
MetaSL

Shading Language Tutorial and Reference

Andy Kopra and Lutz Kettner

Excerpt:

Chapter 4: Rendering state

Preliminary draft — subject to change.

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

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 cloudTM, mental mesh®, RealityServer®, RealityPlayer®, RealityDesigner®, MetaSL®, Metanode®, Phenomenon®, neuray®, iray®, DiCETM, imatter®, Shape-By-Shading®, SPM®, and rendering imagination visibleTM 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 4

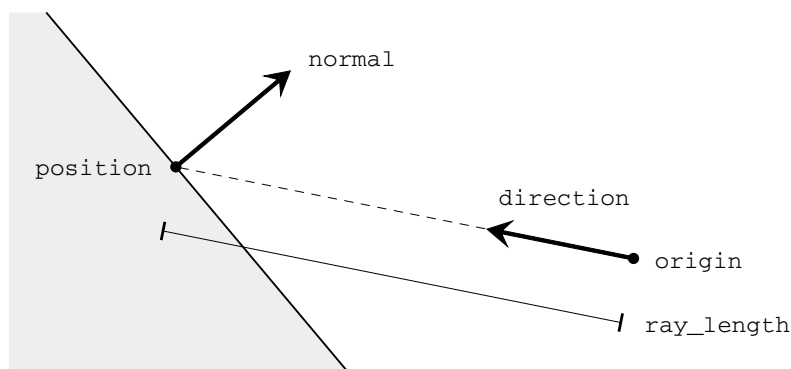
Rendering state

The `Blue` and `Brown` shaders of the previous chapter always return the same result value wherever they are called; they are independent of the context of the ray for which a surface color is being calculated. To provide for shaders that depend upon the context in which the shader is called, MetaSL defines a set of *state variables*. The values of state variables are set by the rendering system and are available in shader statements in the same manner as shader parameters and local variables.

MetaSL also provides a set of *state functions*. Like state variables, these functions are dependent upon the rendering context when they are called in a shader. Together with the library functions, state variables and state functions make up the basic building blocks of MetaSL shader programming.

4.1 Describing rendering state

MetaSL defines over sixty state variables that describe the geometric properties and rendering attributes of cameras, objects, and lights. For example, some state variables describe the geometric relationship between a ray and a surface it strikes.



Geometric relationship of five state variables that describe the intersection of a ray with a surface

Appendix X, *MetaSL state variables*, describes all the state variables, dividing them into their functional categories as well as listing the state variables that are available in each of the six shader types.

For example, five other state variables complete the description of ray intersection in the previous diagram. The complete set of intersection state variables are listed in the following table.

<i>State variable</i>	<i>Type</i>	<i>Description</i>
<code>backside</code>	<code>bool</code>	Set to true when the ray hits the geometry from behind. In this case both the normal and <code>geometry_normal</code> are inverted.
<code>direction</code>	<code>float3</code>	The ray direction. This vector is unit length and is dependent on <code>origin</code> and <code>position</code> .
<code>dot_nd</code>	<code>float</code>	The dot product of the normal and the ray direction. This variable is dependent on <code>normal</code> and <code>direction</code> .
<code>geometry_normal</code>	<code>float3</code>	The true surface normal for the current geometry. This vector is unit length.
<code>inside</code>	<code>bool</code>	Set to true when the current ray is intersecting the surface from inside the volume defined by the surface. The value of <code>inside</code> can be different from <code>backside</code> when the surface normals are oriented such that the front side of the surface faces the interior of the volume.
<code>normal</code>	<code>float3</code>	The surface normal for shading. This vector is unit length.
<code>origin</code>	<code>float3</code>	The ray origin.
<code>position</code>	<code>float3</code>	The intersection point on the surface.
<code>ray_length</code>	<code>float</code>	The length of the current ray. This value is dependent on <code>origin</code> and <code>position</code> .

State variables describing the intersection of a ray with a surface

Some state variables are used frequently in MetaSL programming; others may only be required in special circumstances. The next sections describe three commonly used state variables and show how simple shaders can help visualize their geometric properties.

4.2 Position of the ray intersection

The `position` state variable provides the three-dimensional coordinates of the current surface point that the ray has intersected. This is a necessary component of the rendering state in a surface shader — the ray intersection is the reason the shader has been called.

4.2.1 Displaying position as color

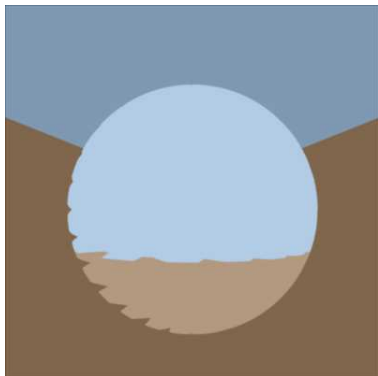
Shader `Height_color` returns one of two colors supplied as input parameters based on a comparison of the `height` input parameter and the `y` component of `position`.

```

1  shader Height_color {
2    input:
3      float height = 0.0;
4      Color below(0.0, 0.0, 1.0, 1.0);
5      Color above(1.0, 1.0, 0.0, 1.0);
6    output:
7      Color result;
8    member:
9      void main() {
10       if (position.y > height)
11         result = above;
12       else
13         result = below;
14     }
15 };

```

The comparison of `position.y` and the `height` input parameter determines which color is returned by the shader.



```

Height_color (
    height: -0.2,
    below: Color(0.7, 0.6, 0.5, 1.0),
    above: Color(0.7, 0.8, 0.9, 1.0) )
ball

Height_color (
    height: 0.2,
    below: Color(0.5, 0.4, 0.3, 1.0),
    above: Color(0.5, 0.6, 0.7, 1.0) )
box

```

In lines 10 to 13 of `Height_color`, the assignment to `result` can be expressed more simply using the *ternary operator* for conditionals, adopted by MetaSL from C. Line 10 of shader `Height_color_v2` avoids the repetition of the assignment to `result` in the `else` clause of the original version of this shader.

```

1  shader Height_color_v2 {
2  input:
3  float height = 0.0;
4  Color below(0.0, 0.0, 1.0, 1.0);
5  Color above(1.0, 1.0, 0.0, 1.0);
6  output:
7  Color result;
8  member:
9  void main() {
10     result = (position.y > height) ? above : below;
11 }
12 };

```

4.2.2 Coordinate systems

The `position` state variable contains the x , y , and z coordinates of a point in three-dimensional space. But what do those numerical values represent? A modeling application defines the coordinates of a model's surfaces, but an animation application will reposition that model throughout the scene, effectively defining new values for the model's coordinates. Which of these values does the `position` state variable actually represent?

To remove this ambiguity and to allow for control over the meaning of scene coordinates in a shader, MetaSL defines six *coordinate spaces*, named `object`, `light`, `world`, `camera`, `raster`, and `internal`. Each coordinate space defines a *coordinate system* which is characterized by how the *coordinate system origin* is defined — where the values of x , y , and z are all equal to zero — and by what the directions of the x , y , and z axes represent in that coordinate space. This chapter will describe the `object`, `world`, and `internal` spaces. (The `light` space is described in Chapter X, “Light shaders”; the `camera` and `raster` spaces are described in Chapter Y, “Lens shaders.”)

The `object` space defines the coordinate system in which a geometric model was constructed. The origin of the space, the orientation of the object with respect to the space, and the meaning of the numerical values will vary from model to model. The `world` space, however, is a single coordinate system throughout the rendered scene. In these terms, placing an object in the scene to be rendered is equivalent to *transforming the object coordinates to world coordinates*.

