Design Specification for MetaSL™ 1.0

Document version 1.0.14
March 20, 2009

Copyright Information

Copyright © 1986-2009 mental images GmbH, Berlin, Germany.

All rights reserved.

This document is protected under copyright law. The contents of this document may not be translated, copied or duplicated in any form, in whole or in part, without the express written permission of mental images GmbH.

The information contained in this document is subject to change without notice. mental images GmbH and its employees shall not be responsible for incidental or consequential damages resulting from the use of this material or liable for technical or editorial omissions made herein.

mental images®, mental ray®, mental matter®, mental mill®, mental queue™, mental q™, mental world™, mental map™, mental earth™, mental mesh™, mental™, Reality™, RealityServer®, RealityPlayer®, RealityDesigner®, MetaSL™, Meta™, Meta Shading™, Meta Node™, Phenomenon™, Phenomena™, Phenomenon Creator®, Phenomenon Editor®, Phenomill™, Phenograph™, neuray®, iray®, imatter®, Cybernator™, 3D Cybernator™, Shape-By-Shading®, SPM®, NRM™, and rendering imagination visible™ are trademarks or, in some countries, registered trademarks of mental images GmbH, Berlin, Germany.

Other product names mentioned in this document may be trademarks or registered trademarks of their respective companies and are hereby acknowledged.

Document build number 49440

Table of Contents

1 Introduction . . . 1
 1.1 Shader building blocks . . . 1
  1.1.1 Re-use of shader components . . . 2
  1.1.2 Organization of shader graphs . . . 2
 1.2 Shader types . . . 3
 1.3 Dataflow . . . 4
 1.4 The MetaSL programming language . . . 5
2 Lexical structure . . . 6
 2.1 Character Set . . . 6
 2.2 Comments . . . 7
 2.3 Tokens . . . 7
 2.4 Operators and Separators . . . 8
 2.5 Identifiers and Typenames . . . 8
 2.6 Reserved Words . . . 9
 2.7 Literals . . . 9
  2.7.1 Boolean Literals . . . 9
  2.7.2 Integer Literals . . . 9
  2.7.3 Floating-point Literals . . . 10
  2.7.4 String Literals . . . 10
3 Data types . . . 11
 3.1 Constructors . . . 12
 3.2 Operators . . . 13
 3.3 Scalars – float, double, half, int and bool . . . 15
  3.3.1 Constructors . . . 15
  3.3.2 Operators . . . 15
  3.3.3 Conversion . . . 16
 3.4 Vectors – float, double, half, int, and bool . . . 18
  3.4.1 Constructors . . . 18
  3.4.2 Members . . . 19
  3.4.3 Operators . . . 20
  3.4.4 Conversion . . . 20
 3.5 Matrices – float, double, and half . . . 22
  3.5.1 Constructors . . . 22
  3.5.2 Members . . . 22
  3.5.3 Operators . . . 23
  3.5.4 Conversion . . . 23
 3.6 Textures and samplers . . . 25
  3.6.1 Constructors . . . 26
  3.6.2 Texture Members . . . 27
  3.6.3 Sampler Members . . . 27

3.6.4 Operators ... 28
3.6.5 Conversion ... 28
3.7 Color ... 30
3.7.1 Constructors ... 30
3.7.2 Operators ... 30
3.7.3 Conversion ... 31
3.8 Spectrum ... 32
3.8.1 Constructors ... 32
3.8.2 Operators ... 32
3.8.3 Conversion ... 32
3.9 Ray ... 33
3.9.1 Constructors ... 33
3.9.2 Members ... 33
3.9.3 Operators ... 33
3.10 String ... 34
3.10.1 Constructors ... 34
3.10.2 Operators ... 34
3.10.3 Conversion ... 34
3.11 Shader ... 35
3.11.1 Constructors ... 35
3.11.2 Members ... 35
3.11.3 Operators ... 36
3.11.4 Conversion ... 36
3.12 Particle_map ... 37
3.12.1 Constructors ... 37
3.12.2 Members ... 37
3.12.3 Operators ... 40
3.12.4 Conversion ... 40
3.13 Light_profile ... 41
3.13.1 Constructors ... 41
3.13.2 Members ... 41
3.13.3 Operators ... 41
3.13.4 Conversion ... 41
4 Arrays ... 42
5 Structures ... 44
6 Enumerations ... 46
7 Typedef ... 48
8 Control flow ... 49
8.1 Loops ... 49
8.2 Branches ... 50
8.3 Jumps ... 51
9 Functions ... 53

 Copyright © 2009 mental images

9.1 Function overloading 53
9.2 Parameter passing 53
9.3 Native functions 54
10 Shader class 56
10.1 Input parameters 56
10.2 Output parameters 57
10.3 Other shader members 58
10.4 Shader methods 59
10.4.1 Shader main method 59
10.4.2 Constructors and destructors 60
10.4.3 Method overloading 61
10.4.4 Parameter passing 62
10.4.5 Shader class inheritance 63
11 Preprocessor 64
12 Annotations 67
12.1 User defined annotations 69
13 Shading state 71
13.1 State variables 71
13.2 State functions 77
13.2.1 Coordinate systems 77
13.2.2 Ray tracing 79
13.2.3 Light shaders 84
13.2.4 Messages and diagnostics 84
14 Illumination 85
14.1 BRDFs 85
14.2 Light iteration 85
14.3 Direct and indirect light 86
14.4 Light Loops 87
15 Sampling 90
16 Shader type reference 92
16.1 Surface 92
16.1.1 State variables 92
16.1.2 State functions 93
16.1.3 Outputs 93
16.2 Light 94
16.2.1 State variables 94
16.2.2 State functions 94
16.2.3 Outputs 95
16.3 Environment 96
16.3.1 State variables 96
16.3.2 State functions 96
16.3.3 Outputs 96

Copyright © 2009 mental images

16.4 Volume 97
16.4.1 State variables 97
16.4.2 State functions 97
16.4.3 Outputs 97
16.5 Displacement 98
16.5.1 State variables 98
16.5.2 State functions 98
16.5.3 Outputs 98
16.6 Lens 100
16.6.1 State variables 100
16.6.2 State functions 100
16.6.3 Outputs 100
17 Standard Library Functions 101
17.1 Math functions 101
17.2 Geometric functions 110
17.3 Texture functions 111
17.4 Derivatives 112
18 Appendix A – The Syntax of MetaSL 113
18.1 A Grammar 113
19 Bibliography 117
20 Changes to this document 118
20.1 Changes since version 1.0.13 118
20.2 Changes since version 1.0.11 118
20.3 Changes since version 3.10 119
20.4 Changes since version 3.9 120
20.5 Changes since version 3.8 121
20.6 Changes since version 3.7 121
20.7 Changes since version 3.6 121
20.8 Changes since version 3.5 122
20.9 Changes since version 3.4 123
20.10 Changes since version 3.3 125
20.11 Changes since version 3.1 126
20.12 Changes since version 3.0 126
20.13 Changes since version 2.9 126
20.14 Changes since version 2.8 126

 Copyright © 2009 mental images

1 Introduction

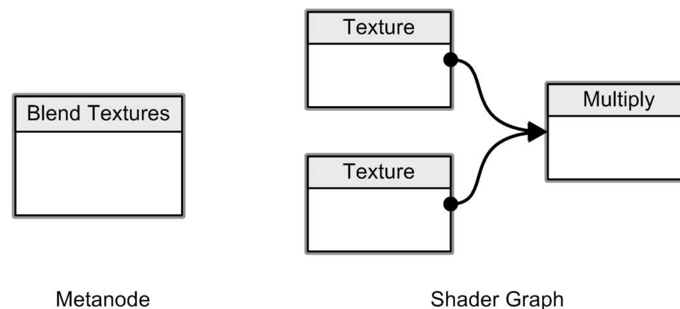
MetaSL™, the Meta Shading Language, is a domain-specific language for the program description of the programmable key components of the rendering process, often referred to as *shaders*. A shader is a user-supplied program which allows parts of the graphics pipeline to be customized giving users control over the final look of the rendered imagery.

MetaSL is a shader language dedicated to providing an abstract shader description. The abstraction protects the shader description from the details of the implementation, allowing the same MetaSL shader to be used on different platforms, in different contexts, or utilizing different rendering algorithms. Because the shader description is protected from the always-changing hardware and software which provide the implementation, a MetaSL shader does not need to be rewritten as graphics technology evolves.

MetaSL is the shading language component of mental mill® [1], which is a system for shader development using both visual methods to assemble shader building blocks and programming with MetaSL, resulting in a tightly integrated visual shader creation, programming, and debugging environment. MetaSL is used used as a shading language for mental ray®, RealityServer®, neuray®, and other renderers.

1.1 Shader building blocks

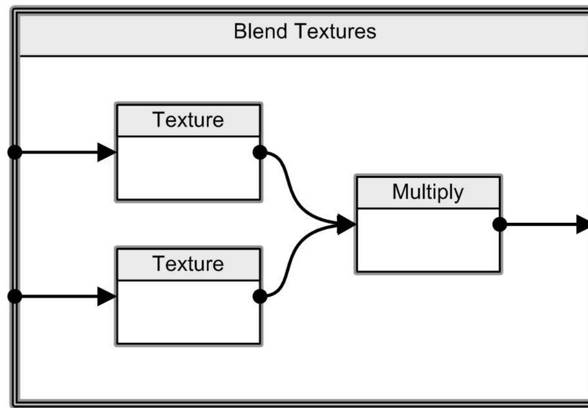
A MetaSL shader is a componentized unit of functionality which can be used as a building block to construct complex shaders. These building blocks are represented as nodes in a shader graph. A node with a MetaSL implementation is referred to as a Metanode™.



Part of the definition of a Metanode is its input and output connection points. The output of one Metanode can be connected to the input of another. A network of connected Metanodes forms a shader graph. A shader graph, or collection of shader graphs, can themselves be encapsulated to form a new type of shader called a Phenomenon™ [2][3]. A Phenomenon node is similar to a Metanode except that it is implemented by a shader graph instead of MetaSL code.

A node in a shader graph, whether a Metanode or a Phenomenon, is an instantiation of a shader type with a set of user supplied values for input parameters and a list of connections from inputs to the outputs of other nodes in the graph.

Copyright © 2009 mental images



Phenomenon

A Phenomenon can also be used in a shader graph, which can in turn define another Phenomenon. This hierarchical structure of shader nodes allows both the efficient organization of complex shader graphs as well as the re-use of shader components. Shader graphs and Phenomena are described using the XMSL file format [4].

1.1.1 Re-use of shader components

A shader graph can become a re-usable component by packaging it into a Phenomenon, allowing the graph to be instantiated in different circumstances as a single Phenomenon node. A Phenomenon defines a set of input and output parameters which form an interface to the Phenomenon. Input parameters expose inputs of nodes in the contained shader graph and outputs similarly expose outputs of the graph. The specification of Phenomenon input and output parameters are typically accomplished with a graphical user interface, which allows users to define Phenomena without writing shader code.

A Phenomenon represents a new type of shader, which can itself be instantiated for use in other shader graphs. Phenomena can be used anywhere a Metanode can be used; Phenomena and Metanodes are implemented differently (a shader graph for the former and MetaSL code for the latter), but share the input/output parameter mechanism as a way of describing their interface, allowing them to cooperate in a shader graph.

Instances of the same Phenomenon type share a common shader graph implementation of the Phenomenon. This allows a user to modify a Phenomenon's shader graph and all instances will automatically inherit the modification. A user can always clone a Phenomenon to separate nodes and protect them from changes or specialize them for a different purpose.

1.1.2 Organization of shader graphs

Shader graphs can become large and complicated to manage, and Phenomena provide an organizing function to mitigate that. Sub-graphs of larger shader graphs can be packaged effectively hiding a portion of the graph, reducing it to a single node which exposes certain inputs and outputs of the graph.

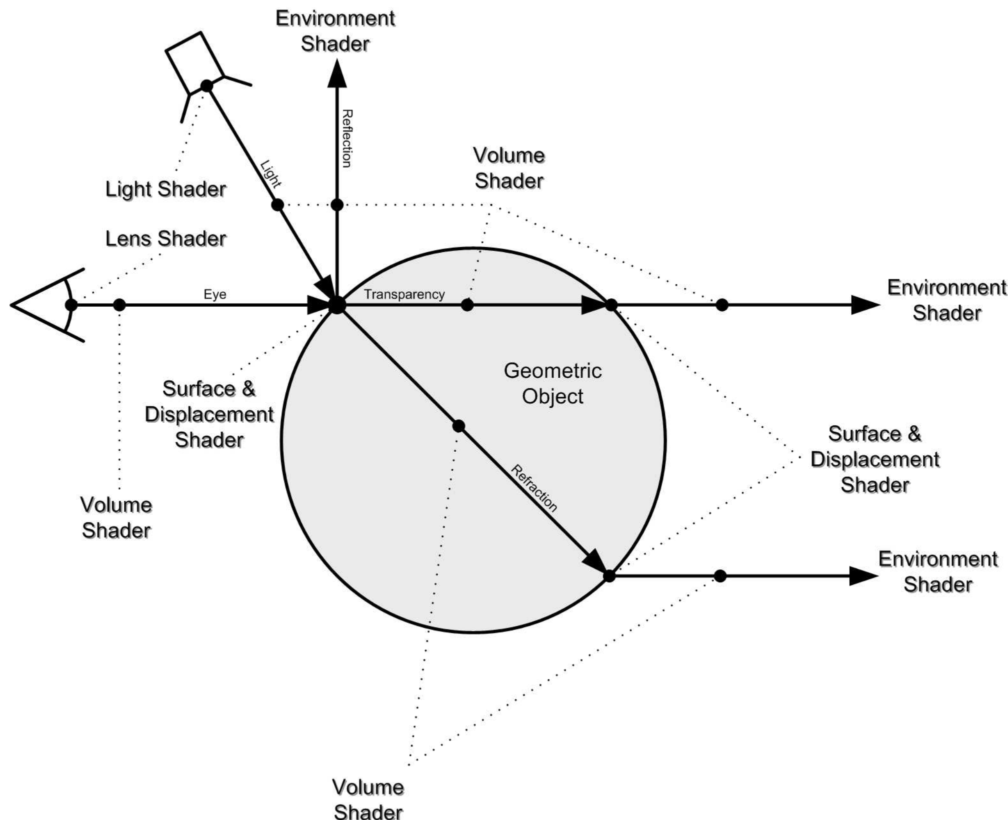
When viewed from the top level, a complex shader may only be represented by a handful of Phenomenon nodes. A user can access the internal graphs of Phenomena, which themselves can be organized into high level components. There is no limit to the number of levels of nesting. At the deepest level shader graphs will consist entirely of Metanodes.

 Copyright © 2009 mental images

1.2 Shader types

A MetaSL shader describes a programmable component of a complete rendering dataflow, for which there are several stages. These stages appear at different points in the graphics pipeline and are responsible for different aspects of the rendering solution.

The following diagram illustrates the various shader types as they relate to a ray as it intersects scene geometry:



The stages are represented by different MetaSL shader types. The MetaSL shader types share the framework provided by MetaSL, but are responsible for different tasks. The primary MetaSL shader types are:

- Displacement – A displacement shader is responsible for modifying the position (and optionally the normal) of surface points to allow the generation of fine grain detail that is difficult to model traditionally.
- Surface – A surface shader is responsible for computing the color of a point on a surface as seen from a particular direction.
- Light – A light shader is responsible for computing the amount of incoming light arriving at a position in space from a light source (or point on a light source in the case of area lights).
- Volume – A volume shader is responsible for computing the effect of a medium on light as it travels through a region of space. Examples include atmospheric effects such as fog as well as fluid constrained to a region of space such as a drinking glass.

Copyright © 2009 mental images

- Environment – An environment shader is responsible for computing the color of a point infinitely far away as observed from a particular direction.
- Lens – A lens shader is responsible for determining the color of a pixel in the rendered image, typically by casting eye rays.

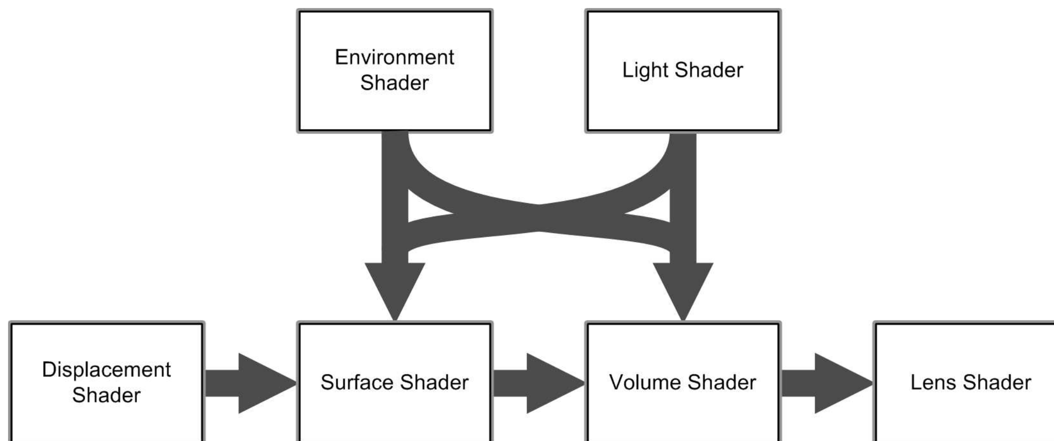
MetaSL shaders represent building blocks which are not necessarily specific to any particular shader type and can be simultaneously used in shader graphs implementing different shader types. The usage of a MetaSL shader, shader graph, or Phenomenon depends on whether it meets the requirements and fulfills the responsibility of the particular shader type.

Since a MetaSL shader can potentially be used as a component of different shader types, a MetaSL shader does not state a shader type as part of its declaration.

1.3 Dataflow

The different MetaSL shader types cooperate in the graphics pipeline to produce the resulting image, and each shader type is responsible for a different portion of the pipeline.

The following diagram illustrates the general dataflow of this pipeline:



A MetaSL material description must define a surface shader at a minimum, but the other shader components are optional.

 Copyright © 2009 mental images

1.4 The MetaSL programming language

The following sections of this document describe the MetaSL shader programming language in detail. The documented is organized into the following major sections:

- Lexical structure (page 6) – A description of the lexical elements of MetaSL, such as comments, literal values, operator symbols and keywords.
- Data types (page 11)– A description of the basic MetaSL data types.
- Data structures (page 42) – A description of the MetaSL constructs used to build data structures from basic MetaSL types.
- Control flow (page 49) – A description of the MetaSL constructs to control the flow of execution.
- Functions (page 53) – A description of MetaSL user defined functions.
- Shader class (page 56) – A description of the shader "class" which is used to define MetaSL shaders.
- Preprocessor (page 64) – A description of the built-in MetaSL preprocessor.
- Annotations (page 67) – A description of the mechanism used to attach metadata to MetaSL shaders.
- Shading state (page 71) – A description of the set of global values and functions available to MetaSL shaders.
- Illumination (page 85) – A description of the handling of lights and illumination in MetaSL.
- Sampling (page 90) – A description of the MetaSL mechanism for generating deterministic quasi-Monte Carlo based sample points.
- Shader type reference (page 92) – A description of the MetaSL shader types including the state variables and functions available to each shader type.
- Standard Library Functions (page 101) – A description of MetaSL's built-in function library.

The constructs of the MetaSL language are each introduced with a fragment of the MetaSL language grammar using EBNF notation. For more information about the MetaSL grammar please refer to *Appendix A* on page 113.

2 Lexical structure

This section describes the lexical structure of MetaSL.

2.1 Character Set

A MetaSL source file is a sequence of characters from a character set. This set comprises at least the following characters:

1. the 52 uppercase and lowercase alphabetic characters:

   ```
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

   a b c d e f g h i j k l m n o p q r s t u v w x y z
   ```

2. the 10 decimal digits:

   ```
   0 1 2 3 4 5 6 7 8 9
   ```

3. the blank or space character
4. the 32 graphic characters:

Character	Name	Character	Name
!	exclamation point	"	double quote
#	number sign	$	dollar sign
%	percent	&	ampersand
'	single quote	(	left parenthesis
)	right parenthesis	*	asterisk
+	plus	,	comma
-	hyphen or minus	.	period
/	slash	:	colon
;	semicolon	<	less than
=	equal	>	greater than
?	question mark	@	at symbol
[	left bracket	\	backslash
]	right bracket	^	circumflex
_	underscore	`	backquote
{	left brace	|	vertical bar
}	right brace	~	tilde

There must also be some way of dividing the source program into lines, typically a newline character or a sequence of newline and carriage return characters. Line endings are significant in delimiting preprocessor directives and one type of comments (see below).

The blank (space) character, tabulators, line endings, and comments (see below) are collective known as *whitespace*. Beyond separating tokens (see below) and the significance of line endings in preprocessor directives, whitespace is ignored.

2.2 Comments

There are two kinds of comments:

- A comment introduced by // extends to the end of the line.
- A comment introduced by /* extends to the first occurrence of */.

Occurrence of // or /* inside a string literal (see below) does not introduce a comment.

2.3 Tokens

The characters making up a MetaSL program are collected into lexical tokens according to the rules presented in the following subsections. There are six classes of tokens: operators, separators, identifiers, typenames, reserved words, and literals.

The compiler always uses the longest possible sequence of characters when reading from left to right to form a token, even if that does not result in a legal MetaSL program, e.g. the sequence of characters "`a--b`" is interpreted as the tokens "`a`", "`--`", and "`b`", which is not a legal MetaSL expression, even though the sequence of tokens "`a`", "`-`", "`-`", "`b`" might constitute a legal expression.

Copyright © 2009 mental images

2.4 Operators and Separators

These are the operators of MetaSL grouped by precedence, from highest to lowest:

<table>
<tr><th>Operation Name</th><th>Operator Expression</th></tr>
<tr><td>scope resolution</td><td>qualified_name::identifier</td></tr>
<tr><td>scope resolution</td><td>qualified_name::typename</td></tr>
<tr><td>member selection</td><td>expression.identifier</td></tr>
<tr><td>subscripting</td><td>expression[expression]</td></tr>
<tr><td>function call</td><td>qualified_name(expression)</td></tr>
<tr><td>value construction</td><td>qualified_type(expression)</td></tr>
<tr><td>post increment</td><td>lvalue++</td></tr>
<tr><td>post decrement</td><td>lvalue--</td></tr>
<tr><td>pre increment</td><td>++lvalue</td></tr>
<tr><td>pre decrement</td><td>--lvalue</td></tr>
<tr><td>not</td><td>!expression</td></tr>
<tr><td>unary minus</td><td>-expression</td></tr>
<tr><td>unary plus</td><td>+expression</td></tr>
<tr><td>member selection</td><td>expression.expression</td></tr>
<tr><td>multiply</td><td>expression * expression</td></tr>
<tr><td>divide</td><td>expression / expression</td></tr>
<tr><td>modulo</td><td>expression % expression</td></tr>
<tr><td>add</td><td>expression + expression</td></tr>
<tr><td>subtract</td><td>expression - expression</td></tr>
<tr><td>less than</td><td>expression < expression</td></tr>
<tr><td>less than or equal</td><td>expression <= expression</td></tr>
<tr><td>greater than</td><td>expression > expression</td></tr>
<tr><td>greater than or equal</td><td>expression >= expression</td></tr>
<tr><td>equal</td><td>expression == expression</td></tr>
<tr><td>not equal</td><td>expression != expression</td></tr>
<tr><td>and</td><td>expression && expression</td></tr>
<tr><td>or</td><td>expression || expression</td></tr>
<tr><td>simple assignment</td><td>lvalue = expression</td></tr>
<tr><td>multiply and assign</td><td>lvalue *= expression</td></tr>
<tr><td>divide and assign</td><td>lvalue /= expression</td></tr>
<tr><td>modulo and assign</td><td>lvalue %= expression</td></tr>
<tr><td>add and assign</td><td>lvalue += expression</td></tr>
<tr><td>subtract and assign</td><td>lvalue -= expression</td></tr>
<tr><td>conditional expression</td><td>expression ? expression : expression</td></tr>
<tr><td>sequencing</td><td>expression , expression</td></tr>
</table>

Unary operators and assignment operators are right-associative; all others are left-associative.

The separators of MetaSL are “`{`”, “`}`”, “`:`”, and “`;`”.

2.5 Identifiers and Typenames

An *identifier* is an alphabetic character followed by a possibly empty sequence of alphabetic characters, decimal digits, and underscores, that is neither a typename nor a reserved word (see below).

