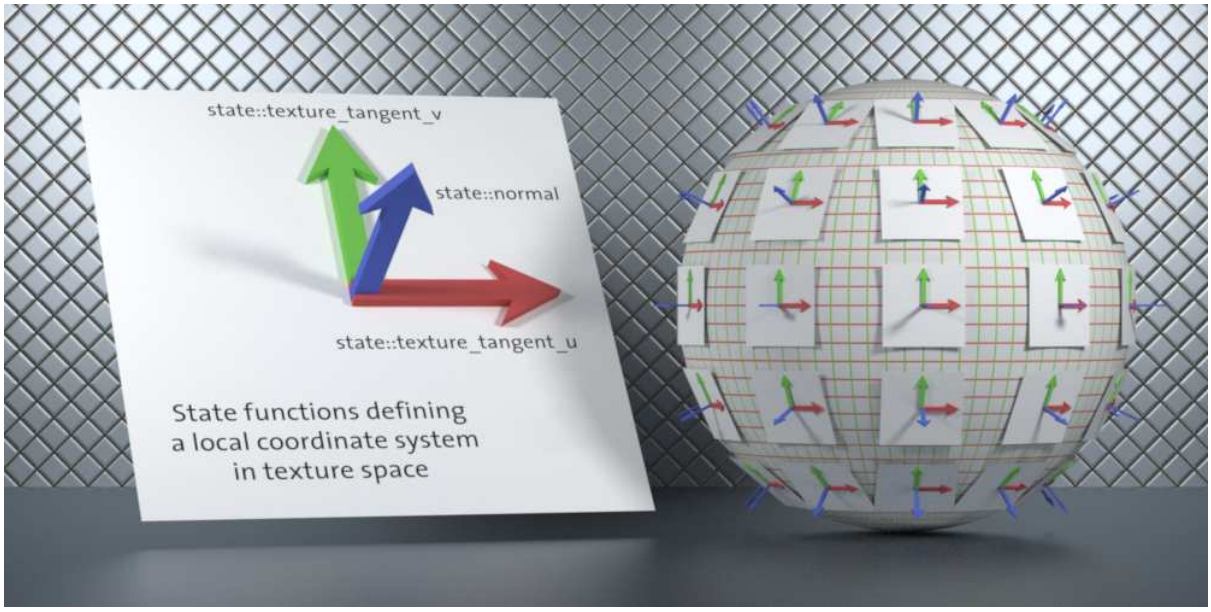
Material Definition Language

Handbook

5 May 2021
Version 1.5



WORK IN PROGRESS — `http://www.mdlhandbook.com`

NVIDIA

Material Definition Language – Handbook

Text, diagrams, scene design and rendering	Andy Kopra
Material design	Andy Kopra, Jan Jordan
Editorial review	Mike Blake, Jan Jordan, Lutz Kettner, Daniel Seibert
MDL Product Manager	Jan Jordan
MDL Specification Lead	Lutz Kettner
Socrates head model	clay master [a]
Ganesha model	Hane3D [b]
Chinese Buddha model	Giimann [c]
Toy train model	Vachagan [d]
Fabric model	Jan Jordan

a. http://www.turbosquid.com/Search/Artists/clay-master
b. http://www.turbosquid.com/Search/Artists/Hane3D
c. http://www.turbosquid.com/Search/Artists/Giimann
d. http://www.turbosquid.com/Search/Artists/Vachagan

Copyright Information

Document build number 345460

Contents

Preface

> ...the brain of the massive (about two tons) stegosaur weighed only about 70 grams, or 2.5 ounces.... By contrast, even the brain of the sheep—which is not a particularly brilliant animal—weighs about 130 grams, greater both in absolute size and even more so relatively to body size.... So far as strength is concerned nothing could stop one of the great dinosaurs when it was on its way; but while it is all very well to be able to go where you are going, the reasons for going and what is seen and understood on the way are even more important.
>
> Weston La Barre, *The Human Animal* (1954), pp.24-25. Quoted in S. I. Hayakawa, *Language in Thought and Action*, 1941; fifth edition, 1991.

The Material Definition Language (MDL) specification is the primary resource for programmers writing an MDL compiler or integrating the use of MDL materials into an application. For those goals, the lower-level details of the language ("What is the legal character set?") are critical.

For an artist creating rendered scenes, however, only one question is important: How can MDL be used to achieve a particular look or visual effect? The language details of MDL—its syntax, the meaning of its predefined types, how these types are combined and used by a renderer—are only interesting as they serve the artist's purpose.

However, both system integrators and artists will need to understand the basic principles of MDL as an approach that differs in some fundamental ways from traditional shading languages. And though a graphical interface may assist in the creation of new materials, many artists remain interested in the implementation details of the systems they use, aware of the additional flexibility and creative control such lower-level understanding can provide.

The structure of the *MDL Handbook* is designed to address the interests of these various audiences. Each chapter begins with background information about principles of physics used in that chapter as well as related issues from the history of rendering in computer graphics. The components of MDL used to implement those principles are presented next. Examples of MDL that implement common real-world materials (plastic, glass, fabric) follow descriptions of MDL to demonstrate the use of MDL in a practical implementation.

Given this structure, a reader interested in an overview of the issues needed to understand MDL could read only the "Background" sections in order. On the other hand, a reader already familiar with MDL can look through the examples for implementation ideas, reading only those examples she finds interesting. For a full understanding of MDL, however, the chapters should be read in order, experimenting with the materials in the examples in each chapter using one of the rendering systems that supports MDL.

Part 1 Language structure

1 The design rationale for MDL

This chapter provides an overview of the various goals of Material Definition Language (MDL) as a description of the physics of light as well as a practical tool in creating images with software.

1.1 Background: Describing appearance

The word *rendering* has several meanings, but they all hover around the notion of transformation. In the drawing and painting traditions of many cultures, a scene in the world is represented by the artist using line and color. From the complexity and detail of the world, the artist's reduced set of marks and hues can still convey an impression of the original scene. In this sense, all representational art is actually abstract, in that it discovers the critical features of appearance and abstracts from them the important features for the picture.

Fig. 1.1 – Young Hare
Albrecht Dürer, 1502

Fig. 1.2 – Detail from Magpies and Hare
Ts'ui Po, 1061

Rendering in three-dimensional computer graphics, however, proceeds in the opposite direction. The descriptions of the scene to be rendered consist of mathematical structures represented by programming languages. The rendering process uses these abstracted descriptions

as input; the goal is a picture that creates an image true to the appearance of the world through all the details supplied by rendering.

Fig. 1.3 – Iray Photoreal rendering of an automobile
Thomas Zancker, Simbild

Fig. 1.4 – Iray Photoreal rendering of an automobile
Bunkspeed

The rendering process of computer graphics—from simple description to pictorial complexity—more closely resembles architectural rendering than painting and drawing. As part of the process of design, a rendering in architecture can serve as a visualization of the completed project, transforming the plan, elevation, and lists of building materials into an image of the building, seen from a particular point of view and a given time of day.

Fig. 1.5 – House by APOLLO Architects and Associates
Jonathan Beals; Iray Photoreal rendering

Fig. 1.6 – Entrance of Hvitträsk
Eliel Saarninen; watercolor

1.1.1 The graphics pipeline

The computer graphics rendering process is often compared to a pipeline, in which mathematically defined shapes and descriptions of their appearance serve as inputs to the renderer, which produces a picture as output. The set of objects to be rendered is called the *scene*.

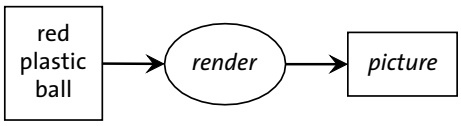


Fig. 1.7 – Object and appearance together

But rather than bundling the color "red" along with the other data that defines the ball, a rendering process could allow color to be specified separately.

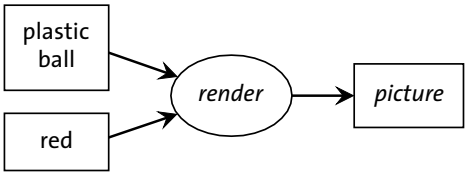


Fig. 1.8 – Separate appearance attribute

But the color attribute is part of the description of the objects as "plastic," so it may be more intuitive to group "red" and "plastic" together as a unit of input for the renderer.

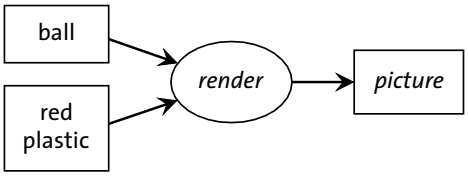


Fig. 1.9 – Separate appearance

The creation of "red plastic" could also be viewed as an input/output process, where "red" is an input to a general description of the appearance of "plastic."

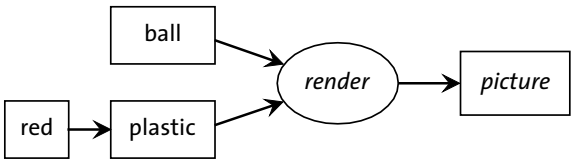


Fig. 1.10 – Attribute as appearance parameter

Now that these three components of the pipeline—the geometric object, the nature of its appearance, and the specific qualities (like color) of that appearance—have been separated, it becomes easier to substitute other objects that can use the same description of appearance.

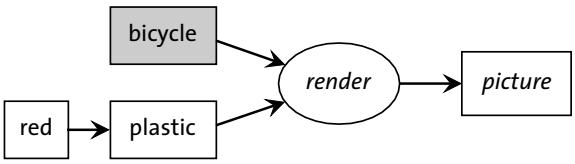


Fig. 1.11 – A different object

This color attribute for the plastic appearance can also be independently changed.

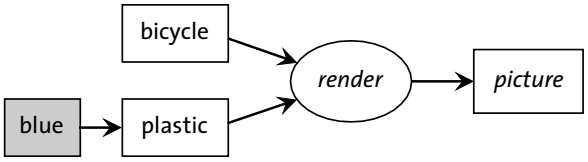


Fig. 1.12 – A different attribute

Furthermore, even the substance (plastic) that was being simulated can be changed independently of the geometric object it describes.

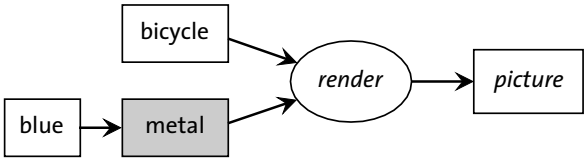


Fig. 1.13 – A different substance

For a rendering system, an object like a bicycle will have some geometric definition for its surface, for example, as a mesh of triangles. To describe an object's generic appearance, the word "material" is typically used in rendering systems, in the sense of something being composed of a particular material or substance. The inputs to the material are similar to the parameters of a function in a programming language.

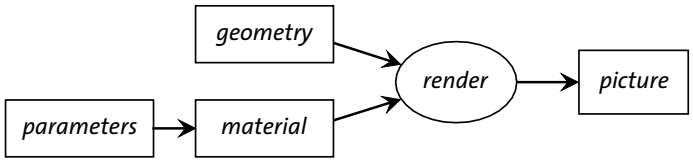


Fig. 1.14 – The general categories

The goal of MDL is to provide a way of defining these materials and their parameters that the rendering system uses to determine the appearance of an object in the resulting picture.

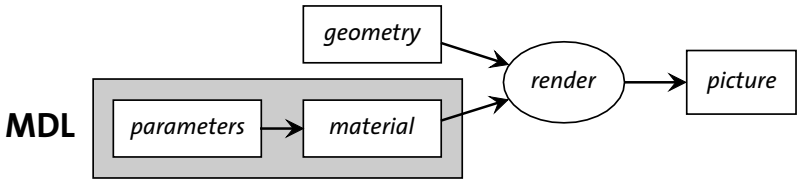


Fig. 1.15 – The domain of MDL

1.2 Basic principles of MDL

A *high-level programming language* can describe a computational process in terms that more closely resemble how the programmer conceptualizes the problem to be solved rather than the actual instructions the computer executes. The important work of the programmer shifts from describing how to implement the machinery of the solution to creating a productive conceptual model of the problem to be solved. When the model has been designed well, problem solving during development—as well as later extensions to the original design—are simplified by the intuitive understanding of the problem that the model provides.

In an analogous manner, MDL is designed as a *high-level description of appearance*. Rather than creating a series of instructions for the rendering system to execute in order to achieve a particular visual effect, an MDL *material* describes the physical characteristics of the object that will produce that effect.

To create an intuitive model of appearance, MDL relies on two common-sense ideas that determine how things look.

1.2.1 Idea #1: Light is only reflected, transmitted and emitted.

If an object is visible, then some light from that object must have traveled to the observer, or, in the case of a photograph, to the camera's film or sensor array. What is the source of that light? It can only have been reflected by the object, traveled through it, or been generated by the object itself.

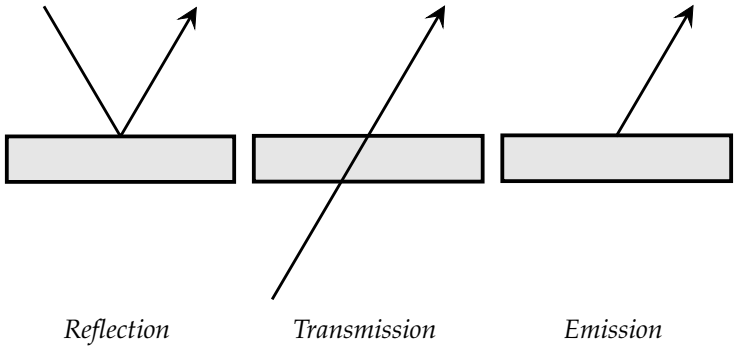


Figure 1.16

The terminology and structure of MDL can be understood as a series of refinements of this small bit of informal (and obvious!) physics.

Though the word "reflection" is typically used to describe an image in a mirror, MDL uses this word to describe any action of light that bounces off a surface. A reflection occurs at the *boundary* of the object and its environment. Similarly, light passing through an object will be affected at the boundary in a variety of ways. For example, light rays are bent, or *refracted*, as they cross the boundary between air and glass. If a glass pitcher contains water, then light passing through both the glass and the water will be bent differently at each boundary: air, glass, water, glass, air.

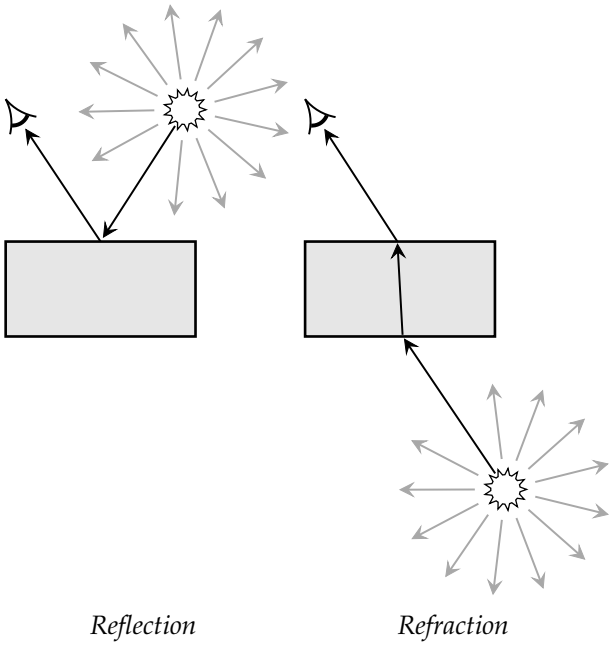


Figure 1.17

Light emitted by an object is considered by MDL to be a phenomenon that occurs at the object's boundary.

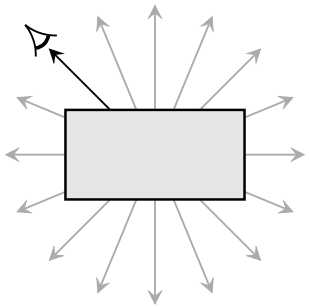


Fig. 1.18 – Emission

However, light that passes through some substances does not pass through it in a straight line. If a small amount of milk is added to the water, then the very small particles in the milk scatter and absorb the light. Light is still refracted when it enters the volume of milky water, but very quickly encounters the particles that affect it through scattering and absorption.

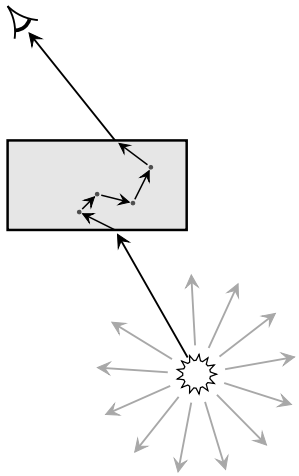


Fig. 1.19 – Volume effects

These four types of light and object interaction—reflection, transmission, emission and volume effects—form the basis for the software structure of MDL.

1.2.2 Idea #2: Angles matter.

In a mirror, the angle from which the light strikes the mirror (the *incident angle*) is the same as the angle of reflection. In MDL, a perfectly reflective surface is called *specular*.

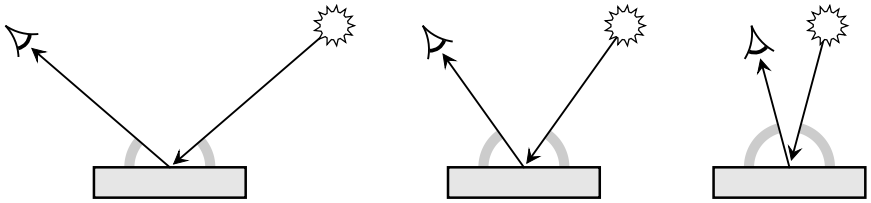


Fig. 1.20 – A perfectly specular surface

In the opposite case to specular reflection, light striking a surface that is then distributed evenly in all directions from that surface is called *perfectly diffuse*.

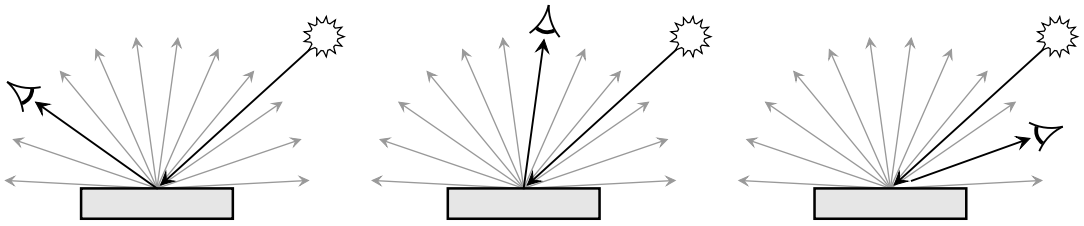


Fig. 1.21 – A perfectly diffuse surface

In the previous diagrams, arrows were used informally only to describe direction. If the length of an arrow also represents the amount of light leaving a point in that direction, then the diagram can describe other types of reflection besides the idealized specular and diffuse.

For example, a rough surface which reflects light more in some directions than others will create reflections that appear to be blurred. The more the light spreads after striking the surface, the blurrier the reflection will appear to be. In MDL, this kind of reflection is called *glossy*.

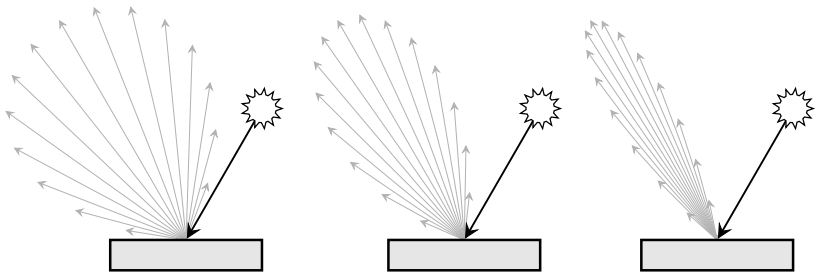


Fig. 1.22 – Surfaces of increasing roughness resulting in decreasing glossiness

Imagining all the directions in which light is reflected from a surface and the respective brightness of each direction (the length of the arrow), a shape is defined that represents the distribution of light from a point, given a certain angle of the light hitting that point.

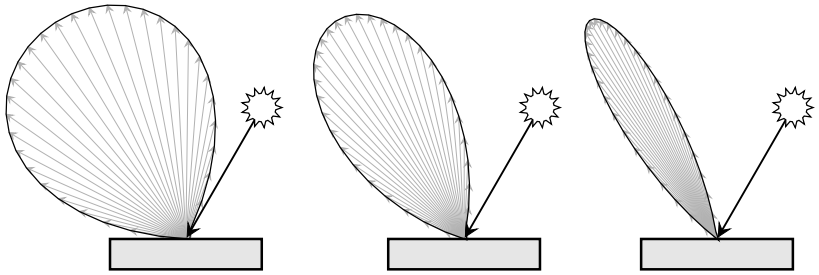


Fig. 1.23 – Examples of the distribution function

Because light reflected from a rough surface is not evenly distributed, the viewing angle will affect how bright the surface appears to be.