By default, all state variables that define spatial coordinates are located in an undefined `internal` space. The `internal` space allows a rendering system that employs MetaSL shaders to use whatever coordinate system is most suitable for its implementation. Shader `Height_color` used the `internal-space` value of

`position` state variable. Because internal space may vary between implementations, there is no guarantee that shader `Height_color` will produce the same image in different implementations of MetaSL; that shader is therefore *platform dependent*.

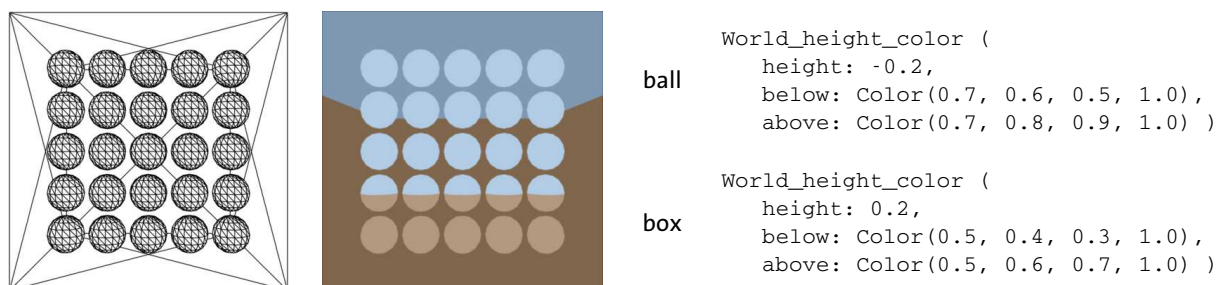
For a shader to be *platform independent*, the shader needs to explicitly transform internal space coordinates to the space required by the shader using the state function `transform_point`. Function `transform_point` can convert a point in one space to the equivalent point in another. The function's three arguments specify the current space of the point, the space to which the point should be transformed, and the point to be transformed. For example, shader `World_height_color` is a modification of shader `Height_color` that uses function `transform_point`:

```

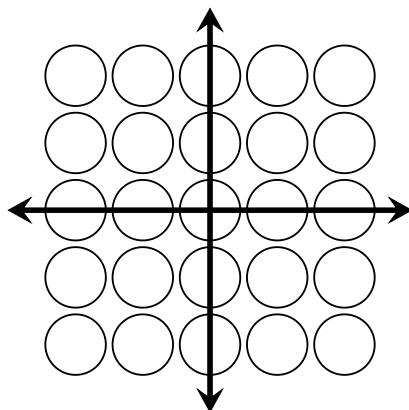
1  shader World_height_color {
2      input:
3          float height = 0.0;
4          Color below(0.0, 0.0, 1.0, 1.0);
5          Color above(1.0, 1.0, 0.0, 1.0);
6      output:
7          Color result;
8      member:
9          void main() {
10             float3 world_position =
11                 transform_point("internal", "world", position);
12             result = (world_position.y > height) ? above : below;
13         }
14     };

```

In shader `World_height_color`, the `position` state variable is transformed from internal space to world space to define the local variable `world_position` on lines 10-11. Rendering a scene with multiple objects using the same shader arguments for the previous scene demonstrates that in world space objects share the same coordinate system.



The shader `World_height_color` is used for all twenty-five spheres in the scene, but the “position” each uses is relative to a single coordinate system.



Two-dimensional example of a world coordinate system used for all objects

A shader that uses `object` space must make assumptions about the actual coordinate values used to define the surfaces of the object. In architectural applications, for example, the scale of coordinates may correspond to the actual size of an object in some linear system of measurement (meters, inches, cubits, *kyynärät*) and the position and orientation may be standardized by conventions defined by the application (for example, the smallest `y` value is 0.0, with the positive `y` axis pointing up).

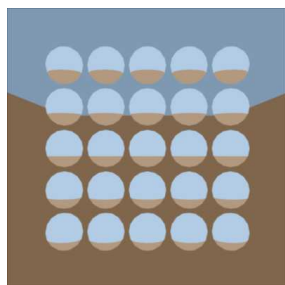
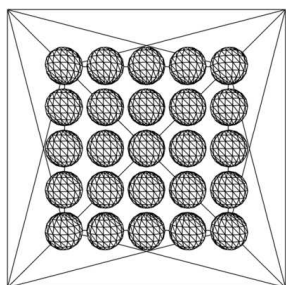
Shader `Object_height_color` is a version of shader `Height_color` that uses object space and is identical in structure to `World_height_color`:

```

1  shader Object_height_color {
2      input:
3          float height = 0.0;
4          Color below(0.0, 0.0, 1.0, 1.0);
5          Color above(1.0, 1.0, 0.0, 1.0);
6      output:
7          Color result;
8      member:
9          void main() {
10             float3 object_position =
11                 transform_point("internal", "object", position);
12             result = (object_position.y > height) ? above : below;
13         }
14     };

```

The spheres in the previous scene are all generated from a single spherical object with a diameter of 1.0 and centered around the origin. The scene description provided to the renderer creates separate versions of the source object — called *instances* — that are placed throughout the scene by a *transformation* within the scene description. It is this transformed coordinate position that `World_height_color` used in its calculation. Shader `Object_height_color` uses the original coordinate values of the model instead.



```

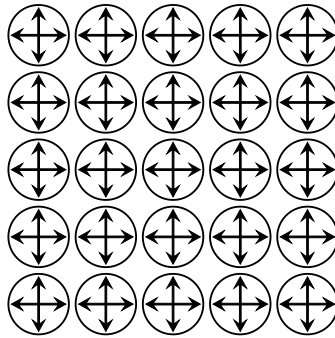
Object_height_color (
    height: -0.2,
    below: Color(0.7, 0.6, 0.5, 1.0),
    above: Color(0.7, 0.8, 0.9, 1.0) )
ball

Object_height_color (
    height: 0.2,
    below: Color(0.5, 0.4, 0.3, 1.0),
    above: Color(0.5, 0.6, 0.7, 1.0) )
box

```

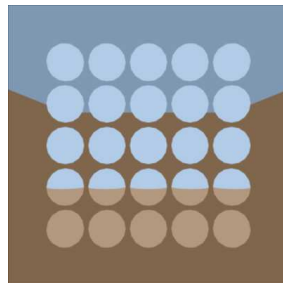
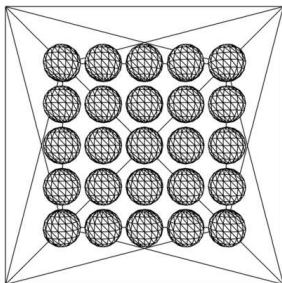
The argument value for parameter `height` in `Object_height_color` is `-0.2`. For each sphere, then, the division into blue and brown occurs a little below the middle of the sphere. The image also displays the variable appearance of the spheres due to perspective — more of the brown is visible for the upper spheres, less for the lower ones.

Remember, however, that the local coordinate system of an object is arbitrarily based upon how it was constructed. In this case, the origin of the source object for all of the sphere instances is in the center of the sphere.



Local coordinate system unique to each object

The MetaSL specification does not require that using the `position` state variable in internal space will signal an error, but such use does make a shader platform dependent, as well as more prone to confusing errors in different rendering environments. Rendering the twenty-five spheres with the untransformed `position` state variable in shader `Height_color` produces this image:



```

Height_color (
ball   height: -0.2,
        below: Color(0.7, 0.6, 0.5, 1.0),
        above: Color(0.7, 0.8, 0.91, 1.0) )

Height_color (
box    height: 0.2,
        below: Color(0.5, 0.4, 0.3, 1.0),
        above: Color(0.5, 0.6, 0.7, 1.0) )

```

For the implementation of this renderer, the internal space appears to be identical to world space, but shader code should never depend upon such an apparent equivalence.