 Copyright © 2009 mental images

A *typename* has the same lexical structure as an identifier, but is the name of a built-in type or type defined by the user with a structure or enumeration declaration.

2.6 Reserved Words

These are the reserved words of MetaSL:

```
abstract     after      annotation  attachment        auto         before
bool2        bool3      bool4       bool              break        bsdf
case         catch      char        class             const_cast   const
continue     decltype   default     delete            do           dynamic_cast
else         enum       event       explicit          export       extern
false        finally    final       foreach           for          friend
global       graph      if          import            inline       inout
input        int2       int3        int4              interface    int
in           long       member      mutable           namespace    native
new          operator   output      out               phaser       phenomenon
private      protected  public      reinterpret_cast  return       scenedata
sealed       shader     short       signed            sizeof       state
static_cast  static     struct      super             switch       technique
template     texture1D  texture2D   texture3D         textureCUBE  this
throw        true       try         typedef           typeid       typename
uniform      union      unsigned    using             varying      virtual
void         while

half       half2      half3      half4      half2x2    half2x3    half3x2    half3x3
half3x4    half4x3    half4x4    half4x2    half2x4

float      float2     float3     float4     float2x2   float2x3   float3x2   float3x3
float3x4   float4x3   float4x4   float4x2   float2x4

double     double2    double3    double4    double2x2  double2x3  double3x2  double3x3
double3x4  double4x3  double4x4  double4x2  double2x4
```

2.7 Literals

Literals are a means to directly denote values of simple types.

2.7.1 Boolean Literals

The boolean literals are `true` and `false`.

2.7.2 Integer Literals

Integer literals can be given in octal, decimal, or hexadecimal base.

A decimal literal is a non-empty sequence of decimal digits.

An octal literal is the digit zero followed by a non-empty sequence of octal digits (the digits from zero to seven inclusive).

Copyright © 2009 mental images

A hexadecimal literal is the digit zero, followed by the character `x` or `X`, followed by a non-empty sequence of hexadecimal digits, i.e. the decimal digits and the letters `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, and `F`.

2.7.3 Floating-point Literals

A floating point literal is a possibly empty sequence of decimal digits, optionally followed by a decimal point (the character period) and a possibly empty sequence of decimal digits, optionally followed by an exponent given by the letter `e` or `E`, an optional sign (`-` or `+`) and a non-empty sequence of decimal digits. Either the decimal point or the exponent need to be present. There has to be at least one digit preceding or following the decimal point.

2.7.4 String Literals

A string literal is a possibly empty sequence of characters not including a double quote or escape sequences, enclosed in double quotes. A string literal may not include a line ending. String literals can be concatenated by juxtaposition. An escape sequence is a backslash followed by one of the following escape codes:

Escape Code	Translation	Escape Code	Translation
`a`	alert (e.g. bell)	`t`	horizontal tab
`b`	backspace	`\`	backslash
`f`	form feed	`'`	single quote
`n`	newline	`"`	double quote

 Copyright © 2009 mental images

3 Data types

MetaSL defines a collection of built-in data types tailored for the types of tasks shaders need to accomplish. In addition, these types can be used to define custom structures as described on page 44.

Below is a list of built-in MetaSL data types with a brief description of each type:

- `float, double, half` – A single real number.
- `int` – An integer, including positive and negative numbers as well as zero.
- `bool` – A Boolean value that is either `true` or `false`.
- `float2, float3, float4, double2, double3, double4, half2, half3, half4` – Vectors with real number elements.
- `int2, int3, int4` – Integer vectors.
- `bool2, bool3, bool4` – Boolean vectors.
- `float2x2, float2x3, float3x2, float3x3, float4x3, float3x4, float2x4, float4x2, float4x4, double2x2, double2x3, double3x2, double3x3, double4x3, double3x4, double2x4, double4x2, double4x4, half2x2, half2x3, half3x2, half3x3, half4x3, half3x4, half2x4, half4x2, half4x4` – Matrices of various dimensions.
- `texture1D, texture2D, texture3D, textureCUBE` – Texture maps.
- `Color` – A color with an alpha component and access to red, green, and blue components.
- `Spectrum` – A purely implementation independent representation of color.
- `Ray` – A pair of `float3` values that define the origin and direction of a ray.
- `String` – A character string.
- `Shader` – A reference to a shader.
- `Particle_map` – A multidimensional spatial database.
- `Light_profile` – A description of light emittance for real world lights.

Fundamental types begin with a lower case letter and are keywords in MetaSL. Other types are capitalized and are not keywords and therefore can be used in general as identifiers. For example it is legal to write the following user defined function:

```
bool Color() { return false; }
```

But it is not legal to write the following:

```
bool int() { return false; }
```

Some MetaSL types have member variables and methods which are accessed by placing a period character (`.`) after the type instance followed by the name of the member. For example `v.x` accesses the `x` member of `v`.

Copyright © 2009 mental images

MetaSL also defines a set of iterator and option types used by other built-in functions and language constructs. These types provide methods to access the data they represent and are described in more details later in this document.

- `Light_iterator` – Used with `foreach`.
- `Sample_iterator` – Used with `foreach`.
- `Texture..._sampler_...` – Versions for different texture coordinate dimensions and return type marked as `...` which are replaced with a specific dimension and type.
- `Particle_map_options` – Used with `Particle_map` to control the evaluation of particle maps.
- `Trace_options` – Used with `trace()`.
- `Occlusion_options` – Used with `occlusion()`.
- `Irradiance_options` – Used with `irradiance()`.

These data types support the assignment operator (=), but no other operators. They also provide a copy constructor that allows an instance to be constructed from another instance, in addition to any specific constructors for each data type.

Note that these iterator and option types cannot be declared as shader input or output parameters.

3.1 Constructors

An instance of a MetaSL data type can be constructed by calling the type's constructor. The constructor syntax consists of the type name, followed by a comma separated list of arguments contained in parenthesis. For example, the following expression evaluates to a value of type `float3`:

```
float3(0.3, 0.1, 0.7)
```

When declaring a variable or shader input or member the type constructor can be invoked by appending the constructor parameters, enclosed in parenthesis, to the variable name in the declaration. If a variable declaration contains an initializer, it will be treated as if the variable was constructed taking the right hand side of the assignment statement as the constructor's parameter.

For example, the following three cases are interpreted to have the same meaning, which is to invoke the `float` type constructor with the literal value `0.0` as a parameter and initialize the variable with the resulting value:

```
float x(0.0);
float y = float(0.0);
float z = 0.0;
```

Cast style syntax also allows conversion from one type to another, but is equivalent to using a constructor. The cast syntax consist of a type name surrounded by parenthesis, which is inserted before the value to cast.

The following two expressions are equivalent:

 Copyright © 2009 mental images

```
(float)1  // cast an int to a float
float(1)  // construct a float from an int
```

Variables, whether shader output parameters, member variables, or local variables, do not have to be initialized with a constructor, however a variable must be initialized before it can be read from, with some exceptions: input parameters do not have to be initialized before they are used and they can only be initialized within the input parameter declaration. Also the `Light_iterator`, `Sample_iterator`, `Particle_map_options`, `Trace_options`, `Occlusion_options`, and `Irradiance_options` types provide default constructors and do not require initialization.

In the following sections, detailed descriptions of each type identify the overloaded constructor versions supported by that type.

3.2 Operators

Most MetaSL types support all or some of a standard set of operators listed below:

Operator	Meaning
=	Assignment
/	Divide
/=	Divide and assign to left operand
+	Add
+=	Add and assign to left operand
-	Subtract or unary negate
-=	Subtract and assign to left operand
*	Multiply
*=	Multiply and assign to left operand
%	Modulo
%=	Modulo and assign
==	Equal
!=	Not equal
<=	Less than or equal
<	Less than
>=	Greater than or equal
>	Greater than
&&	Logical and
!	Logical not
||	Logical or
++	Increment
--	Decrement

Operators can also be overloaded based on the types of operands. Operator overloading allows operators to be implemented to support user defined types or combinations of built-in and user defined types.

The comma operator and ternary conditional operator (represented by the '?' character) are also supported for use in expressions; however it is not possible to overload them. The comma operator evaluates to the value of the right most expression in the comma separated list of expressions and shares its type. The '?' operator evaluates to the value of the second or third operands depending on the result of the first operand.

If the first operand is `true` the result is the second operand otherwise it is the third operand. The second and third operand must have the same type, which defines the type of the expression.

In the following sections, detailed descriptions of each type identify the operators supported by that type.

 Copyright © 2009 mental images

3.3 Scalars – float, double, half, int and bool

A `float, half` or `double` represents an approximation of a mathematical "real" number. MetaSL does not define the amount of precision nor the smallest or largest values representable by these types however it is guaranteed that `double` will have at least as much precision as `float` and `float` will have at least as much precision as `half`. The `float, double,` and `half` types all represent a numeric value, but also provide a hint to the compiler to indicate variables that may require more or less precision.

Different platforms may choose to use different representations including floating or fixed point. Some platforms may use relatively low precision floating point for performance reasons.

An `int` represents a single "whole" number (i.e. a number without a fractional component). MetaSL does not strictly define the underlying representation of `int` that may be used on different platforms, nor does it define the smallest or largest possible values that can be represented.

A `bool` represents a single boolean value with possible values `true` and `false`.

3.3.1 Constructors

All the scalar types can be constructed from other scalar types even if the result is a loss of precision. When `bool` values are converted to numeric values, `true` is assigned the value one and `false` is assigned the value zero. When numeric values are converted to `bool` any non-zero value is assigned the value `true` and zero is assigned the value `false`.

For example:

```
float x(5);
int y(x);
bool z(x);
```

All three constructor calls above are legal and result in three variables initialized with the following values:

Variable	Value
x	5.0
y	5
z	true

3.3.2 Operators

The `float, half, double,` and `int` types support the following operators:

```
=     /     /=    +     +=    -
-=    *     *=    ==    !=    <=
<     >=    >     ++    --
```

The `int` type additionally supports the modulo operators:

%	%=

The bool type supports the following operators:

=	==	!=	&&
!	||		

All scalar types support the ternary ? operator. The first operand is the conditional operand which selects values from the third and fourth operands for the result. The conditional operand must be of type bool. The second and third operands can be any type with the restriction that they must be the same type.

3.3.3 Conversion

When required by use in an expression, an instance of a scalar type will be implicitly converted to another scalar type provided there is no loss of precision. For example, a half can be automatically converted to a float but an explicit cast or constructor call is required to convert a double to a float.

The following table lists the types each scalar type can be automatically converted to:

Type	Can be converted to
bool	int, half, float, or double
int	half, float, or double
half	float or double
float	double

Since conversions from lower precision types to higher precision types are implicit, but conversions that result in a loss of precision require an explicit conversion, lower precision type arguments can be passed as higher precision function parameters provided the parameter is an in type parameter.

Similarly, higher precision type arguments can be passed as lower precision function parameters provided the parameter is an out type parameter. When a parameter is an out parameter, a lower precision argument cannot be substituted.

The built-in arithmetic, assignment, and relational operators are overloaded to support mixing scalar operands with vectors. The implementations of these operators will treat a scalar expression as a vector of the same size as the other vector operand in this case. Each component of the constructed vector will be set to the value of the scalar. Note that all vector operands in an expression must be of the same dimension.

For example:

```
float3 x(1, 2, 3);
float3 y = x - 1;
```

In this example, the int literal value 1 is part of an expression involving x which is a float3. Conceptually, the int is converted to a float which is in turn converted to a float3 using the float value for each

 Copyright © 2009 mental images

component. The resulting value of the `y` variable is `<0, 1, 2>`.

Similarly, instances of type `int` and `bool` can be used in expressions containing vectors. Overloaded versions of the arithmetic operators support mixing integers with scalar or integer vectors and boolean values with boolean vectors.

3.4 Vectors – float, double, half, int, and bool

MetaSL provides two, three, and four component vector types with either `float`, `double`, `half`, `int`, or `bool` component types. Vectors are named by taking the fundamental type name and appending the dimension of the vector (possible values for the dimension are 2, 3, and 4).

For example:

```
float3 f3;  // a three dimensional vector of floats
int2 i2;    // a two dimensional vector of ints
bool4 b4;   // a four dimensional vector of bools
```

3.4.1 Constructors

A vector can be constructed from a scalar provided the element type of the constructed vector matches the scalar type. A vector can be constructed from any other vector of equal size including `Color`, which may result in a loss of precision. A vector can be constructed from other vectors or scalars provided the total number of components is the same and the element types of the constructed vector match the element types of the constructor arguments. A `float3` vector as well as a `float4` vector can be constructed from `Spectrum`.

For example, the following table shows all constructors for the `float3` vector type:

```
float3(float)                float3(bool3)
float3(float,float,float)    float3(int3)
float3(float,float2)         float3(half3)
float3(float2,float)         float3(float3)
float3(Spectrum)             float3(double3)
```

When `bool` values are converted to numeric values, `true` is assigned the value one and `false` is assigned the value zero. When numeric values are converted to `bool` any non-zero value is assigned the value `true` and zero is assigned the value `false`.

When a vector is constructed from a single value, this value will be copied to all elements of the vector.

Some examples:

```
bool   b1 = true;           // a boolean value to work with
int    i0 = 0,   i4 = 4;    // some scalar values to work with
float  s2 = 2.0, s3 = 3.0;  // more scalar values
float4 v4(b1, s2, s3, i4);  // 4-float constructor, implicit conversions to float
float2 v2(v4.xy);           // float2 constructor
float3 v3(i0, v2);          // (float,float2) constructor, implicit conversion of i0
bool3  vb3(v3);             // conversion of equal sized vectors with lost precision
```

These four vector constructor calls result in the three vectors initialized with the following values:

 Copyright © 2009 mental images

Variable	Value
v4	<1.0, 2.0, 3.0, 4.0>
v2	<1.0, 2.0>
v3	<0.0, 1.0, 2.0>
vb3	<false, true, true>

3.4.2 Members

The vector types support member variables to access their components and follow a common scheme to determine which members are available for each vector type.

The 'x', 'y', 'z', and 'w' members provide access to up to four components. A particular vector type will only support the first n components where n is the dimension of the vector. For example, float2 supports the 'x' and 'y' members.

When referring to vector components, multiple components can be accessed at the same time, the result being another vector of the same or different length. Also the order of the components can be arbitrary and the same component can be used more than once. For example given a vector V of length three:

- V.xy Returns a 2 component vector <x,y>
- V.zyx Returns a 3 component vector <z,y,x>
- V.xxyy Returns a 4 component vector <x,x,y,y>

A similar syntax can be used as a mask when writing to a vector. To form a proper lvalue, such that it can be used on the left side of an assignment or as out and inout parameter, a component can occur at most once in the components selection.

```
V.yz = float2(0.0, 0.0);
```

In this example the y and z components are set to 0.0 while the x and w components are left unchanged.

Vector components can also be accessed using array indices and the array index can be a variable.

```
float sum = 0.0;
for (int i=0; i<4; i++)
    sum += V[i];
```

In this example the float4 V has its components summed using a loop.

These rules for accessing vector components apply not only to vector variables and the results of functions which return a vector type, but in general to all vector valued expressions.

```
cross(v1, v2).z
```

In this example the z component of the vector value returned from the cross function is accessed.

3.4.3 Operators

Vectors support math and comparison operators in a component-wise fashion. The operator is applied to each component of the operand vectors independently and the result is a vector of the same size as the operands. The operand vectors must be the same size or one must be a scalar (in which case it is promoted to a vector with the same dimension as the other operand).

The float2, float3, float4, double2, double3, double4, half2, half3, half4, int2, int3 and int4 types support the following operators:

```
=     /     /=    +     +=    -
-=    *     *=    ==    !=    <=
<     >=    >     ++    --
```

The int2, int3, and int4 types additionally support the modulo operators:

```
%     %=
```

The bool2, bool3 and bool4 types support the following operators:

```
=     ==    !=    &&    !     ||
```

The vector types also support the ternary conditional ? operator, which operates in a component-wise fashion. The first operand is the conditional operand which selects values from the second and third operands for the result. The conditional operand must be a single bool value or a boolean vector, with the restriction that if the conditional operand is a vector, the operands must all be vectors of the same length.

```
float2 v1(1.0, 2.0);
float2 v2(2.0, 1.0);
float2 result = v1 < v2 ? float2(3.0, 4.0) : float2(5.0, 6.0);
```

In this example the result variable would hold the value <3.0, 6.0>.

3.4.4 Conversion

A value of type float4 can be used implicitly anywhere a Color is used and it will be automatically converted. The 'x' component maps to the 'r' component of Color, 'y' to 'g', 'z' to 'b', and 'w' to 'a'.

When required by use in an expression, an instance of a vector type will be implicitly converted to another vector type of the same length provided there is no loss of precision.

The following table lists the automatic conversion rules for each vector type:

 Copyright © 2009 mental images

Type	Can be converted to
`bool` vectors	`int`, `half`, `float`, or `double` vectors
`int` vectors	`half`, `float`, or `double` vectors
`half` vectors	`float` or `double` vectors
`float` vectors	`double` vectors

Since conversions from lower precision types to higher precision types are implicit, but conversions that result in a loss of precision require an explicit conversion, lower precision type arguments can be passed as higher precision function parameters provided the parameter is an `in` type parameter.

Similarly, higher precision type arguments can be passed as lower precision function parameters provided the parameter is an `out` type parameter. When a parameter is an `out` parameter, a lower precision argument cannot be substituted.

Copyright © 2009 mental images

3.5 Matrices – float, double, and half

MetaSL provides several matrix types with row and column sizes ranging from two to four. Matrix elements can be of type `float`, `double`, or `half`. Matrix types are named `type[rows]x[columns]` where `type` is one of `float`, `double`, or `half`, `[rows]` is the number of rows and `[columns]` is the number of columns.

Specifically, the built-in matrix types are: `float2x2`, `float2x3`, `float3x2`, `float3x3`, `float3x4`, `float4x2`, `float2x4`, `float4x3`, `float4x4`, `double2x2`, `double2x3`, `double3x2`, `double3x3`, `double3x4`, `double4x2`, `double2x4`, `double4x3`, `double4x4`, `half2x2`, `half2x3`, `half3x2`, `half3x3`, `half3x4`, `half4x2`, `half2x4`, `half4x3`, and `half4x4`.

3.5.1 Constructors

Matrices can be constructed from a series of scalars in row-major order where the number of scalars passed to the constructor matches the number of elements of the matrix.

For example:

```
float4x3 mat(
    1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.5, 0.0);
```

Matrices can also be constructed from a series of vectors where each vector represents a row of the matrix. The dimension of the vectors must match the size of rows in the matrix.

The following example produces a matrix initialized to the same values as that of the previous example above:

```
float3 row0(1.0, 0.0, 0.0);
float3 row1(0.0, 1.0, 0.0);
float3 row2(0.0, 0.0, 1.0);
float3 row3(0.0, 0.5, 0.0);
float4x3 mat(row0, row1, row2, row3);
```

A third way to construct a matrix is from another matrix. The matrix used to construct another matrix must at least have the same number of rows and columns of the constructed matrix, but may have more. For example a `float3x3` can be constructed from a `float4x4`. Note that a `float3x2` cannot be constructed from a `float2x4`; a `float2x4` has more total elements than `float3x2`, but less rows.

3.5.2 Members

The matrix types use array notation to provide access to their members, which are rows of the matrix. An index of zero refers to the first row of the matrix and indices of up to $n - 1$ (where n is the number of rows) provide access to the remaining rows.

The data type of matrix rows are vectors with dimension equal to the number of columns of the matrix. Since rows are vectors and vectors support array syntax to access vertex elements, individual elements of a matrix can be accessed with syntax similar to a multidimensional array.

 Copyright © 2009 mental images

For example:

```
float4x3 mat(
    1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.5, 0.0);
float3 row;
float element;

// row will equal <0.0, 1.0, 0.0> after the assignment
row = mat[1];

// element will equal 0.5 after the assignment
element = mat[3][1]
```

3.5.3 Operators

The matrix types support the following operators:

`=`	`/`	`/=`	`+`	`+=`	`-`
`-=`	`*`	`*=`	`==`	`!=`	

The multiplication operator is supported to multiply two matrices or a matrix and a vector and will perform a linear algebra style multiplication between the two. As should be expected when multiplying two matrices, the number of columns of the matrix on the left must equal the number of rows of the matrix on the right. The result of multiplying a `NxT` matrix with a `TxM` matrix is a `NxM` matrix. A vector can be multiplied on the right or left side provided the number of elements equals the number of rows when the vector is on the left side of the matrix and the number of elements equals the number of columns when the vector is on the right.

The equality and inequality comparison operators, `==` and `!=`, return a scalar `bool`. They require that the operands are matrices of the same dimension.

All other operators are applied component-wise and require that the operands are matrices of the same dimension.

Overloaded versions of the operators are provided to support mixing a matrix and a scalar in an expression. In this case, the scalar operand is applied component-wise to each element of the matrix.

3.5.4 Conversion

When required by use in an expression, an instance of a matrix type will be implicitly converted to another matrix type of the same dimensions provided there is no loss of precision.

The following table lists the automatic conversion rules for each matrix type:

Type	Can be converted to
`half` matrices	`float` or `double` matrices
`float` matrices	`double` matrices

Since conversions from lower precision types to higher precision types are implicit, but conversions that result in a loss of precision require an explicit conversion, lower precision type arguments can be passed as higher precision function parameters provided the parameter is an `in` type parameter.

Similarly, higher precision type arguments can be passed as lower precision function parameters provided the parameter is an `out` type parameter. When a parameter is an `out` parameter, a lower precision argument cannot be substituted.

 Copyright © 2009 mental images

3.6 Textures and samplers

The MetaSL texture types (`texture1D`, `texture2D`, and `texture3D`) represent references to data stored in one, two, or three dimensional uniform grids. The `textureCUBE` type is organized as a collection of six two dimensional grids, one for each direction (`+x`, `-x`, `+y`, `-y`, `+z`, and `-z`).

A data element in a texture can have one of the following MetaSL types: `float`, `float2`, `float3`, `float4`, `double`, `double2`, `double3`, `double4`, `half`, `half2`, `half3`, `half4`, `Color` or `Spectrum`. The texture types do not specify the type of elements. Instead, a texture sampler object facilitates looking up values of a supported type.

A texture sampler object is declared as a local variable in a shader and is initialized with the texture object it is to sample along with other optional parameters.

There are multiple types of sampler objects, one for each combination of return type and coordinate type. Texture coordinates are of type `float`, `float2`, or `float3` corresponding to their coordinate dimension, where cube map samplers behave like 3D texture and expect a `float3` texture coordinate. The lookup result can be of any the texture element types described above. The following table lists all available texture sampler types:

```
Texture1D_sampler_float      Texture2D_sampler_float      Texture3D_sampler_float
Texture1D_sampler_float2     Texture2D_sampler_float2     Texture3D_sampler_float2
Texture1D_sampler_float3     Texture2D_sampler_float3     Texture3D_sampler_float3
Texture1D_sampler_float4     Texture2D_sampler_float4     Texture3D_sampler_float4
Texture1D_sampler_double     Texture2D_sampler_double     Texture3D_sampler_double
Texture1D_sampler_double2    Texture2D_sampler_double2    Texture3D_sampler_double2
Texture1D_sampler_double3    Texture2D_sampler_double3    Texture3D_sampler_double3
Texture1D_sampler_double4    Texture2D_sampler_double4    Texture3D_sampler_double4
Texture1D_sampler_half       Texture2D_sampler_half       Texture3D_sampler_half
Texture1D_sampler_half2      Texture2D_sampler_half2      Texture3D_sampler_half2
Texture1D_sampler_half3      Texture2D_sampler_half3      Texture3D_sampler_half3
Texture1D_sampler_half4      Texture2D_sampler_half4      Texture3D_sampler_half4
Texture1D_sampler_color      Texture2D_sampler_color      Texture3D_sampler_color
Texture1D_sampler_spectrum   Texture2D_sampler_spectrum   Texture3D_sampler_spectrum
```

```
TextureCUBE_sampler_float
TextureCUBE_sampler_float2
TextureCUBE_sampler_float3
TextureCUBE_sampler_float4
TextureCUBE_sampler_double
TextureCUBE_sampler_double2
TextureCUBE_sampler_double3
TextureCUBE_sampler_double4
TextureCUBE_sampler_half
TextureCUBE_sampler_half2
TextureCUBE_sampler_half3
TextureCUBE_sampler_half4
TextureCUBE_sampler_color
TextureCUBE_sampler_spectrum
```

3.6.1 Constructors

When instances of the texture types are declared as input parameters, they can be initialized with a `String` to name the resource from which the texture should be created.

