and connect two instances.

param *type paramname value...* **;** **param** *type paramname value...* **[[** *metadata...* **]]** **;**

Declare an instance value of a shader parameter to be applied to the next `shader` statement. We refer to the parameter values set, which have not yet had their shader declared, as *pending parameters*.

The *paramname* is the name of the parameter whose instance values are being specified.

The *type* is one of the basic numeric or string data types described in Chapter *Data types* (`int`, `float`,`color`, `point`, `vector`, `normal`, `matrix`, or `string`), or an array thereof (indicated by the usual notation of `[size]`). The *type* must match the declared type of the parameter in the shader.

The actual values are listed individually, with multiple values (in the case of an array or an aggregate such as a `color`) simply separated by whitespace. If fewer values are supplied than the total number of array elements or aggregate components, the remainder will be understood to be filled with 0 values. String values must be enclosed in double quotes (`"like this"`).

The `param` statement is terminated by a semicolon (`;`).

Optionally, metadata hints may be supplied enclosed by double brackets, immediately before the semicolon.

shader *shadername layername* **;**

Declares a shader instance, which will receive any pending parameters that were declared since the previous `shader` statement (and in the process, clear the list of pending parameters).

The `shadername` is an identifier that specifies the name of the shader to use as the master for this instance. The `layername` is an identifier that names the layer (e.g., to subsequently specify it as a source or destination for connections).

The `shader` statement is terminated by a semicolon (`;`).

connect *source_layername* **.** *paramname* *destination_layername* **.** *paramname* **;**

Establish a connection between an output parameter of a source layer and an input parameter of a destination layer, both of which have been previously declared within this group. The source layer must have preceded the destination layer when they were declared, and the parameters must exist and be of a compatible type so that it is meaningful to establish a connection (for example, you may connect a `color` to a `color`, but you may not connect a `color` to a `matrix`).

If the named parameters are structures, the two structures must have identical data layout, and establishing the connection will connect each corresponding data member. It is also possible to make a connection of just a single member of a structure by using the usual ``dot'' syntax, for example, for layers `A` and `B`, `connect A.c.x B.y` might connect A's parameter `c`, member `x`, to B's parameter `y` (the types of `c.x` in A and `y` in B must match).

The `connect` statement is terminated by a semicolon (`;`).

Example

```
param string name "rings.tx" ;      # set pending `name'
param float scale 3.5 ;             # set pending `scale'
shader "texturemap" "tex1" ;        # tex1 layer, picks up `name', `scale'
param string name "grain.tx" ;
shader "texturemap" "tex2" ;
param float gam 2.2 ;
shader "gamma" "gam1" ;
param float gam 1.0 ;
shader "gamma" "gam2" ;
param color woodcolor 0.42 0.38 0.22 ;       # example of a color param
shader "wood" "wood1" ;
connect tex1.Cout gam1.Cin ;        # connect tex1's Cout to gam1's Cin
connect tex2.Cout gam2.Cin ;
connect gam1.Cout wood1.rings ;
connect gam2.Cout wood1.grain ;
```

CHAPTER

ELEVEN

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*.

BSDF
: Bidirectional scattering distribution function, a function that describes light scattering properties of a surface.

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 OSL itself).

Closure
: A symbolic representation of a function to be called, and values for its parameters, that are packaged up to 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 parameter* 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.

EDF
: Emission distribution function, a function that describes the distribution of light emitted by a light source.

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 several *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.

Shader
A small self-contained program written in OSL, 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 its unique *instance parameters*, transformation, etc.

Shader function
A function written in OSL 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.

VDF
Volumetric distribution function, a function that describes the scattering properties of a volume.