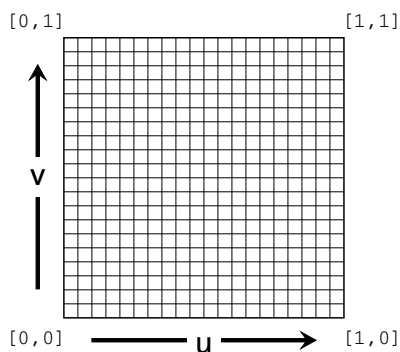
4.3 Texture coordinate state

The value of the `position` state variable is the result of the ray's intersection with the surface of a geometric model. It is an *intrinsic* property of a model defined by surfaces and can therefore always be made available to a shader through the `position` state variable. Other MetaSL state variables store data that are not strictly necessary for the geometric definition of the surface. Like the geometry of the model's surfaces, the values of these optional data types are defined as part of the model's dataset. These *extrinsic* types provide other kinds of information useful to shaders.

4.3.1 Displaying texture coordinates as color

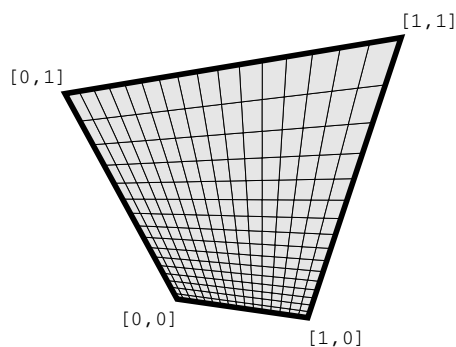
An example of optional model data available through a MetaSL state variable is the array of *texture spaces* stored in the `texture_coordinate` state variable. A texture space is a two-dimensional coordinate

system. By common convention, coordinate values in this space lie within the range of 0.0 to 1.0 in both axes. Traditionally, the equivalent to the x axis is called the u axis; the y axis is called the v axis. These two axes define the set of uv coordinates that locate two-dimensional positions within the space.



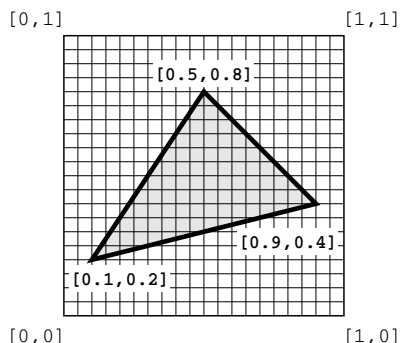
Coordinate system for texture spaces

In the simplest geometric models defined by sets of polygons, the vertex of each polygon may define the uv coordinates of that vertex. The uv coordinate of any point on the polygon is defined by its spatial relationship to the uv coordinates of the vertices of that polygon. Based on the relative position of the vertices, the texture space can be apparently stretched or squeezed.



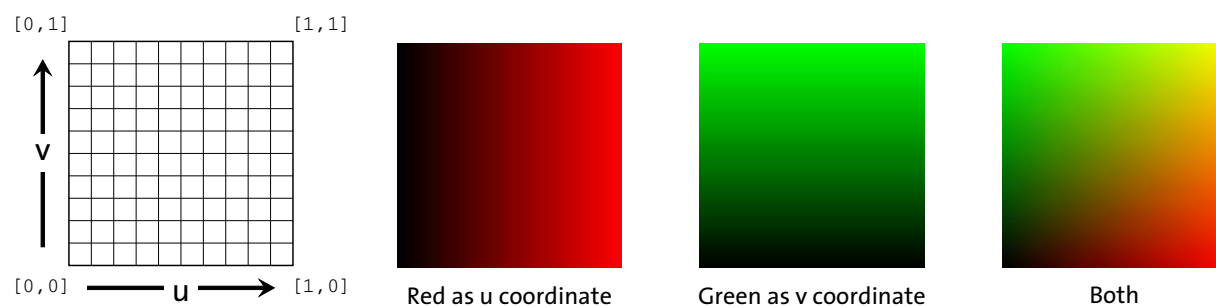
Texture coordinates assigned at the vertices of a polygon

However, the uv coordinates of vertices are arbitrary and may not correspond to the borders of the texture space. For example, the uv coordinates can be chosen to preserve the original proportions of the texture space.



Texture coordinates projected on a triangle

To visualize a model's texture space in a MetaSL shader, the u and v coordinates can be interpreted as the red and green components of the color of the surface.



Visualizing uv coordinates as colors

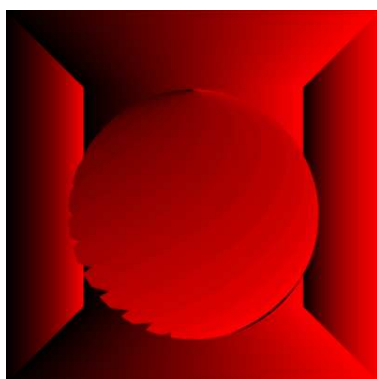
The state variable that stores texture spaces in MetaSL is an array, `texture_coordinate`, allowing the modeling system to store multiple texture spaces for an object. For a simple shader that displays the uv coordinates as colors, the `x` and `y` components of `texture_coordinates[0]` are interpreted as red and green. Boolean input parameters control whether `u` and `v` are included in the output color. Since the uv coordinates in the example scene stay within the range of 0.0 to 1.0, we can use those values directly for the red and green values.

```

1  shader Show_uv {
2      input:
3          bool show_u = true;
4          bool show_v = true;
5          float blue = 0.0;
6      output:
7          Color result;
8      member:
9          void main() {
10             float red = show_u ? texture_coordinate[0].x : 0.0;
11             float green = show_v ? texture_coordinate[0].y : 0.0;
12             result = Color(red, green, blue, 1.0);
13         }
14     };

```

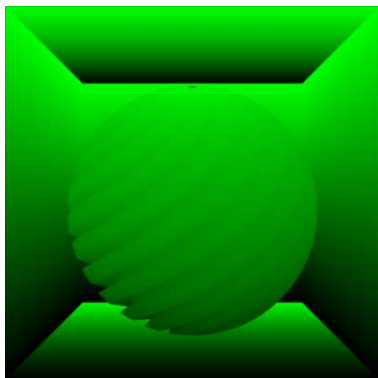
Rendering the scene with this shader displays how the uv coordinates have been defined for these two modes. Each side of the box includes the full range of `u` and `v` values. In contrast, the uv coordinates wrap around the ball.



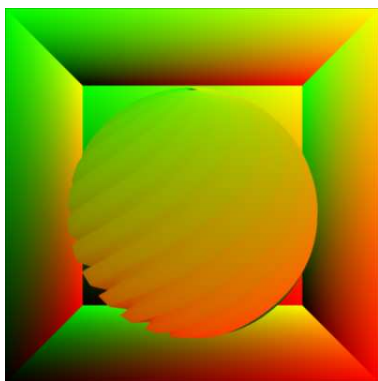
```

scene Show_uv (
    show_v: false )

```



```
scene Show_uv (
    show_u: false )
```

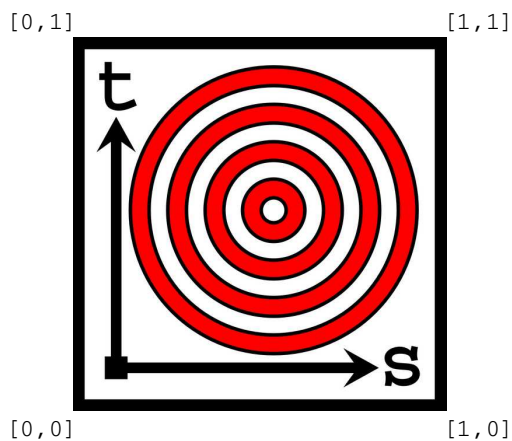


```
scene Show_uv ()
```

In the next section, these uv coordinates of the model will be put to use to define surface color.

