MetaSL
Shading Language Tutorial and Reference

Andy Kopra and Lutz Kettner

Excerpt:

Chapter 3: Basic surface shaders

Preliminary draft — subject to change.

For latest version, see `http://www.metasl.org`

 ©2011 mental images GmbH. All rights reserved.

Copyright Information

Copyright © 1986-2011 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 cloud™, mental mesh®, RealityServer®, RealityPlayer®, RealityDesigner®, MetaSL®, Metanode®, Phenomenon®, neuray®, iray®, DiCE™, imatter®, Shape-By-Shading®, SPM®, 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.

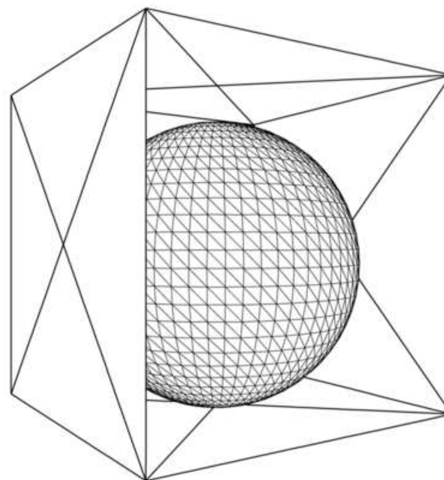
Chapter 3

Basic surface shaders

In the previous chapter, a MetaSL shader was described as a type of function, having input data that controls the way in which a process creates output data. But a shader does not perform its calculations in isolation, but as a component in a rendering system designed with a *plugin architecture*. The role of the shader in the *rendering pipeline* determines the requirements of the input, output, and process that the shader defines. To provide a rendering context for shaders, this chapter begins with a description of a generalized rendering system which will serve as a model for the series of shaders that follow.

3.1 Geometric data and the virtual camera

In a rendering system, a *scene* is the geometric description of surfaces and other data from which the rendering process will produce a picture. A software application that produces scene geometry is typically called a *modeling system* and produces the *three-dimensional models* that represent the objects in the virtual world that will be rendered. Modeling systems can employ various mathematical methods to represent three-dimensional objects. The typical properties of these models assumed by MetaSL will be covered in the course of developing the shaders described in this book.

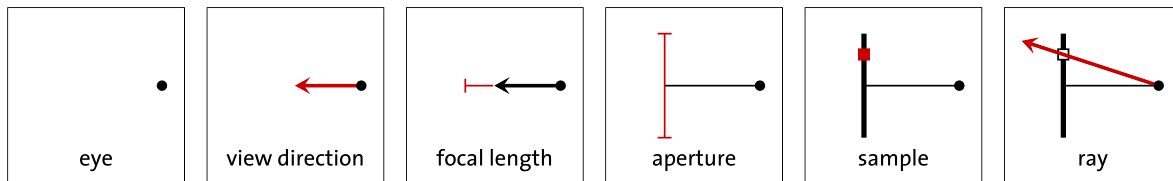


Geometric models composed of triangles

 ©2011 mental images GmbH. All rights reserved.

Rendering systems also require a method for determining what part of the scene should be rendered, in the same way that the position and orientation of a real camera determines the contents of any photograph that it takes. However, the simplest *virtual camera* in a rendering system is defined not by the optics of lenses, but by a small set of geometric elements. (In Chapter *X*, *lens shaders* will extend this simple camera model and how it constructs the final image.)

To explain the default three-dimensional camera assumed by MetaSL, it is helpful to first describe a two-dimensional simplification of the camera geometry.



Constructing and using the virtual camera

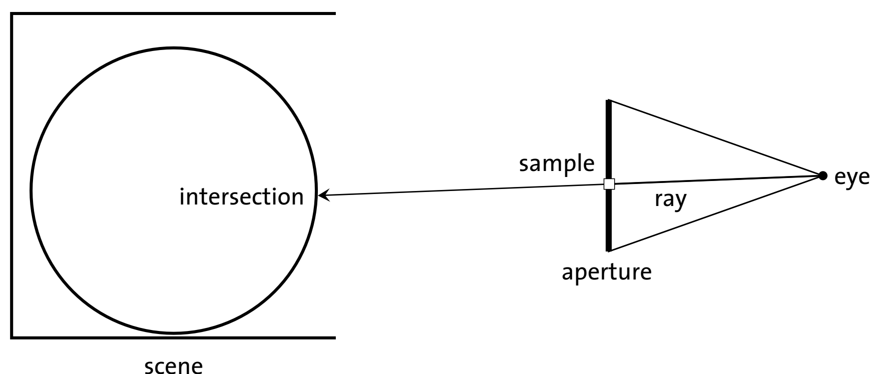
Constructing the virtual camera requires the definition of four geometric elements:

1. Define a point in space, called the *eye*.
2. Define a direction from the eye, called the *view direction*.
3. Define a distance from the eye along the view direction, called the *focal length*.
4. Define a line segment, called the *aperture*, at right angles to the view direction and centered around it, separated from the eye by the focal length.

Rendering an image with this virtual camera requires two more elements:

1. A point in the aperture for which the rendering system will determine a color, called a *sample*.
2. A line beginning at the eye and proceeding through the sample point, called a *ray*.

A ray extended from the aperture into the geometry of the scene may strike an object, called an *intersection* with that object. Extending a ray from the virtual camera into the scene is called *tracing a ray*. A set of samples produced by tracing multiple rays form an *image* in the aperture.

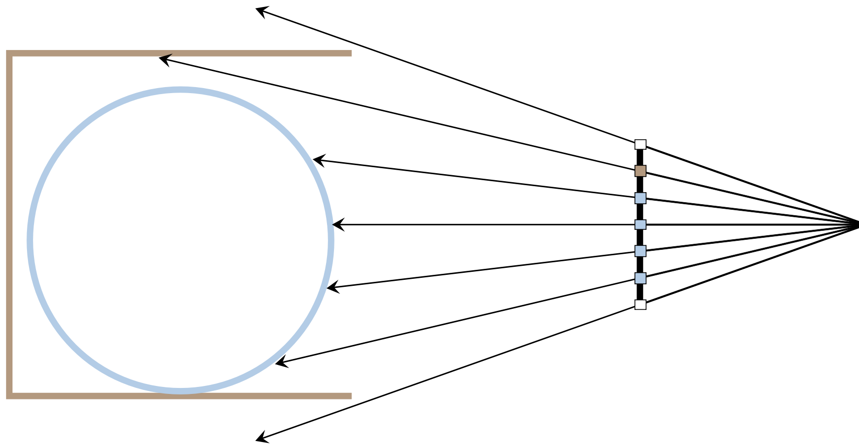


The virtual camera and a ray that intersects an object

 ©2011 mental images GmbH. All rights reserved.

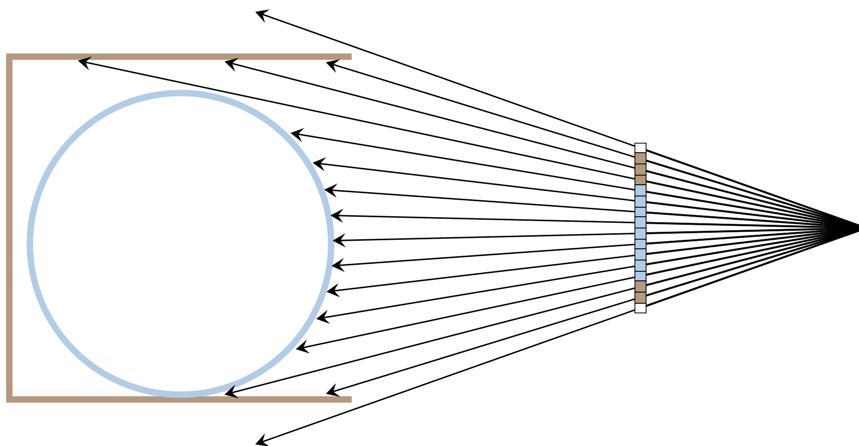
When a ray intersects the surface of an object, the rendering system acquires a color from the *surface shader* associated with the object by executing the `main` function of the shader. This association is part of the definition of the scene. The color that the surface shader calculates for its `result` value is the color defined for the sample through which the ray was traced. For example, a surface shader with a `result` of blue produces a blue sample; a shader `result` of brown produces a brown sample.