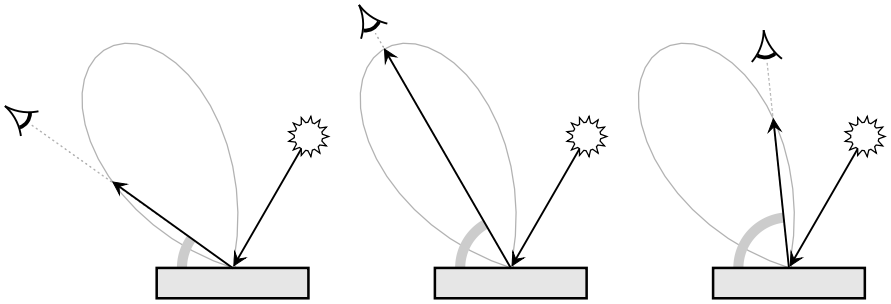


Fig. 1.24 – Varying brightness based on the angle of view with a constant light angle

The way that light is reflected from a surface can be dependent upon the angle of the light that strikes that surface, called the *incident angle*. For some materials, no light will be reflected in the viewing direction for a particular range of incident angles.

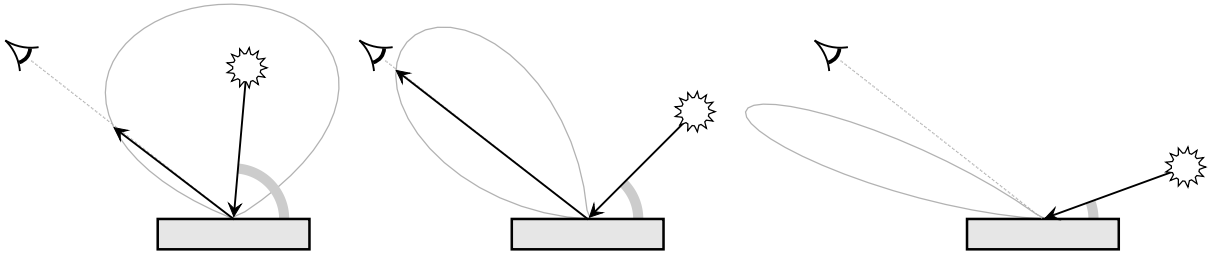


Fig. 1.25 – Varying brightness based on the incident angle of light with a constant view angle

Both the angle of view and the incident angle of the light affect how the light is perceived after striking the surface. A functional relationship exists between these angles and how the light is distributed. The mathematical definition of this relationship is called a *reflectance distribution function*. These two angles also define *directions*—the direction from which a point is being viewed and the direction of the incoming light. This emphasis on direction gives this function the name used for it in the technical literature, the *bi-directional reflectance distribution function*, or BRDF.

However, the behavior of light entering an object can also be described mathematically in the same way. For example, a surface can have a "glossy" transmission that will create a blurry image seen from the opposite side, like frosted Plexiglas. Because light is being transmitted through the object rather than reflected from it, the corresponding distribution function defines the effect of the viewing and incident angles on transmission. Corresponding to the BRDF, this function is called the *BTDF*, the *bi-directional transmission distribution function*.

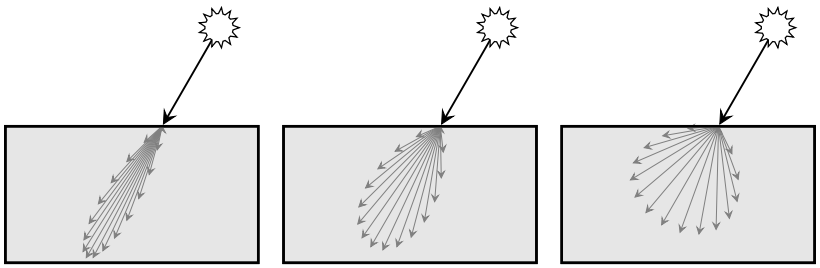


Fig. 1.26 – The effect of increasing surface roughness on transmission

Because of the similarity of the formal properties of the BRDF and the BTDF, MDL considers these two functions as special cases of a function that describes the scattering of light in general. This function is called the *bi-directional scattering distribution function*, or *BSDF*. The BSDF is the key component of MDL that defines the boundary interactions of objects and light.

1.2.3 The orientation of the surface

In all the previous diagrams, the reflecting surface was assumed to be perfectly flat, so that angles could be measured from the surface itself. To define angles for surfaces of any shape, a direction that is 90° from the surface is used as a reference. This direction is called the *normal vector*.

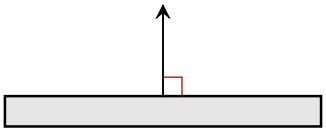


Fig. 1.27 – A normal vector forms a 90-degree angle to the surface

To determine the normal vector at a point on a curved surface, the *tangent* at that point is used as the reference for a virtual "flat" surface at that point. The normal vector can then be used as the frame of reference for the viewing and lighting angles at that point.

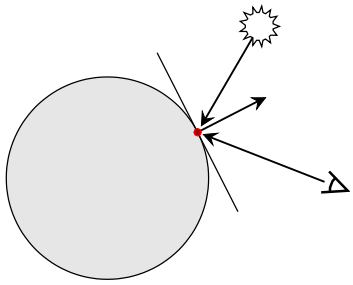


Fig. 1.28 – The normal vector creates a frame of reference for the viewing and lighting angles

For a circle, the tangent at a point is the line that only shares that single point with the circle. For a sphere, the tangent is similarly defined as the plane that only shares that single point with the sphere. This idea of the tangent can be extended to any line or surface.

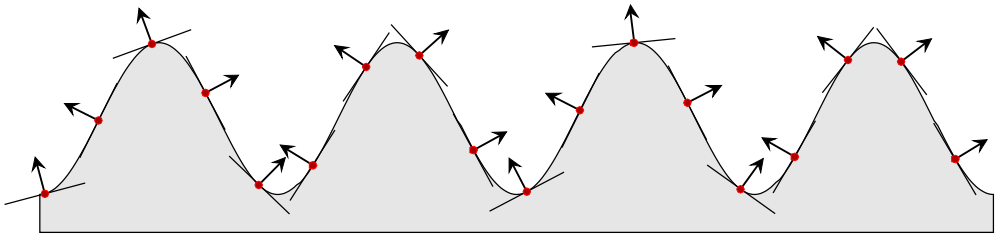


Fig. 1.29 – The normal vector describes the orientation of a surface at any point

For an object in the scene to be rendered using MDL's description of its appearance, the normal vector provides a frame of reference anywhere on the surface. The objects in the scene are defined mathematically—for example, as a set of connected triangles—so that the rendering system can determine the tangent and normal and provide it to the calculations of appearance that MDL defines.

1.3 Appearance in the world

Are the two primary ideas of MDL—the four interactions of light (reflection, transmission, emission, and volume effects) and the importance of angles—sufficient to describe the appearance of objects in the real world? Combining the three surface interactions (reflection, transmission and emission) and the three types of light scattering (diffuse, glossy, and specular), produces nine possibilities. What kinds of objects in the world demonstrate these properties?

Real-world objects usually do not fall so cleanly into one of these nine categories, but often display complex combinations of elemental aspects of appearance. MDL implements such combinations by a software structure designed as a collection of simpler parts. These parts can be used independently or in combination. An intuitive analysis of polished quartz as having both a volume scattering and absorption effect as well as a surface reflection effect is implemented in just that way. This also includes light emission as a candidate for such a combination, for example, a flame in a glass lantern filled with smoke. A material can also be combined with other materials to simulate blending and layering of effects. For example, a material that produces the effect of varnish could be layered over a material that simulates wood grain. As later chapters will show, designing materials in this way also provides flexibility and allows reuse of previously designed materials, so that the varnish material could also be used over a material that simulates oil paint.

The physical composition of an object is not the only factor that determines its appearance. Polishing marble produces a mirror-like finish that does not typically occur in nature, just

as the opacity of frosted glass is only due to the very small-scale roughness of its surface. The geometric structure of a surface—not just the molecular constitution of the object—also contributes to its appearance.

But very small geometric structures may be difficult to construct using triangles or other geometric primitives used in the modeling process. MDL also provides a means to modify the normal vector to simulate a more complex surface than the underlying "real" geometric definition. In Bump mapping: perturbing a surface normal (page 211) this technique is used to create the appearance of a finely woven surface. The geometric detail of the weave is represented by an image that specifies how the normals should be modified, a traditional technique known as *bump mapping*.

Modification of the surface structure itself can be defined by MDL through a technique called *displacement*, used in "Geometric profiles" (page 241) and "Architectural details" (page 269). Displacement adds geometric data to an object—adding triangles to a triangle mesh, for example—rather than only modifying the description of surface orientation that the normal vector represents. These manipulations of the geometric proprieties of objects are defined in MDL in the same way as the attributes of appearance and complement the physically based descriptions of the distribution functions.

1.4 The spirit of MDL

Learning to create materials in MDL is primarily the process of understanding the purpose and use of the three major categories contained within an MDL material: light interaction at a boundary, light interaction within a boundary, and the manipulation of geometry, combined to achieve effects that are significant in the rendering process. Reflection, transmission and emission are furthermore defined as subcategories of surface interaction.

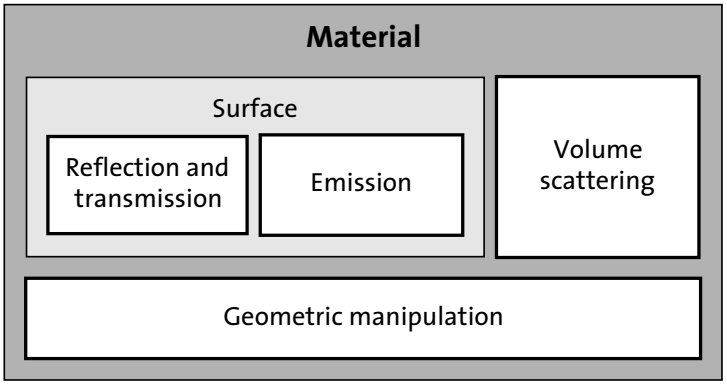


Fig. 1.30 – The main categories of an MDL material

Based on this mental model, here is the definition of an MDL material that will be explored throughout the rest of this book:

> A material is a grouping of rendering properties based on the physics of light as well as on practical and traditional techniques that extend the power of the renderer.

The structure within which that these elemental units and their combinations are specified in the language of MDL is the subject of the next chapter.

2 The structure of a material

This chapter describes the primary syntactic structures of MDL and describes how those structures organize the physical properties of light interaction and emission.

2.1 Background: Form and meaning in language

Linguists typically divide the investigation of language into different fields of study. A single sentence, for example, can be analyzed in several ways—by its structure (syntax), by its meaning (semantics), or by its role in a given context (pragmatics). Programming languages can also be analyzed in this way; it's possible to describe the structure of a program without regard to what the program actually does. Grammatical concepts, like nouns and verbs, can also be very useful ways to think about the components of a programming language—the data as nouns; the actions the program performs as verbs.

But programming languages differ from natural (human) languages in two major ways. First, programming languages are not a matter of human instinct. They are designed and created for a purpose with particular tasks in mind—the language's "application domain." One of the oldest languages still in use, COBOL, makes this clear even from its name, an acronym for "common business-oriented language." Two other of the oldest languages, FORTRAN and LISP, are named in a similar manner, as contractions of "formula translation" and "list processing." Though natural languages also vary in their structural qualities—rhyming is easier in Italian than English; German grammar provides greater flexibility in word order than Chinese—programming languages are consciously designed to favor particular modes of expression. Over time, both natural and programming languages evolve, but once again, this evolution of programming languages is a matter of conscious debate and design.

Second, in natural languages, speaking and understanding speech have evolved into fundamental human capabilities. Writing and reading, on the other hand, are cultural artifacts that may never develop in a given society. However, programming languages only exist in "written" form, a "text" in the language represented as digital data that will be transformed into "instructions" to control the operation of the computer hardware. This fundamental purpose of a programming language—as instructions to a machine—requires precision from the user of that language; the assumptions, interpolations and ambiguity of a natural language can only be sources of error in software.

2.1.1 Types of data

In counting chickens or measuring teaspoons of salt, the difference between "how many" and "how much" are matters of common sense. But "how many" can become "how much" at the butcher shop, and a single salt crystal may be important to a chemist. In day-to-day language, understanding is guided by context to clarify how a number is being used and to what it refers—to length, to weight, to time.

In mathematics, the difference between types of numbers is defined in a formal way. The phrase *natural number* is the formal name for the "counting numbers" from grade school. Combining the natural numbers, zero and the negative of each natural number produces the *inte-*

gers. Numbers that can be expressed as a ratio of two integers are *rational*; numbers that can't be (like π, the ratio of the circumference of a circle to its diameter) are *irrational*.

For many programming languages, the way different types of numbers are represented is the basis of a system of *data types*. These types may be based on categories of mathematics (for example, integers and ratios), but may also take the hardware into account, differentiating between "integer" subtypes that can represent different ranges of the mathematically infinite integers. Other types in a language may represent *characters* (letters, numbers and punctuation) or small sets of related but unique values, like "true" and "false" that have special meaning in the language.

These types represent a single, or *atomic*, value—a number, a character, a truth-value. A *compound* type consists of a set of types, and is a natural way to represent common mathematical ideas, like a point on a plane with an x and y coordinate, or concepts from an application domain, such as a color in computer graphics defined by three numeric values for its red, green, and blue components. If a language allows a compound type to include compound types as well as atomic types, arbitrarily complex types can be defined in a consistent manner, enabling the language to create types that best represent the application domain.

2.1.2 The value of a type

Geometrically, a circle is defined as all the points that are the same distance from a single point. To define a "circle" data type for a drawing program, for example, at least three numbers are therefore required: the two geometric coordinates for the center (*cx* and *cy*) and the *radius*. This compound type (of three numeric atomic types) can be represented by a table of the data type of each component and its name:

Data type: `circle`	
`number`	`cx`
`number`	`cy`
`number`	`radius`

To create a "real" value—to define data that can be used as part of the computation described by software—the components of the circle (cx, cy, and radius) are given numeric values. This can be depicted in a similar table that specifies the name of the component and its value:

A `circle` instance: `circle_1`	
`cx`	`0`
`cy`	`0`
`radius`	`5`

The creation of data of a given type is called *instantiation*, and the datum is called an *instance* of the type. This corresponds to the common-sense use of the word "instance" to describe a realization or example of some general idea or category—"an instance of the painter's earlier work." In the previous diagram, the name of the instance of the `circle` type is `circle_1`.

Defining a data type for color is a similar process. One traditional representation of color in computer graphics specifies the amount of the red, green and blue values. These can be the components of the `color` data type:

Data type: `color`	
`number`	`red`
`number`	`green`
`number`	`blue`

In this simple implementation of color data, the red, green and blue values vary from zero to one, where zero represents no contribution of that component, and one represents the maximum contribution. An instance of the `color` type that creates a dark red color could be defined like this:

A `color` instance: `dark_red`	
`red`	`0.5`
`green`	`0.1`
`blue`	`0.2`

Both the `circle` and `color` data types are compound types consisting of three atomic values of type `number`. A compound type can also contain other compound types. For example, a drawing program could specify an individual color for each circle.

Data type: `tinted_circle`	
`number`	`cx`
`number`	`cy`
`number`	`radius`
`color`	`tint`

The `tint` component of the `tinted_circle` type has type `color`. When an instance of type `tinted_circle` is created, the `tint` component will then be an instance of type `color`.

<table>
<tr><th colspan="3">A <code>tinted_circle</code> instance: <code>dark_red_circle</code></th></tr>
<tr><td><code>cx</code></td><td colspan="2">0</td></tr>
<tr><td><code>cy</code></td><td colspan="2">0</td></tr>
<tr><td><code>radius</code></td><td colspan="2">5</td></tr>
<tr><td rowspan="4"><code>tint</code></td><td colspan="2">A <code>color</code> instance</td></tr>
<tr><td><code>red</code></td><td>0.5</td></tr>
<tr><td><code>green</code></td><td>0.1</td></tr>
<tr><td><code>blue</code></td><td>0.2</td></tr>
</table>

To define and organize the properties of appearance, MDL uses a compound structure called a “struct” that is similar to the examples described above. MDL structs can contain simple numeric values, but also contain instances of data types that represent the distribution functions of the previous chapter. The type of some MDL structs are themselves structs, implementing a hierarchical organization for MDL data types.

2.2 The syntax of the material type

The main categories of MDL described in the previous chapter are organized within a *compound type* called a *material*. The syntax of the material type is similar in principle to the "struct" type of several other programming languages, including C and C++. The material type, in turn, contains other compound types, which may themselves contain compound types. For clarity, the basic syntax of MDL will first be presented through a set of simple examples, followed by a description of how that syntax is actually used in MDL.

2.2.1 The struct type

A struct in MDL treats a set of values as a single data type. The struct is defined by a set of names and types for these values. As in C and many other languages, a primary numeric type in MDL is the `float` (so called because the decimal point can "float" to different positions within a sequence of digits). For a minimal definition of a circle, three values of type `float` could be specified: the x and y coordinates of the center and the length of the radius. In the following example of MDL syntax, this new struct type is named `circle`:

```
struct circle {
    float cx;
    float cy;
    float radius;
};
```

An abstract representation of an MDL struct will help to describe its syntax in a general way. In the following, *italic font* is used for descriptive names while `fixed font` is used for the parts of the struct that remain the same for all types. Three dots (`...`) means that the previous line can be repeated any number of times.

```
struct struct-name {
    type field-name ;
    ...
};
```

To provide actual values to a struct and thereby define data for use in MDL, the *constructor* of the type is executed. A constructor resembles a function call, with the type name used as the function name, and a set of values that provides the values of the struct's fields. If there is more than one value, the values are separated by commas.

```
struct-name ( field-value, ... )
```

The result of using a constructor is the creation of an *instance* of the type. For example, the following creates a circle centered at (0,0) with a radius of five (in which `0`, `0`, and `5`, separated by commas, specify the values of cx, cy, and the radius, respectively):

```
circle(0, 0, 5)
```

As another example, a typical implementation of a simple color type would contain fields for the red, green and blue components of the color:

```
struct color {
    float red;
    float green;
    float blue;
};
```


The constructor for the `color` type to create a dark red color instance provides a list of the red, green and blue values for the desired color:

```
color(0.5, 0.1, 0.2)
```

In certain contexts in MDL, an instance of struct can be used as the value of a *variable*, a name that can be used later to refer to that instance. The syntax of variable creation is the same as in C and C++:

data-type variable-name = *variable-value* ;

For example, this statement would create a variable from the previous instance of struct `color`, calling it `dark_red`:

```
color dark_red = color(0.5, 0.1, 0.2);
```

2.2.2 Default values for fields in a struct

For the fields of complex structs it is useful to define a *default value* for each field. When the constructor creates a new struct value, any field without an explicit value is assigned the default value.

To define a struct with default values for its fields in MDL, a set of *name/value* pairs is specified. Instead of simply defining the name of a field, the name is followed by its default value, separated with an equals sign (=):

```
struct struct-name {
    type field-name = default-value ;
    ...
};
```

For example, this revised definition of the `circle` type defines default values of 0 for cx and cy (the circle is centered at the origin) with a radius of 1.0 (a unit circle):

```
struct circle {
    float cx = 0.0;
    float cy = 0.0;
    float radius = 1.0;
};
```

With default values for the fields, values for individual fields can be specified in the constructor; the other fields in the instance are set to their default values. To specify field values, another name/value syntax is used in which the colon character (:) separates the field name and its value:

struct-name (*field-name*: *field-value*)

If there are multiple field values, each name/value pair is separated by a comma:

```
struct-name (
    field-name-1: field-value-1,
    field-name-2: field-value-2,
    ... )
```

For example, this instance of the `circle` type is centered at the origin (by virtue of the default field values for `cx` and `cy`) but has a radius of 5.0:

```
circle(radius: 5.0)
```

To explicitly specify the `cx` and `cy` values, the field values are separated by commas. Because the field is specified by name in the constructor, the fields can be given in any order.

```
circle(radius: 5.0, cx: 5, cy: 15)
```

For these simple examples, the utility of providing default values for fields in a struct seems limited. However, for more complex structs, default fields values can be used to define a default setting for the struct itself, so that an instance created without specifying any field values still creates a "reasonable" instance for the intended use of that data. Looking ahead, MDL materials make extensive use of default field values for this purpose, so that creating a material without specifying any field values still creates an instance with intuitively reasonable behavior during rendering.