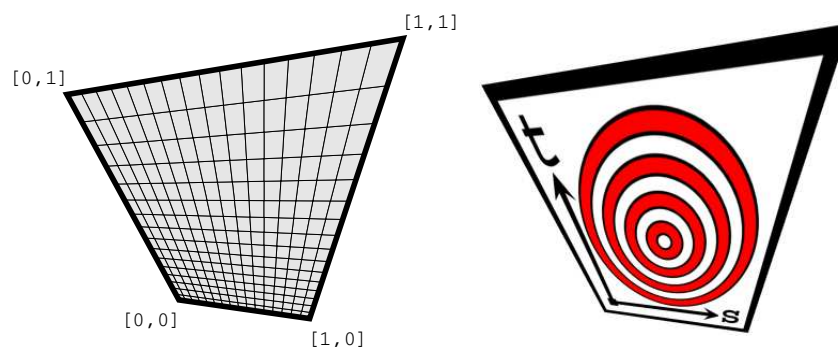
4.3.2 Texture mapping

Texture mapping is a traditional technique in rendering systems to associate the color values of digital imagery with the surfaces of geometric models. The uv coordinates of a surface are interpreted as the two-dimensional coordinates of a digital image, called a *texture map*. The axes of the image space are called *s* and *t*. Coordinate values typically vary from 0.0 to 1.0 in both directions.



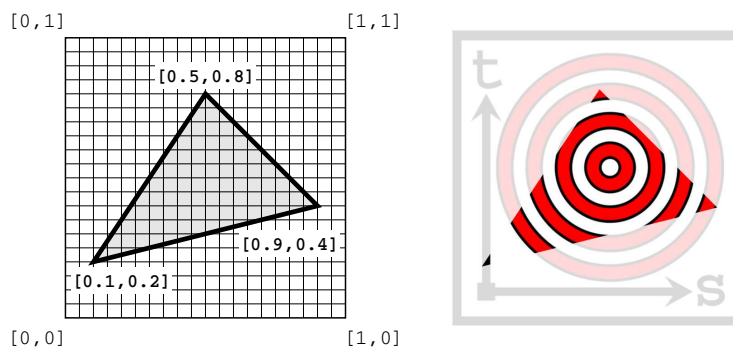
Texture map image showing s and t orientation and coordinates of image corners

The appearance of the image on the surface of the object is dependent upon the interpretation of the uv coordinates in the shader. If the proportional relationship between increments in *u* and *v* vary across the surface, then the image will be distorted accordingly.



Texture distorted based on uv coordinates

However, the uv coordinates of a surface could preserve the relationship of the st coordinates of the texture map, resulting in surface colors that duplicate the proportions of the source image.



Texture proportions preserved by uv coordinates

The library function `tex2D` is the simplest way to acquire colors from, or *sample*, a texture map.

```

1  shader Default_texture {
2  input:
3      Texture2d texture;
4  output:
5      Color result;
6  member:
7      void main() {
8          result = tex2d(texture, texture_coordinate[0].xy);
9      }
10 };

```

The `tex2D` function takes as its first argument a reference to a texture, which can vary between implementations of MetaSL based on the way that external data is defined in the rendering system. The second argument specifies the st coordinates of the texture from which the color should be sampled. In shader `Default_texture`, the uv coordinates are specified as st coordinates — a one-to-one mapping from the texture coordinate space of the object to the image space of the texture.



```
scene Default_texture (
    texture: "st_map.tif" )
```

By modifying the uv coordinates that are acquired from the object through state variable `texture_coordinate`, other relationships between the uv and st spaces can be defined. The color values of the texture can also be modified in any way for their use in the shader. For example, the `Place_texture` shader provides parameters that allow the texture to be repeated and shifted, as well as a color value that is multiplied by the color value of the texture for the shader's final result.

```
1  shader Place_texture {
2    input:
3      Texture2d texture;
4      float u_repeat = 1;
5      float v_repeat = 1;
6      float u_offset = 0;
7      float v_offset = 0;
8      Color tint(1.0);
9    output:
10     Color result;
11    member:
12     void main() {
13       float2 uv = texture_coordinate[0].xy;
14       uv.x = fmod((uv.x * u_repeat) + u_offset, 1.0);
15       uv.y = fmod((uv.y * v_repeat) + v_offset, 1.0);
16       result = tex2d(texture, uv) * tint;
17     }
18  };
```

The previous rendering showed that the sides of the box apparently have uv coordinates that match the 0.0 to 1.0 range of the texture space. The parameters for repeating and offsetting the uv coordinate values in shader `Place_texture` modify the relationship of uv and st coordinates.



```
ball Place_texture (
    texture: "st_map.tif",
    u_repeat: 8,
    v_repeat: 4,
    tint: Color(0.8, 0.9, 1, 1.0) )

box Place_texture (
    texture: "st_map.tif",
    u_repeat: 2,
    v_repeat: 2,
    u_offset: 0.5,
    v_offset: 0.5,
    tint: Color(1, 0.9, 0.8, 1.0) )
```

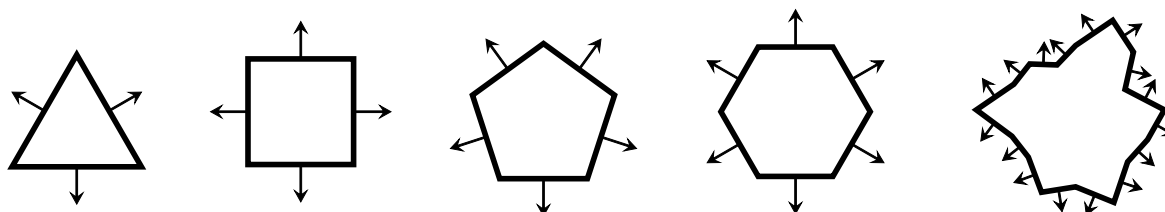
Procedural manipulation of the relationship between the uv and st systems can be effective for regular geometric shapes. For more organic and complex shapes, sophisticated modeling and texture-painting applications are often used to define the uv coordinates of a model and the corresponding texture maps so that no further manipulation of the coordinate systems are required.

4.4 The orientation of the surface

To determine how a surface will look in shaders that simulate direct lighting (described in the next chapter), the orientation of the surface and its relationship to the viewing direction must be considered. MetaSL provides several state variables that will be at the heart of any shader that implements a simulation of light.

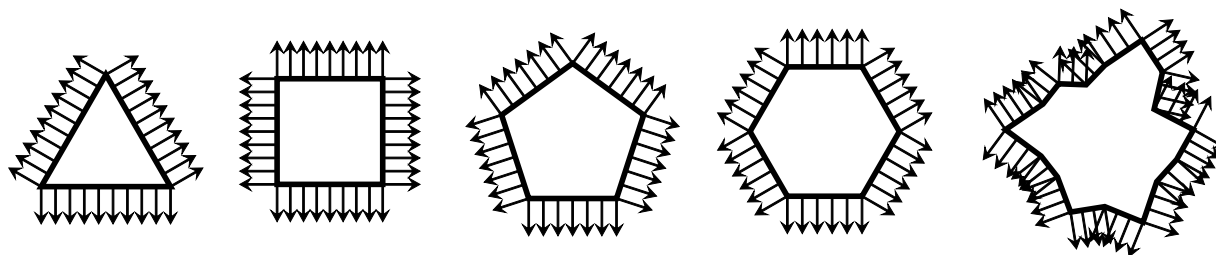
4.4.1 The normal vector

A *normal vector*, or *normal*, is a traditional way of describing the orientation of a surface. The normal is perpendicular to the surface, an extension in three-dimensions of the two-dimensional concept of a right angle.