A ray may not strike any object in the scene. Rays that do not intersect an object are defined instead to intersect an "object" that surrounds the scene and that has no geometric representation called the *environment*. Defining the color resulting from a ray intersecting the environment is the responsibility of *environment shaders* (discussed in Chapter *X*).



Surface shaders executing at ray intersections creating blue and brown samples in an environment of white

To calculate the color of more samples — that is, to create an image of higher resolution — more rays can be traced in exactly the same manner. The rendering system combines these samples to produce the set of pixels that compose the final image.

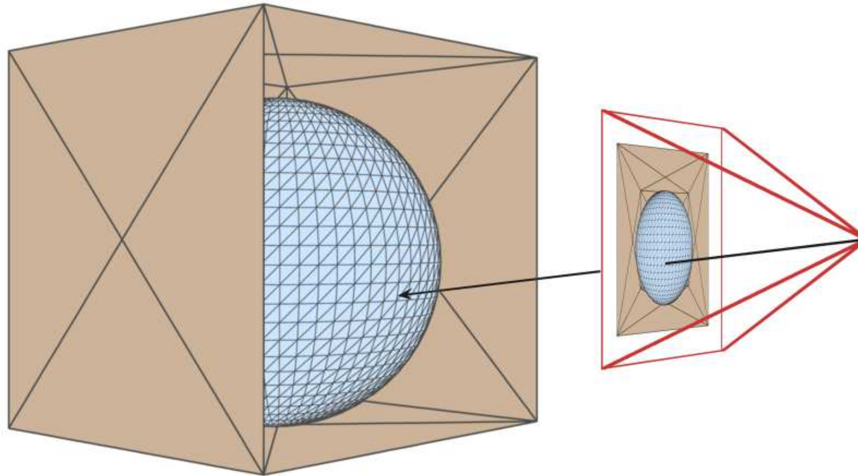


More rays allowing for more samples to produce images of higher resolution

For a three-dimensional virtual camera, the "aperture" becomes a rectangle characterized by the ratio of its width to its height, called the camera's *aspect ratio*. Rays are traced through the aperture into the scene as in

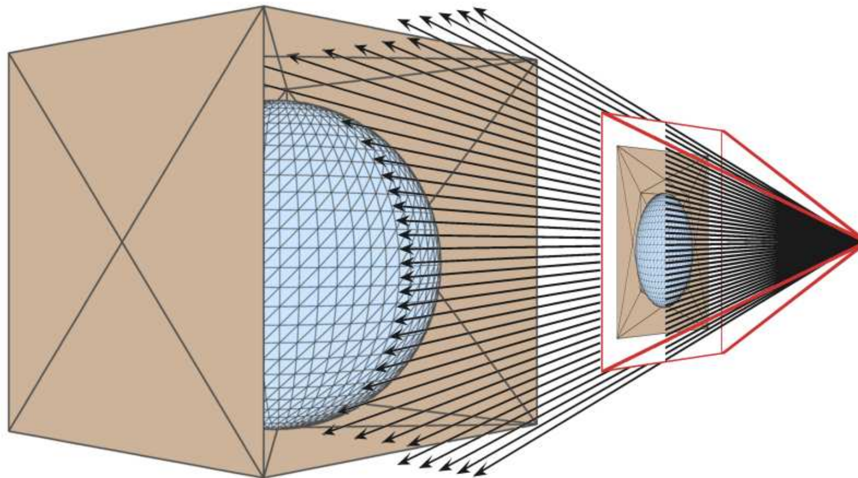
 ©2011 mental images GmbH. All rights reserved.

the two-dimensional case. The colors that result from executing the surface shaders become part of a grid of image pixels. The image is a *projection* of the scene visible through the aperture.