2.2.3 Compound structs

A field in a struct may be a struct itself. For example, to create a more complex version of the `circle` type in the previous section, a field could be added to specify the color of the circle. For example, a field named `tint` of type `color` can be added to the `circle` type, defining a new type that is called `tinted_circle`:

```
struct tinted_circle {
    float cx = 0.0;
    float cy = 0.0;
    float radius = 1.0;
    color tint = color(
        0.5, 0.5, 0.5);
};
```

Notice that the default value of field `tint` is an instance of the `color` type, created by calling the `color` constructor with values of 0.5, 0.5 and 0.5. Even in this simple example, design considerations appear—is gray an appropriate default value for the `tinted_circle` type? Default values for a type are typically chosen with the intended use of the type in an application domain, as will be evident in the more complex types of MDL.

Creating an instance of `tinted_circle` is the same in principle as the simpler circle type. For example, to create a unit circle that is centered at the origin the default field values are sufficient, so that creating a red circle with those properties only requires specifying a value for the `tint` field:

```
tinted_circle(tint: color( 1, 0, 0))
```

Specifying values for all fields does not require that the names be used; the values are associated with the fields by position. The `color` constructor is used for the fourth value to create the correct type for the `tint` field.

```
tinted_circle(2, 4, 10, color(0.5, 0.1, 0.2));
```

As the definitions of structs grow in size and complexity (with nested structs), it can be useful to use field names and multiline layout with meaningful indentation to help clarify the contents of the struct:


```
tinted_circle (cx: 2,
               cy: 4,
               radius: 5,
               tint: color(red: 0.5,
                           green: 0.1,
                           blue: 0.2 ));
```

As in many programming languages, the way that code is formatted on the page can be a source of (surprisingly) intense debate. (Spaces before parentheses? After?) Because multiple space characters are ignored in MDL, the definition of the material type can be formatted in a manner that can clarify the type, name, and default value for each field. As an example, the following format for the previous struct instance will appear clearer to some:

```
tinted_circle (
    cx: 2,
    cy: 4,
    radius: 5,
    tint: color (
        red: 0.5,
        green: 0.1,
        blue: 0.2 ));
```

2.2.4 Accessing struct components with the dot operator

It is sometimes useful to be able to refer to a field value within a struct instance, rather than the entire instance itself. MDL provides the *dot operator* to extract field values from a struct.

The syntax of the dot operator is familiar from many traditional programming languages: the struct values is followed by a dot (`.`), followed by the name of the field to be accessed.

For example, given the previous example of the `circle` struct with default field values, an instance of a `circle` could be used to initialize a variable, named here `upper_circle`:

```
circle upper_circle = circle(0.0, 5.0, 1.0);
```

The field values of `upper_circle` can be extracted and assigned to variables of the appropriate type, in this case, type `float`:

```
float center_y = upper_circle.x;
float center_x = upper_circle.y;
float radius = upper_circle.radius;
```

If a field value that is acquired through the dot operator is itself a struct, then the dot operator could be used again on that value, resulting in a series of field references.

For example, if the variable `dark_red_circle` is declared as an instance of `tinted_circle`:

```
tinted_circle dark_red_circle = tinted_circle(2, 4, 10, color(0.5, 0.1, 0.2));
```

then the red value of `dark_red_circle` could be extracted and saved to a variable of type `float` named `red_component` using the dot operator twice:

```
float red_component = dark_red_circle.tint.red;
```

The dot operator becomes most useful in MDL for extracting parts of existing materials to reuse them in building new materials in a manageable way.

2.3 Design of the MDL material

Given MDL's syntax for nested data structures using the struct type, how can the observations of light's behavior in the first chapter be encoded in MDL? The first requirement is that the definition of a material represent all three possibilities of light reaching the eye from an object's surface: reflection, transmission, and emission:

- Material
- Reflection properties
- Transmission properties
- Emission properties

MDL defines these three basic properties as interactions of light at the surface of an object, so they are combined under a single category:

- Material
  - Surface properties
    - Reflection
    - Transmission
    - Emission

As described in the first chapter, light entering an object may not take a straight and unobstructed path through it. Volume effects are therefore another important component of the material:

- Material
  - Surface properties
    - Reflection
    - Transmission
    - Emission
  - Volume properties

This hierarchical structure forms the physics-based component of an MDL material. But a material is used to describe how a surface should be rendered in a computer program. The "surface" to be rendered is a mathematical description; traditional rendering techniques have included—for efficiency in both specification and computation—geometric manipulations in the rendering phase. To support these techniques, the MDL material adds another component for geometric properties.

- Material
  - Surface properties
    - Reflection
    - Transmission
    - Emission
  - Volume properties
  - Geometric properties

Further practical concerns include how the "back" side of a surface should be rendered ("back-facing"). The basic surface properties are duplicated for the back surface.

- Material
  - Surface properties of front-facing surfaces
    - Reflection
    - Transmission
    - Emission
  - Surface properties of back-facing surfaces
    - Reflection
    - Transmission
    - Emission
  - Volume properties
  - Geometric properties

Two other properties define behavior for the entire material: a value for the index of refraction for light transmission through a surface, and how a surface should be interpreted with respect to volumes. Adding fields for these properties completes a schematic view of MDL's definition of a material:

- Material
  - Surface properties of front-facing surfaces
    - Reflection
    - Transmission
    - Emission
  - Surface properties of back-facing surfaces
    - Reflection
    - Transmission
    - Emission
  - Volume properties
  - Geometric properties
  - Shared properties
    - Index of refraction
    - Surface treated as boundary or volume

These final additions to the outline of an MDL material complement the physics-based descriptions of the first chapter with techniques that extend the power of a material to describe a wider range of appearance characteristics. Simple MDL materials only use a few fields of the struct, so that a full understanding of all material fields is not required until the effect of that field becomes relevant in the rendering.

Fields not specified in a material definition default to values which typically signify that the property the field represents does not contribute to the rendered effect. So, for example, a surface that exclusively reflects light only specifies a reflection property; a surface that exclusively emits light only specifies an emission property. However, the MDL structure as a whole represents all possible ways that light could be seen on a surface, allowing the definition of complex combined effects such as a shiny crystal ball filled with glowing smoke, or, more practically, a fluorescent tube that both reflects and emits light.

2.3.1 The definition of the material struct

The previous section described the set of properties that define appearance in MDL. How is this set implemented in MDL syntax? The MDL *material struct* contains six fields that define these properties as data types in MDL. These fields contain instances of the distribution functions (BSDFs, EDFs, VDFs) as well as other parameters.

Field name	*Description*	*Data type*
`thin_walled`	Distinguishes between boundaries and two-sided objects	`bool`
`surface`	Defines interactions of light and surface	`material_surface`
`backface`	Defines interactions of light with the "back" of a surface	`material_surface`
`ior`	Defines the index of refraction for refracting objects	`color`
`volume`	Defines interaction of light in a volume	`material_volume`
`geometry`	Defines render-time geometric modifications	`material_geometry`

The six fields of the material struct

These six fields are defined in the material struct using the syntax described above in Section 2.2.3 on page 22). Each field in the material struct specifies a default value for the field, and so consists of three parts: the *data type* of the field, the *field name*, and the *default value* of the field that follows an equals sign (=).

Listing 2.1

```
struct material {
    uniform bool      thin_walled = false;
    material_surface  surface     = material_surface();
    material_surface  backface    = material_surface();
    uniform color     ior         = color(1.0);
    material_volume   volume      = material_volume();
    material_geometry geometry    = material_geometry();
};
```

The following is a brief overview of the material struct's fields and their data types; later chapters go into greater depth on how the field values affect the resulting appearance of a material.

`thin_walled`
: In MDL, the distinction between a boundary and a two-sided object is specified by the first field of the material struct, `thin_walled`. The `thin_walled` field has a type of `bool` (for "Boolean"), which can have a value of `true` or `false`. The thin-wall property cannot vary for the surface for which material is defined, so the type modifier `uniform` precedes the `bool` type name. The default value of `thin_walled` is `false`, meaning that the surface associated with the material represents a volume boundary. If the value of `thin_walled` is `true`, then the surface is an object without thickness—a membrane that is infinitely thin.

`surface`
: The `surface` field defines the appearance property for both boundary and thin-walled surfaces. The type of the `surface` field is the `material_surface` struct (described in the next chapter).

Listing 2.2

```
struct material_surface {
  bsdf scattering = bsdf();   Bidirectional scattering distribution function
  material_emission emission = material_emission();
};
```

Note that the `material_surface` struct is a compound struct: the second field of `material_surface` is itself a struct of type `material_emission` that defines light emission:

Listing 2.3

```
struct material_emission {
  edf emission = edf();   Light emission distribution function
  color intensity = color(0.0);
  intensity_mode mode = intensity_radiant_exitance;
};
```

For MDL, reflection, transmission and emission are all considered to be interactions of light and a surface, and are therefore all specified within the compound `material_surface` struct.

`backface`

Both boundary and thin-walled surfaces use the `surface` field for their appearance definition. The two sides of the thin-walled surface can be separately specified using the third field of the material struct, `backface`. Like `surface`, the `backface` field is of type `material_surface`. By default, both sides of a thin-walled surface use the value of `surface`; only by also specifying `backface` can the two sides have separate appearance properties. To have an effect in the appearance of an object, the `backface` field requires that the value of `thin_walled` is set to `true`.

`ior`

The `ior` field defines the index of refraction when the surface is treated as a boundary. To have an effect, the `ior` field requires that the value of the `thin_walled` field is `false`. The type of `ior` is `color`, providing for frequency dependent refraction effects. The value of `ior` cannot vary across a surface, so the type is also designated as `uniform`. The `ior` property uses the MDL "color" type that is more complex than in the example in section 2.1.2.

`volume`

The fifth field, `volume`, is a struct of type `material_volume`, and defines how light is absorbed and scattered in a volume. Because it treats the surface as a boundary of the volume into which light is passing, the value of `thin_walled` must be `false` for this field to have an effect.

Listing 2.4

```
struct material_volume {
   vdf scattering = vdf();          Volume distribution function
   color absorption_coefficient = color();
   color scattering_coefficient = color();
};
```

`geometry`
: The final field of the material struct, `geometry`, is a struct of type `material_geometry`, and is MDL's implementation of traditional modeling techniques that can be part of the rendering process. (Geometric manipulations using the `material_geometry` type are described in Part 5 (page 185).)

Listing 2.5

```
struct material_geometry {
   float3 displacement = float3(0.0);     Modification of the surface geometry
   float cutout_opacity = 1.0;            Modification of the surface opacity
   float3 normal = state::normal();       Modification of the surface orientation
};
```

The default values of the property types in the material struct (for `surface`, `backface`, `volume`, and `geometry`) do not specify argument values for their constructors (the parenthesized list of field values is empty). The default values for the material properties have the following effect on the appearance the material will produce:

Material property	*Effect of its default value*
`material_surface`	No reflection or transmission of light
`material_emission`	No emission of light
`material_volume`	No absorption or scattering of light in the volume
`material_geometry`	No manipulation of the surface

Visual effect of the default values of the material properties

Note that these default values implement the idea of "doing nothing" for each category. This implies that creating materials that implement different types of appearance is accomplished by enabling the appropriate properties of the relevant fields. In other words, more complex effects are created by enabling more components of the material struct, for example, a material that defines both reflective and emissive properties.

2.3.2 Combinations of field values

Not all combinations of field values are viable in the material struct. For example, when the value of `thin_walled` is `true`, the value of the `volume` field is ignored. On the other hand, the `backface` field is used when `thin_walled` is set to true. Three main categories are possible based on the settings of `thin_walled` and `backface`:

Value of `thin_walled`	*Status of* `backface`	*Effect*
`false`	Field not set	The surface is a boundary
`true`	Field not set	The surface is a two-sided object; `surface` field used by both sides
`true`	Field set	The surface is a two-sided object; the value of the `surface` field is used for one side, the value of `backface` for the other

Effect of the values of `thin_walled` *and* `backface`

The following diagram shows all the fields that can be set for the three main categories derived from the settings of `thin_walled` and `backface`.

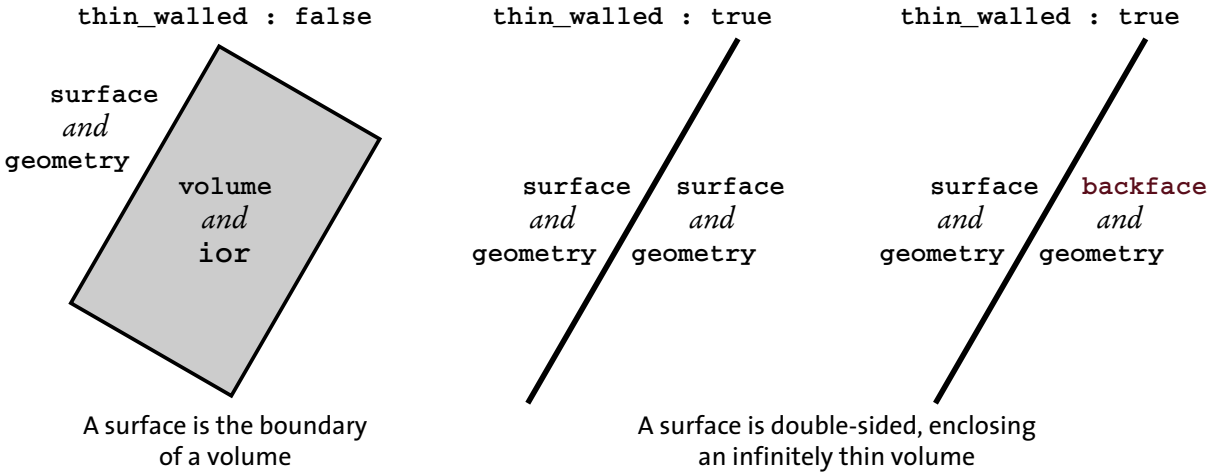
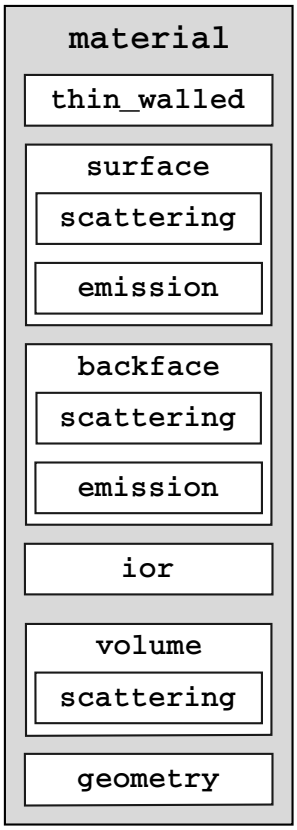


Fig. 2.1 – Permutations of the fields of the material struct

2.3.3 Creating a material

The material struct of Section 2.3.1 (page 25) defines the abstract properties used by MDL to specify appearance. When an an object is rendered using an MDL material, these abstract properties must be made concrete by providing actual attributes for them, for example, specifying a particular hue of red for the abstract type "color." The material struct can be thought of as a container that organizes these values.




```
material (
   thin_walled : boolean-value,
   surface     : material-surface-instance,
   backface    : material-surface-instance,
   ior         : color-instance,
   volume      : material-volume-instance,
   geometry    : material-geometry-instance
)
```

The next chapter uses this container of attributes to define the properties of the fundamental behavior of light described by MDL—the interaction of light with a surface.

Part 2 Light interaction

3 Light at a surface

Now that the previous chapter has presented the syntactic structures of MDL, this chapter can describe the use of MDL materials in the production of imagery by a rendering system. Each end-user rendering application will tailor the presentation of materials to suit the requirements of its interface, though the materials used during the rendering process will all be based on the structural principles described here.

The materials that are presented throughout this handbook can be used in any application that supports MDL and allows the installation of custom materials.

3.1 The simplest material

As section 2.2.1 described, creating an instance of an MDL `material` struct syntactically resembles a function call in a typical programming language. If no arguments are provided for the fields of the struct, the default values are used instead. The simplest possible instance of a `material` struct defines no arguments—the argument list is empty:

```
material()
```

Without arguments, the default values for an instance of the `material` struct (described in section 2.3.1) create a material that does not reflect, transmit, emit, absorb or disperse light. This is hardly a useful material, but can serve as a starting point for a description of MDL syntax.

To use a `material` instance in a rendering system, a name and a list of zero or more parameters are associated with the instance with the equals sign character (=). This association creates a *material definition*.

For example, the following MDL statement creates a material definition named "nil" with no parameters. It uses the default material instance (containing no arguments) described above.

```
material nil() = material();
```

In the terminology of functions in languages like C and C++, the previous statement both declares and defines the `nil` material definition.

declaration `=` *definition* `;`

Typically, however, a material definition is made more complex than this trivial example in two ways:

1. Useful material instances specify argument values.
2. Reusable material definitions provide parameters used by those arguments.

A complete syntax description of a material definition includes those features:

`material` *name*`(`*material-parameters*`) = material(`*material-arguments*`);`

In a manner that is typical for most programming languages, whitespace in MDL (spaces, tabs, newlines) are not significant except as separators between words. MDL code can be divided into separate lines to help convey it structure. Materials developed in the following sections will follow this basic structure:

```
material name (
    material-parameters )
= material (
    material-arguments );
```

The next section describes the arguments of a material instance: the hierarchical structure of material properties and distribution functions.

3.2 A material for diffuse reflection

A rendering system based on a chain of traditional rendering plug-ins that manipulate the rendering process ("shaders") can be presented in a linear way. For example, the simplest shader may simply provide a constant color to use for an object's surface and will serve as a good introduction to shader programming. But even a simple MDL material is a compound structure, consisting of one or more material properties, which in turn contain one or more distribution functions.

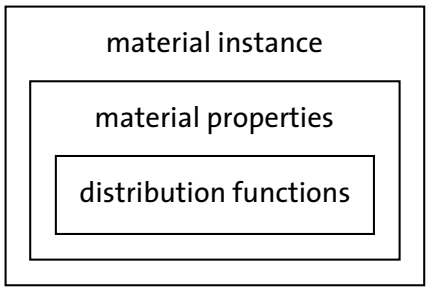


Fig. 3.1 – A model of the nesting of data in an MDL material

Materials can also serve as building blocks to create compound materials, another aspect of hierarchical organization in MDL. The next chapter will describe how these elemental materials can be combined using mixing and layering processes to an arbitrarily complex degree.

This section develops the simplest useful material by describing its hierarchical structure from the top down, from the material definition to the distribution function at its heart. The material reflects light uniformly in all directions, simulating *diffuse reflection*, but does not transmit or emit light. This behavior is a rough simulation of the reflective properties of plaster, so the material definition will be named "plaster".

```
material plaster ()
= material (
    material-arguments );
```

Note that the material has no parameters, so an empty parameter list follows the name `plaster`.