A two-dimensional visualization of the normal vector

In MetaSL, the `geometry_normal` state variable is a vector of length 1.0 that contains the normal vector of the geometric primitive intersected by the ray. For a primitive that is flat, for example, a triangle, the `geometry_normal` state variable has the same value for the entire primitive.



The geometric normal defined across the face of a primitive

In the same way that the previous section visualized the texture space as a color, a shader can convert the x , y , z components of the normal vector into the red, green, and blue components of a color. The range of the vector's x , y , and z components is -1.0 to 1.0 . To convert these to the range for color components — 0.0 to 1.0 — the shader can simply divide the vector's components by 2 and add 0.5.

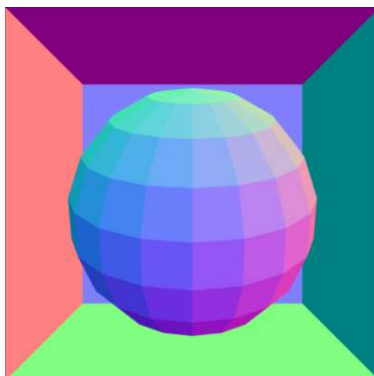
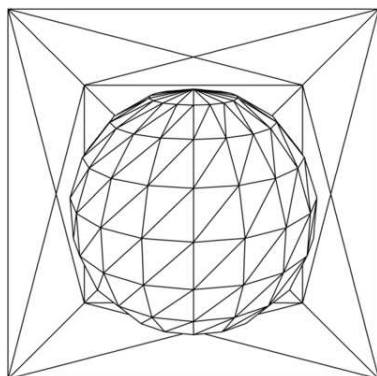
```

1  shader Show_geometry_normals {
2      output:
3          Color result;
4      member:
5          void main() {
6              result = Color(geometry_normal / 2 + 0.5, 1.0);
7          }
8  };

```

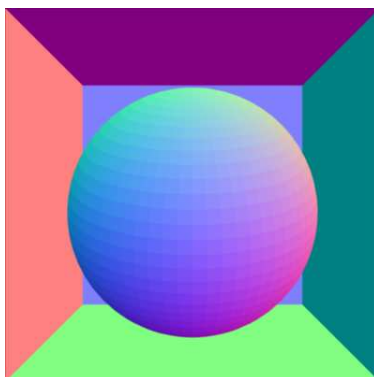
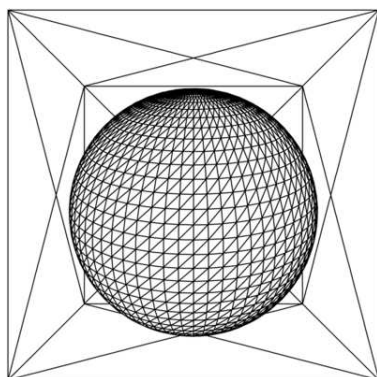
Scalar arithmetic operations on `geometry_normal` — a variable of type `float3` — are performed on all three components in parallel, producing a new value of type `float3`. This expression is used in line 6 for the red, green, and blue components of the `result` parameter of type `Color`. A constant value of 1.0 is specified for the alpha component. This `Color` value of `result` is created with the `Color` constructor that takes two arguments, a value of type `float3` and a value of type `float`.

Rendering a spherical shape made from a small number of triangles clearly displays the different normal values for the triangles. The image also shows that pairs of triangles in each horizontal band are rendered with the same color. These pairs of triangles lie in the same plane — they are *coplanar* — and therefore have the same normal.



scene Show_geometry_normals ()

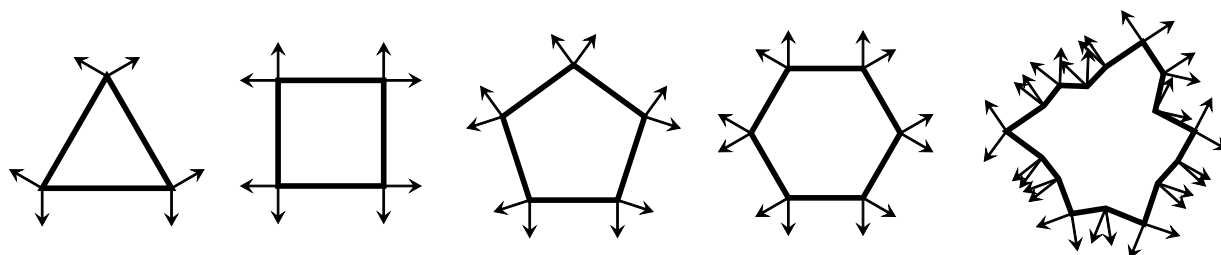
In the next chapter the lighting of a surface will be dependent upon its orientation; the normal vector is the basis of those calculations. Using the `geometry_normal` state variable, creating a better approximation of a perfect sphere requires the use of more triangles — an increase in its *polygonal resolution* — to construct the model.



scene Show_geometry_normals ()

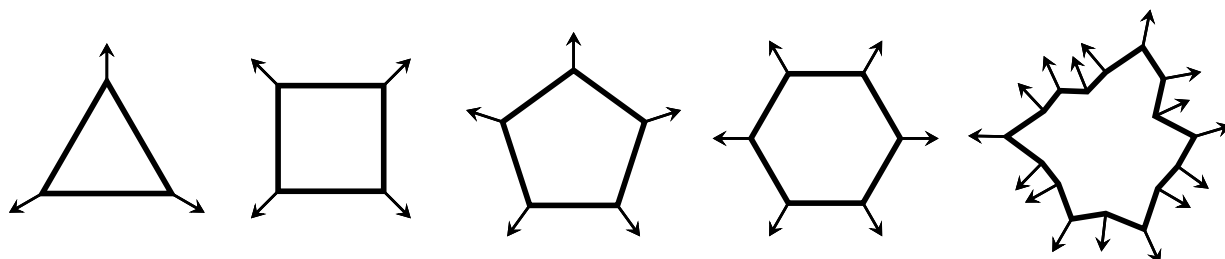
Even with a much higher polygonal resolution, the differing normals of the individual triangles are clearly visible. Any shader result dependent upon that normal will similarly show the same faceted appearance. However, a modeling system can supply additional information to replace the actual surface orientation of a primitive by defining a normal vector that should be used instead in shader calculations. In MetaSL, this *shading normal* is available as the state variable `normal1`. The shading normal is derived from data supplied by the modeling system, in contrast to the geometric normal that is an property of the geometry itself.

For example, a vertex in a polygonal model will be shared by some number of triangles. In the two-dimensional analogy, a vertex will be shared by two sides of the figure, each with a different normal.