For example:

```
input:
  texture2D texture = "bricks.jpg";
```

When a texture is constructed as a local variable or function parameter it can only be constructed from another texture of the same type.

Texture samplers can be constructed from texture instances. The coordinate dimension of the texture sampler must match the type of the texture. The following table lists all constructors available for texture samplers, where `...` can be any of the texture element type names:

```
Texture1D_sampler_...(   texture1D   texture);
Texture1D_sampler_...(   texture1D   texture, float  blur);
Texture1D_sampler_...(   texture1D   texture, float  blur, String filter);

Texture2D_sampler_...(   texture2D   texture);
Texture2D_sampler_...(   texture2D   texture, float  blur);
Texture2D_sampler_...(   texture2D   texture, float2 blur);
Texture2D_sampler_...(   texture2D   texture, float  blur, String filter);
Texture2D_sampler_...(   texture2D   texture, float2 blur, String filter);

Texture3D_sampler_...(   texture3D   texture);
Texture3D_sampler_...(   texture3D   texture, float  blur);
Texture3D_sampler_...(   texture3D   texture, float3 blur);
Texture3D_sampler_...(   texture3D   texture, float  blur, String filter);
Texture3D_sampler_...(   texture3D   texture, float3 blur, String filter);

TextureCUBE_sampler_...( textureCUBE texture);
TextureCUBE_sampler_...( textureCUBE texture, float  blur);
TextureCUBE_sampler_...( textureCUBE texture, float3 blur);
TextureCUBE_sampler_...( textureCUBE texture, float  blur, String filter);
TextureCUBE_sampler_...( textureCUBE texture, float3 blur, String filter);
```

The two optional parameters to the texture sampler constructor are:

- `float... blur` – A bias to the size of the filter kernel such that values that are greater than one increase the size of the kernel and values that are less than one decrease it. The blur parameter is either a vector with a dimension that matches the dimension of the texture coordinate or a scalar. A `float` blur parameter is compatible with texture lookups of any dimension. When not specified in the constructor, the default value is `1`.
- `String filter` – The type of filter kernel to use. The filter kernel parameter must be a literal string with one of the following values:

 Copyright © 2009 mental images

`"default"` – Implementation dependent
`"box"` – Box filter
`"triangle"` – Triangle filter
`"gauss"` – Gaussian filter
`"cmitchell"` – Clipped Mitchell filter
`"clanczos"` – Clipped Lanczos filter
`"none"` – No filter

When not specified in the constructor, the default value is `"default"`, which allows the implementation to select the most suitable filtering strategy for the particular platform.

3.6.2 Texture Members

The MetaSL texture types have the following members:

- `int width` – The width of the texture (in pixels). When looking up a value from a texture, the `x` coordinate refers to the dimension specified by width.
- `int height` – The height of the texture (in pixels). When looking up a value from a texture, the `y` coordinate refers to the dimension specified by height.
- `int depth` – The depth of the texture (in pixels). When looking up a value from a texture, the `z` coordinate refers to the dimension specified by depth.

Not all properties are available for all texture types:

- `texture1D` – width
- `texture2D` – width, height
- `texture3D` – width, height, depth
- `textureCUBE` – width, height. Note that width and height refer to the dimension of a single face of the cube. All faces have the same dimension.

3.6.3 Sampler Members

All the texture sampler types supply methods to lookup data from the texture. There are two versions of the lookup function with one accepting optional texture coordinate derivatives. The following table lists the available methods, where ... can be any of the texture element type names:

```
Texture1D_sampler_...   ::lookup( float  s);
Texture1D_sampler_...   ::lookup( float  s,   float  ds_dx,   float  ds_dy);

Texture2D_sampler_...   ::lookup( float2 st);
Texture2D_sampler_...   ::lookup( float2 st,  float2 dst_dx,  float2 dst_dy);

Texture3D_sampler_...   ::lookup( float3 str);
Texture3D_sampler_...   ::lookup( float3 str, float3 dstr_dx, float3 dstr_dy);
```

```
TextureCUBE_sampler_... ::lookup( float3 str);
TextureCUBE_sampler_... ::lookup( float3 str, float3 dstr_dx, float3 dstr_dy);
```

The following is an example usage of a texture sampler:

```
input:
  texture2D texture;

output:
  Color result;

  void main()
  {
      Texture2D_sampler_color sampler =
          Texture2D_sampler_color(texture);

      result = sampler.lookup(
          texture_coordinate[0].xy,
          ddx(texture_coordinate[0]).xy,
          ddy(texture_coordinate[0]).xy);
  }
```

3.6.4 Operators

Texture types support the following operators:

`=` `==` `!=`

Texture sampler types support the following operator:

`=`

3.6.5 Conversion

All the texture types support an automatic conversion to type `bool`. When used in an expression that forces a conversion to `bool`, a valid texture will evaluate to `true` and an invalid texture will evaluate to `false`. A texture variable that is an input parameter is considered invalid if it is not supplied with actual texture data at render time. A texture local variable or function parameter is invalid if it has not been assigned from a valid texture.

The following example shows a MetaSL code fragment which assumes the existence of a `texture2D` type variable named `texture`.

 Copyright © 2009 mental images

```
if (texture) {
    float pixel_size = 1.0 / texture.width;
}
```

The texture types can also be initialized with a `String` to name the resource from which data should be loaded. Initialization of this type is only permitted for shader input parameters and a literal string must be used.

3.7 Color

Instances of the `Color` type are identical in structure to instances of `float4` although their members are referred to by `[r, g, b, a]` instead of `[x, y, z, w]` to refer to the red, green, blue and alpha components, respectively.

Colors can be used any place a `float4` can be used and all operations that apply to `float4` will work with `Color` as well. The primary purpose of this type is for code readability. Otherwise this type is logically synonymous with `float4`. In particular, for the purpose of shader method overloading or function overloading, `float4` and `Color` are not considered different types.

While the `Color` type shares members and operators in common with `float4` it also provides a meaning for the data – that it represents a color with alpha.

A particular renderer implementation is free to substitute an alternative representation for the `Color` type as long as it supports any access to the `r, g, b,` or `a` components. When used on a platform that utilizes an alternative representation of color, accessing the red, green, or blue components of a `Color` variable may force a conversion to rgb color.

3.7.1 Constructors

Since the `Color` type is synonymous with `float4`, it supports all the constructor versions supported by `float4`, which are: `Color` can be constructed from a single `float` which assigns all its components to the `float` value. `Color` can be constructed from other `float` vectors or `float` scalars provided the total number of components is the same. `Color` can be constructed from any other four-dimensional vector, which may result in a loss of precision. `Color` can be constructed from `Spectrum`.

The following table shows all constructors for the `Color` type:

```
Color(float)                        Color(bool4)
Color(float,float,float,float)      Color(int4)
Color(float,float3)                 Color(half4)
Color(float3,float)                 Color(float4)
Color(float2,float2)                Color(double4)
Color(float,float,float2)           Color(Color)
Color(float,float2,float)           Color(Spectrum)
Color(float2,float,float)
```

3.7.2 Operators

Like `float4`, the `Color` type supports math and comparison operators in a component-wise fashion. The `Color` type supports the following operators:

```
=     /     /=    +     +=    -
-=    *     *=    ==    !=    <=
<     >=    >     ++    --
```

 Copyright © 2009 mental images

The `Color` type also supports the ternary conditional `?` operator, which operates in a component-wise fashion. The first operand is the conditional operand which selects values from the second and third operands for the result. The conditional operand must be a single `bool` or `bool4` type value.

3.7.3 Conversion

A value of type `Color` can be used implicitly anywhere a `float4` is used and it will be automatically converted. The `'r'` component maps to the `'x'` component of `float4`, `'g'` to `'y'`, `'b'` to `'z'`, and `'a'` to `'w'`.

The `Color` type is also assignment compatible with `Spectrum`. When converting from an instance of `Spectrum` the `Color` instance's alpha component is set to one. When converting to `Spectrum` the alpha component is ignored.

There is no other type of automatic conversion supported for `Color`.

3.8 Spectrum

The Spectrum type provides an implementation independent abstraction to represent light. If a shader does not need direct access to the red, green, blue, or alpha components of the Color type, then it is preferable to use Spectrum because it gives the most flexibility to the underlying implementation to use the best representation for that platform.

For example, spectral renderers may use many samples over the visual spectrum of 400 to 700nm wavelengths to provide a highly physically accurate simulation. The Spectrum type does not define the number of samples, nor their range, nor at which frequencies they are sampled at. Standard renderers might implement the Spectrum type with a conventional RGB model.

3.8.1 Constructors

Instances of Spectrum can be constructed from a float (resulting in a monochromatic value), a Color, float3, float4 or another instance of Spectrum.

When constructed from a Color or float4, the alpha component of the Color instance or w component of float4 is ignored. The x, y, and z components of float3 and float4 are interpreted as red, green, and blue, respectively.

3.8.2 Operators

Although the number of components of a Spectrum is not defined by MetaSL, the Spectrum type supports math operators and functions which work in a component-wise fashion.

Specifically, Spectrum supports the following operators:

```
=    /    /=   +    +=   -
-=   *    *=   ==   !=
```

3.8.3 Conversion

The Spectrum type is assignment compatible with Color, float3, and float4. Assignment compatible means that these types have an assignment operator and a constructor accepting a Spectrum parameter.

When converting from Color or float4 to Spectrum, the a and w components are ignored. When converting from Spectrum to Color or float4 the a and w components are set to 1.

 Copyright © 2009 mental images

3.9 Ray

The MetaSL Ray type represents a ray in three dimensional space, which is comprised of a position representing the origin of the ray and a direction vector.

Instances of the Ray type are used by the ray tracing methods described in this document.

3.9.1 Constructors

An instance of Ray can be constructed by passing two vectors: the ray origin and direction, in that order. A ray can also be constructed from another ray.

For example:

```
float3 o(0);      // origin
float3 d(0,0,1);  // direction
Ray ray(o, d);
```

3.9.2 Members

The Ray type has two float3 type members:

- origin - The origin of the ray
- direction - The direction of the ray

For example:

```
float3 o = ray.origin;
float3 d = ray.direction;
```

3.9.3 Operators

The Ray type supports the following operators:

=	==	!=

3.10 String

A string is a sequence of characters of arbitrary length. MetaSL uses strings primarily as parameters to identify options or user defined categories. Because of the restricted need for strings in a shading language, and the fact that some platforms have limited or no support for strings, MetaSL defines a limited set of string handling functionality.

Please see *Appendix A* on page 113 for a detailed description of string literals and the MetaSL character set.

3.10.1 Constructors

A string can be constructed from another string, or an instance of one of the following MetaSL types: `float`, `double`, `half`, `int`, `bool`, `float2`, `float3`, `float4`, `double2`, `double3`, `double4`, `half2`, `half3`, `half4`, `int2`, `int3`, `int4`, `bool2`, `bool3`, `bool4`, `float2x2`, `float2x3`, `float3x2`, `float3x3`, `float4x3`, `float3x4`, `float4x2`, `float2x4`, `float4x4`, `double2x2`, `double2x3`, `double3x2`, `double3x3`, `double4x3`, `double3x4`, `double4x2`, `double2x4`, `double4x4`, `half2x2`, `half2x3`, `half3x2`, `half3x3`, `half4x3`, `half3x4`, `half4x2`, `half2x4`, `half4x4`, `Color`, `Spectrum` or `Ray`.

When constructing a string from a value other than another string, the value is converted into a human readable format (e.g. a `float` may be converted to `"1.5"`). This is useful for diagnostic purposes using the `error()`, `warning()`, `info()`, and `progress()` functions described later in this document.

3.10.2 Operators

The `String` type supports the following operators:

`=`	`+`	`+=`	`==`	`!=`
	`<=`	`<`	`>=`	`>`

The `+` and `+=` perform string concatenation and the `<`, `<=`, `>`, and `>=` comparison operators use lexicographical ordering to determine their result.

3.10.3 Conversion

An instance of `String` can be used to initialize texture, light profile or particle map type parameters (specifically: `texture1D`, `texture2D`, `texture3D`, `textureCUBE`, `Light_profile`, and `Particle_map`) to name the resource from which data should be loaded. Initialization of this type is only permitted for shader input parameters and a literal string must be used.

 Copyright © 2009 mental images

3.11 Shader

The MetaSL `Shader` type is used to represent a reference to a shader. It is used is to allow one shader to explicitly evaluate another shader.

MetaSL input parameters are implicitly connectible in shader graphs. When an input is connected to the output of another shader, the connected shader is automatically evaluated and the result placed in the input variable that is the target of the attachment.

The `Shader` type allows a different type of connection to another shader. In this case the input parameter value is the shader (or a reference to it) and not a numeric value.

The advantage is that a shader can explicitly call another shader and request the value of one or more outputs. The shader can also be called multiple times with different values for its inputs and different state values.

3.11.1 Constructors

An instance of `Shader` can only be constructed from another instance of `Shader`. An input parameter of type `Shader` cannot be initialized or constructed and so it is not possible to specify a default value within the input declaration.

3.11.2 Members

The `Shader` type supports a `call()` method to explicitly call the shader referred to by the `Shader` instance:

```
bool Shader::call(
    String parameter_inout,
    String parameter_name,
    T value,
    ...);
```

The `call()` method accepts a variable number of arguments to indicate zero or more input parameters to override and one or more output parameters to retrieve. Variable arguments are specified in the shader call in groups of three variables:

- `parameter_inout` – A literal string which can have the value `"input"` or `"output"` to specify whether the parameter referred to is an input or an output. Input parameters are specified to override the value of the called shader's inputs. Output parameters are specified to retrieve the result(s) of the shader evaluation.
- `parameter_name` – The name of the parameter.
- `value` – The type of this argument should match that of the parameter referred to. When overriding an input, this argument can be a variable, literal, or expression. When specifying an output to retrieve, this argument is an `out` parameter and a variable must be specified to hold the result. In other words, the argument should be legal on the left hand side of an assignment statement.

An example shader call that assumes the existence of a `Shader` type parameter named `shader` and a `float` named `x`:

```
Color result;
shader.call(
    "input", "amount", 0.5*x,
    "output", "result", result);
```

3.11.3 Operators

The `Shader` type supports the following operators:

`=` `==` `!=`

3.11.4 Conversion

The `Shader` type supports an automatic conversion to type `bool`. When used in an expression that forces a conversion to `bool`, a valid shader will evaluate to `true` and an invalid shader will evaluate to `false`. A shader variable that is an input parameter is considered invalid if it is not supplied with an actual shader at render time. A shader local variable or function parameter is invalid if it has not been assigned from a valid shader.

 Copyright © 2009 mental images

3.12 Particle_map

A `Particle_map` represents a collection of arbitrary values associated with points in space such that the values can be efficiently stored and accessed even if they aren't uniformly distributed. The map can be indexed with a three dimensional point at minimum, and can optionally be indexed with a four dimensional point (space and time) and/or a direction vector.

The values stored in a `Particle_map` can be of any MetaSL type including structures, however any single `Particle_map` will store the same type of data at all points. Additionally an instance of a map will always use the same combination of indices to access data in the map.

3.12.1 Constructors

When instances of `Particle_map` are declared as input parameters, they can be initialized with a `String` to name the resource from which the particle map should be loaded. When a particle map is constructed as a local variable or function parameter it can only be constructed from another particle map instance.

3.12.2 Members

The `Particle_map` class provides methods to access data stored in the particle map.

```
bool Particle_map::lookup(
    Particle_map_options options,
    float3 position,
    String name,
    out Type value,
    ...);

bool Particle_map::lookup(
    Particle_map_options options,
    float4 position,
    String name,
    out Type value,
    ...);

bool Particle_map::lookup(
    Particle_map_options options,
    float3 position,
    float3 direction,
    String name,
    out Type value,
    ...);

bool Particle_map::lookup(
    Particle_map_options options,
    float4 position,
    float3 direction,
    String name,
    out Type value,
    ...);
```

Note that while all elements of the map hold the same collections of named data, calls to the `lookup()` method can request a subset of the data as specified by the provided name/value pairs and subsequent calls to `lookup()` can request different name/value combinations.

The `lookup()` particle map method interpolates nearby points to compute the resulting value. If an insufficient number of points are found, the `lookup()` method returns `false`. The `lookup()` method also returns false if the type of indices used don't match those specified when the map was created.

All versions of the `lookup()` method accept a variable number of name/value pairs, which specify the name of the element to access and a variable in which to place the result.

A MetaSL built-in data type named `Particle_map_options` provides control over how particle maps are evaluated, and has the following methods:

```
void Particle_map_options::set_maximum_distance(
    float max_distance);
```

Specifies the furthest distance from the position argument in which values will be considered. If left unspecified, all points will be considered.

```
void Particle_map_options::set_maximum_angle(
    float max_angle);
```

Specifies the furthest angular distance from the direction argument in which values will be considered. If left unspecified, all directions will be considered. The angle is specified in degrees.

```
void Particle_map_options::set_maximum_points(
    int max_points);
```

Specifies the maximum number of points that will be considered. If more points are found than specified by the `max_points` argument, the points closest in distance will be selected. If left unspecified, all points will be considered.

```
void Particle_map_options::set_search_volume(
    float3 min,
    float3 max);

void Particle_map_options::set_search_volume(
    float4 min,
    float4 max);
```

Specifies a region in three or four dimensional space around the position argument in which values will be considered. These values are specified relative to the given position. If left unspecified, all points will be considered.

 Copyright © 2009 mental images

Note that both `set_maximum_distance()` and `set_search_volume()` can be used to simultaneously restrict the search.

The `Particle_map` class also provides search methods which locates nearby points and returns them in a list instead of interpolating them.

```
int Particle_map::search(
    Particle_map_options options,
    float3 position,
    out float3 positions[],
    String name,
    out Type values[],
    ...);

int Particle_map::search(
    Particle_map_options options,
    float4 position,
    out float4 positions[],
    String name,
    out Type values[],
    ...);

int Particle_map::search(
    Particle_map_options options,
    float3 position,
    float3 direction,
    out float3 positions[],
    out float3 directions[],
    String name,
    out Type values[],
    ...);

int Particle_map::search(
    Particle_map_options options,
    float4 position,
    float3 direction,
    out float4 positions[],
    out float3 directions[],
    String name,
    out Type values[],
    ...);
```

These methods output all points found, within the constraints specified by the `Particle_map_options` argument. The out type array parameters will contain the corresponding positions, directions, and values for those points. These arrays impose additional upper bounds on the number of points returned. If the search finds more points than fit in the arrays, the points closest in distance will be reported. These methods return the number of points stored in the arrays. Array elements that are not used remain unchanged.

All versions of the `search()` method accept a variable number of name/value pairs, which specify the name of the element to access and an array variable in which to place the results.

Copyright © 2009 mental images

3.12.3 Operators

The `Particle_map` type supports the following operators:

`=` `==` `!=`

3.12.4 Conversion

The `Particle_map` type supports an automatic conversion to type `bool`. When used in an expression that forces a conversion to `bool`, a valid particle map instance will evaluate to `true` and an invalid particle map instance will evaluate to `false`. A particle map variable that is an input parameter is considered invalid if it is not supplied with actual data at render time. A particle map local variable or function parameter is invalid if it has not been assigned from another valid particle map instance.

Instances of `Particle_map` can also be initialized with a `String` to name the resource from which data should be loaded. Initialization of this type is only permitted for shader input parameters and a literal string must be used.

 Copyright © 2009 mental images

3.13 Light_profile

The MetaSL `Light_profile` type is used to represent a reference to light profile data, which is typically provided by vendors of real-world physical lights to describe precisely how much light is emitted from a point light source in a particular direction.

3.13.1 Constructors

When instances of `Light_profile` are declared as input parameters, they can be initialized with a `String` to name the resource from which the light profile should be loaded. When a light profiles is constructed as a local variable or function parameter it can only be constructed from another light profile instance.

3.13.2 Members

The `Light_profile` class provides methods to lookup the light intensity emitted in a particular direction:

```
Spectrum Light_profile::lookup();

Spectrum Light_profile::lookup(
    float phi,
    float theta);
```

The first version of the `lookup()` method does not require any parameters. It takes the direction in which to sample from the current state: a vector based at the light origin pointing toward the current position being shaded.

The second version allows the lookup direction to be specified. The `phi` and `theta` arguments specify the horizontal and vertical directions, respectively. They are angles specified in degrees in light space.

The return value is the light intensity emitted in the specified direction as defined by the light profile data.

3.13.3 Operators

The `Light_profile` type supports the following operators:

`=` `==` `!=`

3.13.4 Conversion

The `Light_profile` type supports an automatic conversion to type `bool`. When used in an expression that forces a conversion to `bool`, a valid light profile instance will evaluate to `true` and an invalid light profile instance will evaluate to `false`. A light profile variable that is an input parameter is considered invalid if it is not supplied with actual data at render time. A light profile local variable or function parameter is invalid if it has not been assigned from another valid light profile instance.

Instances of `Light_profile` can also be initialized with a `String` to name the resource from which data should be loaded. Initialization of this type is only permitted for shader input parameters and a literal string must be used.

4 Arrays

MetaSL supports arrays of any of the built-in types or user-defined structures. Arrays can have a fixed size or an unspecified size. An unsized array must ultimately have a size specified before the shader is used.

A fixed size array is declared by following the variable declaration with the array size surrounded by square brackets. The size must be a literal value greater than zero of type `int`.

An unsized array is declared the same way with square brackets, except that the size is omitted.

For example:

```
shader my_shader {
  input:
    int n[4]; // Fixed size array.
    int m[];  // Unsized array.
};
```

All array variables have an `int` type member named `count` which provides the number of elements in the array.

Unsized arrays can be declared as input parameters without specifying their size elsewhere in the shader. Ultimately instances of the shader class will specify a concrete array size (and values) before shader instances are used.

An unsized array as input parameter that has an array initializer of fixed size can still have an array of different size assigned to it later.

Arrays declared as function parameters can also be unsized since their size will be determined by the argument passed to the function call. Arrays declared as local variables must have a fixed size or be initialized with another array before their first use.

For example:

```
float sum_array(float values[]) {
    float sum = 0.0;
    for (int i=0; i<values.count; i++)
        sum += values[i];
    return sum;
}

float a_function() {
    float foo[8]; ... // initialize members of foo
    return sum_array(foo);
}
```

This simple example loops over an array and sums the components. The code for this function was written without actual knowledge of the size of the array but when shading the size will be known. Either the array variable will come from an array shader parameter or a fixed size array declared in a calling function.

 Copyright © 2009 mental images

Arrays can be initialized or constructed by using curly braces to enclose a list of values separated by commas. For example:

```
// an array of scalar values
float list[] = {3.4, 1.5, 0.3, -78.2};
```

Sized arrays can be initialized in the same manner, but the size of the initializer must match the declared size of the array.

Each item in an array's initialization list must be an expression which can be converted to the type of the array's members. For example, arrays of values that have a type which can be initialized with a string can be initialized with a list of strings, provided the array variable is an input parameter.

```
// an array of textures
texture2D list[] = {"bricks.jpg", "rocks.jpg"};
```

Note that MetaSL does not support multidimensional arrays.

Copyright © 2009 mental images

5 Structures

struct_declaration : `struct` *identifier* `{` *field_declaration* { *field_declaration* } `}` `;`

field_declaration : *qualified_type field* { `,` *field* } `;`

field : *identifier* [`[` [*integer_literal*] `]`]

MetaSL supports the definition of user defined structures. A structure is a collection of named variables, possibly of different types.

A structure is declared using the keyword `struct` followed by the name of the structure and the declaration of members enclosed in curly braces. Structure member variables are declared in the same manner as local variable declarations, although initializers are not permitted.

```
struct Color_pair {
    Color a;
    Color b;
};
```

In this example a new type called `Color_pair` is defined that is comprised of two colors.

Structure member variables can be of any built-in type or another user-defined structure type to produce a nested structure. Structure members can also be arrays.

A constructor is automatically generated for user defined structures that accepts a list of parameters corresponding to the structure members in the order they are declared. For example:

```
struct Color_and_scalar {
    Color a;
    float b;
};

Color_and_scalar x = Color_and_scalar(Color(1,1,1,1), 1.0);
```

An array of structures can be initialized with a combination of the syntax for array initialization and automatic structure constructors as follows:

```
Color_and_scalar list[] = {
    Color_and_scalar(Color(1,1,1,1), 1.0),
    Color_and_scalar(Color(0,0,0,1), 0.5),
    Color_and_scalar(Color(1,0,0,1), 0.0)
};
```

If a structure contains a member whose type is one of the types that can be initialized with a string, a string can be passed to the structure constructor provided the structure variable is declared as an input parameter. For example:

Copyright © 2009 mental images

```
struct Texture_and_color {
    texture2D texture;
    Color color;
};

input:
    Texture_and_color tc("bricks.jpg", Color(1,1,1,1));
```

Structures can be declared within shader declarations to prevent name collisions. Structures declared within shader declarations can be directly referred to by methods and parameters of the shader holding the declaration. The structure type can also be referred to by other shaders provided the structure name is qualified with the name of the shader holding the declaration.