Out of the six fields of the `material` struct, the `plaster` material only needs to specify a value for the `surface` field:

```
material plaster ()
= material (
   surface: surface-property );
```

The surface property for the `surface` field is `material_surface` (described in section 2.3.1):

```
material plaster ()
= material (
   surface: material_surface (
      material-surface-arguments ));
```

The `material_surface` struct contains two field values, `scattering` and `emission`. For the `plaster` material, a value will be specified for just the `scattering` field:

```
material plaster ()
= material (
   surface: material_surface (
      scattering: bsdf-instance ));
```

The surface, volume and emission material properties contain instances of distribution functions. For plaster, the `material_surface` property will use an instance of `diffuse_reflection_bsdf`. The distribution functions are components of the standard `df` MDL module, so the module name precedes the BSDF, separated by two colons (`::`).

```
material plaster ()
= material (
   surface: material_surface (
      scattering: df::diffuse_reflection_bsdf (
         bsdf-arguments )));
```

Distribution function `df::diffuse_reflection_bsdf` has two arguments that control the color and roughness of the surface, named `tint` and `roughness`. For the `plaster` material, a `color` value of 70% gray will be specified for the `tint` argument. By not specifying the `roughness` argument, the default value of zero will be used.

```
material plaster ()
= material (
   surface: material_surface (
      scattering: df::diffuse_reflection_bsdf (
         tint: color(0.7) )));
```

Rendering two objects using this material produces the Figure 3.2:

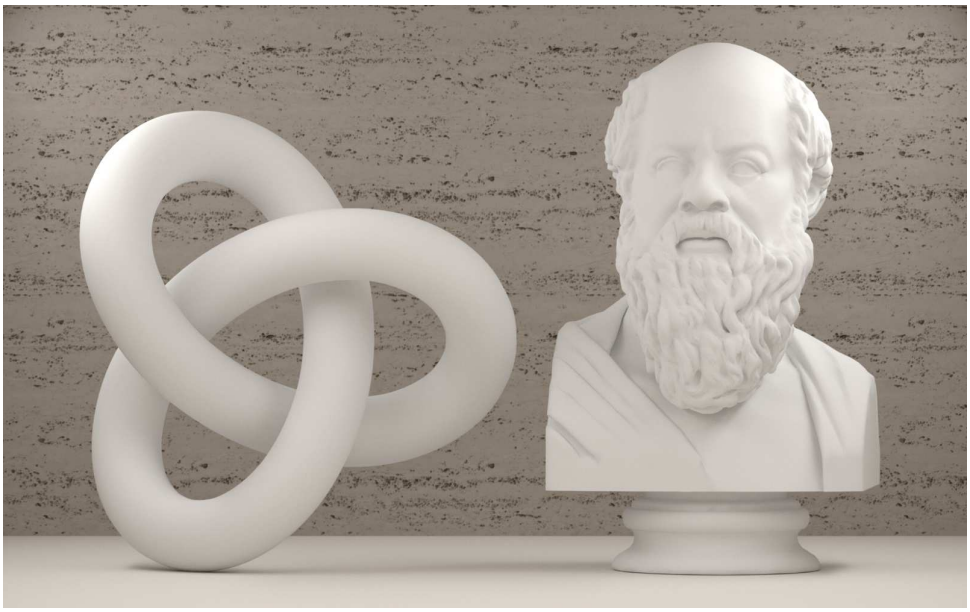
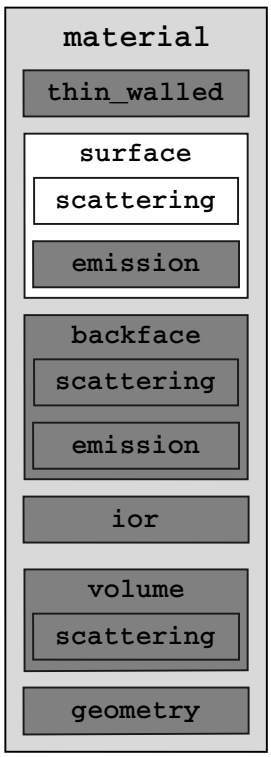


Fig. 3.2 – The appearance of diffuse reflection produced by the `plaster` *material*

For the `plaster` material, only one field value of the `material` struct was specified, `surface`. For the `material_surface` property it contains, only one field value was specified, `scattering`. Materials can be characterized by the subset of field values that they specify, based on the diagram at the end of the last chapter.




```
material plaster ()
= material (
    surface:
      material_surface (
        scattering:
          df::diffuse_reflection_bsdf (
            tint: color(0.7) )));
```

The value of the “surface” field is a “material_surface” instance

The value of the “scattering” field is a BSDF

3.3 The material's role in the rendering system

What does "using a material" mean in a rendering system? The material definition describes a collection of material properties and distribution functions. In a scene to be rendered, a material definition is associated with an object through some specification by the user of the rendering system, for example, through a mouse gesture in a graphical interface or through the syntax of a text-based description of the scene. An *instance of the material definition* is created for this association with an object. Note, however, that the `plaster` material definition has no parameters—every instance of `plaster` will behave in the same way. That is, given this definition of plaster:

```
material plaster ()
= material (
   surface: material_surface (
      scattering: df::diffuse_reflection_bsdf (
         tint: color(0.7) )));
```

...only one kind of instance can be created:

```
plaster()
```

In other words, all instances of `plaster` will produce the same type of appearance.

It would be better to have a more flexible version of the `plaster` material definition, for example, one in which the color can be specified by the user in the interface to the rendering system. Any material definition can be generalized by transforming constant values in the material instance into parameters of the material definition. Constant values in the material instance become *references* to the parameters *declared* by the material in its parameter list:

```
material plaster (
   parameter-declarations )
= material (
   surface: material_surface (
      scattering: df::diffuse_reflection_bsdf (
         parameter-references )));
```

For the current version of `plaster`, this means making the `tint` value of the scattering BSDF into a parameter of the material definition:

Listing 3.1

```
material plaster (
   color tint = color(0.7))      Tint values exposed as a parameter
= material (
   surface: material_surface (
      scattering: df::diffuse_reflection_bsdf (
            tint: tint)));      Using the "tint" parameter
```

By giving the parameter named `tint` a default value of `color(0.7)`, a `plaster` instance without a `tint` value will produce the same effect as before. However, an instance can specify a `tint` value to produce `plaster` instances of any color. For example, this instance would produce diffuse reflections of dark red:

```
plaster(tint: color(0.3, 0.1, 0.1))
```

Materials are often developed incrementally in this way—converting constant values in the material instance into references to parameters of the material definition when it becomes clear that increased flexibility in the use of the material is desirable. For example, the BSDF `diffuse_reflection_bsdf` also has a `roughness` parameter (a coefficient of the Oren-Nayar model on which the BSDF is based). By not including an explicit value for `roughness` in the BSDF instance, the `roughness` default value of `0.0` is used during rendering. The `roughness` parameter can be *exposed* in the material definition interface in the same way as the `tint` parameter:

Listing 3.2

```
material plaster (
  color tint = color(0.7),
  float roughness = 0.0 )                    Degree of roughness as a parameter
= material (
  surface: material_surface (
    scattering: df::diffuse_reflection_bsdf (
      tint: tint,
      roughness: roughness )));              Roughness passed to the BSDF
```

By specifying the same default value of `0.0` for `roughness` in the `plaster` parameters as `roughness` has in `diffuse_reflection_bsdf`, the new `plaster` definition has the same behavior as it did before the addition of the `roughness` parameter. This careful parameterization allows new capabilities for a material even as existing scenes that use the material will continue to render as before.

3.4 Diffuse transmission

The surface distribution functions can control transmission through a surface as well as reflection from it. By replacing the `diffuse_reflection_bsdf` in the `plaster` material definition with `diffuse_transmission_bsdf`, light travels through an object, but is scattered uniformly when it strikes the surface.

Listing 3.3

```
material diffuse_transmission ()
= material (
  surface: material_surface (
      scattering: df::diffuse_transmission_bsdf (
        tint: color(1.0))));                 Diffuse transmission
```

Replacing the `plaster` instances for the two objects with instances of `diffuse_transmission` produces Figure 3.3 (page 36):

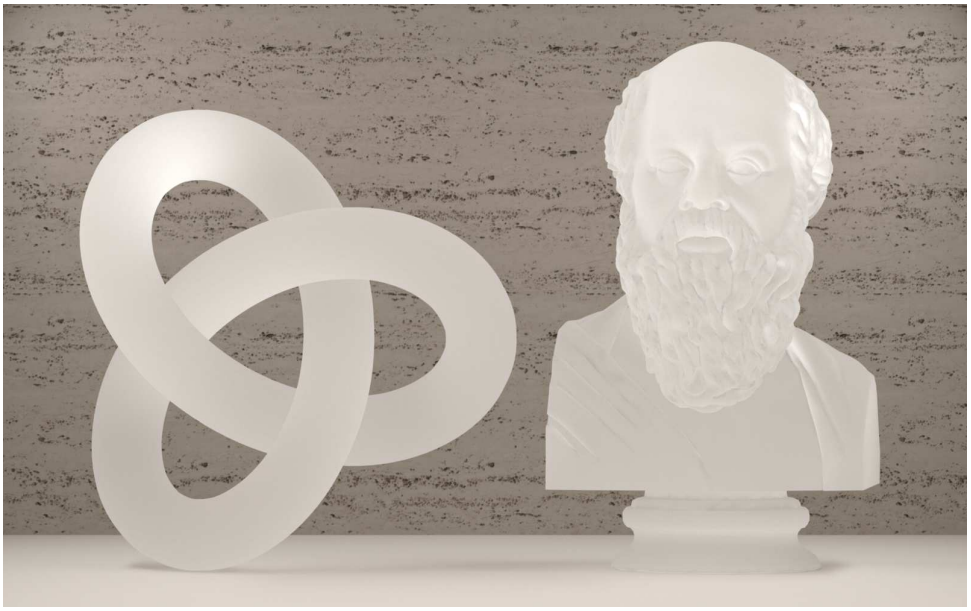


Fig. 3.3 – Diffuse transmission

This material is an idealization of diffuse transmission and isn't physically plausible; an object in nature would also reflect some light as well as transmit it. But a better known use of diffuse transmission can be produced by using the `diffuse_transmission` material with a thin, flat object.

Fig. 3.4 – Diffuse transmission through a thin object

Figure 3.5 (page 40) looks more like frosted glass or plexiglass; the missing diffuse reflections are hardly obvious and may be an acceptable approximation for this particular effect. The `diffuse_reflection` material can also show shadows cast on it in a convincing manner.

Fig. 3.5 – Shadows visible from diffuse transmission through a thin object

A very simple material like `diffuse_transmission`, though not complete in its handling of light interaction, may still be a useful rendering effect in the early phases of design work. Later, when greater accuracy in simulating the final look of a design, materials that better emulate the appearance real world substances could replace this early approximation, used in the same spirit as an artist's sketch.

3.5 Light emission

The discussion of the previous examples of diffuse reflection and transmission ignored a rather important component of the image—the source of light. This is a problem with any presentation of a language like MDL; without light objects cannot be seen, but without objects, light will never be reflected or transmitted to our eyes (or camera). Given the previous description of diffuse reflection and transmission, this section will now fill in the missing piece of the puzzle: the source of illumination in MDL.

3.5.1 Emissive objects

MDL defines light emission as a property of a surface, like reflection and transmission. The `material_surface` property contains two fields to implement this, `scattering` and `emission`:

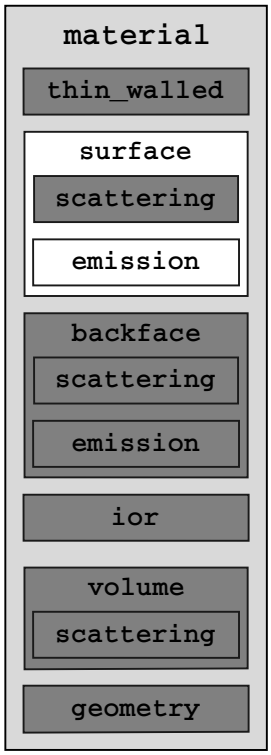
```
struct material_surface {
   bsdf scattering = bsdf();
   material_emission emission = material_emission();
};
```

The `scattering` field is a BSDF, but the `emission` field is defined as another compound type, the `material_emission` struct:

```
struct material_emission {
   edf emission = edf();
   color intensity = color(0.0);
   intensity_mode mode = intensity_radiant_exitance;
}
```

The *emission distribution function*, or *EDF*, in the `material_emission` struct is implemented by type `edf`. For the examples of the next several chapters, the light-emitting objects will use `diffuse_edf`, which emits light uniformly in all directions.

The simplest emissive material definition only defines a value for the `emission` field of the `surface` field of the `material` struct. This example also provides control of the light intensity through the parameterization of the value of the `material_emission`'s `intensity` field (the parameter `tint`).



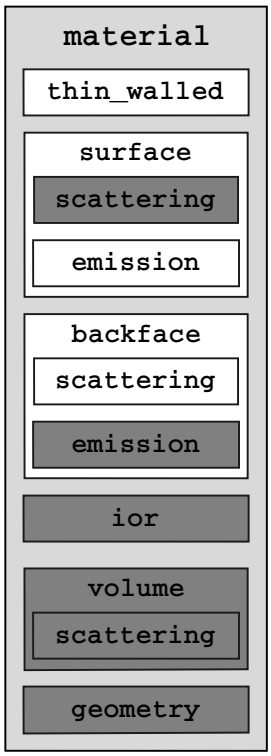

```
material emission (
   color tint = color(1.0) )
= material (
   surface: material_surface (
      emission: material_emission (
         emission: df::diffuse_edf(),
         intensity: tint )));
```

Light emission from front surface

Rendering the objects of the previous scene with ten rectangular polygons that use the `emission` material produces Figure 3.6:

Fig. 3.6 – Emission from the front side of the polygon, no material assigned to the back

The `backface` field of the `emission` material is not defined, so the single polygons only emit light from their "front" side (defined by a geometric convention of the renderer). Defining a diffuse reflection BSDF for the `backface` field and setting `thin_walled` to true defines two different materials for the two sides of the polygon.




```
material emission_with_diffuse_backface (
  color tint = color(1),
  color backtint = color(0.8, 0.78, 0.75) )
= material (
  thin_walled: true,
  surface:
    material_surface (
      emission: material_emission (
        emission: df::diffuse_edf(),
        intensity: tint )),
  backface:
    material_surface (
      scattering:
        df::diffuse_reflection_bsdf (
          tint: backtint )));
```

Light emission from front surface

Diffuse reflection from back surface

The color used for diffuse reflection in the `backface` field is set by default to be the same color for the diffuse reflection of the floor, so that they seem to be made of the same type of real-world material.

Fig. 3.7 – Emission from the front side of the polygon, diffuse reflection from the back

This material, `emission_with_diffuse_backface`, was used to provide illumination for the examples of diffuse reflection and transmission in the beginning of this chapter. A wider angle of view that used in those images shows the structure of the lighting environment that produced them.

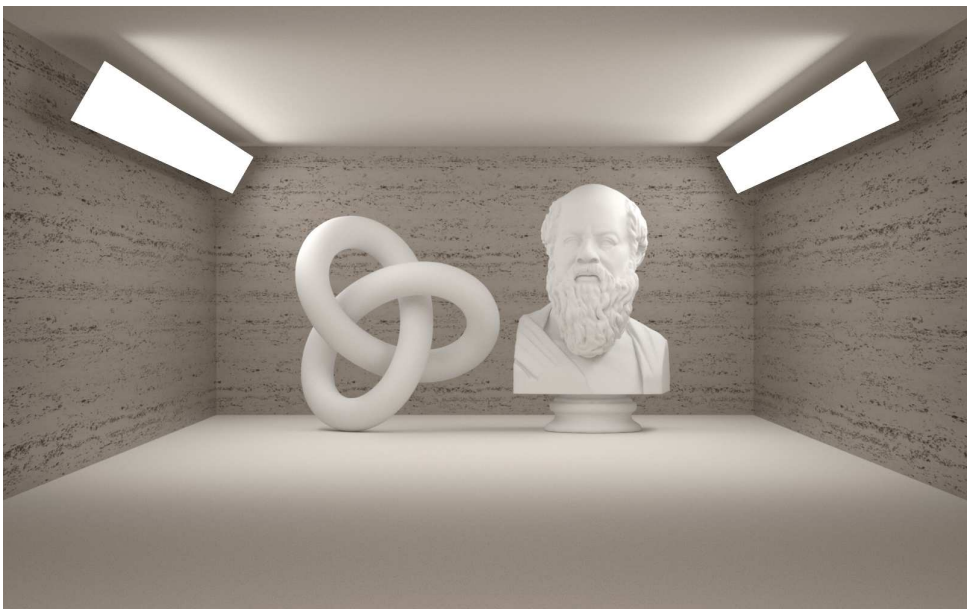


Fig. 3.8 – Lighting environment of the images of the previous section

Like lighting design for photography, the emissive objects in the upper corners are not positioned to be seen in Figure 3.8, but for the lighting effect their shape and placement will provide.

3.5.2 Design considerations

The emissive rectangles used in lighting the pictures at the beginning of this chapter were structured in the scene setup to be easily rotated around their lengthwise axis. By rotating them so that they point straight down, the ceiling is not longer illuminated, and the upper parts of the two objects are no longer lit as well.

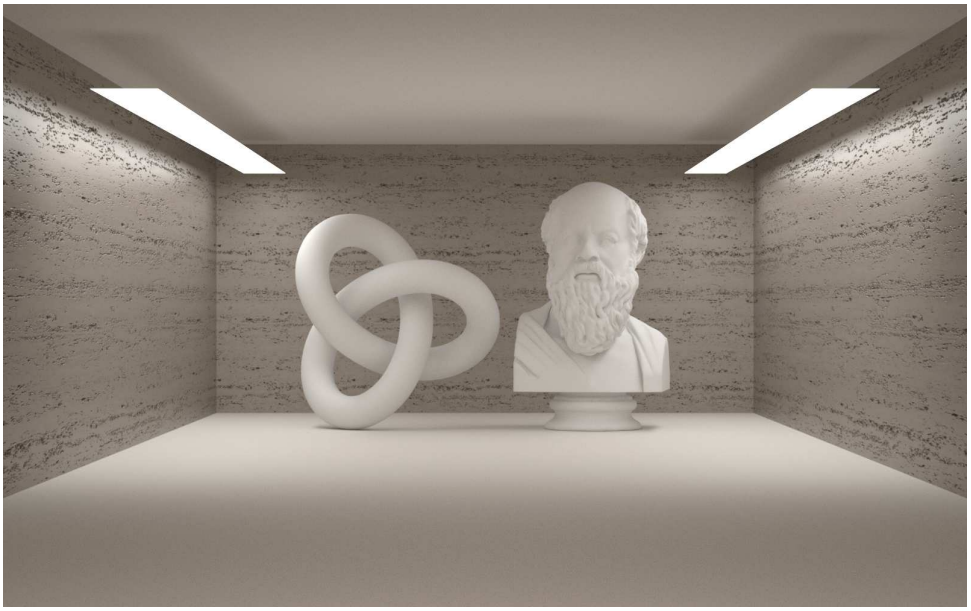


Fig. 3.9 – Emissive objects rotated to point down

Rotating the emissive rectangles so that they point straight up, all light in the scene now comes from diffuse reflection from the ceiling.

Fig. 3.10 – Emissive objects rotated to point up

Changing only two parameters of the geometric structure of the scene—the rotation of the emissive rectangles—different lighting moods can be created. With the physically plausible simulation of lighting effects provided by MDL, lighting scene design can more close resemble

the traditional lighting techniques of photography and cinematography. Technical advances in physical simulation ironically allow the pre-computerized design of traditional Hollywood lighting to once again be a source of artistic inspiration.

3.6 Specular interaction at a surface

As described in "Idea #2: Angles matter (page 11)," the *specular* reflection of light from a surface in MDL describes the reflection from a perfect mirror. Specular reflection, in this sense, is a mathematical idealization, like perfectly uniform (diffuse) reflection. Like diffuse interaction, specular interaction is extended to include "mirror-like" transmission at a surface—light is *refracted* at the surface based on the index of refraction defined by the material.

For diffuse reflection and transmission, MDL provides two distribution functions, `df::diffuse_reflection_bsdf` and `df::diffuse_transmission_bsdf`. However, in specular interaction at a surface, there is a single distribution function, `df::specular_bsdf`.

Listing 3.4

```
bsdf df::specular_bsdf (
    color tint = color(1.0),
    uniform scatter_mode mode = df::scatter_reflect
);
```

Scattering mode defined by an enum

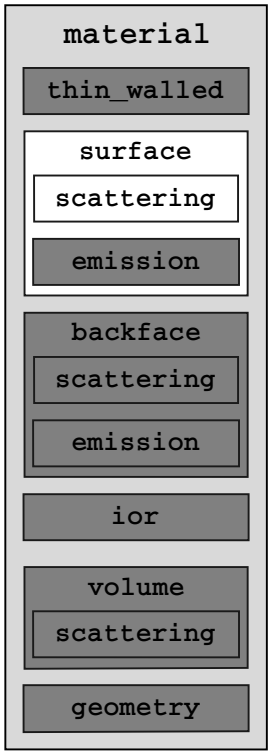
The `mode` field of `specular_bsdf` controls whether light is reflected, transmitted, or both. To implement a set of choices for use as argument values in constructors and functions, MDL provides an *enumerated type*, defined by the keyword `enum`. The enumerated type used by `specular_bsdf` to control reflection and transmission is called `scatter_mode`. The definition of an `enum` in MDL contains a list of the possible values that a value of that `enum` type can have.

```
enum scatter_mode {
    scatter_reflect,
    scatter_transmit,
    scatter_reflect_transmit
};
```

All of the remaining materials described in this chapter will use `scatter_mode` to choose between reflection, transmission, or a combination of the two.

3.6.1 Specular reflection

Pure specular reflection—a perfect mirror surface—is the default value of the `df::specular_bsdf` function. The simplest material that implements a mirror does not specify any argments for the BSDF:




```
material specular()
= material(
   surface: material_surface(
      scattering:
          df::specular_bsdf()));
```
Default specular reflection

The default `tint` value for `df::specular_bsdf` is `color(1.0)`—in other words, all light is reflected by the surface. Such a perfect reflector is impossible in nature. To define a material that reflects 50% of the incoming light, a `tint` value of `color(0.5)` is used:

Listing 3.5

```
material mirror()
= material(
   surface: material_surface(
      scattering: df::specular_bsdf(
            tint: color(0.5),
         mode: df::scatter_reflect)));
```
Color value to simulate 50% reflection

In the `mirror` material, the `mode` argument for `df::specular_bsdf` is also specified so that this material can serve as the pattern for the other types of specular interaction.