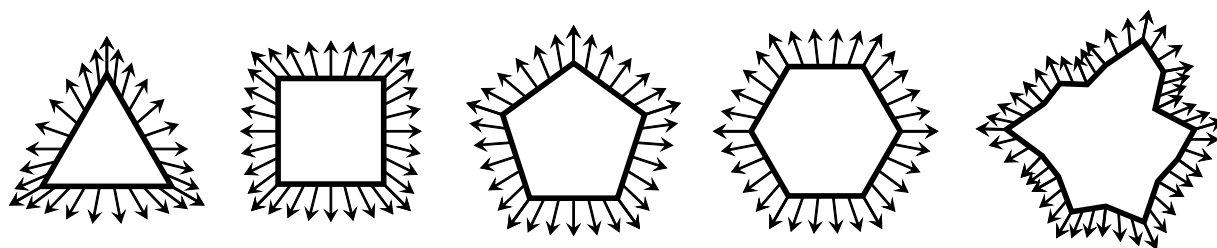
Vertices shared by normals

All the normal vectors at a vertex can be combined into a new normal by calculating their *vector average*. In the simplest case, the sum of all the corresponding x , y , and z components are each divided by the number of vectors. This new vector of coordinate sums is converted to a vector of length 1.0 by dividing by the length of the vector. This process of converting a vector to an equivalent vector of length 1.0 is called (in another use of the word “normal”) *normalizing the vector*. This new averaged vector of length 1.0 is called the *vertex normal*.



Averaging normals sharing a vertex

Now instead of using the same geometric normal across the face of a primitive, a normal is calculated as the *weighted average* of the vertex normals, proportional to the distance of the surface point from the vertices.



A weighted average between averaged vertex normals

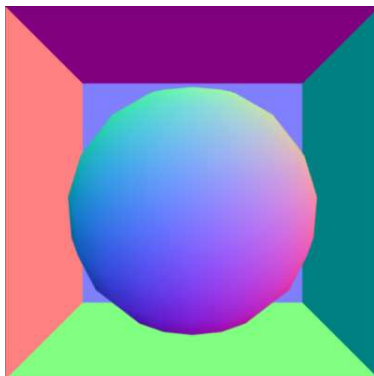
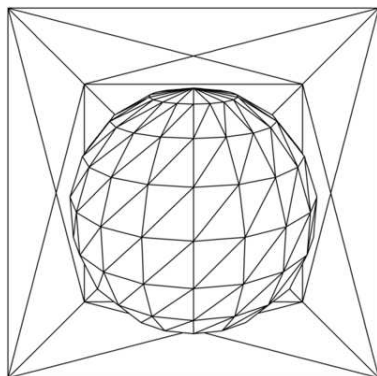
The MetaSL code for the shading normal is the same as for geometric normal; only the state variable has changed.

```

1  shader Show_normals {
2      output:
3          Color result;
4      member:
5          void main() {
6              result = Color(normal / 2 + 0.5, 1.0);
7          }
8  };

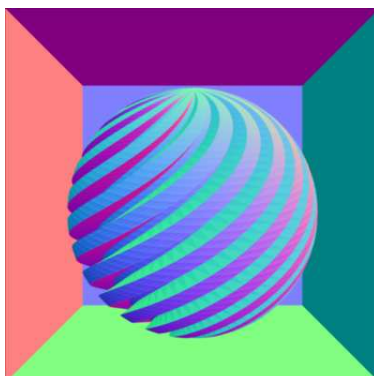
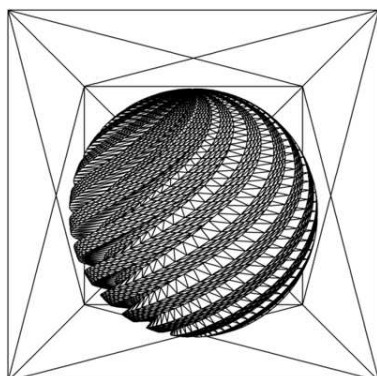
```

Now the spherical model of low polygonal resolution has a smooth appearance. Only the angular profile indicates the actual construction of the model.



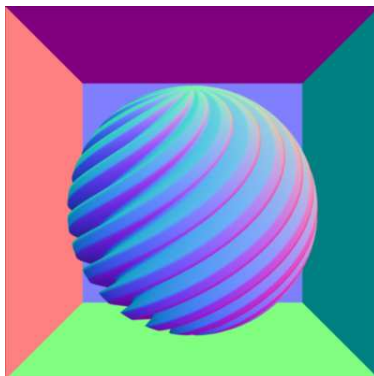
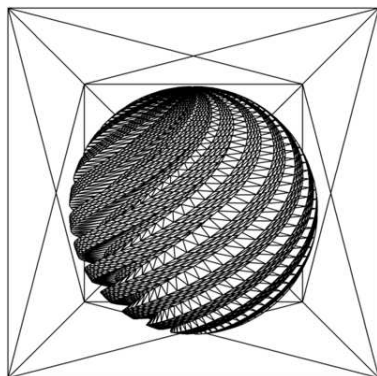
```
scene Show_normals ()
```

More complex models may depend upon higher-polygonal resolution for the representation of detail, but still use the manipulations of the shading normal for smaller scale effects. For example, the geometric normal in the following image clearly shows the sharp polygonal angles of the spiral bands.



```
scene Show_geometry_normals ()
```

The shading normal creates a slightly rounded effect due to the averaging of the normals at the edges.



```
scene Show_normals ()
```

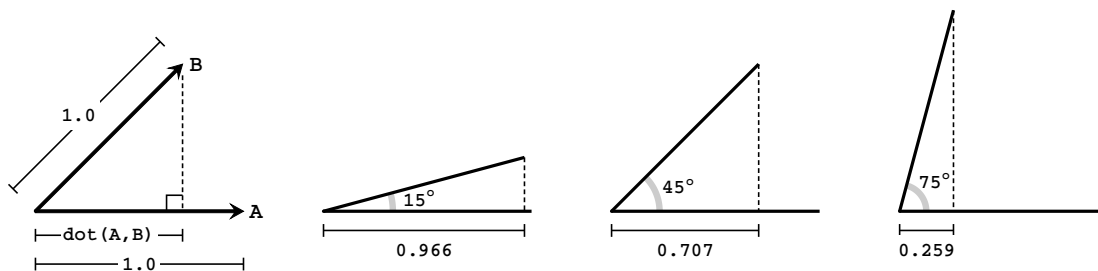
Though the code of these two shaders for the visualization of the normal vectors is extremely simple, they can be very helpful in verifying the correct construction of a model and in identifying problems that may be difficult to diagnose in more complex shading effects.

4.4.2 The dot product

The `normal` state variable defines the orientation of the surface. Lighting calculations also require a description of the viewing direction; many lighting effects are *view dependent*, changing as the relationship of the viewer to the scene changes.

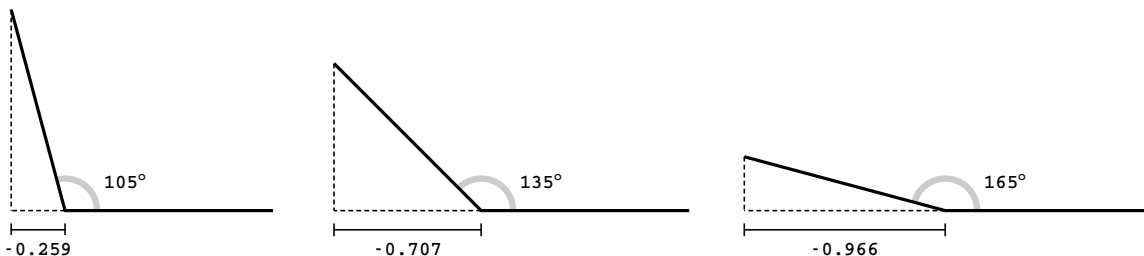
To characterize the relationship of the surface orientation to the viewing direction, MetaSL defines the state variable `dot_nd`, which is the *dot product* of the `normal` and `direction` state vectors. A geometric visualization of the dot product involves the *projection* of one vector on another — like a shadow from a

light shining directly above. If both vectors are *unit vector* (have a length of 1.0), then the length of the projection is the value of the dot product.