The following is an example of a structure declared within a shader:

```
shader My_shader {
  input:
    struct Color_and_scalar {
      Color a;
      float b;
    };

    Color_and_scalar x = Color_and_scalar(Color(1,1,1,1), 1.0);
};

shader Another_shader {
  input:
    My_shader::Color_and_scalar x =
        My_shader::Color_and_scalar(Color(1,1,1,1), 1.0);
};
```

Note that a declaration of a structure always defines a new type name and anonymous structures are not supported.

Copyright © 2009 mental images

6 Enumerations

enum_declaration	:	`enum` *identifier* `{` *enum_item_declaration* { `,` *enum_item_declaration* } `}` `;`
enum_item_declaration	:	*identifier* [`=` *expression*]

MetaSL provides the capability to define an enumeration as a convenient way to represent a set of named integer constants.

An enumeration is declared using the keyword `enum` followed by a comma separated list of identifiers enclosed in curly braces.

For example:

```
enum Detail { LOW, MEDIUM, HIGH };
```

The enumerators can be explicitly assigned values as well.

For example:

```
enum Detail {
    LOW    = 1,
    MEDIUM = 2,
    HIGH   = 3
};
```

This example defines a new type called `Detail` with possible values of `LOW`, `MEDIUM` and `HIGH`. Enumeration type values can be implicitly cast to integers which results in an integer with the explicitly assigned value. If explicit values aren't specified, each element is assigned the value of its predecessor plus one. The first element is assigned the value zero.

The values associated with an `enum` are not required to be unique within the type. For example, the following are both legal `enum` declarations:

```
enum Bool_states {
    ON     = 1,
    YES    = 1,
    OFF    = 0,
    NO     = 0
};

enum Foo {
    A = 2,
    B = 1,
    C       // C will implicitly be assigned the value 2
};
```

 Copyright © 2009 mental images

An enumeration type can be declared inside or outside a shader declaration. Declaring an enumeration type inside a shader is useful to prevent naming conflicts with other types.

When an enumeration is declared in a shader, methods and parameters of the shader can refer to the enumeration type and values directly. The enumeration type and values can still be referred to by methods and parameters of other shaders, but such references must be qualified by prefixing the shader name followed by two colons to the enumeration type or value.

The following example illustrates the declaration of an enumeration type inside a shader declaration. The second shader is able to refer to the enumeration type by qualifying references with the name of the shader holding the declaration.

```
shader My_shader {
  input:
    enum Size {
      SMALL, MEDIUM, BIG
    };

    Size size = SMALL;
};

shader Another_shader {
  input:
    My_shader::Size size = My_shader::MEDIUM;
};
```

Note that a declaration of an enum always defines a new type name and anonymous enums are not supported. Additionally, the enum elements themselves define a symbol within the scope where the enum is declared. If an enum element name conflicts with a previous definition, a compile error will result.

7 Typedef

type_declaration	:	`typedef` *qualified_type identifier* ;
qualified_type	:	[[[::] *identifier*] ::] *typename*

The `typedef` specifier allows a new type name to be introduced which acts as a synonym for an existing type name. To declare a new type name, the `typedef` specifier is followed by the name of an existing type and after that the new name, which thereafter acts as a synonym for the existing type.

For example:

```
typedef int INT;
```

This defines a new identifier `INT` which is syntactically synonymous with `int`.

Declarations that use `typedef` to define a type name can only appear at the file scope level. The new type name is valid from the location at which it is declared to the end of the file.

Once a type name is defined it cannot be redefined to name a different type, nor can an existing type name be redefined to refer to a different type. The only exception are system defined non-fundamental types. These are the types defined by MetaSL that are not keywords (and are capitalized instead of lower case).

For example the following is legal:

```
typedef int Ray;
```

The following is not legal:

```
typedef float3 half3;
```

Note that `typedef` does not introduce a new type, but instead only an additional name for an existing type. This means that for the purpose of differentiating parameters to match overloaded functions, the type name isn't enough for two parameters to be considered different.

For example the following is not legal:

```
typedef int INT;

void my_function(INT x) {}
void my_function(int x) {} // error -- redefinition of my_function
```

 Copyright © 2009 mental images

8 Control flow

MetaSL supports the familiar programming constructs that control the flow of a shader's execution. Specifically these are:

- The loop statements `for`, `while`, and `do-while`. The keywords `continue` and `break` are supported to control execution of the loop.
- The branch statements `if` with optional `else` clauses and `switch` statements with cases and optional defaults.
- A `return` statement to terminate a function or method and return a value if the function or method does not have a void return type.

8.1 Loops

for_statement	:	`for (` [*expression*] `;` [*expression*] `;` [*expression*] `)` *statement*
	|	`for (` *variable_declaration* [*expression*] `;` [*expression*] `)` *statement*
while_statement	:	`while (` *expression* `)` *statement*
do_while_statement	:	`do` *statement* `while (` *expression* `)`
foreach_statement	:	`foreach (` *expression* `)` *statement*

A for loop is declared using the keyword `for` followed by three expressions separated by semicolons and enclosed in parenthesis. The first expression is evaluated once before the loop begins. The second expression is evaluated once at the beginning of each iteration of the loop and will terminate the loop if it evaluates to `false`. The third expression is evaluated once at the end of each iteration of the loop. The loop body follows the `for` statement.

For example:

```
for (int i=0; i<10; i++) {
    // ...
}
```

Note that the first expression can declare variables which are visible in scope within the second and third expressions as well as the loop body.

The `continue` statement can be used in the body of the `for` loop to jump to the end of the current loop iteration.

The `break` statement will terminate the loop without further evaluation of the expressions in the `for` statement.

The `do` and `while` loop constructs are similar to each other. A `while` loop begins with the keyword `while` followed by an expression enclosed in parenthesis followed by the loop body. The body of the `while` loop will be executed as long as that expression evaluates to `true`.

When used in `do` or `while` loops, the `continue` statement will jump control to the end of the loop. The `break` statement will terminate the loop.

A `do` loop begins with the keyword `do` followed by the loop body and a while statement. The body will execute until a `break` statement is encountered or the expression in the `while` statement evaluates to `false`. The `while` test will be performed at the end of each iteration through the loop instead of at the beginning.

For example:

```
// a while loop
while (x<n) {
   if (x==0) break;
   // ...
}

// a do loop
do {
    // ...
} while (x<n);
```

The `foreach` statement is used with an iterator to enumerate lights or samples as described in the *Illumination* (page 85) and *Sampling* (page 90) sections of this document.

8.2 Branches

conditional_statement	:	`if (` *expression* `)` *statement* [`else` *statement*]
switch_statement	:	`switch (` *expression* `) {` { *switch_case* } `}`
switch_case	:	`case` *expression* `:` { *statement* }
	|	`default :` { *statement* }

An `if` statement is declared using the keyword `if` followed by an expression contained in parenthesis followed by a statement that will be executed if the expression evaluates to `true`.

The `else` keyword can optionally follow indicating a statement that will be executed if the expression evaluates to `false`. In nested conditional statements, an else statement matches the closest un-matched if statement.

For example:

```
if (x<n) {
    // statements that execute if x<n
} else {
    // statements that execute if x>=n
}
```

 Copyright © 2009 mental images

Switch statements allow control to jump to a selected body of statements based on an integral value. A switch statement is declared using the keyword `switch` followed by an expression enclosed by parenthesis. The expression must evaluate to an `int` valued expression. Following this is a series of `case` blocks enclosed in curly braces.

The case blocks are declared using the keyword `case` followed by a colon, then a constant integral value, and finally the statements of the case block. If the value of the expression in the `switch` statement equals the case value, that case block will be executed. Execution will fall through to the subsequent case block unless control is terminated with a `break` statement.

An optional default case block can be declared using the `default` keyword followed by a colon and then the statements to be executed if no other case statement matches the expression value.

For example:

```
switch (x)
{
    case 0:
        // statements that will be executed if x==0
        break;
    case 1:
        // statements that will be executed if x==1
        break;
    case 2:
        // statements that will be executed if x==2
        break;
    default:
        // statements that will be executed if x!= 0, 1, or 2
        break;
}
```

8.3 Jumps

break_statement	:	`break ;`
continue_statement	:	`continue ;`
return_statement	:	`return` [*expression*] `;`

The return statement terminates execution of the current function and returns control to the calling function or completes execution of the shader if the current function is the `main()` method of the shader.

A return statement is declared using the `return` keyword followed by an optional return value. Functions that are declared to have a `void` return type do not return a value otherwise the return statement must include a value that matches the return type of the function.

The `break` statement is used to terminate a loop or `switch` statement. The `continue` statement is used to terminate an iteration of a loop and proceed to the next iteration of the loop.

The `return`, `break`, and `continue` statements all cause the flow of control to jump to a new location in the code.

 Copyright © 2009 mental images

9 Functions

function_declaration	:	[`native`] *qualified_type identifier* `(` [*parameter* { `,` *parameter* }] `)` `;`
	|	*qualified_type* `operator` *operator* `(` [*parameter* { `,` *parameter* }] `)` `;`
function_definition	:	[`native`] *qualified_type identifier* `(` [*parameter* { `,` *parameter* }] `)` *compound_statement*
	|	*qualified_type* `operator` *operator* `(` [*parameter* { `,` *parameter* }] `)` *compound_statement*

MetaSL can define global functions outside of shader declarations that can be called from methods of shaders or from other functions. Global functions can access the shading state.

A function is declared and defined in the main scope of a source file. The declaration consists of the return type, followed by the function name, followed by a list of function parameters surrounded by parenthesis. The parameter list is a comma-separated list of parameter declarations, each comprised of a type name followed by the parameter name (and an optional array specifier). For example:

```
float3 average_normals(float3 norm1, float3 norm2);
```

Functions must have a declaration that appears before any reference to the function is made. The function body can be included as part of the declaration or a separate function definition can occur later in the source file, after the function declaration.

9.1 Function overloading

Function names can be overloaded such that multiple functions with the same name can be defined, provided they differ at least by the number or types of parameters. It is not sufficient for overloaded functions to only differ by return type.

If a function is called with a set of arguments which match one overloaded version of the function exactly, that version of the function is used. Otherwise, if there is precisely one match if implicit conversions are allowed on the arguments, that version of the function is used with the conversions performed on the arguments. If two or more matches are found, the call is considered ambiguous.

9.2 Parameter passing

parameter	:	[`in` | `out` | `inout`] *qualified_type identifier* [`[` [*integer_literal*] `]`]

Parameters to functions are semantically passed by value. Calling parameter declarations can be qualified with one of the following qualifiers to allow a function to modify a calling parameter and allow that modification to be visible to the caller:

- `in` – The parameter value is copied into the function being called, but not copied out.
- `out` – The parameter value is not copied into the function being called and is undefined if read by the called function. The called function can however set the parameter's value and that result will be copied back to the variable passed by the caller.

Copyright © 2009 mental images

- inout – The parameter value is both copied into the function being called and copied out to the variable passed by the caller.

The in, out, and inout qualifiers should appear before the parameter name and type in the function declaration, as in the following example:

```
bool my_function(in float x, out Color result1, out Color result2);
```

If an in, out or inout qualifier is not specified, the parameter is assumed to be an in parameter.

The copying behavior described above specifies the semantics of function parameter passing. A particular MetaSL implementation is not likely to actually copy values unless it is necessary to maintain the behavior as described.

Note: An output parameter must be initialized before it is first used. For a structured output parameter, all fields must be initialized. Using an uninitialized output parameter gives undefined behavior. For example:

```
// c.a is not written, accessing this output gives undefined behavior
void get_color(out Color c) { c.rgb = float3(1.0); }
```

9.3 Native functions

MetaSL also includes a facility to allow the implementation of specific functions using the language of the target platform. These functions are called native functions because they are implemented in the native language of the target platform.

A prototype for a native function must still be provided as part of the MetaSL shader. The prototype begins with the native keyword to indicate that it is a native function. The implementation of the function is provided in a separate file that is passed along (unmodified) to the native compiler for the target platform.

For example:

```
native bool my_native_function(out Color result);
```

A native function can also have a MetaSL implementation. The MetaSL implementation will be used on platforms where a native implementation is not provided. The function is defined and implemented the same way as a regular function except the presence of the native keyword which indicates that a native implementation should be used when found for a particular platform.

For example:

```
native bool my_native_function(out Color result)
{
    return false;
}
```

 Copyright © 2009 mental images

The downside of native functions is that a MetaSL shader that uses a native function is only compatible with platforms on which an implementation of the native function is available, but this can be mitigated by providing a MetaSL fallback implementation.

10 Shader class

shader_declaration : `shader` *identifier* [: *identifier*]
{ *shader_member_declaration* { *shader_member_declaration* } }
[*annotation*] ;

shader_member_declaration : *enum_declaration*
| *struct_declaration*
| *function_declaration*
| *function_definition*
| *variable_declaration*
| `input :`
| `output :`
| `member :`

All components of a MetaSL shader are grouped together in a shader class denoted with the `shader` keyword. A shader class describes a set of shaders with common characteristics and behaviors, while at the same time allowing members of the set to differ. An instantiation of a shader class, called a shader instance, describes a particular use of a shader class.

A shader class is declared with the following syntax:

```
shader my_shader {
    // Contents of the shader are found here
};
```

The body of the shader class defines the shader's inputs, outputs, and other members as described in the following sections. Variable declarations in the shader body that are given before any `input`, `output` or `member` qualifier, are by default member variables. Note that the names given to input, output and other declarations must be unique within the shader class.

A single MetaSL source file can contain any number of shader class declarations. Although a shader class can be instantiated any number of times by a MetaSL compliant application, only shader classes are defined in MetaSL source code and not shader instances. However, an application can pass shader instances as references to other shaders using the MetaSL `Shader` data type, see Section ShaderType.

10.1 Input parameters

A shader can have zero or more input parameters which the shader uses to determine its result value. Each instance of a shader class receives its own copy of input parameter values, which allows shader instances to produce different results while sharing common functionality as defined by the shader class.

The input parameters of shader instances may store literal values or can be attached to the result of another shader, however a shader doesn't need to be concerned with this possibility. A shader can refer to an input parameter as it would to any another variable, although input parameters can only be read and not written to within a shader.

Input parameters can be of any of the built-in types (described in the *Data types* section on page 11) excluding the iterator and options types (listed on page 12), or they can be of any of the custom structure

 Copyright © 2009 mental images

types (described in the *Structures* section on page 44).

A shader declares its input parameters in a section denoted by the `input:` label followed by a declaration of each parameter.

For example:

```
shader my_shader {
  input:
    Color c0;
    Color c1;
};
```

This example declares a shader with two color input parameters.

Like a local variable declared in a function, an input parameter may have an initializer, although it is not required. For example:

```
shader my_shader {
  input:
    float x = 1;
};
```

The initializer can contain an expression that includes references to state variables or other input parameters. Any shader parameters referred to in the initializing expression must be declared before the initialized parameter. For example:

```
shader my_shader {
  input:
    int texture_space;
    float2 texture_uv = texture_coordinate[texture_space];
};
```

For more information about state variables, see the *Shading state* section on page 71.

Texture input parameters can be initialized with a string which identifies the texture resource (typically the file name of the texture). Texture type variables can only be initialized with a string if they are input parameters. When a texture variable is declared as a local variable, it can only be initialized with another texture of the same type.

10.2 Output parameters

A shader must have at least one output parameter, but may have more than one. Output parameters store a shader's result. The purpose of a shader is to compute some function of its input parameters (if any) and store the result of that function in its output parameters. Similarly to input parameters, each shader

instance receives its own copy of the shader class's output parameters.

Output parameters can be of any of the built-in types (described in the *Data types* section on page 11) excluding the iterator and options types (listed on page 12), or they can be of any of the custom structure types (described in the *Structures* section on page 44). However they cannot contain unsized arrays.

A shader declares its output parameters in a section denoted by the `output:` label followed by a declaration of each parameter. For example:

```
shader my_shader {
  output:
    Color ambient_light;
    Color direct_light;
};
```

This example declares a shader with two color outputs. Many shaders will only have a single output parameter, which by convention is named "result".

Note: An output parameter declaration cannot contain an initializer as part of its declaration. However, an output parameter must be initialized in the body of the shader code before it is first used. For a structured output parameter, all fields must be initialized. Using an uninitialized output parameter gives undefined behavior.

10.3 Other shader members

Shaders can declare other variables which are not input or output parameters, but instead store other values read by the shader. When initializing before rendering begins, a shader can compute in its constructor values and store them in member variables. During rendering member variables are read-only. Member variables are designed to hold values that are computed once before rendering begins but do not change thereafter. This avoids redundantly computing a value each time the shader is called.

Member variables are declared in a section denoted by the `member:` keyword. For example:

```
shader my_shader {
  input:
    float amount;

  output:
    Color result;

  member:
    float data[1024];
};
```

Each shader instance receives its own copy of member variables.

 Copyright © 2009 mental images

10.4 Shader methods

In addition to input, output, and member variables, shader classes can define methods which act on those variables to produce their result. Shader methods can be declared to return values of any type and can accept any number of calling parameters of any type.

A method is declared in the body of a shader class. The declaration consists of the return type, followed by the method name, followed by a list of method parameters surrounded by parenthesis. The parameter list is a comma-separated list of parameter declarations, each comprised of a type name followed by the parameter name (and an optional array specifier). For example:

```
float3 average_normals(float3 norm1, float3 norm2);
```

The definition of a shader method can be placed inline inside the shader class body along with the method declaration or outside of the shader class. Placing method definitions outside of the shader class can help keep the class declaration concise and readable. When a shader method is defined outside of the shader class, the shader class must still contain a declaration of the method without the implementation.

A method that is defined outside of the shader class must have the method name prefixed with the name of the shader class followed by two colons. For example:

```
shader My_shader
{
  member:
    float3 average_normals(float3 norm1, float3 norm2);
}

float3 My_shader::average_normals(
    float3 norm1, float3 norm2)
{
    return normalize(norm1+norm2);
}
```

Parameter types and return types of a method defined outside of the shader class are looked up in the scope of the shader class.

All shader methods are visible to other methods of the same shader class regardless of the order in which they are declared. A shader method can call another shader method even if the called method is declared later in the file. The only restriction is that method calls cannot be recursive. Input, output, and member variables are visible to all methods, but only output parameters are modifiable.

10.4.1 Shader main method

The primary task of a shader is to compute one or more result values. This is implemented in the shader class by a method named `main`. This method is called when the renderer needs the result of the shader or when the shader is attached to an input parameter of another shader and that shader needs the parameter's value.

The return type of this method is always `void` which means it doesn't return a value. Instead the result of

Copyright © 2009 mental images

a shader should be placed in its output parameters. The `main` method never has any parameters; it takes its input and output from the shader's input and output parameters.

A simple shader example which blends two colors:

```
shader mix_colors {
  input:
    Color c0;
    Color c1;
    float mix;

  output:
    Color result;

    void main() {
        result = c0*(1.0-mix) + c1*mix;
    }
};
```

Like other shader methods, the `main` method can be declared inline in the shader body (as illustrated in the above example) or outside of the shader definition.

10.4.2 Constructors and destructors

A shader class can optionally define special methods called constructors and destructors. A constructor is a method that is called to allow the shader to perform initialization before rendering begins and a destructor is called when the shader is no longer needed.

Shader member variables are only writable inside constructors. A shader can use a constructor to perform a calculation and place the result in a member variable to avoid repeating the calculation while rendering.

Note that constructors and destructors cannot accept calling parameters.

If present, a shader's constructor will be called for each instance of a shader class before the first use of the shader instance. Within the constructor, shader parameters can be read, but not written. Member variables can be written within the constructor, which is its primary purpose. Additionally some state variables and methods can be accessed.

If present, a shader's destructor will be called for each instance of a shader class some time after the last use of the shader instance. The same rules apply with respect to accessing shader and state variables for destructors as do for constructors.

The exact time of constructor and destructor calls and the number of shader instances and therefore constructor and destructor calls created are implementation dependent. Therefore, side-effects in constructor or destructor implementations should be better avoided or carefully synchronized.

A constructor is a method sharing the same name as the shader class. A destructor also shares the name of the shader class but is prefixed with a tilde (~) character.

The following provides an example of constructor and destructor syntax:

 Copyright © 2009 mental images

```
shader My_shader
{
  member:
    My_shader()  {    // constructor
        // ...
    }

    ~My_shader() {    // destructor
        // ...
    }
};
```

10.4.3 Method overloading

Shader methods can be overloaded by defining another method with the same name as an existing method, but with a different set of calling parameters. Overloaded versions of a method must have a different number of parameters or parameters that differ in type (or both). It is not sufficient for overloaded methods to only differ by return type.

If a shader method is called with a set of arguments which match one overloaded version of the method exactly, that version of the method is used. Otherwise, if there is precisely one match if implicit conversions are allowed on the arguments, that version of the method is used with the conversions performed on the arguments. If two or more matches are found, the call is considered ambiguous.

When a shader method and a global function share the same name, they are overloaded, but in different scopes. To resolve overloaded functions when called within a shader method, first shader methods matching the name of the function call are considered. If a match is found (even if implicit type conversions are required), then the matching shader method is used. If not, global functions which match the name are considered.

For example:

```
void foo(int x) {}
void foo(Color x) {}

shader My_shader {
  input:
    // this method shadows the global function foo(int x)
    void foo(float x) {}

    void main()
    {
        // this will call the shader method foo(float x)
        foo(13);

        // this will call the global function foo(Color x)
        foo(Color(1));
    }
}
```

10.4.4 Parameter passing

parameter : [`in` | `out` | `inout`] *qualified_type identifier* [`[` [*integer_literal*] `]`]

Parameters to methods are semantically passed by value. Calling parameter declarations can be qualified with one of the following qualifiers to allow a method to modify a calling parameter and allow that modification to be visible to the caller:

- `in` – The parameter value is copied into the method being called, but not copied out.
- `out` – The parameter value is not copied into the method being called and is undefined if read by the called method. The called method can however set the parameter's value and that result will be copied back to the variable passed by the caller.
- `inout` – The parameter value is both copied into the method being called and copied out to the variable passed by the caller.

The `in`, `out`, and `inout` qualifiers should appear before the parameter name and type in the method declaration, as in the following example:

```
bool my_method(in float x, out Color result1, out Color result2);
```

If an `in`, `out` or `inout` qualifier is not specified, the parameter is assumed to be an `in` parameter.

The copying behavior described above specifies the semantics of method parameter passing. A particular MetaSL implementation is not likely to actually copy values unless it is necessary to maintain the behavior as described.

 Copyright © 2009 mental images

Note: An output parameter must be initialized before it is first used. For a structured output parameter, all fields must be initialized. Using an uninitialized output parameter gives undefined behavior. For example:

```
shader my_shader {
    // c.a is not written, accessing this output gives undefined behavior
    void get_color(out Color c) { c.rgb = float3(1.0); }
};
```

10.4.5 Shader class inheritance

MetaSL shader classes support the concept of class inheritance, which allows a shader class to be derived from another class as a means to specialize the behavior of the parent class.

A shader declares an optional parent class as part of the shader declaration by stating the name of the parent shader following the name of the child shader and separated by a colon.

For example:

```
shader my_parent_shader {
    // ...
};

shader my_shader : my_parent_shader {
    // ...
};
```

This allows variations of a shader type, called a sub-class to be implemented while sharing parts of the shader that are common to all variations. A shader definition can only inherit from a single parent shader.

A shader sub-class inherits all parameters and methods from the parent shader, including those of ancestor shader classes. The child shader cannot redefine members of the parent shader, with the exception of methods. A child shader can override (or overload) a method of the parent shader to provide an alternative implementation. A child shader can also introduce new input, output, and member variables.

When a child shader overrides a method of the parent shader, all other methods of the child shader, whether inherited or not, will use the new overridden implementation instead of the parent shader implementation.

Shader class constructors of base shaders are called before the constructors of the derived shader, which may be missing. Destructors are called in the reverse order; the destructors of base shaders are called after the destructors of the derived shader, which may be missing.