Using the `mirror` material to render the scene from the diffuse reflection examples creates Figure 3.11:

Fig. 3.11 – Pure specular reflection

Notice the hard edges of the reflections. Pulling the camera back shows the lighting environment of Figure 3.11 (page 46):

Fig. 3.12 – Lighting environment of Figure 3.12

Changing the lighting environment Figure 3.12 only by rotating the light-emissive polygons creates smoother lighting transitions for some of the reflections in Figure 3.13 (page 48):

Fig. 3.13 – Effect of different lighting environment on the specular surfaces

Figure 3.14 is a close-up view of Figure 3.13:

Fig. 3.14 – Close-up of Figure 3.14

Subjectively, Figure 3.14 may appear to better represent a mirror-like surface than the previous close-up image, Figure 3.13. Why? Typical photographs of highly polished metallic surfaces (jewelry in a catalog, for example) are shot in a conventional "table-top" style, with diffuse reflective surfaces and stragetically placed spotlights. Perhaps the combination of hard and soft transitions in the reflections is more like typical photographs of metallic surfaces.

Creating a circular arrangement of small polygons, each with an emissive surface pointed twoard the ceiling, produces a diffuse lighting environment without the hard lighting transitions of the previous images.

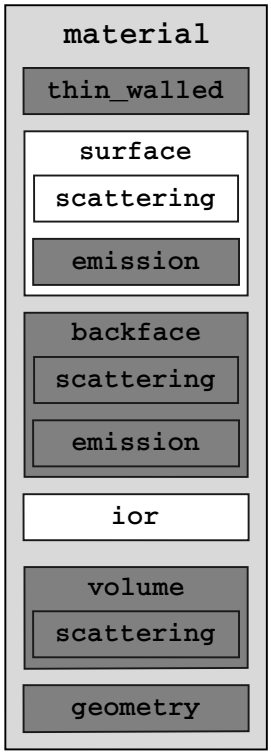
Fig. 3.15 – Small emissive surfaces pointing toward the ceiling to light specular surfaces by diffuse reflection

Fig. 3.16 – Close-up of Figure 3.16

An important quality of the surface—its apparent roughness—has been increased *only by changing the lighting environment*. The strong influence of the lighting environment on the interpretation of the physical nature of a surface has significant implications for materials intended for design visualization and the user interface in which they are presented.

3.6.2 Specular transmission

Like specular reflection, specular transmission at a surface is defined by `df::specular_bsdf` but with `df::scatter_mode` as the value for the `mode` field. However, by specifying a value greater than 1.0 for the `material`'s *index of refraction* (field `ior`), the direction of light is modified at the surface boundary, or *refracted*.




```
material specular_transmission (
    color tint = color(1) )
= material (
    ior: color(1.3),                    Index of refraction
    surface: material_surface (
       scattering: df::specular_bsdf (
          tint: tint,
             mode: df::scatter_transmit ));    Transmission mode
```

In material `specular_transmission`, the `tint` argument to `df::specular_bsdf` is exposed as a parameter to the material. By default, the color of the light that passes through the object is not modified, only the light's direction through refraction.

Fig. 3.17 – Specular transmission without color modification of the light

However, by specifying a value less than 1.0 for the color components of the tint, the light is reduced by that factor at each surface interaction. Multiple surface interaction occur when light enters and leaves the object, but also due to multiple interactions within the object due to *internal reflection*.

Fig. 3.18 – Specular transmission with a tint of `color(0.75)`

In Figure 3.18, the same value was used for the red, green and blue components of the `tint` parameter, resulting in grays of varying darkness. Colors are similarly scaled by each interaction, but only changes of intensity result—there are no changes of hue or saturation.

Fig. 3.19 – Specular transmission with a tint of `color(0.8, 0.4, 0.4)`

The change of the color of the light due to the `df::specular_bsdf` distribution function only occurs at surface interactions. In the physical world, light can be affected by the medium through which it travels, typically called in computer graphics, "particpating media." This phenomenon is modelled MDL by *volume distribution functions*, which will be described in *[a later chapter]*.

3.6.3 Specular reflection and transmission

Combining specular reflection and transmission only requires a change to the `mode` value to `df::scatter_reflect_transmit`. Like the pure transmission case, the `ior` is also significant.

Listing 3.6

```
material specular_reflection_transmission()
= material(
    ior: color(1.3),
  surface: material_surface(
     scattering: df::specular_bsdf(
       tint: color(0.6, 0.7, 0.8),
         mode: df::scatter_reflect_transmit )));
```

Index of refraction

Combined specular reflection and transmission

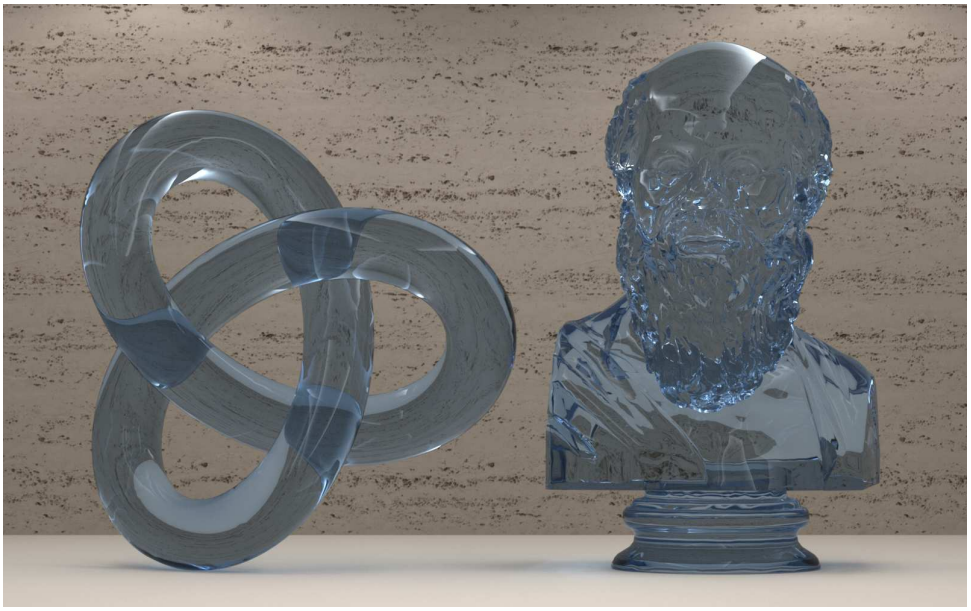


Fig. 3.20 – Combined reflection and transmission with BSDF `df::scatter_reflect_transmit`

As was the case with specular reflection, specular transmission is heavily dependent upon the spatial and lighting context for its believeability and visual effect. Upon reflection, this makes sense—these materials depend to such a high degree on *what* they can reflect and transmit.

3.7 Glossy interaction

By this point, the pattern in this chapter of the simple materials that define surface interaction should be clear. The structure of the `material` remains almost the same, with the addition of a value for the `ior` field when transmission is defined by the distribution function. The parameters of the distribution functions vary, but have similarities based on whether the distribution function defines reflection, transmission, or both.

For *glossy* interaction, a further attribute of surface interaction is defined. *Anisotropy* describes a variation in the result of measurement based on the direction in which that measurement is taken. Physical materials like brushed aluminum display anisotropic reflection due to the fine grooves in the surface that enhance reflection at right angles to the direction of the grooves.

In the glossy distribution functions of MDL, the proportions of reflection in the two axes of direction on the surface (typically defined by the geometric model itself) are defined by fields named `roughness_u` and `roughness_v` (for the canonical axes names *u* and *v*).

Listing 3.7

```
bsdf simple_glossy_bsdf(
    float roughness_u,
    float roughness_v = roughness_u,
    color tint = color(1.0),
    float3 tangent_u = state::texture_tangent_u(0),
    uniform scatter_mode mode = scatter_reflect,
```

Separate aniosotropic control in the u and v directions

By default, the value of `roughness_v` takes the value of `roughness_u`. The typical effect of a default value for a parameter in MDL is to "do nothing," whatever that might mean for each particular parameter. In the case of the default value of `roughness_v`, doing nothing means to duplicate the required value for `roughness_u`, so that that glossy reflections are *isotropic*—light interaction is not directionally dependent.

The following sections demonstrate isotropic reflection. Anisotropic reflection is a fundamental component of the implementation of iridescent silk described in "Fabric" (page 145).

3.7.1 Glossy reflection

The following material, `glossy_reflection`, uses the default behavior of `roughness_v` to define isotropic reflection. The other field values specified in `df::simple_glossy_bsdf` are the same as in many of the previous materials.

Listing 3.8

```
material glossy_reflection ()
= material (
  surface: material_surface (
    scattering: df::simple_glossy_bsdf (
      tint: color(0.6),
      roughness_u: 0.3,
      mode: df::scatter_reflect ));
```

Roughness in u also used for v

Reflection mode

With a `roughness` value of 0.3 for `df::simple_glossy_bsdf`, material `glossy_reflection` creates highlights over the surface of the object that emphasize the position of the emissive surfaces that light the scene.

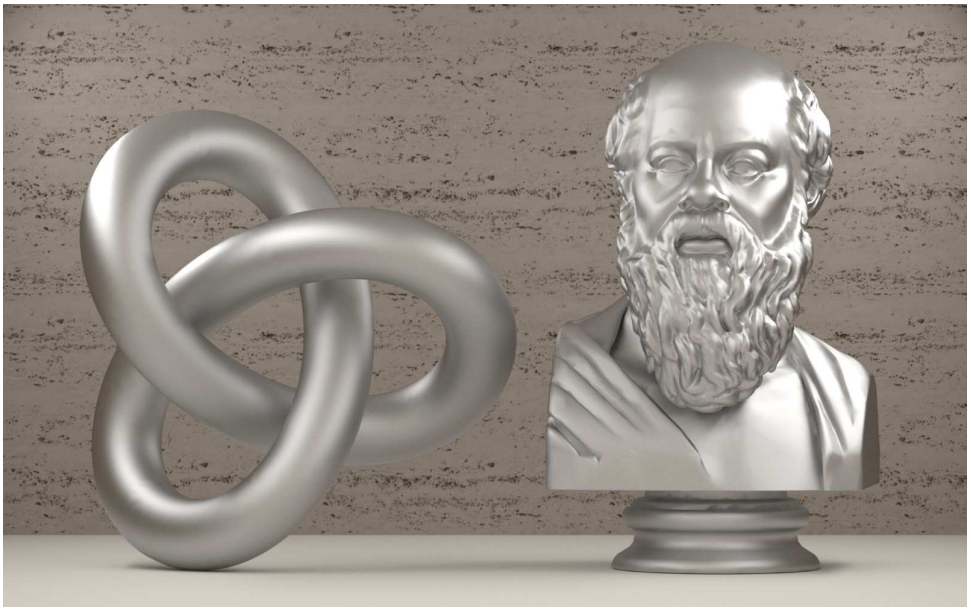


Fig. 3.21 – Glossy reflection with a roughness value of 0.3

3.7.2 Glossy transmission

As in the specular equivalent to glossy transmission, the index of refraction specifies the degree of refraction of lighting striking the surface.

Listing 3.9

```
material glossy_transmission ()
= material (
   ior: color(1.3),    Index of refraction ;
   surface: material_surface (
      scattering: df::simple_glossy_bsdf (
        tint: color(1),
          roughness_u: 0.15,    Roughness used for both u and v

          mode: df::scatter_transmit ));    Transmission mode
```

However, rendering the objects with `glossy_transmission` results in very dark areas that do not correspond with our intuition of a transparent object, glossy or not.

Fig. 3.22 – Glossy transmission—no reflection

Though they were less visible in the specular case, the dark areas represent areas in which *total internal reflection* reduced the light intensity to zero. Such an object is never seen in nature because of the unusual definition of the surface—only transmision, without any reflection at all. This unnatural condition was also true for the material that only defined specular transmission, but the error was not so easily seen.

3.7.3 Glossy reflection and transmission

The physically reasonable material for glossy interaction defines both reflection and transmission:

Listing 3.10

```
material glossy_reflection_transmission ()
= material (
   ior: color(1.3),
   surface: material_surface (
      scattering: df::simple_glossy_bsdf (
         tint: color(1),
         roughness_u: 0.15,
         mode: df::scatter_reflect_transmit ) ));
```

Combined reflection and transmission

Using the `glossy_reflection_transmission` material in Figure 3.23 (page 56), the apparent lack of light in the dark areas is compensated by the reflection component.

Fig. 3.23 – Combined glossy reflection and transmission

3.8 Lighting techniques

The previous sections have shown how simple the textual content of an MDL material can be yet still produce complex and suggestive imagery. The interaction of emissive and non-emissive materials provide a variety of design possibilities.

3.8.1 Geometric constructions

A *luminaire* is the mechanical structure that houses the light-emitting component of a lamp or lighting fixture. Using MDL, diffuse and emissive materials can be combined to construct simulated lamps and lighting devices that provide complex illumination effects.

For example, imagine a box, open on one end, constructed of diffuse materials, and containing a single emissive surface.

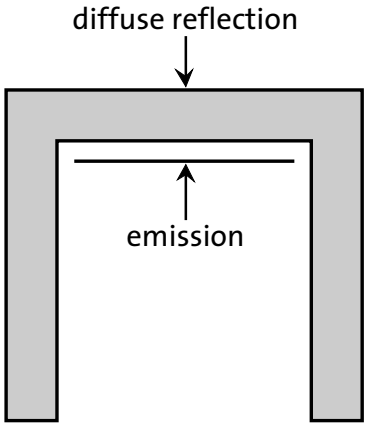


Fig. 3.24 – Surrounding an emissive surface with a diffuse reflection surface

Inserting a set of these objects in a scene, positioned to enhance their effect on the illumination of the room with the light intensity adjusted accordingly, can produce an image like Figure 3.25:

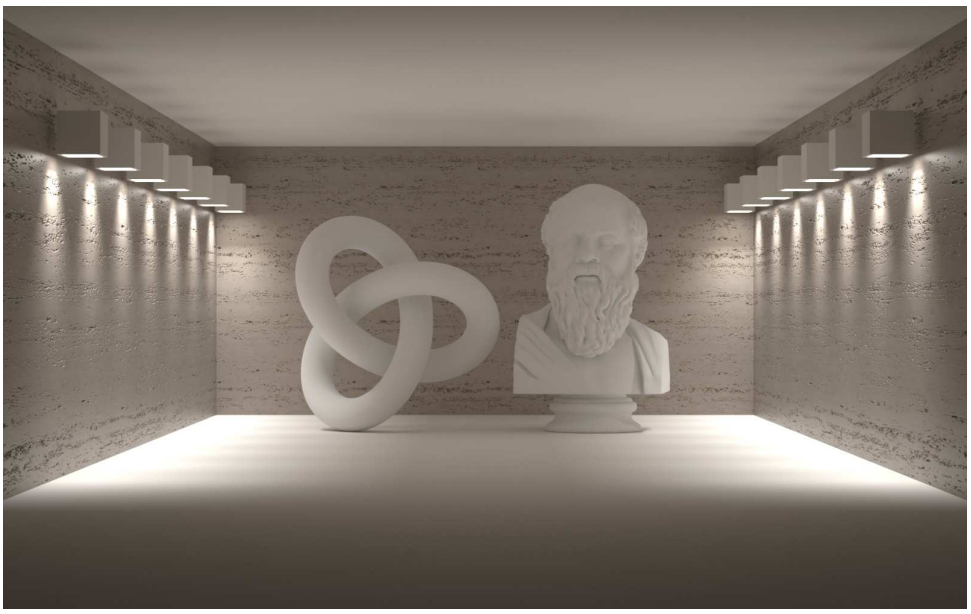


Fig. 3.25 – Emissive surfaces contained within non-emissive objects

These geometric objects, simple to construct with a modeling application, could also be rigged to change size (widening or narrowing the beam of emitted light) or orientation (like spotlights), providing animated effects that would not required special features in the modeling and animation packages.

3.8.2 A global lighting environment

The `base` MDL module provides a number of utility functions to simplify common tasks in the development of materials. The `base` module also provides a function to simulate the *environmental illumination* like that produced by the sun and the sky. The `base` functions are described in Appendix *[the appendix that will describe the base functions]*, but an example of the `base::sun_and_sky` function will be useful as a conclusion to this chapter on surface interaction.

The way in which the environmental illumination is specifed is renderer-dependent, but here is an example of a set of argument values for `base::sun_and_sky`:

```
base::sun_and_sky (
   sun_direction: -1.0 1.0 -1.0,
   horizon_blur: 0.5,
   sun_disk_intensity: 0.04,
   sun_disk_scale: 1.0 )
```

Specifying this function for environmental illumination and using the scene definition for the previous renderings produces Figure 3.26 (page 58):

Fig. 3.26 – Specular reflections using the sun and sky simulation from the MDL "base" module

Notice that in a close-up view, the visual appearance of the objects remains that of mirror-like surfaces reflecting the sky.

Fig. 3.27 – Close-up of sun and sky simulation

However, our assumption that the objects are reflective is not supported by any obvious visual clues (for example, other nearby objects that are visible in reflections). The objects are actually white and blue, with small areas of light brown and tan. This coloring is in fact a stereotype of an airbrush painting technique to produce chrome. Perhaps photographs of cars in the desert outside Los Angeles first created this visual sterotype that was then adopted by airbrush artists. No matter its origin, this visual convention further suggests that creating effective materials may also be dependent upon historical and cultural circumstances.

4 Glass

Chapter 3, "Light at a surface," presented the simplest MDL implementation of specular reflection and refraction, resulting in objects that resembled glass. This chapter begins with that simple model, extending the simulation of glass to include various real-world properties such as the absorption and scattering of light within the object.

4.1 Geometric structure and the appearance of glass

For the example renderings in this chapter, the three geometric objects have been each been modeled with three different thicknesses of glass to demonstrate its importance in refraction, absorption and subsurface scattering.

Shape of the geometric models *Cross section showing the thickness of the glass*

Fig. 4.1 – Geometric models rendered in this chapter

The simplest specular reflection and refraction model uses the `df::specular_bsdf` distribution function for the `scattering` parameter of the `material_surface` property.

Listing 4.1

```
material specular_reflect_transmit (
   color tint = color(1.0),
   uniform color ior = color(1.3)) =
material (
   ior: ior,
   surface: material_surface (
       scattering: df::specular_bsdf (      Specular BSDF
         tint: tint,
           mode: df::scatter_reflect_transmit)));   Mode for both reflection and transmission
```

Rendering with this material produces Figure 4.2. Note the effect of internal reflection in the thicker wall of the vase on the left.

```
specular_reflect_transmit(
  tint:
    color(0.8, 0.9, 1.0))
```

Figure 4.2

As a simplification for the rendering of thin sheets of window glass or other objects in which refractive effects are not typically visible, the `thin_wall` property of the `material` struct can be set to `true`.

Listing 4.2

```
material specular_reflect_transmit_thin_walled (
   color tint = color(1.0),
   uniform color ior = color(1.3)) =
material (
   thin_walled: true,    Set thin_walled property
   ior: ior,
   surface: material_surface (
      scattering: df::specular_bsdf (
         tint: tint,
         mode: df::scatter_reflect_transmit)));
```

With the `thin_wall` property, light rays are not refracted through the object, though the reflections of the emissive surfaces lighting the scene are still visible, with a more subtle effect of the reflected floor at the bottom of the objects.

```
specular_reflect_transmit_thin_walled(
  tint:
    color(0.64, 0.72, 0.80))
```

Figure 4.3

The following sections of this chapter demonstrate other possible properties of glass: glossy reflection and transmission, the Fresnel effect, and subsurface scattering.

4.2 Glossy reflection and transmission

The glass-like example (page 55) of glossy reflection and refraction in Chapter 3 used a "solid" geometric model in the rendered scene. In simulating glass objects like vases and drinking glasses, light will pass through four surfaces of the geometric model—the outer and inner boundaries of the glass form. In glossy transmission through the object, the thickness of the glass form will affect the apparent blurring of the background as seen through the glass.