A single ray and the projected scene in a three-dimensional virtual camera

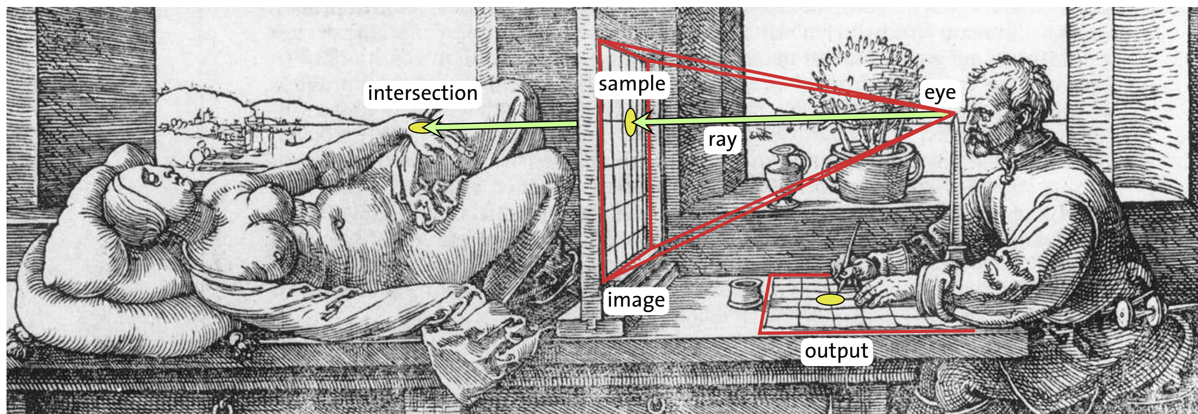
Increasing the number of pixels in the final image, like the two-dimensional case, it simply a matter of increasing the density of rays sent through the aperture into the scene.



A series of samples in a three-dimensional virtual camera

Using a grid in front of an eye position to project an image was a fundamental idea in the development of perspective during the Renaissance. A famous woodcut by Albrecht Dürer contains all the components of the virtual camera described so far. Dürer also includes the final rendering step. For rendering systems, pixel values are written to a digital file; Dürer's draughtsman records his sampling results by drawing on paper ruled by a grid.

 ©2011 mental images GmbH. All rights reserved.



Finding the virtual camera in traditional perspective drawing
*(*Draughtsman Drawing a Recumbent Woman, *Albrecht Dürer, 1525, woodcut)*

3.2 The shader's role in the scene

The previous section described the way in which a rendering system uses the surface shader of a geometrically defined object to calculate the color of a sample. The scene definition will include the association of shader and object as well as specific values to be used for the shader's input parameters. These specific values supplied for the input parameters are known as *shader arguments*.

The relationship between the abstract definition of a shader and its use in rendering is analogous to the definition and use of a function. A *function definition* contains the MetaSL statements that calculate its return value; a *function call* supplies the function with arguments from which the return value is calculated.

```
float multiply(float a, float b) {
    return a * b;
}
```

Function definition

```
multiply(6, 7)
```

Function call

In the same way, a *shader definition* contains the MetaSL parameters and functions that will calculate its result value. A *shader call* includes the actual values to be used by the shader in calculating the result value to be used by the rendering system.

```
shader Multiply {
  input:
    float a = 1;
    float b = 1;
  output:
    float result;
  member:
    void main() {
        result = a * b;
    }
};
```

Shader definition

```
Multiply (
    a: 6,
    b: 7 )
```

Shader call

 ©2011 mental images GmbH. All rights reserved.

Though functions and shaders appear to be very similar in their use of inputs to calculate outputs, there are two structural differences to note:

- A function's arguments are defined by their position in the function call, so all arguments must be supplied. A shader, in contrast, includes the parameter name followed by a colon character (`:`), before the argument value. Arguments can therefore be supplied in any order. If an input parameter defines a default value, the argument for that parameter may be omitted in the shader call; the default value for that argument will be used in the shader's methods.
- A function can provide output values either by using the `return` statement or by assigning a value to `out` or `inout` parameters. A shader can only specify output values by assignment to the parameters defined in the output section.

A function or shader that multiplies two numbers is trivial, but the same dichotomy of definition and call apply for functions and shaders at any level of complexity. A call for the `blend` shader of the last chapter requires arguments of type `Color3`.

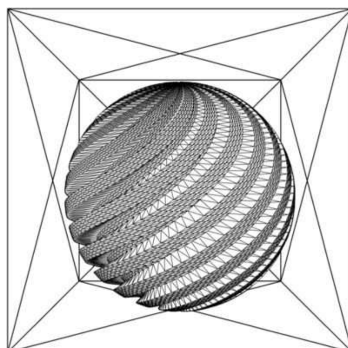
```
shader Blend {
  input:
    Color3 color_1 = Color3(0.0, 0.0, 0.0);
    Color3 color_2 = Color3(1.0, 1.0, 1.0);
    float color_1_fraction = 0.5;
  output:
    Color3 result;
  member:
    void main() {
        result = color_1 * color_1_fraction +
                 color_2 * (1.0 - color_1_fraction);
    }
};
```