11 Preprocessor

The MetaSL language definition includes a preprocessor which supports macro substitution, conditional compilation, and the inclusion or referencing of named files.

A preprocessor directive is indicated using the # character and consumes at least one line of the source file. The # character must be the first non-white-space character on a line. Directives can be continued onto the following line by ending the current line with the \ character.

The preprocessor supports the following directives:

- `#define`
- `#undef`
- `#if`
- `#ifdef`
- `#ifndef`
- `#else`
- `#elif`
- `#endif`
- `#include`
- `#import`
- `#native`
- `#version`

The syntax and usage of these preprocessor directives are as follows:

`#define` *identifier token-string*
`#define` *identifier* `(` *identifier , ... , identifier* `)` *token-string*

Defines a symbol for macro substitution. Instances of the given identifier are replaced throughout the source file with the specified sequence of tokens. The definition is applied from the line in which the directive appears until the end of the file, or until the symbol definition is removed using the `#undef` directive.

The `#define` directive can also be used to define parameterized macros. Arguments to macros are substituted in each instance where the parameter appears in the macro definition. For example:

```
#define square(a) ((a)*(a))
```

This defines a macro named `square` which accepts a single parameter. For example `square(6)` would expand to `((6)*(6))`.

The preprocessor continues expanding macros until no further substitutions are encountered. This allows an identifier in the token string of a definition to itself be expanded by another definition.

 Copyright © 2009 mental images

`#undef` *identifier*

The `#undef` directive removes a previously defined macro and is effective from the point in the file where `#undef` occurs until the end of the file.

`#if` *expression text* { `#elif` *expression text* } [`#else` *text*] `#endif`
`#ifdef` *identifier text* { `#elif` *expression text* } [`#else` *text*] `#endif`
`#ifndef` *identifier text* { `#elif` *expression text* } [`#else` *text*] `#endif`

Conditional compilation allows entire blocks of text to be ignored by the MetaSL compiler.

The `#if` and `#elif` directives are handled by evaluating each expression in the order it appears in the file until one of them evaluates to a non-zero value. The text associated with the non-zero expression is considered for compilation while text in other blocks is ignored (including any other preprocessing directives in the text). If none of the expressions evaluate to a non-zero value, the text provided in the optional `#else` case will be selected.

The `defined()` macro is also supported, but can only be used in the conditional expression part of an `#if` or `#elif` directive. The `defined()` macro accepts an identifier as a parameter and evaluates to `true` if the identifier is defined and `false` otherwise.

The `#ifdef` directive selects text to be compiled if the given identifier has been previously defined with the `#define` directive. The `#ifndef` directive selects text if the given identifier has not been defined.

`#include` < *resource-identifier* >
`#import` < *resource-identifier* >

The `#include` directive is supported to add other MetaSL source files to the current file. This allows structure definitions and shader base classes to be shared across files.

The `#import` directive is similar to the `#include` directive except that instead of injecting the included source code during preprocessing, a reference to the imported module is made.

Functions and user defined types implemented in imported modules are available for use by the module they are imported into. Preprocessor definitions made in the parent file will not affect the implementation in the imported file and similarly, preprocessor definitions in the imported file will not affect the parent file. Only global functions and user defined types are imported.

Since the imported file is independent of the files in which it is imported into, it can be compiled more efficiently than directly inserting the code with the `#include` directive. This makes the `#import` directive useful for sharing libraries of global functions among shaders.

`#native` < *resource-identifier* >

The `#native` directive specifies the location of an implementation dependent resource containing implementations of native functions used by the shader.

Native function implementations may be text files or precompiled libraries or any other format required by the particular platform. The exact meaning of specified identifier is defined be the platform on which the native function is implemented.

Copyright © 2009 mental images

`#version` *version-identifier*

The `#version` directive specifies the MetaSL language version. Future versions of MetaSL may introduce new language features, but in all cases a MetaSL implementation must support all previous versions of MetaSL shaders (i.e., MetaSL is always backward compatible).

The version number may be used by a future version of MetaSL to recognize shader source code as a particular version of MetaSL and enable new features.

If the `#version` directive appears in a file it must only appear once and must be on the first line of code that is not filled with white-space or comments.

If the `#version` directive is not present, the source code is assumed to be MetaSL version 1.0.

MetaSL has built-in preprocessor definitions for the following symbols:

- `PI` – `3.14159265358979323846f`
- `TWO_PI` – `6.28318530717958647692f`
- `HALF_PI` – `1.57079632679489661923f`
- `FLOAT_MIN` – The smallest `float` value supported by the current platform.
- `FLOAT_MAX` – The largest `float` value supported by the current platform.
- `DOUBLE_MIN` – The smallest `double` value supported by the current platform.
- `DOUBLE_MAX` – The largest `double` value supported by the current platform.
- `HALF_MIN` – The smallest `half` value supported by the current platform.
- `HALF_MAX` – The largest `half` value supported by the current platform.
- `INT_MIN` – The smallest `int` value supported by the current platform.
- `INT_MAX` – The largest `int` value supported by the current platform.

 Copyright © 2009 mental images

12 Annotations

annotation : `{ {` *identifier* `(` [*expression*] `) ; } }`

MetaSL defines a mechanism called annotations to attach metadata to various components of a shader as well as the shader itself. Annotations can be applied to:

- Shader parameters
- Shader classes

Annotations are placed in blocks immediately after the declaration they are annotating. The list of one or more annotations is enclosed in curly braces. An individual annotation looks like a function call with no return value where the name of the function is the annotation and the parameters are values associated with the annotation. These values need to be constant expressions. It is not an error to have several annotations of the same name.

An annotation block has the following form:

```
{
    annotation_name(param1, param2, ...);
    annotation_name(param1, param2, ...);
    ...
}
```

Annotations can be used to attach any type of metadata to elements of a shader, but the most common case is metadata to describe the user interface for a shader or shader parameter.

Annotations of a shader class are not inherited by a derived shader class. Annotations of a shader parameter are inherited with the parameter.

The following are standard annotations to represent common metadata:

`soft_range(`
`T min,`
`T max)`
	Specifies a range of useful values for the parameter, however the parameter value can exceed this range.
	
	`min` – The minimum value of the range. The type `T` should match the type of the parameter.
	
	`max` – The maximum value of the range. The type `T` should match the type of the parameter.

`hard_range(`
`T min,`
`T max)`
	Specifies bounds for the parameter that cannot be exceeded.
	
	`min` – The minimum value of the range. The type `T` should match the type of the parameter.
	
	`max` – The maximum value of the range. The type `T` should match the type of the parameter.

`display_name(` `    String name)`	Specifies a name to use when the element is displayed in a user interface. `name` – The name to be displayed, which can be friendlier to readers than a typical identifier.
`external_name(` `    String name)`	Specifies an alternative name for the element to match the requirements of an external application. `name` – The external name, which is not constrained by the requirement to be a legal MetaSL identifier.
`hidden()`	Specifies that the element should not be visible in an application's user interface. For example, a hidden parameter would not show up in a parameter editing user interface.
`description(` `    String description)`	Specifies a description of the element. An application providing a user interface for the shader can use this metadata to provide help for the user.
`author(` `    String name)`	Specifies the name of the shader author.
`contributor(` `    String name)`	Specifies the name of a contributing shader author.
`copyright_notice(` `    String name)`	Specifies copyright information.

`created(`
`    int year,`
`    int month,`
`    int day,`
`    String notes)`
	Specifies the date on which the shader was created along with a string to hold creation notes.

`modified(`
`    int year,`
`    int month,`
`    int day,`
`    String notes)`
	Specifies a date on which the shader was modified along with a string to hold notes related to the modification.

 Copyright © 2009 mental images

`version_number(`
`    int version,`
`    ...)`
	Specifies a series of version numbers indicating the version of a shader. The first number in the series indicates the major version of the shader. The last number indicates the revision number. Any numbers in-between identify branches in a source control system.
	
	A larger major version number always indicates a more recent version of the shader. Given the same major version number, a larger revision number also indicates a more recent version of the shader.
	
	A branch number that appears in-between the major version and revision number locates the branch within a source control system, however it is possible that a version from an earlier branch is more current than a later branch as indicated by the revision number.
	
	For example the version number `1.1.12` indicates a more recent version than `1.2.11` because they both have the same major version (1), but the revision number (12) is larger indicating that shader is more current even though it comes from an earlier branch.

`key_words(`
`    String word,`
`    ...)`
	Specifies a list of keywords related to the shader or its function. These keywords can be used to search libraries of shaders.

12.1 User defined annotations

annotation_declaration : `annotation` *identifier* `(` [*parameter* { `,` *parameter* }] `)` `;`

A shader can declare annotations using any name (provided it is a legal identifier) and combination of parameters. User defined annotations should be declared before they are used. An annotation is declared using the `annotation` specifier followed by the name of the annotation and optional annotation parameters surrounded in parenthesis.

For example:

```
annotation my_annotation(float f, String s);
```

This declares an annotation named `my_annotation` which accepts a `float` and a `String` parameters. It could be used to annotate a shader parameter as follows:

```
input:
  Color p(0)
  {
      my_annotation(1.5, "a string");
  };
```

User defined annotations don't strictly have to be declared before they are used. MetaSL compilers don't interpret user defined annotations and so they are not required to successfully compile a shader. However use of undefined annotations will trigger a compiler warning. This can be useful to identify cases where

the use of a declared annotation was intended, but a typo or spelling error caused the annotation to be undefined.

 Copyright © 2009 mental images

13 Shading state

MetaSL defines a set of variables that provide shaders access to rendering state that typically describes some varying aspect of the current fragment being shaded or a uniform property of the render pass. These state variables represent values that are commonly used by shaders.

State variables are global variables that are visible to all shader methods and global functions.

State variables can be thought of as implicit parameters of a shader that by default are supplied values from the renderer implementation rather than directly from user specified values. However it is possible to override state variables with shader parameters of the same name provided the type of shader has write access to the state variable. A shader parameter which overrides a state variable effectively allows the state variable to be attachable in a shader graph since attachments can be made to the input parameter.

In addition to overriding state variables with parameter values, a shader can also write directly to certain state variables. Modifications to state values, whether through overriding parameters or assignment in the shader code, will not affect other nodes in the same shader graph.

Modifications to state values do have the ability to affect shaders that are invoked indirectly by ray tracing or light loops. For example, if the `normal` state variable is modified before a light loop, the light shader will see the modified normal. Modifications to the state will also affect subsequent explicit evaluation of variables of type `Shader` invoked with `Shader::call()`.

Not all state variables are modifiable. Furthermore, some read-only state variables are dependent on the values of other state variables. When a state value is modified, dependent state variables will update to reflect that new value.

Some state variables are particular to certain types of shaders. For example, light shaders have access to `light_position`, but surface shaders do not. This document contains descriptions of the state variables supported by each shader type and whether they are modifiable by a shader of that type.

In addition to state variables, MetaSL also defines a set of state methods which can be called from shader methods. Like state variables, some state methods have restrictions which allow them to only be called from particular shader types.

The names of global state variables and functions are not keywords in the MetaSL language and can be used as identifiers in user defined declarations. When a state variable or function name is used as an identifier for another purpose, the state variable or function is shadowed within the scope of the declaration.

13.1 State variables

The following is a list of all possible state variables. Note that all values are in internal space.

`float3` `origin`	The ray origin
`float3` `position`	The intersection point on the surface
`float3` `direction`	The ray direction. This vector is unit length and is dependent on `origin` and `position`.

Copyright © 2009 mental images

float ray_length	The length of the current ray. This value is dependent on origin and position
float3 normal	The surface normal for shading. This vector is unit length.
float dot_nd	The dot product of the normal and the ray direction. This variable is dependent on normal and direction.
float3 geometry_normal	The true surface normal for the current geometry. This vector is unit length.
float3 motion	Tangential motion vector
float2 raster	The raster coordinates for the fragment being rendered
float2 parametric_uv	The parametric uv coordinate for the surface
float4 texture_coordinate[]	The array of texture spaces.
float3 texture_tangent[]	The array of tangent vectors for each texture space. The tangent vector is a unit length vector in the plane defined by the surface normal, which points in the direction of the positive u axis of the corresponding texture space.
float3 texture_binormal[]	The array of binormal vectors for each texture space. The binormal vector is a unit length vector in the plane defined by the surface normal, which points in the general direction of the positive v axis of the corresponding texture space, but is orthogonal to both the surface normal and the tangent of the corresponding texture space.
float3x3 tangent_space[]	The array of tangent space matrices for each texture space. These matrices are available as a convenience and are constructed from the texture_tangent, texture_binormal, and normal as the x, y, and z axis of the coordinate system, respectively.
float3 texture_du[]	The array of surface derivatives with respect to the u direction of the specified texture space. This array contains derivatives for each texture space. It is similar to texture_tangent except that the vectors in texture_du are not normalized or necessarily orthogonal to texture_dv or normal.
float3 texture_dv[]	The array of surface derivatives with respect to the v direction of the specified texture space. This array contains derivatives for each texture space. It is similar to texture_binormal except that the vectors in texture_dv are not normalized or necessarily orthogonal to texture_du or normal.
int animation_frame	The current frame number

 Copyright © 2009 mental images

float animation_time	The time of the current sample in seconds, including the time within the shutter interval
float shutter_time	The point in time within the shutter interval in seconds. The shutter_time will have the value 0.0 at the beginning of the shutter interval. This value is dependent on animation_time and shutter_open.
float shutter_position	The normalized position within the shutter interval. This variable will have the value 0.0 at the start of the interval and 1.0 at the end, and is dependent on shutter_open, shutter_close and animation_time.
float shutter_open	The time in seconds of the beginning of the shutter interval.
float shutter_close	The time in seconds of the end of the shutter interval.
float shutter_duration	The length of the shutter interval in seconds. This variable is dependent on shutter_open and shutter_close.
bool backside	Set to true when the ray hits the geometry from behind. In this case both the normal and geometry_normal are inverted.
bool inside	Set to true when the current ray is intersecting the surface from inside the volume defined by the surface. The value of inside can be different from backside when the surface normals are oriented such that the front side of the surface faces the interior of the volume.
String ray_type	The type of the current ray, see the ray tracing section 13.2.2 below
String ray_shader	The type of the current shader, see the ray tracing section 13.2.2 below
float importance	The effect of the current shader on the final pixel color (ranges from 0-1)
float incident_ior	The index of refraction of the medium that contains the incident ray. The range of this value is >=1.
float refracted_ior	The index of refraction of the medium which will contain the refracted ray. The range of this value is >=1.
bool orthographic	Set to true if all eye rays are parallel
float focal_length	The camera's focal length
float aperture	The camera's aperture

`float` `aspect_ratio`	The pixel aspect ratio (width / height)
`float` `near_clip`	The distance to the near clipping plane
`float` `far_clip`	The distance to the far clipping plane
`float2` `camera_offset`	The camera plane offset in pixels
`float` `dof_radius`	The depth of field radius
`float` `dof_focus`	The distance from the camera position to the focal plane
`int` `image_x_resolution`	The width of the image in pixels
`int` `image_y_resolution`	The height of the image in pixels
`int` `window_left`	When rendering a cropped region, this specifies the size of the left margin in pixels
`int` `window_right`	When rendering a cropped region, this specifies the size of the right margin in pixels
`int` `window_top`	When rendering a cropped region, this specifies the size of the top margin in pixels
`int` `window_bottom`	When rendering a cropped region, this specifies the size of the bottom margin in pixels
`String` `light_type`	The type of the light object for which the light shader has been invoked. The value can be one of the following, unless it is an area light, in which case `light_area` is greater than zero: `"light_point"` – A *point light*. Light is cast in all directions from the position of the light, `light_position`. `"light_spot"` – A *spot light*. Light is cast only within a cone originating in `light_position` and pointing in the `light_direction` direction with the `light_spread` opening angle. `"light_infinite"` – An *infinite light*. Light rays are parallel and are emitted in the `light_direction` direction from a point infinitely far away. `"light_planar"` – A *planar light*. Light rays are parallel as they are for an infinite light except light is only cast on the side of the plane defined by `light_direction` and `light_position`.

 Copyright © 2009 mental images

float3 light_position	The position of the sample point on the light. It is not defined for infinite lights. For planar lights, the light position lies on the line defined by light_direction and the surface position.
float3 light_direction	The direction of the light. This vector is unit length and points away from light_position. It is not defined for point lights.
float3 light_to_surface	A unit vector that points from the light position to the surface position, except for infinite lights where it is defined as light_direction.
float light_distance	The distance between the surface point and the light point. It is not defined for infinite lights.
float light_dot_nl	The dot product of the surface normal and the negated light_to_surface unit vector.
float3 light_normal	The normal at the sample point on the surface of an area light. This vector is unit length.
float2 light_uv	The surface parameter of the sample point on an area light.
float light_area	The total area of the light emitting surface of an area light.
float light_area_delta	The area associated with the current light sample for an area light.
float4 light_texture_coordinate[]	Additional texture spaces for the surfaces of area lights.
float3 light_texture_tangent[]	The array of tangent vectors for each light texture space. The tangent vector is a unit length vector in the plane defined by the light surface normal, which points in the direction of the positive u axis of the corresponding texture space.
float3 light_texture_binormal[]	The array of binormal vectors for each light texture space. The binormal vector is a unit length vector in the plane defined by the light surface normal, which points in the general direction of the positive v axis of the corresponding texture space, but is orthogonal to both the surface normal and the tangent of the corresponding light texture space.
float light_spread	Defines a cone centered around light_direction. Light is not cast outside the cone. The value of light_spread is given in radians and measures the angle between light_direction and the side of the cone.
float light_spread_cos	The cosine of light_spread. This value is provided so that the light_spread can be directly compared to the dot product of light_to_surface and light_direction. If the result of this dot product is less than light_spread_cos then the surface point is outside the cone and is not illuminated.

float light_distance_limit	Specifies the distance from light_position beyond which no light is received.
Color volume_input	The input value to which the volume shader should apply its effect. For example a volume shader may compute its result by interpolating between the volume_input value and another color that is computed as a result of the ray traveling through the volume.

 Copyright © 2009 mental images

13.2 State functions

The following is a list of all state functions:

13.2.1 Coordinate systems

State vectors are always provided in "internal" space. Internal space is undefined and can vary across different platforms. If a shader can perform calculations independently of the coordinate system then it can operate with the state vectors directly, otherwise it will need to transform state vectors into a known space.

The state provides functions to transform vectors, points and normals between the following coordinate systems:

- `"internal"`
- `"object"`
- `"world"`
- `"camera"`
- `"light"`
- `"raster"`

```
float4x4 get_transform(
    String from,
    String to)
```

Get a transformation matrix to transform from one space to another including user-defined coordinate systems. The `from` and `to` parameters can be the name of a user-defined coordinate system or `"internal"`, `"object"`, `"world"`, `"camera"`, `"light"`, or `"raster"`.

The matrix returned from `get_transform()` assumes that vectors are considered to be column vectors and multiplied on the right hand side of the matrix.

In other words, the translation components of the matrix, `Tx`, `Ty`, and `Tz` would be located in the following positions of a matrix that contained only translation:

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
float3 transform_point(
    String from,
    String to,
    float3 pt)
```

Transform a point from one coordinate system to another. The `from` and `to` parameters can be the name of a user-defined coordinate system or `"internal"`, `"object"`, `"world"`, `"camera"`, `"light"`, or `"raster"`.

`float3 transform_vector(`
`String from,`
`String to,`
`float3 v)`
	Transform a vector from one coordinate system to another. The translation component of the coordinate systems is ignored when transforming the vector. The `from` and `to` parameters can be the name of a user-defined coordinate system or `"internal"`, `"object"`, `"world"`, `"camera"`, `"light"`, or `"raster"`.

`float3 transform_normal(`
`String from,`
`String to,`
`float3 n)`
	Transform a normal from one coordinate system to another. As with vector transformations, the translation component of the coordinate systems is ignored. Additionally the transpose of the inverse transformation matrix is used. The `from` and `to` parameters can be the name of a user-defined coordinate system or `"internal"`, `"object"`, `"world"`, `"camera"`, `"light"`, or `"raster"`.

Note that for all transform functions, if the `from` argument is `"internal"` then the point to transform should be a state vector or a value that his been previously transformed into `"internal"` space otherwise the result is undefined and may differ across implementations. For example the result of:

```
transform_point("internal", "world", float3(1,0,0))
```

is undefined since the definition of `"internal"` is implementation dependent.

 Copyright © 2009 mental images

13.2.2 Ray tracing

The ray that is responsible for the current intersection state is described by the `ray_type`, `ray_shader`, `is_ray_dispersal_group()` and `is_ray_history_group()` state variables and functions. These variables and functions use the following strings to describe attributes of the ray:

A ray has exactly one of the following types as indicated by the `ray_type` state variable:

- `"eye"` – First generation ray with its origin at the eye position
- `"transparent"` – Transparency ray into the current object
- `"refract"` – Refraction into the current object
- `"reflect"` – Reflection away from the current object
- `"shadow"` – Shadow ray
- `"occlusion"` – Ambient occlusion ray
- `"environment"` – Environment ray

A ray can be a member of at most one of the following dispersal groups as indicated by the `is_ray_dispersal_group()` state function:

- `"specular"` – Specular transparency, reflection, or refraction
- `"glossy"` – Glossy transparency, reflection, or refraction
- `"diffuse"` – Diffuse transparency, reflection, or refraction

A shader is a member of exactly one of the following groups as indicated by the `ray_shader` state variable:

- `"surface"` – Surface shader
- `"volume"` – Volume shader
- `"photon"` – Global illumination or caustic photon
- `"light"` – Light shader
- `"displace"` – Displacement shader
- `"environment"` – Environment shader
- `"lens"` – Lens shader

A ray can have zero or more of the following history flags as indicated by the `is_ray_history_group()` state function:

- `"lightmap"` – Lightmap shader call
- `"final_gather"` – Final gather ray

- "hull" – Pass through a hull surface
- "volume" – Ray origin is in empty space

Predefined Trace_options, Occlusion_options, and Irradiance_options classes hold parameters used by the trace(), occlusion(), and irradiance() functions described in the next section. A shader can declare an instance of these types once and pass it to multiple trace, occlusion, or irradiance calls. These option classes support the assignment operator (=).

When a Trace_options, Occlusion_options or Irradiance_options instance is declared, all its member values are initialized to default values. The shader can then call various 'set' methods to change the values to something other than the default.

The methods of the Trace_options class are as follows:

Trace_options

void set_near(float near)	Set the distance from the origin of the ray at which intersection testing begins.
void set_far(float far)	Set the distance from the origin of the ray at which intersection testing ends.
void set_ray_type(String ray_type)	Set the ray type for the ray. Defaults to "reflect". This parameter must be a literal string or a shader parameter. The string "continue" can be specified as a ray type in which case the current ray type will be propagated to the new ray and tracing will continue as if the current intersection did not take place.
void set_ray_dispersal_group(String raygroup)	Set the ray dispersal group. Defaults to "specular". This parameter must be a literal string or shader parameter.

void set_importance(
    Color importance)
void set_importance(
    Spectrum importance)
	Specify the relative importance of the contribution from this ray with a value that ranges from 0 to 1 for each component.

void enable_environment(bool enabled)	Enables or disables the environment shader. If the environment shader is enabled (which is the default), it will be called if the ray doesn't intersect with scene geometry or the trace depth is exhausted.

 Copyright © 2009 mental images

`void set_data( String name, T value)`	Associates arbitrary data with rays that can be read by shaders as the ray is traced. This data is associated with the ray and all child rays until it is cleared by a shader. This data is set in addition to data that has been set by ancestors of this ray. The data is read and cleared with the state functions `get_ray_data()` and `clear_ray_data()` described below. `name` – Specifies a name for the data. If a previous value with the same name exists it will be overwritten. `value` – Specifies the value to store. The type of this parameter can be any MetaSL type. It is important to note that the use of ray data can have costly effects on the performance of ray tracing and may even prevent the use of delayed ray tracing if the data size is too large. For that reason ray data should be used sparingly, preferably only to store a small number of boolean flags.

The methods of the `Occlusion_options` class are as follows:

`Occlusion_options`

`void set_near( float near)`	Set the distance from the origin of the ray at which intersection testing begins.
`void set_far( float far)`	Set the distance from the origin of the ray at which intersection testing ends.
`void set_cone( float3 axis, float angle, float angle_exp=1)`	Specify the cone over which the occlusion factor will be calculated. The `axis` parameter is specified in internal space and defaults to the state normal, the `angle` parameter defaults to 90 and the `angle_exp` parameter defaults to 1. The `angle` parameter is measured in degrees.
`void set_importance( Color importance)` `void set_importance( Spectrum importance)`	Specify the relative importance of the contribution from this ray with a value that ranges from 0 to 1 for each component.

The methods of the `Irradiance_options` class are as follows:

`Irradiance_options`

`void set_near( float near)`	Set the shortest distance from the origin of the ray at which incoming light is considered.
`void set_far( float far)`	Set the furthest distance from the origin of the ray at which incoming light is considered.

`void set_cone(`
`    float3 axis,`
`    float angle,`
`    float angle_exp=1)`
	Specify the cone over which irradiance will be calculated. The `axis` parameter is specified in internal space and defaults to the state normal, the `angle` parameter defaults to 90 and the `angle_exp` parameter defaults to 1. The `angle` parameter is measured in degrees.

`void set_importance(`
`    Color importance)`
`void set_importance(`
`    Spectrum importance)`
	Specify the relative importance of the contribution from this ray with a value that ranges from 0 to 1 for each component.

The following state functions provide the ability to trace individual rays as well as collections of rays (e.g. `occlusion()`).

`bool is_ray_dispersal_group(` `    String group)`	Returns true if the current ray is a member of the given group. This parameter must be a literal string or function parameter.
`bool is_ray_history_group(` `    String history)`	Returns true if the current ray is a member of the given history group. This parameter must be a literal string or function parameter.

`bool trace(`
`    Ray ray,`
`    Ray ray_dx,`
`    Ray ray_dy,`
`    Trace_options options,`
`    String gather_output_name,`
`    out T result,`
`    ...)`
	Trace the given ray. This function returns false if nothing was hit or the trace depth was exhausted. If a hit is encountered one or more output values are stored in the given result variables.
	
	This function can accept a variable number of pairs of `gather_output_name` and `result` parameters. The `gather_output_name` argument specifies the output on the surface shader of the surface hit by the ray to be retrieved. Typically this is a `Color` or `Spectrum` output, but may be any MetaSL data type. All surface shaders should at minimum have an output named `"result"`.
	
	The `gather_output_name` argument can also refer to a state variable, in which case the name should be prefixed with `"State::"`, for example: `"State::normal"`. This retrieves that value of the specified state variable as it appears at the intersection point of the cast ray. For clarity a shader parameter name can also be prefixed with `"Shader::"`. If no prefix is specified, an output parameter is assumed.
	
	For each specified `gather_output_name` the value of that output will be stored in the corresponding result variable. The type of the result variable must match the type of the requested output otherwise the `trace()` function will return false.

`int get_trace_depth(` `    String ray_type)`	Get the accumulated trace depth of the given ray type. The `ray_type` parameter must be a literal string and the value `"total"` can be used to request the total accumulated trace depth of all the ray types.

 Copyright © 2009 mental images

`bool get_ray_data(`
`    String name)`
`    out T value)`
	Retrieves data set by ancestors of the current ray using `Trace_options::set_data()`.
	
	`name` – Specifies the name of the data to access. If no data with the given name exists, this method will return `false`.
	
	`value` – Specifies a variable to hold the resulting value. The type of this parameter can be any MetaSL type.
	Once data is attached to a ray, the data will remain attached for all child rays.

`void clear_ray_data(`
`    String name)`
 	
`void clear_ray_data()`
	Removes data set by ancestors of the current ray using `Trace_options::set_data()`. This removal is effective for this shader and for child rays, but has no effect on the ancestors. The first version of `clear_ray_data()` removes data matching the specified name. If no data exists matching the given name, the method does nothing. The second version clears all ray data.

`float occlusion(` `    Occlusion_options options)`	Get the ambient occlusion for the given point. Returns 1.0 for full occlusion and 0.0 for no occlusion.
`Spectrum irradiance(` `    Irradiance_options options)`	Gets the irradiance for the given point, which is the total amount of incoming indirect light arriving at the point within the cone specified by the `options` parameter.

13.2.3 Light shaders

The following state functions are available to light shaders:

`bool get_shader_parameter(`
`    String parameter_name,`
`    out Type value)`
	Get a parameter value from the shader that invoked the light shader. The `parameter_name` argument must be a literal string. The type of the `value` argument can be one of the basic built-in types.

`Spectrum shadowing()`	Computes the shadow factor for the current light sample with respect to the surface point being shaded. The method used to determine the amount of shadow attenuation (for example shadow maps or ray traced shadows) can vary depending on the target platform.

13.2.4 Messages and diagnostics

The following functions allow shaders to output messages and diagnostic information:

`void error(`
`    String format, ...)`
`void warning(`
`    String format, ...)`
`void info(`
`    String format, ...)`
`void progress(`
`    String format, ...)`
	These functions all output messages within their respective categories. The format string follows the C language printf function format specifier convention and is followed by a variable number of arguments.

 Copyright © 2009 mental images

14 Illumination

One of the most important aspects of shading is determining the amount of illumination reflected in a particular direction at a point in space. This involves properties of the surface as described by the bidirectional reflectance distribution function (BRDF), objects in the scene that emit light (direct light) and the light reflected off other objects in the scene (indirect light).

MetaSL defines constructs to assist shaders in the computation of illumination. A MetaSL shader can choose the level of control it has over the process by using high level functions provided be MetaSL, which are the easiest to use and help to ensure physical correctness, or explicit light loops, which give complete control to the shader and allow artistic styles that may not require a physically correct simulation.

The advantage of the high level direct and indirect light functions as compared to explicit light loops goes beyond ease of use. Because the enumeration of lights is entirely under control of the renderer implementation in these cases, it is able to apply optimizations that might not have been otherwise possible.

14.1 BRDFs

A BRDF is a function which computes the outgoing radiance in a particular direction given the incoming irradiance from another direction.

MetaSL allows a shader to utilize a predefined BRDF in the computation of illumination. The BRDF can be one of the following BRDF types:

- Ashikhmin-Shirley
- Cook-Torrance
- Oren-Nayar
- Phong
- Ward

All BRDF types have parameters that can be attached to MetaSL based Metanodes or Phenomena. For example, a BRDF may have a `Color` input for a diffuse reflection color, which can be attached to the result of a texture lookup node or other shader graph.

14.2 Light iteration

The computation of illumination naturally involves enumerating light sources in a scene. MetaSL provides mechanisms to enumerate lights while providing shaders with control over which lights are considered.

It is often useful to break out light sources into categories and treat groups of lights differently, or to channel the results of groups of lights into different frame buffers.

The MetaSL constructs used to enumerate direct light sources allow the enumeration to be controlled by parameters specified using the built-in `Light_iterator_options` data type, which provides methods to specify option parameters.

Light_iterator_options

`void set_categories( String category, ...)`	This method allows a list of strings to be provided to specify light categories which should explicitly be included or excluded in the enumeration. If this option is provided, lights that are not members of the specified set of categories are ignored. A category name can be prefixed with a '-' character to indicate that lights that are members of the specified category should be excluded, otherwise they are included.
`void set_shadows( bool on_off)`	This method allows the computation of shadows in the light shader to be enabled or disabled. Shadows are enabled by default.
`void set_cone( float3 axis, float angle)`	This method allows a cone to be specified that is used to define which light samples are considered for shading. Light samples coming from a direction that is outside the cone region are ignored. If the cone is not specified, the default behavior is to use the surface normal for the `axis` parameter and the value `90` for the `angle` parameter. This results in a hemisphere shape centered around the surface normal effectively rejecting all light samples coming from behind the surface. The `angle` parameter is specified in degrees. As another example, to consider light samples from all directions including behind the surface use: `set_cone(normal, 180)`.

For example:

```
Light_iterator_options options;
options.set_categories("key", "fill", "-rim");
options.set_shadows(false);
```

This example creates a light iterator options such that only lights with the "key" or "fill" attribute will be considered, but skip any lights that also had the "rim" attribute. Also in this example shadow calculations are disabled.

14.3 Direct and indirect light

MetaSL provides two functions to assist a shader in the computation of illumination. These functions use the BRDF specified by the material to determine direct and indirect illumination.

Multiple versions of the direct and indirect light functions are provided to allow the result to be retrieved as components or a single value. A `String` type parameter called `type` specifies which component or components will be retrieved. The `type` parameter can have one of the following values:

- "d" – Compute the diffuse component
- "s" – Compute the specular component
- "g" – Compute the glossy component
- "ds" – Diffuse and specular
- "dg" – Diffuse and glossy
- "sg" – Specular and glossy

 Copyright © 2009 mental images

Diffuse refers to light that is generally reflected equally in all directions. Specular refers to light that is only reflected in a single direction (sometimes referred to as perfect or ideal specular reflection). Glossy refers to light that is mostly reflected in a particular direction.

The `direct_light()` function computes the contribution of light sources on the shading point such that only light which does not interact with another medium or object is considered.

`void direct_light(` `    out Spectrum result)`	The total direct light is computed and stored in the `result` parameter.

`void direct_light(`
`    String type,`
`    out Spectrum result)`
	One or two components of direct light are computed and stored in the `result` parameter as specified by the `type` string.

`void direct_light(`
`    String type,`
`    out Spectrum result1,`
`    out Spectrum result2)`
	Two components of direct light are computed and stored in the two result parameters in the order specified by the `type` parameter.

`void direct_light(`
`    out Spectrum diffuse_result,`
`    out Spectrum specular_result,`
`    out Spectrum glossy_result)`
	All components of direct light are computed and stored in the result parameters.

The `indirect_light()` function computes the contribution of light which has been reflected off other objects in the scene or passes through another medium.

`void indirect_light(` `    out Spectrum result)`	The total indirect light is computed and stored in the `result` parameter.

`void indirect_light(`
`    String type,`
`    out Spectrum result)`
	One or two components of indirect light are computed and stored in the `result` parameter as specified by the `type` string.

`void indirect_light(`
`    String type,`
`    out Spectrum result1,`
`    out Spectrum result2)`
	Two components of indirect light are computed and stored in the two result parameters in the order specified by the `type` parameter.

`void indirect_light(`
`    out Spectrum diffuse_result,`
`    out Spectrum specular_result,`
`    out Spectrum glossy_result)`
	All components of indirect light are computed and stored in the result parameters.

14.4 Light Loops

MetaSL also allows finer grain control over the evaluation of illumination through the use of explicit light loops. A light loop iterates over light sources allowing the shader to accumulate the result of each light.

The `foreach` statement is used with an instance of `Light_iterator` to enumerate lights. Each iteration through the loop represents a single illumination sample. Dimensionless light sources such as point lights

Copyright © 2009 mental images

only require a single sample while area lights may be sampled multiple times. Sampling is handled by the light loop and so the shader doesn't need to treat area lights differently from other light sources.

A Light_iterator is declared as a local variables and used in the foreach statement. Inside the body of the foreach statement, the light iterator variable provides the results of the light shader for the given sample.

For example:

```
Light_iterator light;
foreach (light) {
    // statements that refer to 'light'
}
```

The Light_iterator class supports an optional constructor which accepts an instance of Light_iterator_options to control the enumeration of lights. The Light_iterator class supports the assignment operator (=).

A Light_iterator instance has the following read-only members:

Light_iterator

float3 point	The sample point on the light. This value is equal to the state variable light_position in the corresponding light shader.
float3 direction	A unit vector pointing from the surface point to the sample point on the light, except for infinite lights where it is defined as the negated light direction. This value is equal to the negated value of the state variable light_to_surface in the corresponding light shader.
float distance	The distance from the surface point to the sample point on the light. It is not defined for infinite lights. This value is equal to the state variable light_distance in the corresponding light shader.
float dot_nl	The dot product of the surface normal and direction. This value is equal to the state variable light_dot_nl in the corresponding light shader.
Spectrum contribution	The light contribution for this sample including attenuation from shadows or volumes
Spectrum raw_contribution	light contribution for this sample without shadow or volume attenuation
Spectrum shadow	The attenuation from volumes and shadows
int count	The number of samples taken so far
bool get_light_parameter(String parameter_name, out T value)	Find the light shader input or output parameter with the specified name and return its result in the value out argument. Returns false if the parameter is not found. The type of the value argument can be one of the basic built-in types.

 Copyright © 2009 mental images

Inside the light loop a shader can use the results provided by light iterator members to compute the contribution from the given light sample. It can maintain complete control by expressing the BRDF calculation directly in the body of the light loop, which allows for non-physically correct effects.

Alternatively, there are versions of the `direct_light()` function which use the material's BRDF to compute the illumination result. Instead of computing the result for all lights, these versions compute the result for a single light sample as identified by the light iterator parameter.

Function	Description
`void direct_light( Light_iterator light, out Spectrum result)`	The total direct light is computed and stored in the `result` parameter.
`void direct_light( Light_iterator light, String type, out Spectrum result)`	One or two components of direct light are computed and stored in the `result` parameter as specified by the `type` string.
`void direct_light( Light_iterator light, String type, out Spectrum result1, out Spectrum result2)`	Two components of direct light are computed and stored in the two result parameters in the order specified by the `type` parameter.
`void direct_light( Light_iterator light, out Spectrum diffuse_result, out Spectrum specular_result, out Spectrum glossy_result)`	All components of direct light are computed and stored in the result parameters.

The versions of `direct_light()` that accept a `Light_iterator` argument can only be called from within the body of a `foreach` loop and must pass the iterator used by that `foreach` statement. Similarly, members of a `Light_iterator` instance can only be accessed within the body of the `foreach` loop using the iterator. Furthermore, the same `Light_iterator` instance cannot be used simultaneously by multiple nested `foreach` statements.

15 Sampling

MetaSL provides a mechanism for generating deterministic quasi-Monte Carlo based sample points which can be used to sample functions for the numerical evaluation of integrals. Note that these sample points are designed for the efficient computation averages of light, i.e. integrals. The points therefore are highly correlated and hence may not be suited for some aspects of modeling in which case pseudo random numbers can be more beneficial (e.g. for placing objects in a random manner).

Sampling is expressed using the `foreach` construct in a similar manner as it is used with light loops. A `Sample_iterator` type facilitates the iteration process. A variable of type `Sample_iterator` is declared and used in the `foreach` statement to start a sample loop. The syntax looks like the following:

```
Sample_iterator si;
foreach (si) {
    // statements that refer to 'si'
}
```

The iterator enumerates all quasi-Monte Carlo sample points. Each iteration provides a stream of numbers in the range [0, 1), which correspond to subsequent components (i.e., dimensions) of the current quasi-Monte Carlo sampling point. This stream is subsequently read component by component (i.e., dimension after dimension) using the `sample()` method of the `Sample_iterator` class. Alternative versions of the `sample()` method return multiple successive components as vectors, i.e. calling `sample3()` is equivalent to calling `sample()` three times.

- `sample()` – Returns a one dimensional `double` type sample point
- `sample2()` – Returns a two dimensional `double2` type sample point
- `sample3()` – Returns a three dimensional `double3` type sample point
- `sample4()` – Returns a four dimensional `double4` type sample point

Note that the `sample()`, `sample2()`, `sample3()`, and `sample4()` methods can only be accessed within the foreach loop using the iterator. Furthermore, the same `Sample_iterator` instance cannot be used simultaneously by multiple nested `foreach` statements.

In the following example the sample iterator produces two dimensional sample points:

```
Sample_iterator si;
foreach (si) {
    double2 p = si.sample2();

    // statements that use the sample point p
}
```

 Copyright © 2009 mental images

The Sample_iterator constructor optionally accepts a parameter to specify the number of sample iterations. If this parameter is omitted or has a value of 0, the sample loop will continue until a break statement or a return statement is encountered, typically when a certain quality is reached. A value greater than 0 indicates a finite number of samples are desired. The following MetaSL code illustrates finite and adaptive sampling:

```
// finite sampling - 16 samples
Sample_iterator fsi(16);
foreach (fsi) {
    double2 p = fsi.sample2();

    // statements that use the sample point p
}

// adaptive sampling - unknown number of samples
Sample_iterator asi();
foreach (asi) {
    double2 p = asi.sample2();

    // statements that use the sample point p

    if (quality_reached(...))
        break;
}

// take 8 samples into the hemisphere
Sample_iterator it(8);
foreach (it) {
    double2 samp = it.sample2();
    double3 dir = sample_hemisphere(samp); // user defined function to sample the hemisphere

    // trace a ray...
}
```

Sampling loops cannot be nested.

If the number of samples is known then it is preferable to use the finite sampling method. This makes it possible for some renderer implementations to generate improved sampling patterns.

16 Shader type reference

The following sections describe the different shader types in detail, including a description of the applicability of state variables and functions to each shader type.

State variables may be read-only or writable, depending on the shader type. In addition some state variables are dependent on the values of other state variables. In the following sections, the accessibility for each state variable as it applies to each shader type is listed using the following symbols:

- `R` – Read only
- `RW` – Read and write
- `RD` – Read only and dependent

16.1 Surface

A surface shader is responsible for computing the color of a point on a surface as seen from a particular direction. The result or results of surface shaders typically involve computing the resulting illumination of scene light sources.

16.1.1 State variables

Surface shaders have access to the following state variables:

State Variable	Access	State Variable	Access
origin	R	backside	R
position	RW	inside	R
direction	RD	ray_type	R
ray_length	RD	ray_shader	R
normal	RW	window_left	R
dot_nd	RD	importance	R
geometry_normal	R	incident_ior	R
motion	R	refracted_ior	R
raster	R	orthographic	R
parametric_uv	R	focal_length	R
texture_coordinate	R	aperture	R
texture_tangent	R	aspect_ratio	R
texture_binormal	R	near_clip	R
tangent_space	R	far_clip	R
texture_du	R	camera_offset	R
texture_dv	R	dof_radius	R
animation_frame	R	dof_focus	R
animation_time	R	image_x_resolution	R
shutter_time	R	image_y_resolution	R
shutter_position	R	window_left	R
shutter_open	R	window_right	R
shutter_close	R	window_top	R
shutter_duration	R	window_bottom	R

 Copyright © 2009 mental images

16.1.2 State functions

Surface shaders have access to the following state functions:

get_transform()	transform_point()	transform_vector()	transform_normal()
is_ray_dispersal_group()	is_ray_history_group()	trace()	get_trace_depth()
get_ray_data()	clear_ray_data()	occlusion()	irradiance()
error()	warning()	info()	progress()
direct_light()	indirect_light()		

16.1.3 Outputs

A surface shader can have one or more `Color` or `Spectrum` type outputs to hold its result. A surface shader may provide a single `Color` or `Spectrum` output to specify the total result and/or multiple outputs to define the result as a series of components.

A surface shader should always have at least one primary output named `"result"`. This convention allows the interoperability of different shaders since it can be assumed that every surface shader generates a `"result"` output.

The benefit of componentized outputs is that each output can be routed into a separate frame buffer. Post processing can then be applied to components individually and they can be composited to produce the final rendered image.

The component outputs of a surface shader can be used by a MetaSL compiler implementation to automatically derive a shader that produces a single component result or a subset of components. This supports rendering passes where certain components are stored in frame buffers and other individual components are re-rendered.

The output names given to surface shader results indicate named frame buffers into which the result is placed. A user can re-route surface shader outputs without modifying the shader source code by placing the surface shader in a Phenomenon and attaching surface shader outputs to user named Phenomenon outputs.

A surface shader can optionally have a `Color`, `Spectrum`, or `float` output named `"transparency"`. The transparency output indicates a factor describing the amount of light that is able to pass through the surface, with a value of zero indicating that the surface is completely opaque. If a surface shader does not have a transparency output, it is considered to be opaque. A surface shader should not premultiply its result color with `1-transparency` since that will be handled by the renderer.

Different rendering algorithms tend to handle transparency differently. The MetaSL shader abstraction allows a surface shader to be used seamlessly with the different techniques for handling transparency utilized by different platforms.

A `Spectrum` type output is assumed to have an alpha component equal to one. A `Color` type output allows the surface shader to specify the alpha component. If the alpha component is not specified, it is computed by the renderer using the transparency factor and the coverage of the surface over the current pixel.

Copyright © 2009 mental images

16.2 Light

A light shader is responsible for computing the amount of incoming light arriving at a position in space from a light source (or point on a light source in the case of area lights).

16.2.1 State variables

Light shaders have access to the following state variables:

State Variable	Access	State Variable	Access
origin	R	orthographic	R
position	R	focal_length	R
direction	R	aperture	R
ray_length	R	aspect_ratio	R
normal	R	near_clip	R
dot_nd	R	far_clip	R
geometry_normal	R	camera_offset	R
motion	R	dof_radius	R
raster	R	dof_focus	R
parametric_uv	R	image_x_resolution	R
texture_coordinate	R	image_y_resolution	R
texture_tangent	R	window_left	R
texture_binormal	R	window_right	R
tangent_space	R	window_top	R
texture_du	R	window_bottom	R
texture_dv	R	light_position	R
animation_frame	R	light_direction	R
animation_time	R	light_to_surface	R
shutter_time	R	light_distance	R
shutter_position	R	light_dot_nl	R
shutter_open	R	light_normal	R
shutter_close	R	light_uv	R
shutter_duration	R	light_texture_coordinate	R
backside	R	light_texture_tangent	R
inside	R	light_texture_binormal	R
ray_type	R	light_type	R
ray_shader	R	light_spread	R
window_left	R	light_spread_cos	R
importance	R	light_distance_limit	R
incident_ior	R	light_area	R
refracted_ior	R	light_area_delta	R

16.2.2 State functions

Light shaders have access to the following state functions:

 Copyright © 2009 mental images

`get_transform()`	`transform_point()`	`transform_vector()`	`transform_normal()`
`is_ray_dispersal_group()`	`is_ray_history_group()`	`trace()`	`get_trace_depth()`
`get_ray_data()`	`clear_ray_data()`	`get_shader_parameter()`	`shadowing()`
`error()`	`warning()`	`info()`	`progress()`
`direct_light()`	`indirect_light()`		

16.2.3 Outputs

A light shader must have at least one `Spectrum` or `Color` type output to hold the result of the light shader. If a light shader returns a `Color`, the alpha component is ignored. This is the value that appears in the `raw_contribution` member of a `Light_iterator`, after the renderer has applied a weighting based on the number of samples. The light shader computes the incident radiance arriving at the shade point for the particular light sample.

A light shader can also specify the attenuation resulting from shadowing by declaring a `Spectrum`, `Color` or `float` output parameter named `light_shadow`. The shadow factor is accessible in the light loop through the light iterator's `shadow` member variable.

A light shader is not required to include a shadow output parameter and if the shadow output is not specified the shadow factor will be computed automatically. In addition the light iterator's `contribution` member variable will be automatically computed as the product of `raw_contribution` and `shadow`.

16.3 Environment

An environment shader is responsible for computing the color of a point infinitely far away as observed from a particular direction.

16.3.1 State variables

Environment shaders have access to the following state variables:

State Variable	Access	State Variable	Access
origin	R	focal_length	R
direction	R	aperture	R
raster	R	aspect_ratio	R
animation_frame	R	near_clip	R
animation_time	R	far_clip	R
shutter_time	R	image_x_resolution	R
shutter_position	R	image_y_resolution	R
shutter_open	R	camera_offset	R
shutter_close	R	dof_radius	R
shutter_duration	R	dof_focus	R
ray_type	R	window_left	R
ray_shader	R	window_right	R
importance	R	window_top	R
orthographic	R	window_bottom	R

16.3.2 State functions

Environment shaders have access to the following state functions:

get_transform()	transform_point()	transform_vector()	transform_normal()
is_ray_dispersal_group()	is_ray_history_group()	get_trace_depth()	get_ray_data()
error()	warning()	info()	progress()

16.3.3 Outputs

An environment shader has a single `Color` or `Spectrum` output in which its result is placed.

 Copyright © 2009 mental images

16.4 Volume

A volume shader is responsible for computing the effect of a medium on light as it travels through a region of space. Examples include atmospheric effects such as fog as well as fluid constrained to a region of space such as a drinking glass.

The state variable `volume_input` specifies the input color which is affected by the volume the current ray is traveling through. Typically a volume shader will modify this color to produce its result.

16.4.1 State variables

Volume shaders have access to the following state variables:

State Variable	Access	State Variable	Access
origin	R	focal_length	R
position	RW	aperture	R
direction	RD	aspect_ratio	R
ray_length	RD	near_clip	R
normal	RW	far_clip	R
raster	R	image_x_resolution	R
animation_frame	R	image_y_resolution	R
animation_time	R	camera_offset	R
shutter_time	R	dof_radius	R
shutter_position	R	dof_focus	R
shutter_open	R	window_left	R
shutter_close	R	window_right	R
shutter_duration	R	window_top	R
ray_type	R	window_bottom	R
ray_shader	R	volume_input	R
importance	R	inside	R
orthographic	R		

16.4.2 State functions

Volume shaders have access to the following state functions:

get_transform()	transform_point()	transform_vector()	transform_normal()
is_ray_dispersal_group()	is_ray_history_group()	trace()	get_trace_depth()
get_ray_data()	clear_ray_data()	error()	warning()
info()	progress()		

16.4.3 Outputs

A volume shader has a single `Color` or `Spectrum` output in which its result is placed.

16.5 Displacement

A displacement shader is responsible for modifying the position (and optionally the normal) of surface points to allow the generation of fine grain detail that is difficult to model traditionally.

16.5.1 State variables

Displacement shaders have access to the following state variables:

State Variable	Access	State Variable	Access
position	R	animation_frame	R
normal	R	animation_time	R
geometry_normal	R	shutter_time	R
parametric_uv	R	shutter_position	R
texture_coordinate	R	shutter_open	R
texture_tangent	R	shutter_close	R
texture_binormal	R	shutter_duration	R
tangent_space	R		
texture_du	R		
texture_dv	R		

16.5.2 State functions

Displacement shaders have access to the following state functions:

```
get_transform()      transform_point()    transform_vector()   transform_normal()
error()              warning()            info()               progress()
```

The transform state functions are restricted to the coordinate systems `"internal"`, `"object"`, and `"world"`. Note that a transformation to the `"world"` coordinate system might have negative impact on renderers that support object instancing because multiple instances with a shared object might have different displacements in world coordinates and can no longer be shared.

16.5.3 Outputs

A displacement shader specifies the amount of displacement through a specially named output called `displacement_amount`. The type of this output can be one of the following:

- `float` – Specifies the amount in which to move the point in the direction of the surface normal.
- `Color` – The average intensity of the red, green, and blue components specifies the amount in which to move the point in the direction of the surface normal.
- `float3` – Specifies the absolute amount of displacement in internal space (regardless of the surface normal).

In addition, a displacement shader can optionally have an output of type `float3` named `displacement_normal` which specifies a new value for the surface normal. If the normal is not specified it will be computed automatically.

 Copyright © 2009 mental images

Any additional outputs are interpreted as additional attributes generated by the displacement shader and will be attached to the point being displaced. These attributes are accessible by other shaders later in the rendering pipeline, such as the surface shader. When the name of an attribute attached to a point matches the name of an input parameter of the surface shader, the point attribute will be used to drive the value of that input parameter.

16.6 Lens

A lens shader is responsible for determining the color of a pixel in the rendered image, typically by casting eye rays.

A lens shader is invoked for each pixel (or sub-sample) of the image with the output of the lens shader representing the resulting color for that pixel. Typically a lens shader will cast one or more eye rays to compute the resulting color.

16.6.1 State variables

Lens shaders have access to the following state variables:

State Variable	Access	State Variable	Access
origin	R	focal_length	R
direction	R	aperture	R
raster	R	aspect_ratio	R
animation_frame	R	near_clip	R
animation_time	R	far_clip	R
shutter_time	R	image_x_resolution	R
shutter_position	R	image_y_resolution	R
shutter_open	R	camera_offset	R
shutter_close	R	dof_radius	R
shutter_duration	R	dof_focus	R
ray_type	R	window_left	R
ray_shader	R	window_right	R
importance	R	window_top	R
orthographic	R	window_bottom	R

16.6.2 State functions

Lens shaders have access to the following state functions:

```
get_transform()           transform_point()       transform_vector()       transform_normal()
is_ray_dispersal_group()  is_ray_history_group()  trace()                  get_trace_depth()
error()                   warning()               info()                   progress()
```

16.6.3 Outputs

Like a MetaSL surface shader, a lens shader can have multiple outputs to route its results into different frame buffers. The name of each output indicates the named frame buffer into which the output should be placed.

 Copyright © 2009 mental images

17 Standard Library Functions

MetaSL defines a standard library of built-in functions. Many of these functions support and return parameters of many of the fundamental MetaSL types (e.g., float, int, etc.), or some combination of these parameters. A list of overloaded versions of each standard library function is included with each description.

Every standard library function that accepts a `float` or vector of `float` elements has overloaded versions that accept `half` or `double`. Only the `float` versions are listed below. The corresponding `half` and `double` versions just replace consistently all occurrences of `float` in the function signature by `half` or `double`, respectively.

Every standard library function that has a `float4` vector in its signature also works with the synonymous `Color` type. Since `float4` and `Color` are not considered different types for the purpose of function overloading, there are no separate functions listed below for the `Color` type.

Built-in functions are globally available to all shaders. The names of standard library functions are not keywords in the MetaSL language and can be used as identifiers in user defined declarations. When a built-in function name is used as an identifier for another purpose, the function is shadowed within the scope of the declaration.

17.1 Math functions

Function	Description

`float abs(float x)`
`int abs(int x)`
`float2 abs(float2 x)`
`float3 abs(float3 x)`
`float4 abs(float4 x)`
`int2 abs(int2 x)`
`int3 abs(int3 x)`
`int4 abs(int4 x)`
`Spectrum abs(Spectrum x)`
	Returns the absolute value of each vector component

`float  acos(float x)`
`float2 acos(float2 x)`
`float3 acos(float3 x)`
`float4 acos(float4 x)`
`Spectrum acos(Spectrum x)`
	Returns the arc cosine of each vector component

`bool all(float x)`
`bool all(int x)`
`bool all(bool x)`
`bool all(float2 x)`
`bool all(float3 x)`
`bool all(float4 x)`
`bool all(int2 x)`
`bool all(int3 x)`
`bool all(int4 x)`
`bool all(bool2 x)`
`bool all(bool3 x)`
`bool all(bool4 x)`
`bool all(Spectrum x)`
	Returns true if all components are true or not equal to zero, false otherwise.

Copyright © 2009 mental images

`bool any(float x)`
`bool any(int x)`
`bool any(bool x)`
`bool any(float2 x)`
`bool any(float3 x)`
`bool any(float4 x)`
`bool any(int2 x)`
`bool any(int3 x)`
`bool any(int4 x)`
`bool any(bool2 x)`
`bool any(bool3 x)`
`bool any(bool4 x)`
`bool any(Spectrum x)`
	Returns true if any component is true or not equal to zero, false otherwise.

`float  asin(float x)`
`float2 asin(float2 x)`
`float3 asin(float3 x)`
`float4 asin(float4 x)`
`Spectrum asin(Spectrum x)`
	Returns the arc sine of each vector component

`float  atan(float x)`
`float2 atan(float2 x)`
`float3 atan(float3 x)`
`float4 atan(float4 x)`
`Spectrum atan(Spectrum)`
	Returns the arc tangent of each vector component

`float  atan2(`
`    float y,`
`    float x)`
`float2 atan2(`
`    float2 y,`
`    float2 x)`
`float3 atan2(`
`    float3 y,`
`    float3 x)`
`float4 atan2(`
`    float4 y,`
`    float4 x)`
`Spectrum atan2(`
`    Spectrum y,`
`    Spectrum x)`
	Returns the arc tangent of `y/x` for each vector component. The signs of `y` and `x` are used to determine the quadrant of the result.

`float  ceil(float x)`
`float2 ceil (float2 x)`
`float3 ceil (float3 x)`
`float4 ceil (float4 x)`
`Spectrum ceil (Spectrum x)`
	Rounds each component of the given vector to the nearest integer that is greater than the component's value and returns the component results in a vector

 Copyright © 2009 mental images

Function	Description

`float clamp(`
`    float x,`
`    float min,`
`    float max)`
`int clamp(`
`    int x,`
`    int min,`
`    int max)`
`float2 clamp(`
`    float2 x,`
`    float2/float min,`
`    float2/float max)`
`float3 clamp(`
`    float3 x,`
`    float3/float min,`
`    float3/float max)`
`float4 clamp(`
`    float4 x,`
`    float4/float min,`
`    float4/float max)`
`int2 clamp(`
`    int2 x,`
`    int2/int min,`
`    int2/int max)`
`int3 clamp(`
`    int3 x,`
`    int3/int min,`
`    int3/int max)`
`int4 clamp(`
`    int4 x,`
`    int4/int min,`
`    int4/int max)`
`Spectrum clamp(`
`    Spectrum x,`
`    Spectrum/float min,`
`    Spectrum/float max)`
	Compares the `x` parameter to `min` and `max` in a component-wise fashion. If `x` is less than `min` then `min` is selected. If `x` is greater than `max` then `max` is selected. Otherwise `x` is selected. Note that the `min` and `max` parameters can be single values, regardless of whether the `x` parameter is a vector or a scalar.

`float  cos(float x)`
`float2 cos(float2 x)`
`float3 cos(float3 x)`
`float4 cos(float4 x)`
`Spectrum cos(Spectrum x)`
	Returns the cosine of each vector component. The angles specified by `x` are in radians.

`float  degrees(float x)`
`float2 degrees(float2 x)`
`float3 degrees(float3 x)`
`float4 degrees(float4 x)`
`Spectrum degrees(Spectrum x)`
	Converts each component of the given vector from radians to degrees

Copyright © 2009 mental images

`float exp(float x)`
`float2 exp(float2 x)`
`float3 exp(float3 x)`
`float4 exp(float4 x)`
`Spectrum exp(Spectrum x)`
	Applies the exponential function to each component of `x` by raising the constant `e` to the power `x`.

`float exp2(float x)`
`float2 exp2(float2 x)`
`float3 exp2(float3 x)`
`float4 exp2(float4 x)`
`Spectrum exp2(Spectrum x)`
	Applies the exponential function to each component of `x` by raising the value two to the power `x`.

`float floor(float x)`
`float2 floor(float2 x)`
`float3 floor(float3 x)`
`float4 floor(float4 x)`
`Spectrum floor(Spectrum x)`
	Rounds each component of the given vector to the nearest integer that is less than the component's value and returns the component results in a vector

`float fmod(`
`float a,`
`float b)`
`float2 fmod(`
`float2 a,`
`float2/float b)`
`float3 fmod(`
`float3 a,`
`float3/float b)`
`float4 fmod(`
`float4 a,`
`float4/float b)`
`Spectrum fmod(`
`Spectrum a,`
`Spectrum/float b)`
	Returns `a` modulo `b`, in other words, the remainder of `a/b`. The component-wise result has the same sign as `a`. Note that the `b` parameter can be single valued, regardless of whether the `a` parameter is a vector or single valued.

`float frac(float x)`
`float2 frac(float2 x)`
`float3 frac(float3 x)`
`float4 frac(float4 x)`
`Spectrum frac(Spectrum x)`
	Returns the positive fractional part of each vector component

`bool is_nan(float x)`
`bool2 is_nan(float2 x)`
`bool3 is_nan(float3 x)`
`bool4 is_nan(float4 x)`
	Returns `true` if the given parameter no longer represents a valid number. This can occur as the result of an invalid operation such as taking the square root of a negative number.

`bool is_finite(float x)`
`bool2 is_finite(float2 x)`
`bool3 is_finite(float3 x)`
`bool4 is_finite(float4 x)`
	Returns `true` if the given parameter represents a value that is not infinite (i.e., `-infinity < x < infinity`).

 Copyright © 2009 mental images

`float lerp(`
`float a,`
`float b,`
`float x)`
`float2 lerp(`
`float2 a,`
`float2 b,`
`float2 x)`
`float3 lerp(`
`float3 a,`
`float3 b,`
`float3 x)`
`float4 lerp(`
`float4 a,`
`float4 b,`
`float4 x)`
`Spectrum lerp(`
`Spectrum a,`
`Spectrum b,`
`Spectrum x)`
	Linearly interpolates between `a` and `b` based on the value of `x`. The result is: `a*(1-x) + b*x`. The `x` parameter can be either a scalar or a vector, providing the vector is the same type as `a` and `b`.

`float log(float x)`
`float2 log(float2 x)`
`float3 log(float3 x)`
`float4 log(float4 x)`
`Spectrum log(Spectrum x)`
	Computes the natural logarithm of each component of `x`

`float log2(float x)`
`float2 log2(float2 x)`
`float3 log2(float3 x)`
`float4 log2(float4 x)`
`Spectrum log2(Spectrum x)`
	Computes the base 2 logarithm of each component of `x`

`float log10(float x)`
`float2 log10(float2 x)`
`float3 log10(float3 x)`
`float4 log10(float4 x)`
`Spectrum log10(Spectrum x)`
	Computes the base 10 logarithm of each component of `x`

`float max(float a, float b)`
`int max(int a,int b)`
`float2 max(float2 a, float2 b)`
`float3 max(float3 a, float3 b)`
`float4 max(float4 a, float4 b)`
`int2 max(int2 a, int2 b)`
`int3 max(int3 a, int3 b)`
`int4 max(int4 a, int4 b)`
`Spectrum max(Spectrum a, Spectrum b)`
	Compares each component of `a` and `b` and selects the component that is greater than the other.

Copyright © 2009 mental images

```
float  min(float a, float b)
int    min(int a, int b)
float2 min(float2 a, float2 b)
float3 min(float3 a, float3 b)
float4 min(float4 a, float4 b)
int2   min(int2 a, int2 b)
int3   min(int3 a, int3 b)
int4   min(int4 a, int4 b)
Spectrum min(Spectrum a, Spectrum b)
```

Compares each component of a and b and selects the component that is less than the other.

```
float  modf(
    float x,
    out float i)
float2 modf(
    float2 x,
    out float2 i)
float3 modf(
    float3 x,
    out float3 i)
float4 modf(
    float4 x,
    out float4 i)
Spectrum modf(
    Spectrum x,
    out Spectrum i)
```

Splits x into integral and fractional parts. The fractional part is returned and the integral part is stored in i. Both the fractional and integer parts have the same sign as x.

 Copyright © 2009 mental images

Function	Description

`float pow(`
`    float a,`
`    float b)`
`int pow(`
`    int a,`
`    int b)`
`float2 pow(`
`    float2 a,`
`    float2/float b)`
`float3 pow(`
`    float3 a,`
`    float3/float b)`
`float4 pow(`
`    float4 a,`
`    float4/float b)`
`int2 pow(`
`    int2 a,`
`    int2/int b)`
`int3 pow(`
`    int3 a,`
`    int3/int b)`
`int4 pow(`
`    int4 a,`
`    int4/int b)`
`Spectrum pow(`
`    Spectrum a,`
`    Spectrum/float b)`
	Raises each component of `a` to the power `b`. Note that the `b` parameter can be single valued, regardless of whether the `a` parameter is a vector or single valued.

`float  radians(float x)`
`float2 radians(float2 x)`
`float3 radians(float3 x)`
`float4 radians(float4 x)`
`Spectrum radians(Spectrum x)`
	Converts each component of the given vector from degrees to radians

`float  round(float x)`
`float2 round(float2 x)`
`float3 round(float3 x)`
`float4 round(float4 x)`
`Spectrum round(Spectrum x)`
	Rounds each component of `x` to the nearest integer value

`float  rsqrt(float x)`
`float2 rsqrt(float2 x)`
`float3 rsqrt(float3 x)`
`float4 rsqrt(float4 x)`
`Spectrum rsqrt(Spectrum x)`
	Returns the reciprocal of the square root of each component of `x`

`float  saturate(float x)`
`float2 saturate(float2 x)`
`float3 saturate(float3 x)`
`float4 saturate(float4 x)`
`Spectrum saturate(Spectrum x)`
	Clamps each component of `x` so that `0.0 <= x <= 1.0`

`float sign(float x)`
`int sign(int x)`
`float2 sign(float2 x)`
`float3 sign(float3 x)`
`float4 sign(float4 x)`
`int2 sign(int2 x)`
`int3 sign(int3 x)`
`int4 sign(int4 x)`
`Spectrum sign(Spectrum x)`
	Returns 1 if `x` is greater than `0`, -1 if `x` is less than `0`, and `0` otherwise, on a component-wise basis.

`float  sin(float x)`
`float2 sin(float2 x)`
`float3 sin(float3 x)`
`float4 sin(float4 x)`
`Spectrum sin(Spectrum x)`
	Returns the sine of each component of `x`. The angles specified by `x` are in radians.

`void sincos(`
`    float x,`
`    out float s,`
`    out float c)`
`void sincos(`
`    float2 x,`
`    out float2 s,`
`    out float2 c)`
`void sincos(`
`    float3 x,`
`    out float3 s,`
`    out float3 c)`
`void sincos(`
`    float4 x,`
`    out float4 s,`
`    out float4 c)`
`void sincos(`
`    Spectrum x,`
`    out Spectrum s,`
`    out Spectrum c)`
	Computes the sine and cosine of x simultaneously, which can be more efficient than calculating them individually when both are required. The angles specified by `x` are in radians.

 Copyright © 2009 mental images

Function	Description

`float smoothstep(float a, float b, float x)`
`float2 smoothstep(float2 a, float2 b, float2/float x)`
`float3 smoothstep(float3 a, float3 b, float3/float x)`
`float4 smoothstep(float4 a, float4 b, float4/float x)`
`Spectrum smoothstep(Spectrum a, Spectrum b, Spectrum/float x)`
	Returns `0` if `x` is less than `a` and 1 if `x` is greater than `b` in a component-wise fashion. A smooth curve is applied in-between so that the return value varies continuously from `0` to 1 as `x` varies from `a` to `b`. Note that the `x` parameter can be single valued, regardless of whether the `a` and `b` parameters are vectors or single valued.

`float sqrt(float x)`
`float2 sqrt(float2 x)`
`float3 sqrt(float3 x)`
`float4 sqrt(float4 x)`
`Spectrum sqrt(Spectrum x)`
	Returns the component-wise square root of `x`

`float step(float a, float b)`
`float2 step(float2 a, float2 b)`
`float3 step(float3 a, float3 b)`
`float4 step(float4 a, float4 b)`
`Spectrum step(Spectrum a, Spectrum b)`
	Returns `0` if `b` is less than `a` and 1 otherwise, on a component-wise basis

`float tan(float x)`
`float2 tan(float2 x)`
`float3 tan(float3 x)`
`float4 tan(float4 x)`
`Spectrum tan(Spectrum x)`
	Returns the component-wise tangent of `x`. The angles specified by `x` are in radians.

`float2x2 transpose(float2x2 x)`
`float2x3 transpose(float3x2 x)`
`float3x2 transpose(float2x3 x)`
`float3x3 transpose(float3x3 x)`
`float4x2 transpose(float2x4 x)`
`float2x4 transpose(float4x2 x)`
`float3x4 transpose(float4x3 x)`
`float4x3 transpose(float3x4 x)`
`float4x4 transpose(float4x4 x)`
	Computes the transpose of the matrix `x`. Note that overloaded versions to support matrices with `double` and `half` elements are also provided.

17.2 Geometric functions

`float3 cross(`
`    float3 a,`
`    float3 b)`
	Returns the cross product of `a` and `b`, which must be of type `float3`

`float distance(`
`    float a,`
`    float b)`
`float distance(`
`    float2 a,`
`    float2 b)`
`float distance(`
`    float3 a,`
`    float3 b)`
`float distance(`
`    float4 a,`
`    float4 b)`
	Returns the distance between `a` and `b`

`float dot(`
`    float a,`
`    float b)`
`float dot(`
`    float2 a,`
`    float2 b)`
`float dot(`
`    float3 a,`
`    float3 b)`
`float dot(`
`    float4 a,`
`    float4 b)`
`int dot(`
`    int2 a,`
`    int2 b)`
`int dot(`
`    int3 a,`
`    int3 b)`
`int dot(`
`    int4 a,`
`    int4 b)`
	Returns the dot product of `a` and `b`

 Copyright © 2009 mental images

`float2 faceforward(`
`    float2 n,`
`    float2 i,`
`    float2 ng)`
`float3 faceforward(`
`    float3 n,`
`    float3 i,`
`    float3 ng)`
`float4 faceforward(`
`    float4 n,`
`    float4 i,`
`    float4 ng)`
	If the dot product of `i` and `ng` is less than zero this function returns `n` otherwise it returns `-n`

`float length(float x)`
`float length(float2 x)`
`float length(float3 x)`
`float length(float4 x)`
	returns the length of `x`

`float2 normalize(float2 x)`
`float3 normalize(float3 x)`
`float4 normalize(float4 x)`
	Scales the vector `x` by the reciprocal of its length to give it a length of 1. If the length of `x` is zero the result of `normalize()` is undefined.

`float3 reflect(`
`    float3 i,`
`    float3 n)`
	Reflects the vector `i` about the normal vector `n`. It is assumed that `i` is pointing toward the surface represented by the normal `n`.

`float3 refract(`
`    float3 i,`
`    float3 n,`
`    float eta)`
	Generates the refraction vector given the normal vector `n`, the incoming ray direction `i`, and the ratio `eta` of the index of refraction of the medium entered and the index of refraction of the medium left. If the angle between `i` and `n` is too large for the given `eta`, this function returns a zero vector.

17.3 Texture functions

As a convenience, functions are also provided to lookup a `Color` directly from a texture without declaring a sampler object. These functions are equivalent to declaring a sampler object with default parameters and calling its lookup method.

There are overloaded versions of each function that allow the derivatives of the texture coordinates to be specified.

`Color tex1D(`
`    texture1D tex,`
`    float coord)`
`Color tex1D(`
`    texture1D tex,`
`    float coord,`
`    float coord_dx`
`    float coord_dy`
	Sample a one dimensional texture and return a color value.

`Color tex2D(`
`    texture2D tex,`
`    float2 coord)`
`Color tex2D(`
`    texture2D tex,`
`    float2 coord,`
`    float2 coord_dx`
`    float2 coord_dy`
	Sample a two dimensional texture and return a color value.

`Color tex3D(`
`    texture3D tex,`
`    float3 coord)`
`Color tex3D(`
`    texture3D tex,`
`    float3 coord,`
`    float3 coord_dx`
`    float3 coord_dy`
	Sample a two dimensional texture and return a color value.

`Color texCUBE(`
`    textureCUBE tex,`
`    float3 coord)`
`Color texCUBE(`
`    textureCUBE tex,`
`    float3 coord,`
`    float3 coord_dx`
`    float3 coord_dy`
	Sample a cube texture and return a color value.

17.4 Derivatives

`float  ddx(float a)`
`float2 ddx(float2 a)`
`float3 ddx(float3 a)`
`float4 ddx(float4 a)`
`Spectrum ddx(Spectrum a)`
 	
`float  ddy(float a)`
`float2 ddy(float2 a)`
`float3 ddy(float3 a)`
`float4 ddy(float4 a)`
`Spectrum ddy(Spectrum a)`
	These functions return an approximation of the partial derivative of `a` with respect to the x and y coordinates of a rectangular surface element defined by the renderer. Typically x and y correspond to the x and y directions of raster space, but can be defined relative to another coordinate system, which would likely be the case for secondary rays. The expression `a` is restricted to a linear combination of state variables, which must be mentioned literally in `a`.

 Copyright © 2009 mental images

18 Appendix A – The Syntax of MetaSL

This section describes the syntactic structure of MetaSL.

18.1 A Grammar

We give a grammar for MetaSL using Wirth's extensions of Backus Normal Form. The left-hand side of a production is separated from the right hand side by a colon. Alternatives are separated by a vertical bar. Optional items are enclosed in square brackets. Curly braces indicate that the enclosed item may be repeated zero or more times. Non-terminal and meta-symbols are given in *italic* font. Terminal symbols except identifiers, typenames, and literals are given in `teletype` font.