This material for glossy reflection and transmission is similar in form to the example of Chapter 3, but provides material parameters for the tint, index of refraction and the surface roughness

Listing 4.3

```
material glossy_reflect_transmit (
   color tint = color(1.0),
   float roughness = 0.0,
   uniform color volume_ior = color(1.5)) =
let {
   bsdf surface_bsdf = df::simple_glossy_bsdf(   Glossy distribution function
      tint: tint,
      roughness_u: roughness,   Parameter for the degree of glossiness
      mode: df::scatter_reflect_transmit);   Mode for both reflection and transmission
} in
material (
   ior: volume_ior,
```

```
    surface: material_surface (
      scattering: surface_bsdf));
```

Rendering with this material produces Figure 4.4.

```
glossy_reflect_transmit(
  roughness: .02,
  tint:
    color(0.8, 0.9, 1.0))
```

Figure 4.4

4.3 Variable transmission at edges

Section "Fresnel layering in a material" (page 106) described the affect of the viewing angle on reflection for thin coatings like varnish. For glass, the Fresnel effect can also be modeled using `df::fresnel_layer` to combine the reflection and refraction components.

Listing 4.4

```
material glossy_reflect_transmit_layers (
   color reflection_color = color(1.0),
   color transmission_color = color(1.0),
   float roughness = 0.0,
   uniform color volume_ior = color(1.5),
   uniform color layer_ior = color(1.5)) =
let {
   bsdf reflection = df::simple_glossy_bsdf(
     tint: reflection_color,
     roughness_u: roughness,
     mode: df::scatter_reflect);
```

Reflection

```
   bsdf transmission = df::simple_glossy_bsdf (
     tint: transmission_color,
     roughness_u: roughness,
     mode: df::scatter_transmit);
```

Transmission


```
    bsdf surface_bsdf = df::fresnel_layer (
      ior: layer_ior,
      layer: reflection,
      base: transmission);
} in
material (
  ior: volume_ior,
  surface: material_surface (
    scattering: surface_bsdf));
```

Fresnel layering of reflection and transmission

The `glossy_reflect_transmit_layers` material includes the glossy roughness parameter to the distribution functions as a parameter to the material itself. By default, the `roughness` parameter is 0.0; that is, by default the material produces specular reflection and refraction. The reflection and refraction colors default to white (`color(1.0, 1.0, 1.0)`).

In the first example rendering, only the reflection color has been changed from the default white to blue. Due to the Fresnel layering, the blue tint of the reflection is most produced at the edges, for example, at the bottom of the vase.

```
glossy_reflect_transmit_layers(
  reflection_color:
    color(0.8, 0.9, 1.0))
```

Figure 4.5

In Figure 4.6, the reflection color is white and the transmission color is now set to blue.

```
glossy_reflect_transmit_layers(
  transmission_color:
    color(0.8, 0.9, 1.0))
```

Figure 4.6

The effect of the combination of reflection and transmission with Fresnel layering can be made clearly visible by defining an extreme value for the reflection. In Figure 4.7, the reflection color has been set to pure red. The pure red within the glass object shows the effect of the increase of reflection predicted by the Fresnel equations; the angles of incidence within the glass creates *total internal reflection*.

```
glossy_reflect_transmit_layers(
  reflection_color:
    color(1, 0, 0),
  transmission_color:
    color(0.8, 0.9, 1.0))
```

Figure 4.7

One of the physical properties of dielectric materials like glass and plastic is that dielectric reflections are not changed in color by the object's color. This is modeled in the MDL material with a default value of white for reflections. The unrealistic reflection color (from the standpoint of physics) of red provides a clear demonstration of total internal reflection, however.

Figure 4.8 (page 65) uses a light yellow reflection color very close to white which, though still not a realistic property of dielectrics, creates a subtle patina effect.

```
glossy_reflect_transmit_layers(
  reflection_color:
    color(1.0, 0.95, 0.9),
  transmission_color:
    color(0.8, 0.9, 1.0))
```

Figure 4.8

With a roughness value of 0.02 the glossy reflection and transmission effects can be seen using the same near-white reflection color as before:

```
glossy_reflect_transmit_layers(
  reflection_color:
    color(1.0, 0.95, 0.9),
  transmission_color:
    color(0.8, 0.9, 1.0),
  roughness: 0.02)
```

Figure 4.9

4.4 The appearance of color through absorption

The previous materials all defined the color of light transmission through glass as a *boundary effect* in which the light is changed in color only at its intersection at the object's surface. The *volume property* of the material struct defines how light is modified as it moves *through* an object. The volume property is an instance of the `material_volume` struct:

Listing 4.5

```
struct material volume {
    vdf scattering = vdf();          Volume distribution function
    color absorption_coefficient = color();
    color scattering_coefficient = color();
};
```

An instance of the `material_volume` struct is added to the material struct as the argument to the `volume` field.

Listing 4.6

```
material glossy_reflect_transmit_absorb (
  color surface_color = color(1.0),
  float roughness = 0.0,
  uniform color absorption = color(1.0),     Parameter used in the "material_volume" property
  uniform color volume_ior = color(1.5)) =
let {
  bsdf surface_bsdf = df::simple_glossy_bsdf(
    tint: surface_color,
    roughness_u: roughness,
    mode: df::scatter_reflect_transmit);
} in
material (
  ior: volume_ior,
  surface: material_surface (
    scattering: surface_bsdf),
  volume: material_volume(                   Volume definition
    absorption_coefficient: absorption));
```

In an image rendered using material `gloss_reflect_transmit_absorb` the "thickness" of the glass – the distance between boundaries—affects the degree of absorption of the red, green, and blue components of the absorption color.

```
glossy_reflect_transmit_absorb(
  absorption:
    color(10, 6, 2))
```

Figure 4.10

The parameter for absorption is a *probability density* defined for one meter in world space. A dependency on world space implies that the numerical values of the geometric model will affect how much absorption will occur. For the numerical values of the geometric model in Figure 4.10, the absorption coefficient required to produce an effect similar to the previous renderings (determined experimentally) was `color(10,6,2)`—hardly an intuitive quantity.

To improve the visual control of absorption, the following material calculates the absorption coefficient based on the distance at which full absorption would occur and the color that such full absorption would produce.

Listing 4.7

```
material glossy_reflect_transmit_absorb_falloff (
  color surface_color = color(1.0),
  uniform color transmission_color = color(1.0),
  uniform float transmission_distance = 1.0,
  float roughness = 0.0,
  uniform float volume_ior = 1.5) =
let {
  bsdf surface_bsdf = df::simple_glossy_bsdf(
    tint: surface_color,
    roughness_u: roughness,
    mode: df::scatter_reflect_transmit);
  color absorption_coefficient =
    transmission_distance <= 0 ?
    color(0.0) :
    -math::log(transmission_color) / transmission_distance;
} in
material (
```

Color produced by full absorption

Distance at which full absorption occurs

Calculation of absorption coefficient

```
    ior: color(volume_ior),
    surface: material_surface (
      scattering: surface_bsdf),
    volume: material_volume(
      absorption_coefficient: absorption_coefficient));
```

Volume property

The two parameters of color and distance, `transmission_color` and `transmission_distance`, provide an interface to this material that resembles the previous materials based on boundary calculations. In Figure 4.11, the same color is used for the `transmission_color` as was used for the colors of the previous materials; the `transmission_distance` was adjusted experimentally to create the desired effect.

```
glossy_reflect_transmit_absorb_falloff(
  transmission_color:
    color(0.8, 0.9, 1.0),
  transmission_distance: .05)
```

Figure 4.11

The example renderings of the `glossy_reflect_transmit_absorb_falloff` material have all used the default `roughness` value of 0.0, producing specular reflection and transmission. Figure 4.12 (page 69) adds a `roughness` value of 0.02 to the material parameters of Figure 4.11.

```
glossy_reflect_transmit_absorb_falloff(
  transmission_color:
    color(0.8, 0.9, 1.0),
  transmission_distance: .05,
  roughness: 0.02)
```

Figure 4.12

4.5 Subsurface scattering as a model of glass

In addition to controlling the absorption of light in a volume, the MDL volume property specifies how light is *scattered*. Like the absorption coefficient, the scattering coefficient is a probability density defined for one meter in world space. The following material includes scattering in the volume property, using the same strategy as in the previous material to create intuitive parameters of distance and color for scattering as well as for absorption.

Listing 4.8

```
material subsurface_scattering (
  color surface_color = color(1.0),
  uniform color scattering_color = color(0.05, 0.05, 0.05),
  uniform color transmission_color = color(0.1, 0.95, 0.65),
  uniform float transmission_distance = 0.3,
  float roughness = 0.0,
  uniform float volume_ior = 1.5) =
let {
  bsdf surface_bsdf = df::simple_glossy_bsdf(
    tint: surface_color,
    roughness_u: roughness,
    mode: df::scatter_reflect_transmit);
} in
material (
  ior: color(volume_ior),
  surface: material_surface (
    scattering: surface_bsdf),
  volume: material_volume(
```

Absorption and scattering parameters

```
scattering: df::anisotropic_vdf(
  directional_bias: -0.5),
```

Volume distribution function

```
scattering_coefficient:
  (transmission_distance <= 0) ?
  color(0.0) :
  -math::log(scattering_color) / transmission_distance,
```

Scattering coefficient

```
absorption_coefficient:
  (transmission_distance <= 0) ?
  color(0.0) :
  -math::log(transmission_color) / transmission_distance));
```

Absorption coefficient

The scattering color describes *how much* of the light is scattered, so that darker colors scatter the light to a smaller degree that brighter colors. Using this material with a scattering color of dark gray produces Figure 4.13:

```
subsurface_scattering(
  transmission_color:
    color(0.8, 0.9, 1.0),
  scattering_color:
    color(.1, .1, .1),
  transmission_distance: .1
)
```

Figure 4.13

With a scattering color that has a non-gray hue, the red, green and blue components of the light are scattered by different amounts. In Figure 4.14 (page 71), the scattering color is `color(0.1,0.1,0.3)`. Because more of the blue component is scattered than the red and green components, the result is a tinting effect of yellow—the complementary color of blue. The yellow combines with the blue absorption color to produce the overall effect of a variation between blue and green.

Figure 4.14

5 Light in a volume

> Every physically-based simulation ultimately concerns physical *quantities*; that is, measurable attributes of phenomena or of matter. Implicit in any equation expressing a relationship among physical quantities is a system of units in which the quantities were measured. Thus, each algebraic equation with a physical interpretation has a corresponding *dimensional equation* showing how the units combine. Unfortunately, the units for radiometric quantities can be troublesome, owing to the overwhelming number of distinct and sometimes inconsistent definitions. As we begin to consider domains outside of radiative transfer the problem grows worse; each field has its own bewildering assortment of units and terminology. This unfortunate fact is largely historical, resulting from independent development of many fundamentally related fields.
>
> James Arvo, *Transfer Equations in Global Illumination*, 1993.

So far, the materials of the previous chapters have almost exclusively described the interaction of light at a surface. In the chapter on glass, the last two sections (page 66) used the *volume property* of a material to modify the color and opacity of the glass object. But for purposes of design, such treatment is often similar to the use of surface properties in its aesthetic intent.

Fig. 5.1 – Volumetric effects in short distances as an attribute of design

To simulate natural phenomena like smoke or fog, a geometric object serves in MDL as an enclosure for volumetric rendering effects. Now the rendering is no longer concerned only with surfaces; instead, a surface bounds a three-dimensional region of space within which the rendering system simulates the phenomenon.

5.1 Background: Categories of interaction in a volume

The physical medium through which light passes as it strikes objects, making them visible, also participates in modifying the light. The absence of such a *participating medium* in older rendering systems simulated a perfect vacuum, but such interactions cannot be ignored when rendering calculations are based on physical reality. Alas, issues of efficiency must also be considered. How these volumetric effects actually appear suggests implementation strategies for rendering.

For example, the four photographs in Figure 5.2 to Figure 5.5 depict physical situations that renderings systems could attempt to represent:

Fig. 5.2 – Small bubbles

Fig. 5.3 – Drops of milk in water

Fig. 5.4 – Milky water

Fig. 5.5 – Drops of milk in milky water

How these images might be simulated can be considered a matter of scale. In a physical sense, all four are the result of light's interaction with geometric objects, from the pinprick-sized water bubbles to the microscopic casein protein molecules in the milk. As computational power has increased, modeling very small objects geometrically, often in combination with complex motion definition, called a *particle system*,[1] has enabled the visually convincing depiction of dust storms, fire, and the spray of ocean waves. This technique could be used to model the structure and motion in Figure 5.2 (page 74) and Figure 5.3 (page 74) and the trail of milk drops in Figure 5.5 (page 74).

However, the cloudy appearance of the last two photographs is best served by a *statistical* approach. The aggregate behavior of light's interaction with an extremely large number of objects—the protein molecules in milk—can be defined by scaling factors used in mathematical models implemented by the rendering system. The models assume that very small *particles* are suspended in a *medium*; the numerical inputs, or *coefficients*, of the models define how light is affected by the particles. These coefficients form the basis for the parameters of MDL's *volume distribution functions*.

The models for volumetric effects may be developed using known properties of the medium that can be represented mathematically. For example, the droplets of water suspended in clouds are small enough that surface tension makes them almost perfectly spherical in shape. That geometric observation, along with well-known properties of light's interaction with water, can form the basis for the mathematical modeling of clouds and fog. Conversely, a mathematical model can be derived from a large number of measurements of light's behavior in a medium. The accuracy of this *heuristic* technique for model development can be tested against further measurements, with the model improved to an acceptable degree of accuracy.

Like the material definitions for reflection and transmission, MDL provides an abstraction of the mathematical models implemented in the rendering system to produce volumetric effects. Materials containing a volume distribution function can be combined in the same manner as other materials (as described in "Combining distribution functions" (page 93)), enabling complex behavior from a small set of primitive materials.

5.1.1 Modeling a volume

The two primary MDL parameters describing volumetric interactions define the modification of light as it passes through a *given distance* of the medium. These two parameters answer questions about the behavior of the medium:

Absorption
: How much light in a given distance is absorbed by the medium?

Scattering
: How much light in a given distance is scattered by the medium?

In the simplest case, light is scattered by a particle evenly in all directions, similar to the purely diffuse reflection of light by a surface. However, light may also be scattered unevenly between the forward and backward directions of the light, called *forward scattering* and *backward scattering*. In pure forward scattering, there is no change to the light's direction after it interacts with a particle; in pure backward scattering, the light's direction is reversed, directly backward to the source. In nature, light's interaction with particles is typically a combination of the two scattering directions.

1. `https://cal.cs.umbc.edu/Courses/CS6967-F08/Papers/Reeves-1983-PSA.pdf`

Direction
: In which direction in the medium is the light scattered?

Particles in a physical medium may also emit light. MDL does not model this behavior, but emission must be considered for a complete accounting of the appearance of a volume.

Emission
: Do the particles of the medium emit light, and if so, how much?

5.2 The volume distribution function

The previous chapters often relied on familiar experiences in the physical world to develop an intuition for their materials—reflections in mirrors, the sheen of silk, the indistinct shapes seen through frosted glass. The effect of these materials only depended on the shape of the surfaces to which they were assigned. However, volumetric effects in MDL are dependent on the *distance* that light has traveled through the medium. "Distance" in this case is defined by the scale of the geometric model as represented by its three-dimensional coordinates. One set of parameters for a volume distribution function can therefore produce different visual results from two objects of the same shape if the scale of their numerical representations do not match.

The volume property in a material also displays the *shape of the light emission*, while the illumination of surfaces can only suggest it. This shape can be controlled by a combination of emissive and non-reflecting materials in the same manner as the container used in "Geometric constructions" (page 56). Figure 5.6 shows the geometric structure of the light source used for many of the rendered images in this chapter.

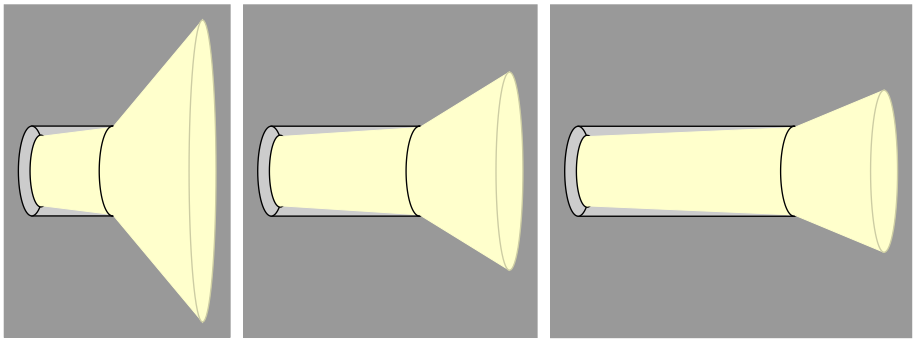


Fig. 5.6 – An emissive polygon inside a non-reflective container

"The spread of light emission" (page 87) also demonstrates controlling the emission shape with the emission distribution function itself.

In addition to considerations of object scale and emission shape, the materials with a volume property differ from surface-based materials in that they can obscure the very effect they produce. To demonstrate these effects of absorption and scattering, the light source and spheres of Figure 5.7 (page 77) are enclosed in a thin rectangular box to provide a somewhat two-dimensional slice of the three-dimensional effects of absorption and scattering. Figure 5.8 (page 77) uses a sub-surface material similar to the glass example described in the beginning of the chapter. For the examples that follow, the enclosing object is oriented as in Figure 5.10 (page 77).

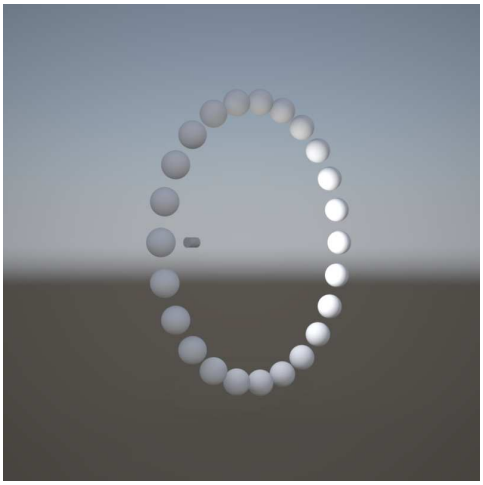


Fig. 5.7 – Set of diffuse reflecting spheres and a light source

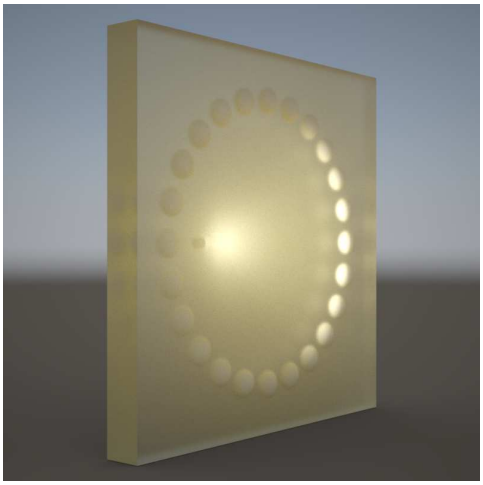


Fig. 5.8 – Spheres and a light source in an object

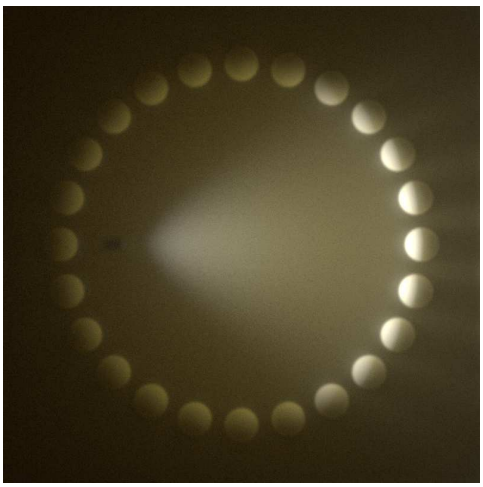


Fig. 5.9 – The orientation for examples

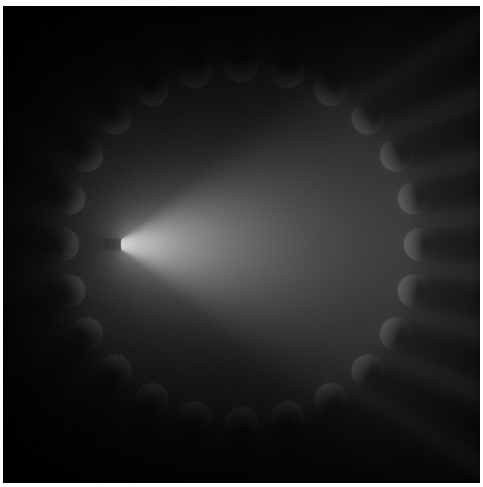


Fig. 5.10 – A simple material with a volume property

To simplify the demonstration of volumetric effects, Figure 5.10 uses the `isotropic_volume` material in Listing 5.1.