The length of the projection of one unit vector on another is the dot product of the two vectors.

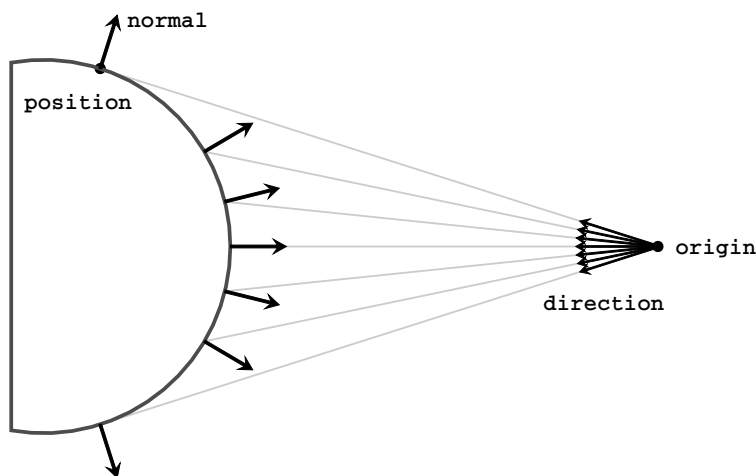
For angles between 90 and 270 degrees, the notion of projection is generalized to include projection on a number line that extends in the negative direction.



The dot product is negative for angles between 90 and 270 degrees.

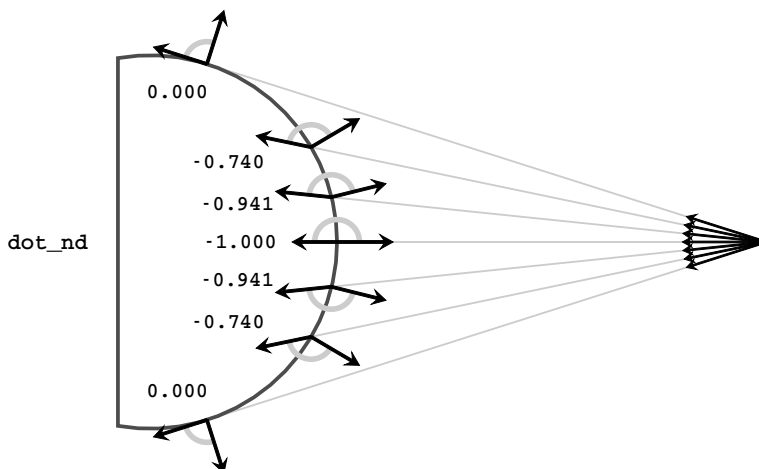
The dot product also has a trigonometric meaning — it is the cosine of the angle between the two vectors. This also becomes clear in two special cases. When the vectors are coincident — when they have an angle between them of zero — the dot product is 1.0. When the vectors form an angle of ninety degrees, the dot product is zero.

Four state variables can help demonstrate how the `dot_nd` state variable is defined using the dot product. The `direction` vector points from the `origin` to `position`, the ray's intersection point on the surface. The orientation of that point on the surface is defined by the state variable `normal`.



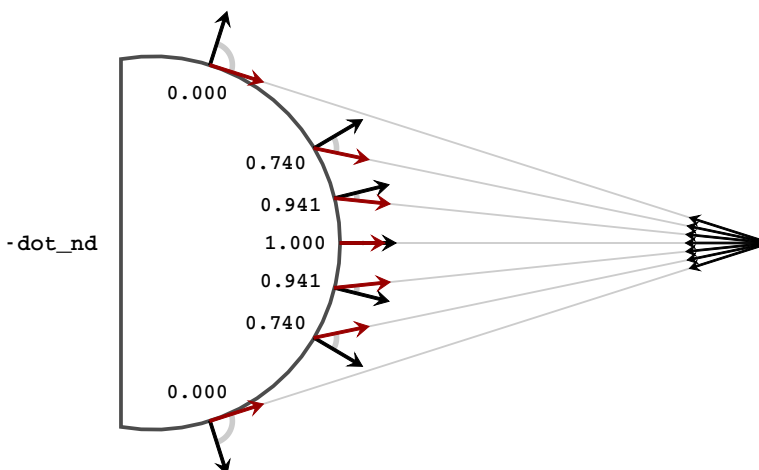
State variables describing the intersection of an object with the primary ray

The `dot_nd` state variable is the dot product of `normal` and `direction`. Notice that when the direction vector is tangent to the surface, the normal vector is at ninety degrees and the dot product is zero.



State variable `dot_nd`, the dot product of normal and direction.

Negating the `dot_nd` state variable is equivalent to reversing one of the vectors. Some rendering systems use the vector *toward* the eye to describe the viewing direction. This is equivalent in MetaSL to negating the `direction` vector.



Geometric representation of state variable `dot_nd` multiplied by -1.0

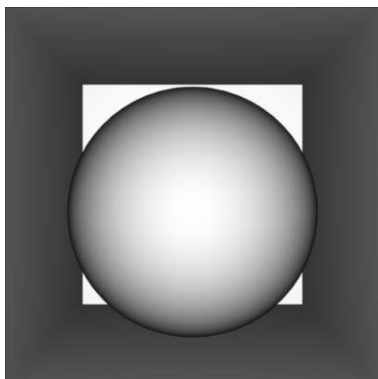
The negation of `dot_nd` has a very useful property. If the surface normal is pointed directly toward the eye, the value is 1.0. If the surface normal is at ninety degrees to the eye direction, the value is 0.0. This means that `-dot_nd` is a measure of the degree to which a surface faces the eye. A simple shader can turn this number into a grayscale value, in which white means that the surface is directly facing the eye and darker values show surfaces seen more at an angle.

```

1  shader Front_facing {
2      output:
3          Color result;
4      member:
5          void main() {
6              result = -dot_nd;
7          }
8  };

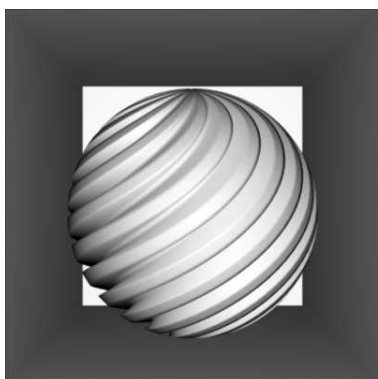
```

For example, a sphere rendered with `Front_facing` clearly shows the variation from 1.0 in the center to 0.0 on the edge.



```
scene Front_facing ()
```

This simple shader is very useful in visualizing the structure of a geometric model. Like the `Show_normals` shader, it provides a good visual description of the orientation of the surface.



```
scene Front_facing ()
```

Scaling a color value based on surface orientation will be the basis of the direct light simulation shaders of the next chapter. Those shaders may need to calculate the dot product directly using the `dot` library function. An equivalent shader to `Front_facing` that explicitly calls the `dot` function can use the `normal` and `direction` state variables for the `dot` function's arguments.

```
1  shader Front_facing_v2 {
2    output:
3      Color result;
4    member:
5      void main() {
6          result = -dot(normal, direction);
7      }
8  };
```

Using the `dot` function, a similar shader can visualize the geometric normals in the same way.

```
1  shader Front_facing_geometry {
2    output:
3      Color result;
4    member:
5      void main() {
6          result = -dot(geometry_normal, direction);
7      }
8  };
```



```
scene Front_facing_geometry ()
```

Mathematical functions like the dot product depend upon the relative geometric relationship between vectors and therefore do not require transformations to a different coordinate space. The relationship of the vectors — not their absolute coordinate values — determines the result of the function.