```
compilation_unit         :  { global_declaration }

global_declaration       :  enum_declaration
                         |  struct_declaration
                         |  type_declaration
                         |  function_declaration
                         |  function_definition
                         |  annotation_declaration
                         |  constructor_definition
                         |  destructor_definition
                         |  shader_declaration

enum_declaration         :  enum identifier { enum_item_declaration { , enum_item_declaration } } ;

enum_item_declaration    :  identifier [ = expression ]

struct_declaration       :  struct identifier { field_declaration { field_declaration } } ;

field_declaration        :  qualified_type field { , field } ;

field                    :  identifier [ [ [ integer_literal ] ] ]

type_declaration         :  typedef qualified_type identifier ;

function_declaration     :  [ native ] qualified_type identifier ( [ parameter { , parameter } ] ) ;
                         |  qualified_type operator operator ( [ parameter { , parameter } ] ) ;

constructor_declaration  :  identifier ( ) ;

destructor_declaration   :  ~ identifier ( ) ;

function_definition      :  [ native ] qualified_type identifier ( [ parameter { , parameter } ] )
                                compound_statement
                         |  qualified_type operator operator ( [ parameter { , parameter } ] )
                                compound_statement

constructor_definition   :  identifier ( ) compound_statement

destructor_definition    :  ~ identifier ( ) compound_statement
```

parameter	:	[`in` | `out` | `inout`] *qualified_type identifier* [`[` [*integer_literal*] `]`]
shader_declaration	:	`shader` *identifier* [`:` *identifier*] `{` *shader_member_declaration* { *shader_member_declaration* } `}` [*annotation*] `;`
shader_member_declaration	:	*enum_declaration*
	|	*struct_declaration*
	|	*function_declaration*
	|	*constructor_declaration*
	|	*destructor_declaration*
	|	*function_definition*
	|	*constructor_definition*
	|	*destructor_definition*
	|	*variable_declaration*
	|	`input :`
	|	`output :`
	|	`member :`
variable_declaration	:	*qualified_type variable* [*annotation*] { `,` *variable* [*annotation*] } `;`
variable	:	*identifier* [`[` [*integer_literal*] `]`] [`=` *expression*]
	|	*identifier* [`[` [*integer_literal*] `]`] [`= {` *expression* `}`]
	|	*identifier* `(` *expression* `)`
local_declaration	:	*variable_declaration*
statement	:	*empty_statement*
	|	*expression_statement*
	|	*conditional_statement*
	|	*switch_statement*
	|	*while_statement*
	|	*do_while_statement*
	|	*for_statement*
	|	*foreach_statement*
	|	*break_statement*
	|	*continue_statement*
	|	*return_statement*
	|	*compound_statement*
empty_statement	:	`;`
expression_statement	:	*expression* `;`
conditional_statement	:	`if (` *expression* `)` *statement* [`else` *statement*]
switch_statement	:	`switch (` *expression* `) {` { *switch_case* } `}`
switch_case	:	`case` *expression* `:` { *statement* }
	|	`default :` { *statement* }

 Copyright © 2009 mental images

while_statement	:	`while (` *expression* `)` *statement*
do_while_statement	:	`do` *statement* `while (` *expression* `)`
for_statement	:	`for (` [*expression*] `;` [*expression*] `;` [*expression*] `)` *statement*
	|	`for (` *variable_declaration* [*expression*] `;` [*expression*] `)` *statement*
foreach_statement	:	`foreach (` *expression* `)` *statement*
break_statement	:	`break ;`
continue_statement	:	`continue ;`
return_statement	:	`return` [*expression*] `;`
compound_statement	:	`{` { *local_declaration* | *statement* } `}`
qualified_name	:	[[[`::`] *identifier*] `::`] *identifier*
qualified_type	:	[[[`::`] *identifier*] `::`] *typename*

Copyright © 2009 mental images

```
expression             :  boolean_literal
                       |  integer_literal
                       |  float_literal
                       |  string
                       |  qualified_name
                       |  expression . expression
                       |  expression [ [ expression ] ]
                       |  qualified_type ( [ expression ] )
                       |  qualified_name ( [ expression ] )
                       |  expression ++
                       |  expression --
                       |  ++ expression
                       |  -- expression
                       |  + expression
                       |  - expression
                       |  ! expression
                       |  expression * expression
                       |  expression / expression
                       |  expression % expression
                       |  expression + expression
                       |  expression - expression
                       |  expression && expression
                       |  expression || expression
                       |  expression <= expression
                       |  expression < expression
                       |  expression >= expression
                       |  expression > expression
                       |  expression == expression
                       |  expression != expression
                       |  expression = expression
                       |  expression += expression
                       |  expression -= expression
                       |  expression *= expression
                       |  expression /= expression
                       |  expression %= expression
                       |  expression ? expression : expression
                       |  expression , expression
                       |  ( expression )

string                 :  string_literal {string_literal }

operator               :  ! | * | / | % | + | - | && | || | <= | < | >= | > | == | !=
                       |  = | += | -= | *= | /= | %=

annotation             :  { { identifier ( [ expression ] ) ; } }

annotation_declaration :  annotation identifier ( [ parameter { , parameter } ] ) ;
```

 Copyright © 2009 mental images

19 Bibliography

[1] “mental mill Functional Overview”, mental images, 2008.

[2] “mental ray Phenomena”, mental images, 1997.

[3] “System and method for generating and using systems of cooperating and encapsulated shaders and shader DAGs for use in a computer graphics system”, European patent 98930966.1-2201 and US patent 6,496,190 B1.

[4] “mental mill XML”, mental images, 2008.

20 Changes to this document

20.1 Changes since version 1.0.13

- Reordered sections to reflect a conventional bottom-up order: Moved the shader class section behind the functions section and moved the lexical structure definition from the appendix to the beginning of the document after the introduction.
- Added new reserved words for potential future use: `abstract`, `after`, `attachment`, `auto`, `before`, `bsdf`, `catch`, `char`, `const_cast`, `decltype`, `delete`, `dynamic_cast`, `event`, `explicit`, `export`, `extern`, `final`, `finally`, `friend`, `graph`, `import`, `inline`, `interface`, `long`, `mutable`, `namespace`, `new`, `phaser`, `phenomenon`, `reinterpret_cast`, `scenedata`, `sealed`, `short`, `signed`, `sizeof`, `state`, `static_cast`, `super`, `throw`, `try`, `typeid`, `typename`, `uniform`, `unsigned`, `using`, `varying`.
- Added a restriction to displacement shaders that they can use the coordinate system transformations only for the coordinate systems `"internal"`, `"object"`, and `"world"`. Clarified also that the `"world"` coordinate system might have negative impact on renderers with instancing support.
- Added constructors and destructors the the grammar in the appendix.
- Clarified in the shader class section that iterator and options types cannot be used as input or output parameters, which is stated in the data types section.
- Clarified the scope of the `clear_ray_data` state functions.
- Clarified that ray data set in `Trace_options::set_date` will be combined with ray data that ancestors of the current ray have been set.
- Clarified that it is not an error to have several annotations of the same name.
- Renumbered former section 3.6 *Conversion* to subsection 3.5.4 in section 3.5 *Matrices – float, double and half*.

20.2 Changes since version 1.0.11

- Removed underspecified geometric standard library function overloads for the `Spectrum` datatype: `distance`, `dot`, `faceforward`, `length`, and `normalize`.
- Added overload resolution definition for method and function calls.
- Clarified that the expression `a` in the standard library functions `ddx` and `ddy` is restricted to a linear combination of state variables, which must be mentioned literally in `a`.
- Clarified that a `float3` output of a displacement shader specifies the absolute amount of displacement in internal space (regardless of the surface normal).
- Clarified the behavior of annotations together with inheritance.
- Clarified that annotation parameters have to be constant expressions
- Clarified that tabulators are whitespaces.
- Clarified the associativity of the operators.
- Removed leftover mentionings of `^^` in the grammar, an operator that was dropped in document version 3.4.

 Copyright © 2009 mental images

- Added operators `%` and `%=` to operator precedence table and grammar.
- Corrected wrong spelling of `Light_iteration_options` to `Light_iterator_options` in two places.
- Clarified that the identifier argument to the `#include`, `#import`, and `#native` preprocessor statements identifies a resource and is not restricted to the *identifier* definition in the grammar.
- Clarified that the identifier argument to the `#version` preprocessor statement identifies a version and is not restricted to the *identifier* definition in the grammar.

20.3 Changes since version 3.10

- Changed versioning of this document. The version `3.10` became `1.0.10`, where `1.0` corresponds to the MetaSL version described, and `10` is a number incrementing for each edition of this document.
- Removed material phenomena from this edition.
- Removed shader techniques from this edition.
- Removed static members and static constructors from this edition.
- Disallow nested sampling loops.
- Changed the `search` method signatures of the `Particle_map` type to return `int` instead of `void`. These methods change in that they report only as many points as fit into the output arrays provided and return the actual number of points written into these arrays.
- Changed `Light_iterator` members to be read-only.
- Added remark that the exact timing and number of constructor and destructor calls for a shader are implementation dependent. Their implementation should therefore be better side-effect free or carefully synchronized.
- Added that the `Spectrum` type is also permissible for the `Color` output of volume and environment shaders as well as for the transparency output of surface shaders.
- Clarified the meaning of assignment compatible in the conversion section of the `Spectrum` data type.
- Clarified that parameter types and return types of a method defined outside of the shader class are looked up in the scope of the shader class.
- Clarified that an unsized local array variable can be initialized by another array even if that is unsized.
- Clarified that an unsized array as input parameter that has an array initializer does not become fixed size. It can have an array of different size assigned to it later.
- Clarified that a foreach loop with a sampler can also be terminated with a return statement.
- Revised the set of valid constructors for vector types and `Color`, effectively restricting the constructors which may result in a loss of precision to those from a vector of equal size.
- Clarified that shader variable declarations are by default member variables if no `input` or `output` qualifier has been given before in the shader body.
- Clarified in the automatic conversions section for the `String` type that a `String` can be automatically converted to a `Particle_map`.
- Clarified that `Color` as a synonym for `float4` is not considered a different type than `float4` for the purpose of shader method overloading or function overloading.

- Since Color as a synonym for float4, removed redundant function signatures for standard library functions that listed overloads for float4 and Color.
- Clarified that when a child shader overrides a method of the parent shader, all other methods of the child shader, whether inherited or not, will use the new overridden implementation instead of the parent shader implementation.

20.4 Changes since version 3.9

- Renamed texture samplers to conform better to naming conventions established for textures. The new names are for example Texture2D_sampler_color or TextureCUBE_sampler_float instead of the old names Texture_sampler_2d_color or Texture_sampler_CUBE_float. The rule is that the name particles 2D or CUBE are exceptions that are just glued to the texture part of a typename without underscore.
- Renamed texture standard library functions to tex1D, tex2D, and tex3D instead of tex1d, tex2d, and tex3d following the naming rule for these name particles.
- Resolved conflict; texture sampler do not support operator == and !=.
- Clarified that arrays have to be at least of size 1.
- Clarified the description of light shader state variables and light iterator members, especially light direction and dot_nl and their dependency on the light type.
- Clarified that shader output parameters and function output parameters must be initialized before they are used. Added an example that structured types, such as Color need to have all fields initialized.
- Specified the calling order of base shader constructors and destructors.
- Clarified that the equality and inequality comparison operators, == and !=, for matrices return a scalar bool.
- Clarified that the value range of 0 to 1 for the Trace_options::set_importance parameter is componentwise. Likewise for Occlusion_options::set_importance and Irradiance_options::set_importance.
- Clarified the meaning of the parameter eta of the standard library function refract.
- Added missing reserved words to table in Appendix A: annotation, float2, float3, float4, texture1D, texture2D, texture3D, textureCUBE.
- Clarified the vector constructor description that a single scalar can be used to initialize a vector.
- Clarified the meaning of the state variable ray_shader.
- Clarified that the wavelength sample range for the Spectrum type is an example and that also a conventional RGB model would be permissible.
- Clarified that the units of state variables window_left, window_right, and window_top is in pixels.
- Clarified the order of shader and technique names in the case where a technique name must be qualified.
- Clarified the use of scoped shader and technique names for externally defined shader methods in techniques.
- Clarified missing types for texture members, which are int width, int height, and int depth.
- Clarified that a function definition can also have the native keyword.

 Copyright © 2009 mental images

20.5 Changes since version 3.8

- Added a reference in the introduction and bibliography to the mental mill XML documentation, which describes the XMSL Phenomenon description format and its XML Schema.
- Removed references to the obsolete `reflection_ray and refraction_ray` functions.
- Removed references to the obsolete `trace_immediate` function.
- Clarified that shader constructors cannot accept calling parameters.
- Clarified that iterator and options classes cannot be declared as calling parameters.
- Clarified that light and sample iterators can only be accessed within the `foreach` loop currently using the iterator.
- Clarified that an iterator cannot be simultaneously used by multiple nested `foreach` statements.
- Clarified that the items of an array's initializer list must be expressions which can be converted to the type of the array's elements.
- Clarified how enum values are assigned when some, but not all of the enum element values are not explicitly stated.
- Clarified that it is legal to assign the same integral values to elements of the same enum type.
- Clarified that enum elements define symbols which cannot conflict with previous definitions.

20.6 Changes since version 3.7

- Relaxed the constraint that input parameter initializers must be constant. Now they can contain references to state variables or other shader input parameters.
- Fixed a minor typo in the constructor section.

20.7 Changes since version 3.6

- Removed the `trace()` function and renamed `trace_immediate()` to `trace()`.
- Removed the built-in BRDF types "Measured", "Blend", and "Switch".
- Removed the built-in functions `reflection_ray()` and `refraction_ray()`.
- Fixed a typo where the `shadow` component of the `Light_iterator` class was incorrectly capitalized.
- Added new standard library functions `is_nan()` and `is_finite()`.
- Noted that if a shader declares techniques, but does not designate one of them as the default, and does not have a match for a particular technique selection, then the shader will be considered unsuitable to run in that context.
- Clarified the descriptions in the Sampling section and changed the results of the sample methods to `double` from `float`.
- Reduced the set of writable state variables to `position` and `normal` for surface and volume shaders.

Copyright © 2009 mental images

- Corrected the definition of `light_dot_nl` to state that it is the dot product of the negative light direction and the surface normal. The same correction was also applied to the `Light_iterator` member `dot_nl`.
- Corrected the description of light shader outputs to note that a light shader computes the incident radiance, not irradiance.
- Corrected the statement that variables or shader parameters must be initialized before their first use. Input parameters do not have to be initialized, but output parameters and local variables do. Additionally the iterator and options classes don't require initialization before first use.
- Clarified that matrix operator section to note that operators other than the multiplication operator are applied in a component-wise fashion and that scalar operands can be mixed with matrices in expressions.
- Added the module operator (%) to the list of MetaSL operators to replace supporting the integer version of `fmod`. This operator is supported only by the `int, int2, int3,` and `int4` types.
- Removed the `Color` version of the `dot()` function since an implicit conversion exists to convert a `Color` to a `float4`.
- Corrected the `int` versions of the `dot()` function to return an `int` not a `float`.
- Removed the `int` versions of the `fmod()` function since implicit conversions exist to convert to `float` types and the modulo operator (%) is now supported for `int` types.
- Removed the `int` versions of the `exp()` and `exp2()` functions since implicit conversions exist to convert to `float` types.
- Clarified that overloaded assignment and relational operators are provided to allow the mixing of scalar and vector operands in an expression (and not just arithmetic operators).
- Clarified that a `Light_iterator` has an optional constructor that accepts a `Light_iterator_options` argument.
- Clarified that the options and iterator classes only support the assignment operator. Also clarified that they provide copy constructors.
- Clarified that the normalize() function is undefined when the given vector has zero length.

20.8 Changes since version 3.5

- Clarified that `get_light_parameter()` can access light shader input and output parameters.
- Noted in the data type section that `"-"` can be both the subtract operator and unary negate.
- Noted that the `float`, `half`, and `double` representations are approximations.
- Noted that the `Color` type supports the `?` operator.
- Added a reference from the `String` data type description to the MetaSL grammar section to describe the character set used for strings.
- Noted that variables can be declared without a constructor call to initialize them as long as they are eventually initialized before they are read.

Copyright © 2009 mental images

- Unified the rule for automatically applying conversions to scalar types to include vectors and matrices. The paragraph describing the automatic promotion of matrix types has been removed since it is covered by these conversion rules. Added tables to the scalar, vector, and matrix data type section describing what automatic type conversions are allowed that are considered to not represent a loss of precision.
- Added an explanation to the description of the state function `get_transform()` which describes the format of the returned matrix.
- Made some minor adjustments and clarification to the intro section of the document.
- Added page number references to several places where there was a reference to a section only by name.
- Updated the list of reserved words in Appendix A to include the fundamental types `bool`, `int`, `half`, `float`, `double`, including vector and matrix versions.

20.9 Changes since version 3.4

- Switched many of the basic types to type names more familiar because of their use in other languages. `Scalar` becomes `float`, `Int` becomes `int`, `Bool` becomes `bool`. Vectors are named `[typename][dimension]` (e.g., `float3` instead of `Vector3`), and matrices are named `float[n]x[m]` (e.g., `float4x4`).
- Re-introduced the `Spectrum` type as a purely implementation independent representation of color without an alpha component. The MetaSL APIs (such as `Light_iterator::contribution` will use `Spectrum`. `Spectrum` is assignment compatible with `Color`.
- Added a note to the standard function library section explaining that there are `double` and `half` versions of all standard library functions that accepts floats or float vectors. It would have been to verbose to list every single overloaded function prototype. I did add versions for `Spectrum`.
- Changed `Occlusion_options` and `Irradiance_options` `set_cone()` methods to take the `axis` parameter in internal space instead of tangent space. The default value if not specified by the shader is now the state normal instead of `<0,0,1>`.
- Fixed the shader type illustration to note that light rays can be affected by a volume shader.
- Added short introduction of the MetaSL language that introduces EBNF notation (via a reference to Appendix A).
- Added a clarification in the section describing coordinate systems that transforming from `"internal"` to another space is undefined unless the given point is known to already be in `"internal"` space (e.g., a state vector).
- Removed the restriction that built-in operators cannot be overridden.
- Clarified that structures and arrays that contain textures (or other types that can be initialized with a string) can also be initialized with a string when declared as input parameters.
- Noted that a `do` statement must be accompanied with a `while` statement.
- Changed the ray history flag `"final gather"` to `"final_gather"` for consistency since we avoid use of the space character elsewhere.

Copyright © 2009 mental images

- Merged in the addition of `Trace_options::set_ray_data()` and state functions `get_ray_data()` and `clear_ray_data()` from the MetaSL changes document. They were overlooked when merging in changes to versions 3.3 of this document.
- Replaced the reference to the C language in the preprocessor section with explicit definitions of all preprocessor directives. This was the last reference to another language in the spec, so the spec should now stand on its own.
- Added additional descriptions to the function section to clarify how overloaded functions and methods are resolved with respect to class versus global scope.
- Added a description to the Scalar data type section explaining the implications of automatic `Int` to `Scalar` conversions as they applied to `in` and `out` function parameters.
- Clarified that `irradiance()` returns the total incoming indirect light (not all light).
- Added a sentence to the light shader type description to clarify that what a light shader computes as its result is irradiance.
- Dropped the `^^` logical exclusive or operator. This will be supported but deprecated in the current MetaSL compiler.
- Added a modification to allow native functions to provide a MetaSL implementation. The native keyword indicates that a native implementation should be used when found on the target platform, otherwise the MetaSL implementation provides a fallback.
- Added support for `typedef` to allow user defined type names.
- Noted that state variables will be available in global functions.
- Added new state variable `inside`.
- Added notes to the descriptions of data types, state variables, and standard library functions explaining that these system defined identifiers are not keywords and can be overridden by user defined types (i.e., they live in a submerged namespace).
- Noted in the constructor sections for scalars and vectors that constructors for these types can accept values of different precision even if the result is a loss of precision. A constructor is semantically equivalent to a cast so for example, it should be possible to construct an `int` from a `float`.
- Replaced the `ray_shader` value `"regular"` with separate `"surface"` and `"volume"` flags and added new flags: `"environment"`, and `"lens"`.
- Removed the `"probe"` `is_ray_history_group()` flag since it would never be set to true as seen by a shader.
- Added the ability to pre-declare user defined annotations so that use of undefined annotations (which could be the result of a typo) could be caught and generate warnings. This includes an addition to the grammar.
- Added new system defined annotations: `author`, `contributor`, `copyright_notice`, `created`, `modified`, `version_number` and `key_words`.
- Noted in the surface shader description that surface shaders should at least have one primary output named `"result"` so they can interoperate with other shaders. A references to this has also been added to the descriptions of `trace()` and `trace_immediate()`.
- Replaced the built-in preprocessor definitions `SCALAR_MIN` and `SCALAR_MAX` with `FLOAT_MIN`, `FLOAT_MAX`, `DOUBLE_MIN`, `DOUBLE_MAX`, `HALF_MIN`, and `HALF_MAX`.
- Various grammatical and wording improvements and other minor adjustments.

 Copyright © 2009 mental images

20.10 Changes since version 3.3

- A general reorganization has been applied to the spec to improve clarity and readability. This also involves many minor changes and restructuring that isn't practical to list here.
- Defined the `Shader` type and explicit shader calls.
- Introduced static member variables.
- Replaced events with constructors and destructors.
- Merged in definition of displacement shaders.
- Added `Light_profile` type and functionality.
- Renamed `Map3d` to `Particle_map` and added a description of functionality.
- Added description of BRDF nodes and related `direct_light()` and `indirect_light()` functions.
- Added state functions to compute reflection and refraction (glossy and ideal) including the calculation of ray differentials.
- Added texture properties width, length, and height as well as conversion to `Bool` to test the validity of a texture.
- Added `"default"` option for a filter type for textures to indicate the use of the preferred filtering type of the platform.
- Noted that state variables will be visible in all shader methods.
- Removed explicit state transformation matrices since they are redundant. Shaders will use transform_*() methods.
- Made texture state variables unsized arrays instead of length 256.
- Removed `Spectrum` type. `Color` will fill this role for spectral rendering.
- Change the importance state variable in `Trace_options`, `Occlusion_options`, and `Irradiance_options` to be a color instead of a scalar.
- Added `light_area` state variable.
- Added `light_area_delta` state variable.
- Removed the `trace_environment()` method since it duplicates `trace_immediate()` with a ray type of `"environment"`.
- Added `enable_environment()` to `Trace_options`.
- Added the possibility to pass `"continue"` to `Trace_options` to allow a shader to cause ray tracing to continue as if the last intersection didn't occur.
- Removed the discussion of the rule file for techniques from the spec. The selection of technique will be implementation dependent.
- Clarified that `ddx()` and `ddy()` might not always be relative to screen space.
- Moved the discussion of MetaSL levels to an appendix.
- Fixed the type of `light_dot_nl` which was state as a `Vector3` when it should have been a `Scalar`.
- Fixed typo so that the `Int` versions of `fmod` now correctly return an `Int`.
- Added `SCALAR_MIN`, `SCALAR_MAX`, `INT_MIN`, and `INT_MAX` to the list of preprocessor definitions.

20.11 Changes since version 3.1

- Changed the name of the functions `tex1D`, `tex2D`, `tex3D`, and `texCUBE` to `tex1d`, `tex2d`, `tex3d`, and `texcube` to maintain consistent capitalization. The old versions containing capitalized characters are supported for backward compatibility.

20.12 Changes since version 3.0

- Removed the section describing programmable BRDF shaders. Programmable BRDFs are being replaced with a collection of predefined BRDFs, which are implemented in the core of MetaSL compliant renderers, including a very general BRDF model which will cover most cases where a user specified BRDF is required.

  A future extension to MetaSL will provide generic programmability of BRDF shaders including control over sampling of the BRDF.

20.13 Changes since version 2.9

- Removed the section describing the `global` modifier for parameter declarations. The keyword `global` is still a reserved word for possible use in a future version of MetaSL.

20.14 Changes since version 2.8

- For clarity, changed ``pt'' to ``n'' in the signature of the `transform_normal()` state function.
- Added a statement to the Matrix section indicating that matrices can be constructed from other matrices.
- Added an overloaded version of the `sign()` function that accepts an `Int` parameter.
- Fixed the `Color` version of the `asin()` function which was incorrectly labeled `acos()`.
- Added overloaded versions of several functions to allow single valued versions of some parameters to be mixed with vector parameters. For example, the `x` parameter of the `smoothstep()` function can be a `Scalar` when the `a` and `b` parameters can be vectors or colors. The functions that were modified are: `clamp(), smoothstep(), distance(), dot(), pow(), fmod(), and length()`. In some cases (such as `distance()`), a `Scalar` version was added for consistency.
- Added a comment in the Vector section to note that Boolean logic and comparison operators can mix single and vector values.
- Added `Light_iterator, Sample_iterator, Texture_sampler, Trace_options, Occlusion_options`, and `Irradiance_options` to the Data types section. These aren't new types (although `Irradiance_options` is new), but weren't previously listed in the Data types section.
- In the Type conversion section, removed the statement that `Int` vectors can be implicitly converted to `Scalar` vectors of the same size.
- Added a note that `Scalar` vectors can be constructed from `Int` vectors of the same size.

 Copyright © 2009 mental images

- Added a comment in the shader class section noting that identifiers declared at shader scope must be unique with respect to each other.
- Changed the operators and operator overloading section to note that the '?' operator is not overloadable.
- Added the new irradiance() state function.
- Added the new Irradiance_options built-in type.
- Fixed the descriptions of any() and all() to state they evaluate to true when the components are not equal to zero instead of just greater than zero.

Copyright © 2009 mental images