Listing 5.1

```
material isotropic_volume(
    color scattering = color(1.0),
    color absorption = color(1.0))
= material (
    surface: material_surface(
      scattering: df::specular_bsdf(
        tint: color(1.0),
        mode: df::scatter_transmit)),
    volume: material_volume(
      absorption_coefficient: absorption,
      scattering_coefficient: scattering));
```

Volume parameters

Surface property to act as a clear, non-refracting container for the volume

Volume property

The `isotropic_volume` material defines a clear, non-refracting surface property and serves as a enclosure for the scattering of light defined in the volume property. By default, light is scattered with equal probability in all directions, thus "isotropic." This material is used in the examples of the next sections.

As a further simplification for the initial demonstrations of the parameters of the volume distribution function, only gray values are used for the parameter values. Color (that is, red, green and blue components with differing values) is added as a parameter to the volume material in sections "Colored scattering" (page 85) and "Colored absorption" (page 86).

5.3 Scattering and absorption

Figure 5.11 shows the combined effect of scattering and absorption using the `isotropic_volume` in Listing 5.1 (page 77).[2]

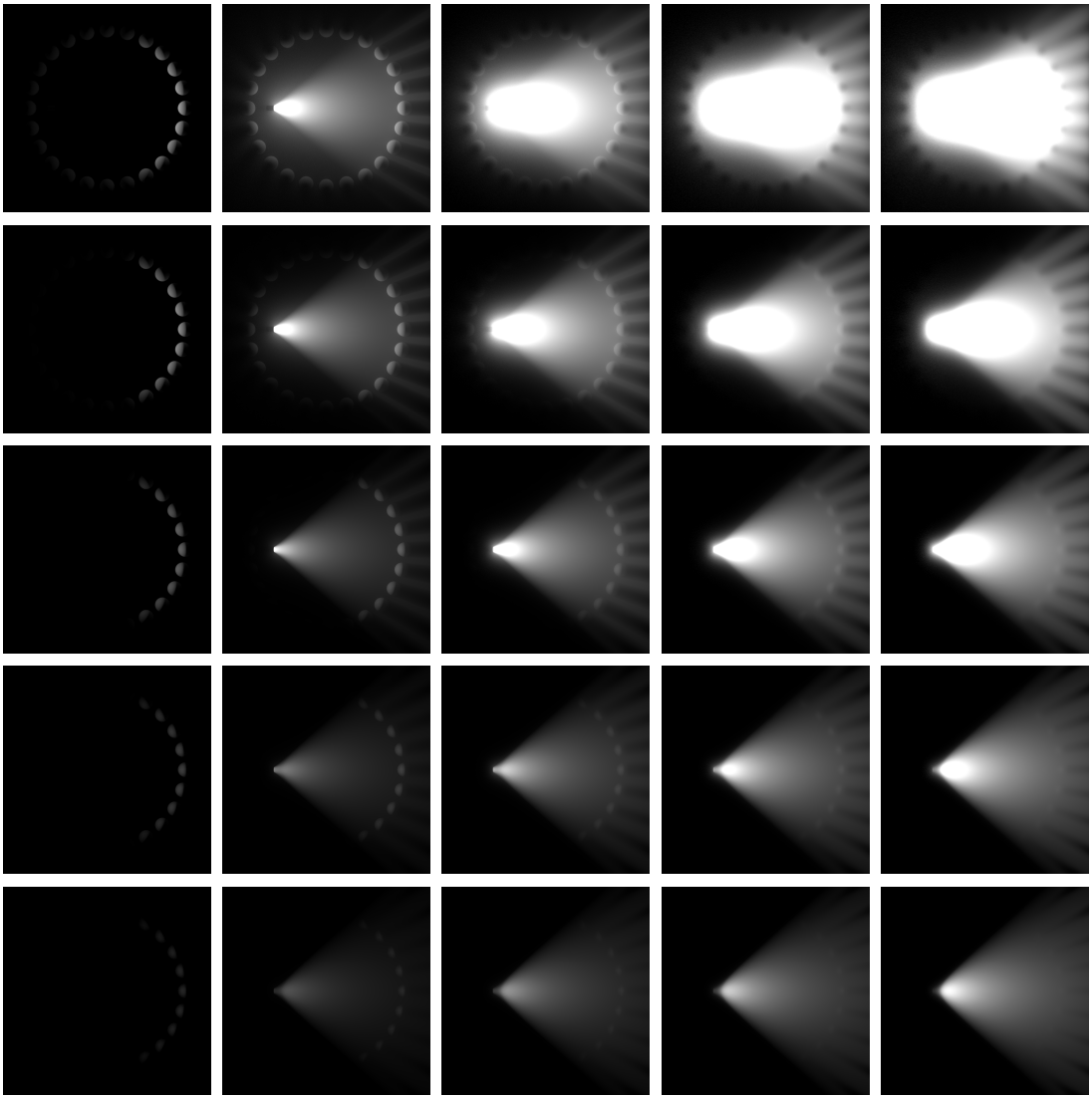


Fig. 5.11 – Scattering increases from left to right; absorption increases from top to bottom

2. The HTML version of this book at `http://mdlhandbook.com/mdl_handbook/` can display the images at higher resolution.

5.4 Scattering direction

In Figure 5.11 (page 78), the `isotropic_volume` material scatters photons from a particle with all possible directions being equally likely. This is the default behavior of the `scattering` parameter of the `material_volume` property. Listing 5.2 defines `scattering` as the volume distribution function `df::anistropic_vdf`. Its single parameter, `directional_bias`, has values from -1.0 to 1.0. A value of -1.0 defines pure backward scattering; 1.0 is pure forward scattering. A value of 0.0 defines the directional behavior of isotropic scattering. The lines with comments in Listing 5.2 are the additions made to Listing 5.1 (page 77) to provide directional control of scattering.

Listing 5.2

```
material anisotropic_volume(
   color absorption = color(1.0),
   color scattering = color(1.0),
   float directional_bias = 0.0,      Control of photon direction after scattering by particle
   uniform color ior = color(1.0))
= material (
   ior: ior,
   surface: material_surface(
      scattering: df::specular_bsdf(
         tint: color(1.0),
         mode: df::scatter_transmit)),
   volume: material_volume(
       scattering: df::anisotropic_vdf(
          directional_bias: directional_bias),   Volume distribution function includes
                                                 directional control
      absorption_coefficient: absorption,
      scattering_coefficient: scattering));
```

Figure 5.12 uses a small emissive polygon facing to the right to show the scattering effects with varying values of the `directional_bias` parameter.

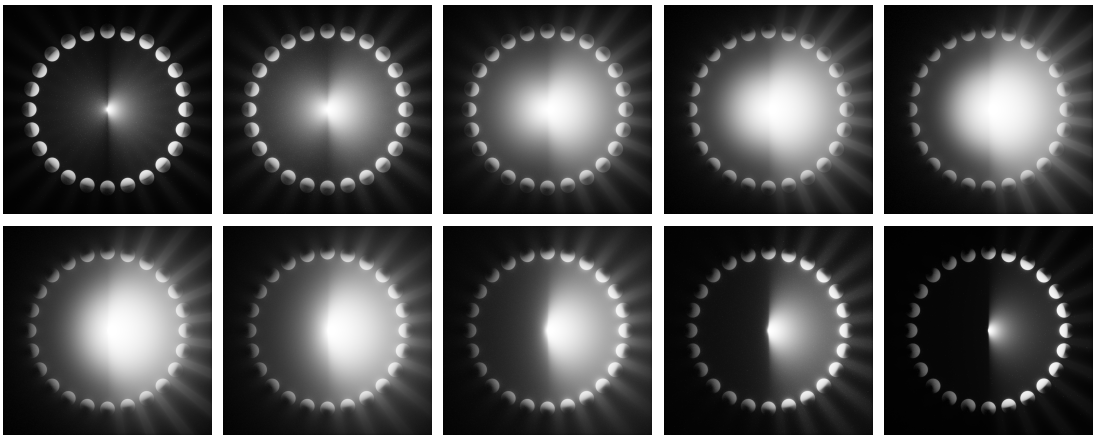


Fig. 5.12 – Parameter `directional_bias` *from -0.99 to 0.99 in even increments*

The rendered images in the rest of this chapter use `isotropic_material` for volumes, but physical simulations of complex anisotropic behavior will also need to take scattering direction into account.

5.5 An experimental basis for volume modeling

Considering the photon-and-particle model of participating media, the images of Figure 5.11 (page 78), after some reflection, may make intuitive sense. But what is the relationship of these varying scattering and absorption parameters to the physical world?

A research paper in 2006 described a means for determining the scattering and absorption parameters based on real-world measurements:

> In this paper, we present a simple device and technique for robustly estimating the properties of a broad class of participating media that can be either (a) diluted in water such as juices, beverages, paints and cleaning supplies, or (b) dissolved in water such as powders and sugar/salt crystals, or (c) suspended in water such as impurities. The key idea is to dilute the concentrations of the media so that single scattering effects dominate and multiple scattering becomes negligible, leading to a simple and robust estimation algorithm.
>
> Srinivasa G. Narasimhan, et al., "Acquiring scattering properties of participating media by dilution",[3] SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, July 2006, pp. 1003-1012.[4]

The researchers created a device with which they could measure the properties of various dilutions of a given medium. Figure 5.13 is an illustration from the paper—two photographs of the device and the accompanying caption:

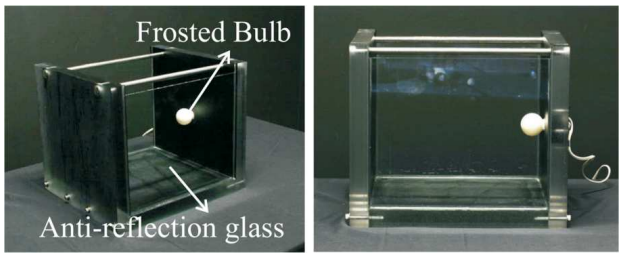


Figure 3: Two views of the apparatus used to measure scattering properties of water-soluble media. A glass tank with rectangular cross-section is fitted with a small light bulb. The glass is anti-reflection coated. Different volumes of participating media are diluted with water in the tank, to simulate different concentrations. A camera views the front face of the tank at normal incidence to avoid refractions at the medium-glass-air boundaries.

Fig. 5.13 – Figure 3 from "Acquiring scattering properties of participating media by dilution"

The researchers photographed the result of various media to estimate its scattering and absorption parameters. Figure 5.14 from the paper demonstrates the difference between high degrees of scattering and absorption:

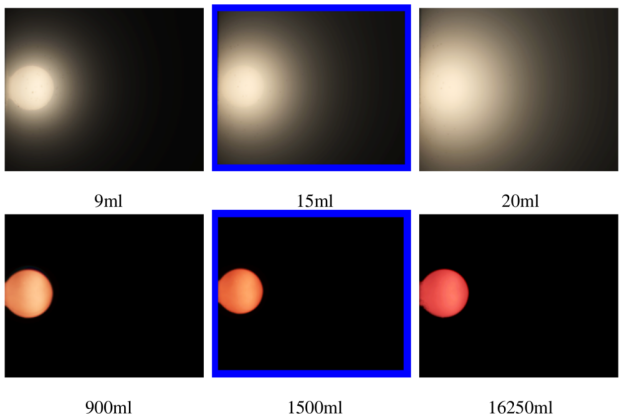


Figure 6: Images illustrating different degrees of scattering and absorption. [Top row] Images of milk at various concentrations. Since milk is a highly scattering liquid, we observe an increase in blurring with increasing concentration. [Bottom Row] Images of red wine at various concentrations. Red wine is a highly absorbing liquid, showing only a saturation of the bulb color with increasing concentration, and no blurring. The highlighted images are chosen for estimating the parameters.

Fig. 5.14 – Figure 6 from "Acquiring scattering properties of participating media by dilution"

3. `https://cseweb.ucsd.edu/~ravir/dilution.pdf`
4. `https://dl.acm.org/doi/abs/10.1145/1179352.1141986`

How are the parameters of the MDL volume property related to the behavior of light in real-world terms? For example, one rendering system implementation of MDL uses the Henyey-Greenstein phase function[5] to implement the effects of scattering, but other implementations are possible. MDL does not define *how* the volume property should be calculated; it only provides control over the "amount" of absorption and scattering, with optional control of the direction of scattering (in the anisotropic case).

Given the complexity of the resulting images, the two volume materials in this chapter are syntactically quite simple. This chapter focuses on ways which the capabilities of MDL volumes can be explored, concentrating on those aspect of volumetric rendering that are valid for any rendering system that implements MDL.

5.6 The volume's enclosing object

To produce an apparent environment with the visual effect of participating media, MDL requires an *enclosure object*. A material is assigned to this object in the same manner as the materials of the previous chapters. For example, the cube in Figure 5.15 is assigned the diffuse reflection material from the section "A material for diffuse reflection" (page 34). The cubes in Figure 5.16 to Figure 5.18 are assigned the `isotropic_material` from Listing 5.2 (page 79), but with different values for scattering and absorption.

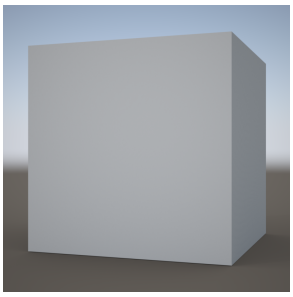


Fig. 5.15 – Diffuse reflection

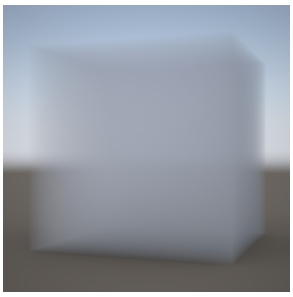


Fig. 5.16 – Scattering only

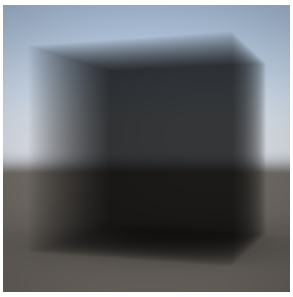


Fig. 5.17 – Absorption only

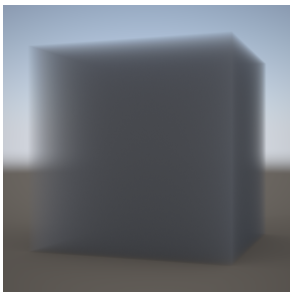


Fig. 5.18 – Scattering and absorption

5.6.1 Scattering

With a light source within the enclosure object, increasing the value of the scattering parameter gradually floods the entire volume with light. The default volume distribution function in the volume property scatters the light with an equal probability in all directions.

Figure 5.19 (page 82) shows the result of increasing the value of the scattering parameter. The absorption parameter has been set to zero.

5. L.G. Henyey, J.L. Greenstein, *Diffuse radiation in the galaxy*, Astrophysical Journal 93:70-83, 1941.

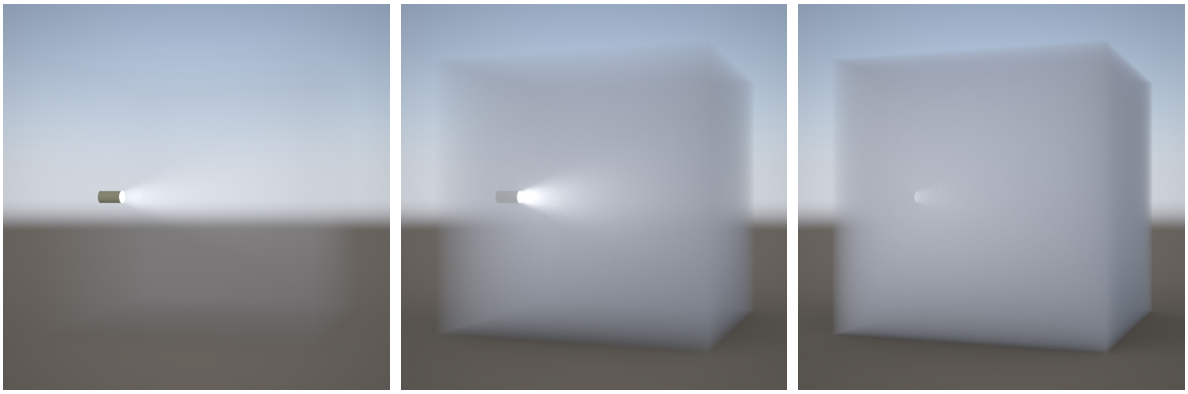


Fig. 5.19 – Increasing scattering

5.6.2 Absorption

With no absorption defined for the volume property, light will continue to be scattered throughout the enclosure object based only on the value of the scattering parameter. However, the light in a *given direction* will be decreased when the scattering parameter increases, as the obscured light source in Figure 5.19 demonstrates.

Absorption is cumulative; the light at a point in the volume enclosure is dependent on how far the "photon" has traveled to reach that point. The higher the scattering value, the more frequently the photon has "collided" with a particle, increasing the distance it has traveled and the absorption of its energy.

Figure 5.20 shows the result of increasing the value of the absorption parameter. Scattering is set to a non-zero value; with a value of zero no light would be visible in the volume enclosure.

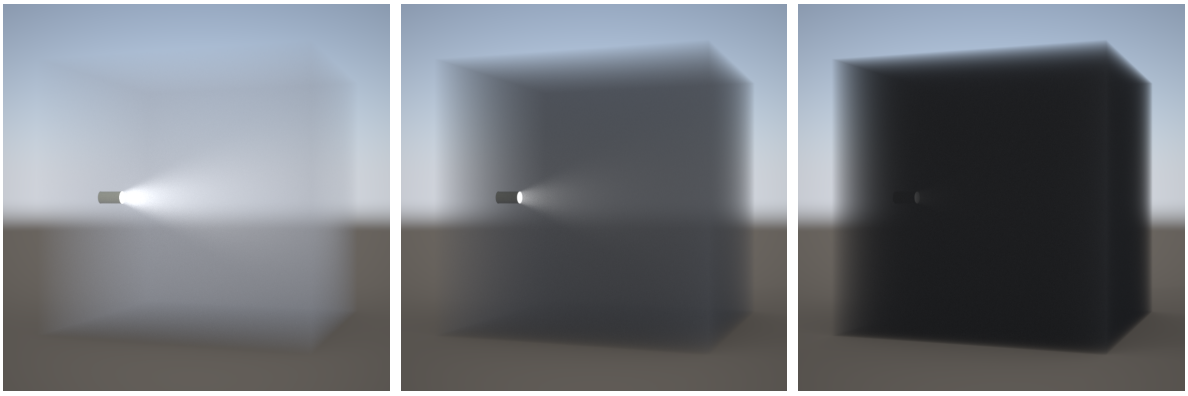


Fig. 5.20 – Increasing absorption

5.7 Volume enclosures as scene elements

If the apparent enclosure for the volume is not transparent, two objects with two different materials are required: one pair for the volume enclosure and one for the object that is part of the rendered scene, a *scene object*. Figure 5.21 (page 83) shows a volume enclosure that is smaller than the scene object. In Figure 5.22 (page 83), the objects are the same size.

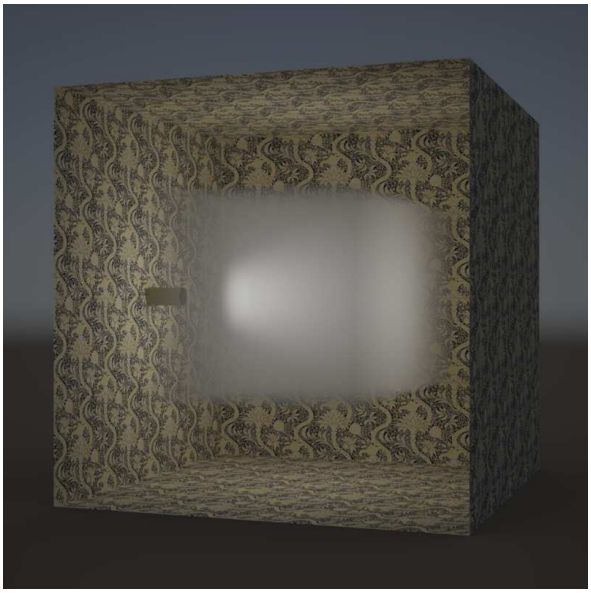


Fig. 5.21 – Scene object and smaller volume enclosure

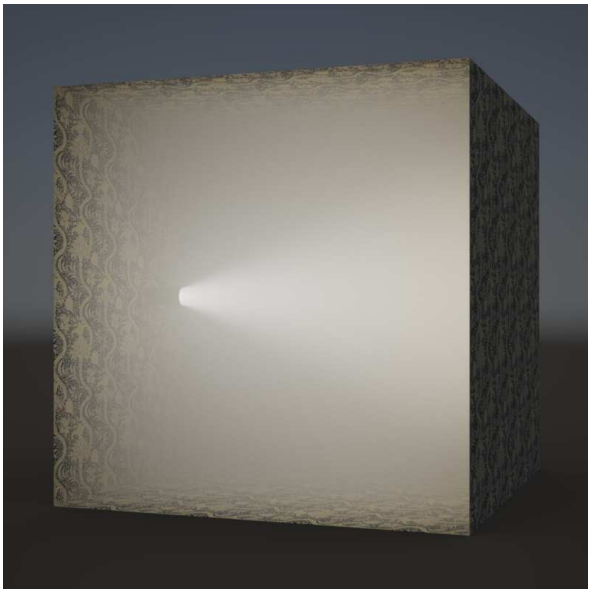


Fig. 5.22 – Matching scene object and volume enclosure

The inside of the scene object Figure 5.22 is much brighter than in Figure 5.21, in which the light reaches the scene object and is then scattered back into the volume enclosure.