Shader definition

```
Blend (
    color_1: Color3(0.7, 0.8, 0.9),
    color_2: Color3(0.7, 0.6, 0.5),
    color_1_fraction: 0.7 )
```

Shader call

For the example in shaders in the next several chapters, an example shader will be accompanied by a rendered image of a box containing a ball with spiral grooves. The shader calls used in rendering the image are listed to the right of the image. The box and the ball may be rendered with the same shader and arguments, with the same shader using different arguments, or two different shaders altogether.



scene	*shader call*
	OR
ball	*shader call*
box	*shader call*

The structure of the rendering examples

 ©2011 mental images GmbH. All rights reserved.

3.3 The simplest shader

The last chapter described the typical structure of a shader as input values processed to produce output values. But it is also possible to have a shader without any inputs at all. In that case, the shader produces the same result value every time it is called.

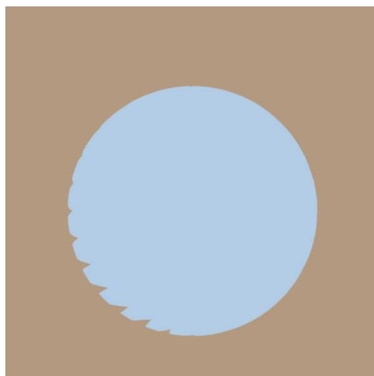
For example, the shader `Blue` always produces the same `Color` value that is created on line 6 and assigned to the `result` output parameter.

```
1  shader Blue {
2    output:
3      Color result;
4    member:
5      void main() {
6          result = Color(0.7, 0.8, 0.9, 1.0);
7      }
8  };
```

A shader without input parameters can be used to hide details that should not be changed by the person using the shader, but only by the shader's author. Such a shader could represent a design choice that should only be changed in a controlled and limited way. Enclosing a design decision within a shader is another example of encapsulation, where details have been hidden behind the layer of an interface.

```
1  shader Brown {
2    output:
3      Color result;
4    member:
5      void main() {
6          result = Color(0.7, 0.6, 0.5, 1.0);
7      }
8  };
```

Without any input parameters, a shader call will still represent the input parameters, but only by an empty pair of parentheses.



ball `Blue ()`

box `Brown ()`

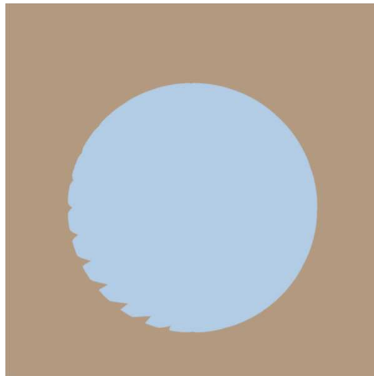
The process of expanding the definition of input parameters from "one or more input values" to "zero or more input values" is an example of *generalization* where an existing structure is newly understood to be a special case of a more general structure.

The process of generalization also occurs when data defined as constant within a shader is defined instead as parameters to the shader, or *parameterized*. For example, given the `Blue` and `Brown` shaders above, a shader that parameterizes the internal color provides an input parameter that defines that color.

 ©2011 mental images GmbH. All rights reserved.

```
1  shader Constant_color {
2    input:
3      Color color = Color(1.0, 1.0, 1.0, 1.0);
4    output:
5      Color result;
6    member:
7      void main() {
8          result = color;
9      }
10 };
```

Now the color to be used for the surface is defined by the shader call as the argument to the `color` parameter.



ball
```
Constant_color (
   color: Color(0.7, 0.8, 0.9, 1.0) )
```

box
```
Constant_color (
   color: Color(0.7, 0.6, 0.5, 1.0) )
```

Moving the definition of color in the `Blue` and `Brown` shaders to a parameter in the `Constant_color` shader is an example in miniature of the process of *parameterization*, in which a shader becomes more generally useful by adding input parameters that replace constant values in the shader's member functions. This is often the process that drives the evolution of a shader in a production environment as users request additional features from shader programmers. Many shaders in this book will also be developed by increasing the scope of their applicability in this way.

 ©2011 mental images GmbH. All rights reserved.