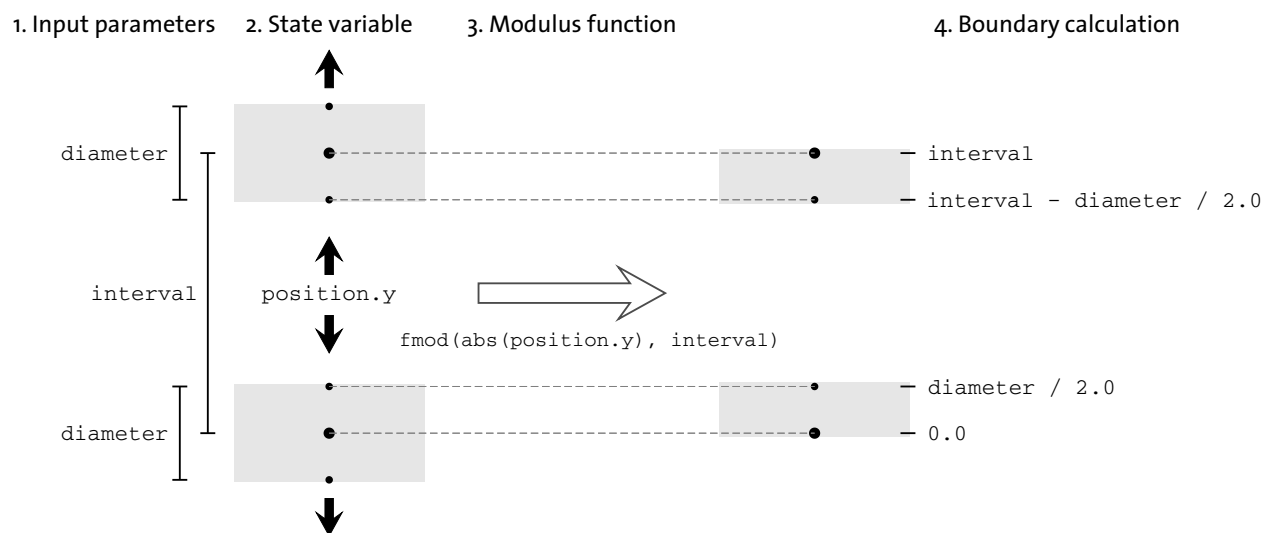
4.5 State variables and shader design

Shader design begins with a visual goal, some effect that should be produced in the rendered image. The initial design questions are simple: What is the problem to be solved? What do I initially have as data to use for my solution? What can I do with the data to produce that solution?

Many shaders have four phases of definition and calculation that further refine these general questions.

1. *Input parameters* — What controls for the shader’s calculations should be provided to the users of the shader?
2. *Available state variables* — What state variables are available for this shader type and are required for the shader’s calculations?
3. *Transformation of state* — How should the state be transformed for the calculations of the shader?
4. *Calculation of result* — How is the final result calculated?

It may often be helpful to accompany these questions with a geometric diagram to help clarify the shader’s required parameters and intermediate results. For example, a surface shader can draw lines at a consistent height throughout a scene, similar to “contour lines” that display elevation in a map. Input parameter control the spacing between lines, called the *interval*, and the *diameter* of the lines. The placement of the lines is dependent upon the *y* component of the *position* state variable, modified by the `abs` and `fmod` library functions. The resulting values are then limited to a range of 0.0 to the *interval* value. Defining the *radius* to be half the diameter, then a sample is part of the line to be drawn if it is less than the radius distance from 0.0 or from the *interval* value. By splitting line drawing in half this way, the lines are always centered around even multiples of *interval*.



Visualizing the geometry of drawing lines based on position.y

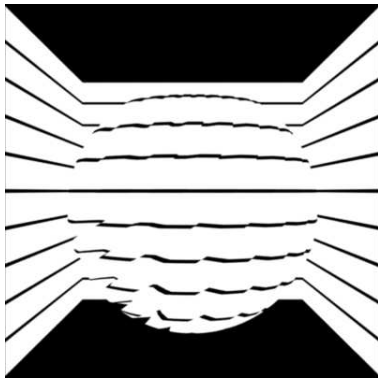
The initial implementation of a shader — especially for beginning shader programmers — can use local variables to describe all of the intermediate steps in its calculations, making the algorithm implemented by the code explicit in the code itself.

```

1  shader Height_lines {
2      input:
3          float diameter = 0.01;
4          float interval = 0.1;
5          Color base_color = Color(1,1,1,1);
6          Color line_color = Color(0,0,0,1);
7      output:
8          Color result;
9      member:
10     void main() {
11         float3 world_position =
12             transform_point("internal", "world", position);
13         float y_mod = fmod(abs(world_position.y), interval);
14         float radius = diameter / 2.0;
15         float lower_boundary = radius;
16         float upper_boundary = interval - radius;
17         bool in_lower_region = y_mod < lower_boundary;
18         bool in_upper_region = y_mod > upper_boundary;
19         result = (in_lower_region || in_upper_region) ? line_color : base_color;
20     }
21 };

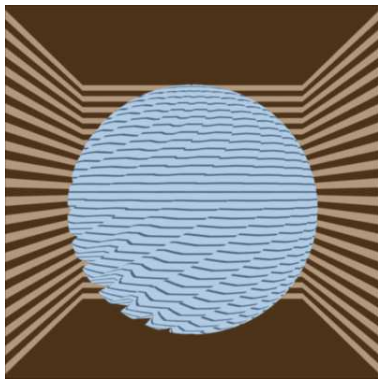
```

Rendering with the default values for `Height_lines` draws black lines on white, with lines of diameter 0.01 separated by 0.1. Note that bottom and top sides of the box are complete drawn in the line color, since all points of those triangles are within the line-drawing region.



```
scene Height_lines ( )
```

This rendering defines explicit values for all shader parameters.



```

ball
    Height_lines (
        base_color: Color(0.7, 0.8, 0.9, 1.0),
        line_color: Color(0.3, 0.4, 0.5, 1.0),
        diameter: 0.005,
        interval: 0.025 )

box
    Height_lines (
        base_color: Color(0.7, 0.6, 0.5, 1.0),
        line_color: Color(0.3, 0.2, 0.1, 1.0),
        diameter: 0.025,
        interval: 0.05 )

```

An equivalent shader to `Height_lines` might only use local variables as a way of avoiding redundant calculation; `y_mod` and `radius` are used twice in the final result, so they should be calculated only once.

```

1  shader Height_lines_v2 {
2      input:
3          float diameter = 0.01;
4          float interval = 0.1;
5          Color base_color = Color(1,1,1,1);
6          Color line_color = Color(0,0,0,1);
7      output:
8          Color result;
9      member:
10         void main() {
11             float3 world_position =
12                 transform_point("internal", "world", position);
13             float y_mod = fmod(abs(world_position.y), interval);
14             float radius = diameter / 2.0;
15             result = (y_mod < radius || y_mod > interval - radius) ?
16                 line_color :
17                 base_color;
18         }
19     };

```

In this simple shader implementation, the shorter form still makes the separation of the result into the two regions based on `interval` and `diameter` relatively clear. However, the shader programmer should be careful about optimizing execution performance through brevity of expression. A compiler will often do a good job of making redundant code more efficient, but it can be quite difficult to understand and correct code that has been written without the (human) reader in mind.