5.8 Objects within a volume

The area of a shadow in a volume is made visible to varying degrees by the values of the scattering and absorption parameters. Reflected color may be visible in a volume where an object is illuminated, as in the slightly yellow haze to the left of the yellow block in Figure 5.23 and Figure 5.24. Color bleeding turns the rightmost face of the blue block on the left green.

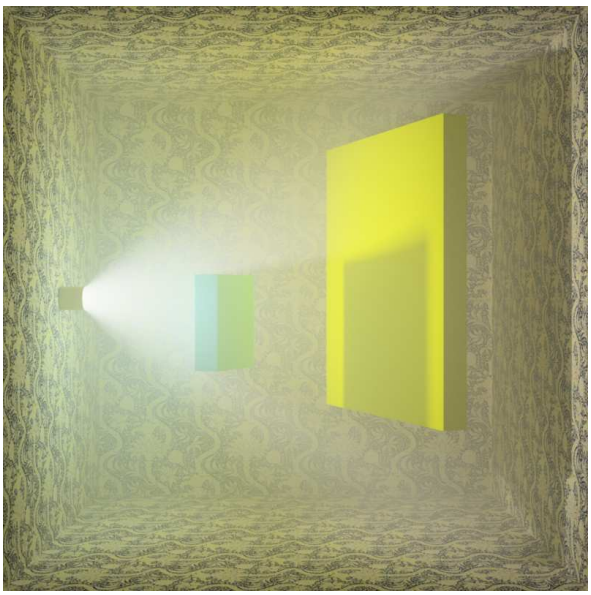


Fig. 5.23 – Diffuse reflection without volume absorption

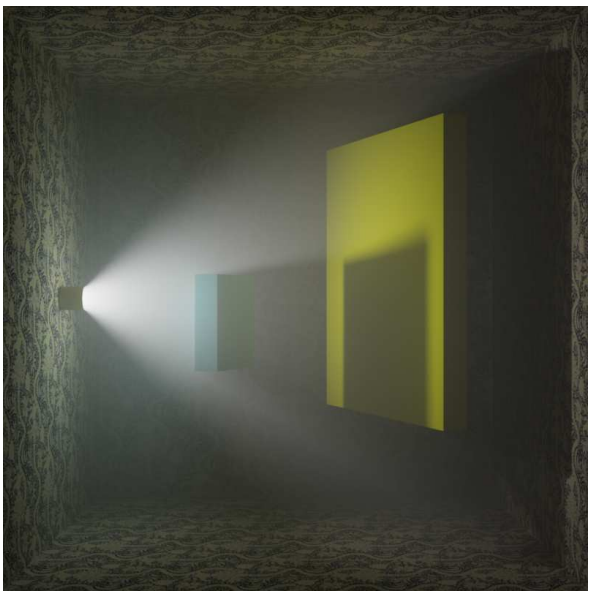


Fig. 5.24 – Diffuse reflection with volume absorption

The effects of an object's color are more easily seen when light is transmitted through an object. In Figure 5.25 and Figure 5.26, the blocks are assigned as specular transmission material with an index of refraction of 1.0

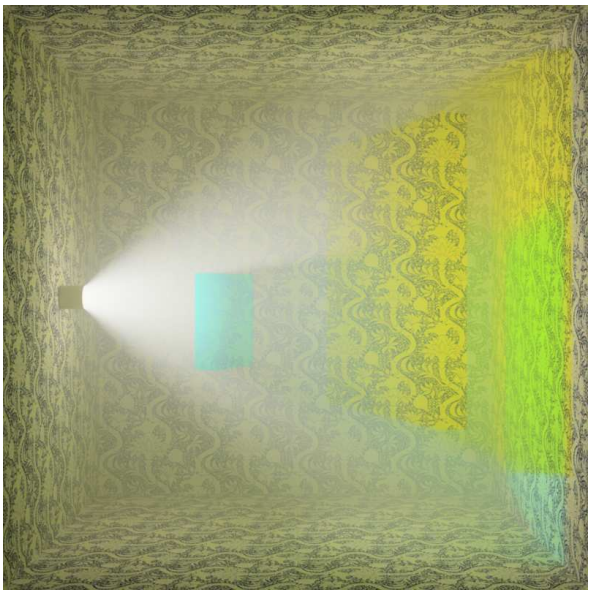


Fig. 5.25 – Transmission without volume absorption

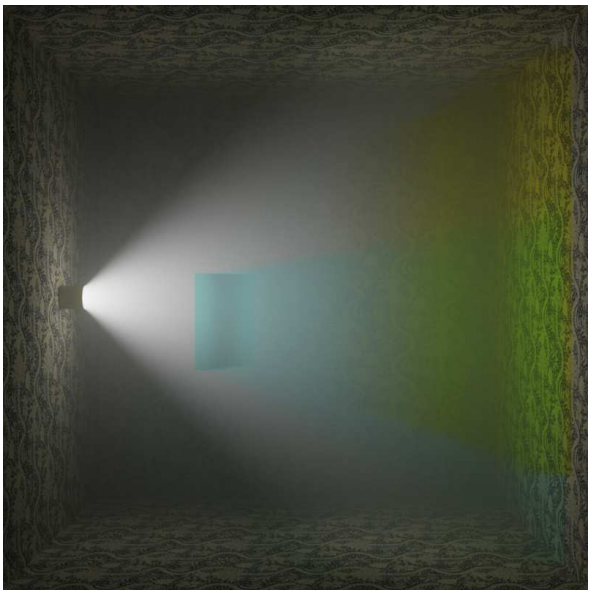


Fig. 5.26 – Transmission reflection with volume absorption

In the same way that the refraction of light that strikes a diffuse surface produces caustics, refracted light in a volume produces *volume caustics*. In Figure 5.27 and Figure 5.28, the material of the lens-shaped objects contains the `df::specular_bsdf` with a `mode` value of `df::scatter_transmit` and an index of refraction of 1.52.

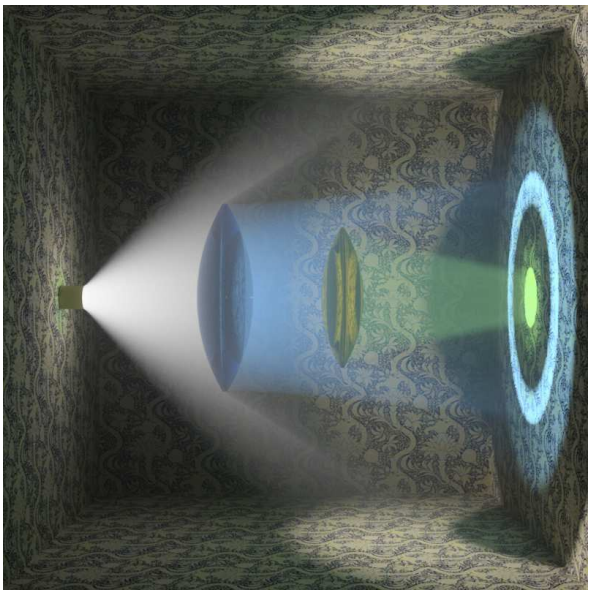


Fig. 5.27 – Volume caustics with low-level scattering

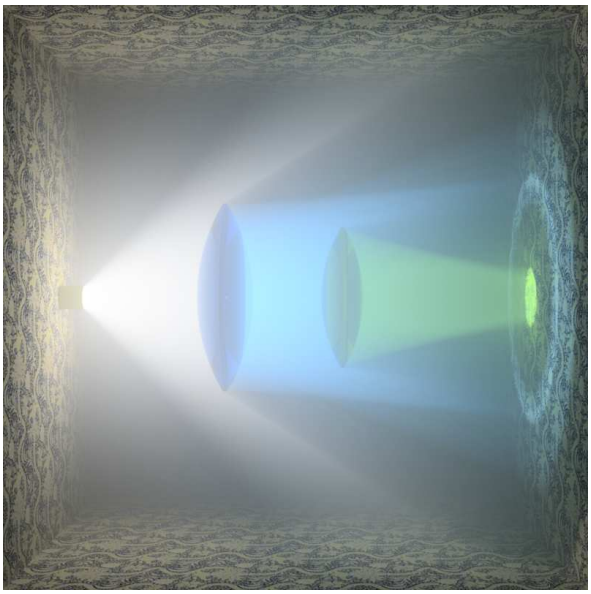


Fig. 5.28 – Volume caustics with high-level scattering

5.9 Light color

Unlike the previous examples, the emissive materials in Figure 5.29 and Figure 5.30 use color for the `intensity` value of the `material_emission` property. In Listing 5.3, the `red_light` material has no parameters, and defines the emission intensity as a color:

Listing 5.3 – Diffuse emission with red intensity

```
material red_light()
= material (
   thin_walled: true,
   surface: material_surface (
      emission: material_emission (
         emission: df::diffuse_edf(),
          intensity: color(1.0, 0.0, 0.0))));   Light "color"
```

The green and blue lights are defined in the same manner. The surfaces struck by the three lights in Figure 5.29 show the various colors produced by combination—cyan, magenta and yellow. The scattered light in Figure 5.30 show the same result in the way the light colors are combined in the overlapping region of the lights.

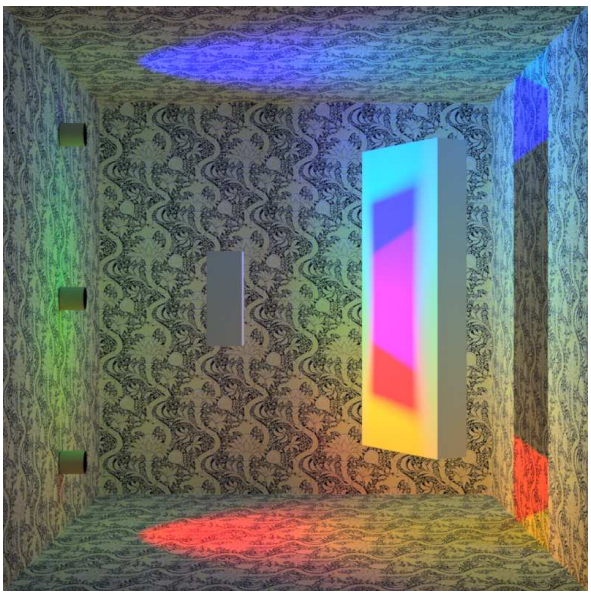


Fig. 5.29 – No scattering or absorption

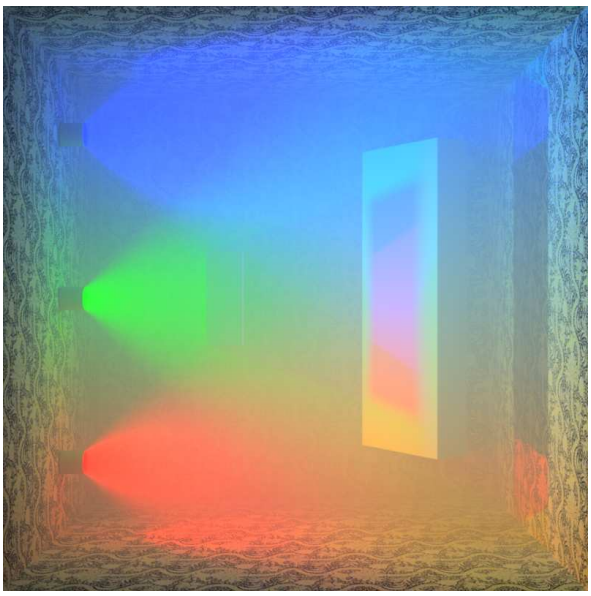


Fig. 5.30 – Scattering and absorption

5.10 Colored scattering

The previous section used colored (non-gray-scale) lights. Figure 5.31 (page 86) and Figure 5.32 (page 86) return to "white" light, but use a color for the scattering parameter instead.

Listing 5.4 – Using blue for the scattering parameter

```
material blue_scattering()
= material (
   surface: material_surface(
```

```
    scattering: df::specular_bsdf(
      tint: color(1.0),
      mode: df::scatter_transmit)),
  volume: material_volume(
    absorption_coefficient: color(0.3),   "White" absorption
    scattering_coefficient: color(0.1, 0.1, 0.15)));   "Blue" scattering
```

Intuitively, the "blue" value for scattering means that more blue light is scattered, so the visible effect of the scattered light becomes "more blue" as the parameter value becomes "more blue."

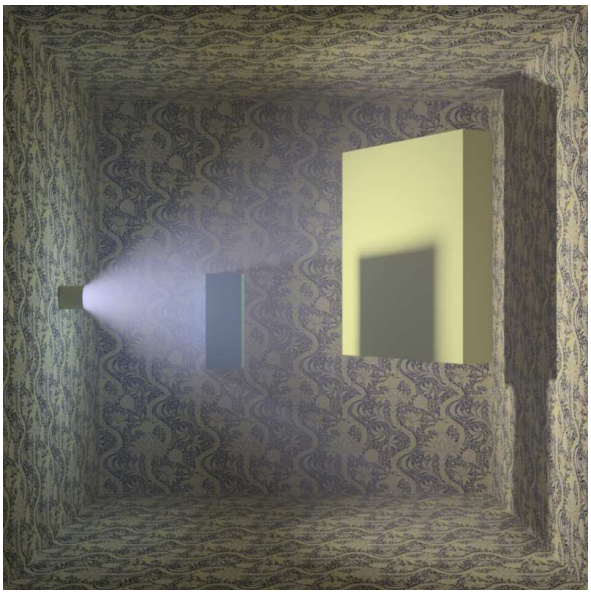


Fig. 5.31 – Scattering: `color(0.1, 0.1, 0.15)`

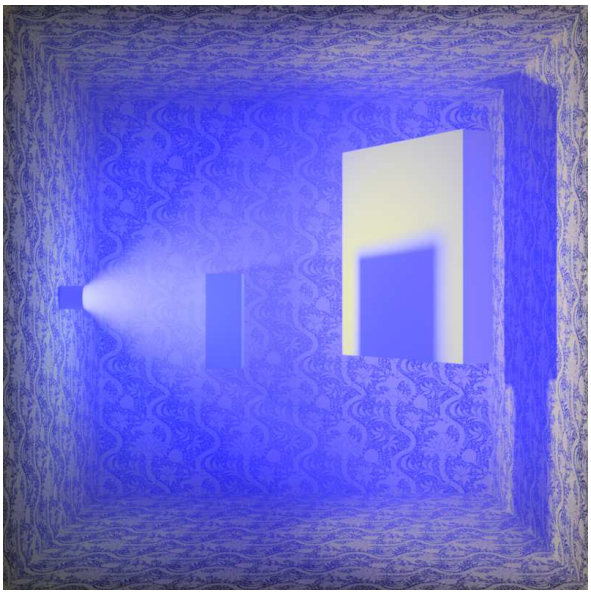


Fig. 5.32 – Scattering: `color(0.1, 0.1, 0.3)`

The previous paragraph begins to have an proliferation of quotation marks; using an arbitrary color as a scattering parameter is not an intuitive concept from our day-to-day experience.

5.11 Colored absorption

Absorption can also be defined as a color, but the parameter value produces the opposite color in the visual result. Removing blue is equivalent to increasing the red and green, resulting in a yellow hue.

Listing 5.5 – Using blue for the absorption parameter

```
material blue_absorption()
= material (
  surface: material_surface(
    scattering: df::specular_bsdf(
      tint: color(1.0),
      mode: df::scatter_transmit)),
  volume: material_volume(
```

```
        absorption_coefficient: color(0.0, 0.0, 0.2),   "Blue" absorption, resulting in yellow
        scattering_coefficient: color(0.5)));           "White" scattering
```

As more blue is absorbed, the color of the volume is dominated by yellow, shown in the difference between Figure 5.33 and Figure 5.34:

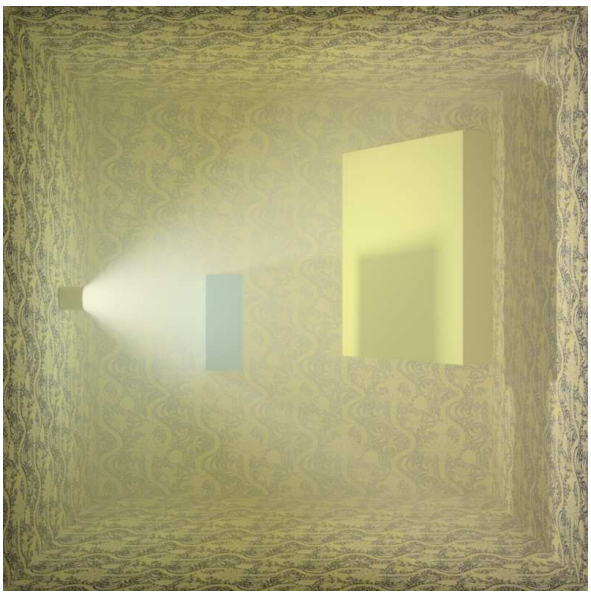


Fig. 5.33 – Absorption: `color(0.0, 0.0, 0.2)`

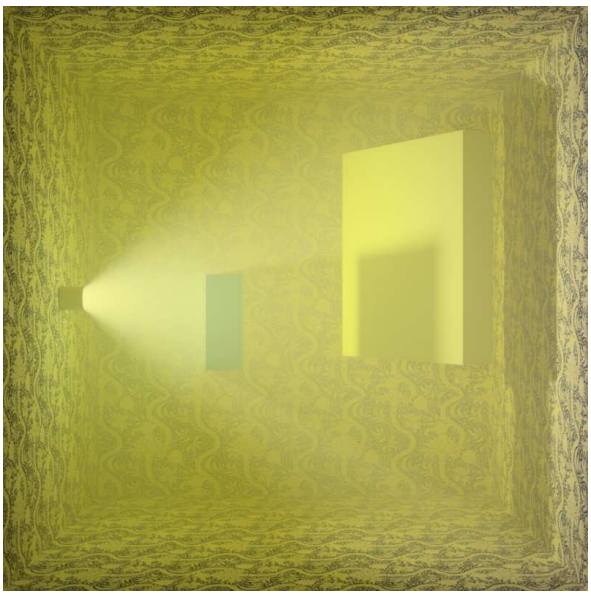


Fig. 5.34 – Absorption: `color(0.0, 0.0, 0.9)`

5.12 The spread of light emission

The light container (page 76) earlier in the chapter shaped the extent of light emission through a non-reflective object that surrounded an emissive polygon. In all the examples so far, light emission has been implemented by the emission distribution function `df::diffuse_edf`. The word "diffuse" has the same meaning here as in "diffuse reflection"—the light is emitted evenly in all directions in the hemisphere above each emissive point.

Listing 5.6 – Diffuse emission

```
material emission (
  color tint = color(1.0))
= material (
  surface: material_surface (
    emission: material_emission (
        emission: df::diffuse_edf(),   Diffuse emission
      intensity: tint)));
```

MDL provides another emission distribution function, `df::spot_edf`, which defines the angle of emission, the light's *spread*. The functions' `spread` parameter is the angle in radians around the normal vector of an emissive point. This radian measure varies from 0 to π. In Listing 5.7, the material `spotlight_emission` normalizes `spread` with the parameter `spread_factor` as a value in the range 0.0 to 1.0.

Listing 5.7

```
material spotlight_emission(
   uniform float spread_factor = 1.0,
   color tint = color(1.0))
= material (
   thin_walled: true,
   surface: material_surface (
      emission:  material_emission (
          emission: df::spot_edf (      "Spotlight" emission
           exponent: 1.0,
             spread: spread_factor * math::PI,    Normalized spread value
           global_distribution: true),
        intensity: tint)));
```

In Figure 5.35, the light object of previous examples is replaced by an emissive polygon that approximates a circle. The `spread` parameter of the ten images varies evenly from 1.0 to 0.1. The spread value of 1.0 in the first image is equivalent to the effect produced by `df::diffuse_edf` as light is emitted evenly toward points in the hemisphere.

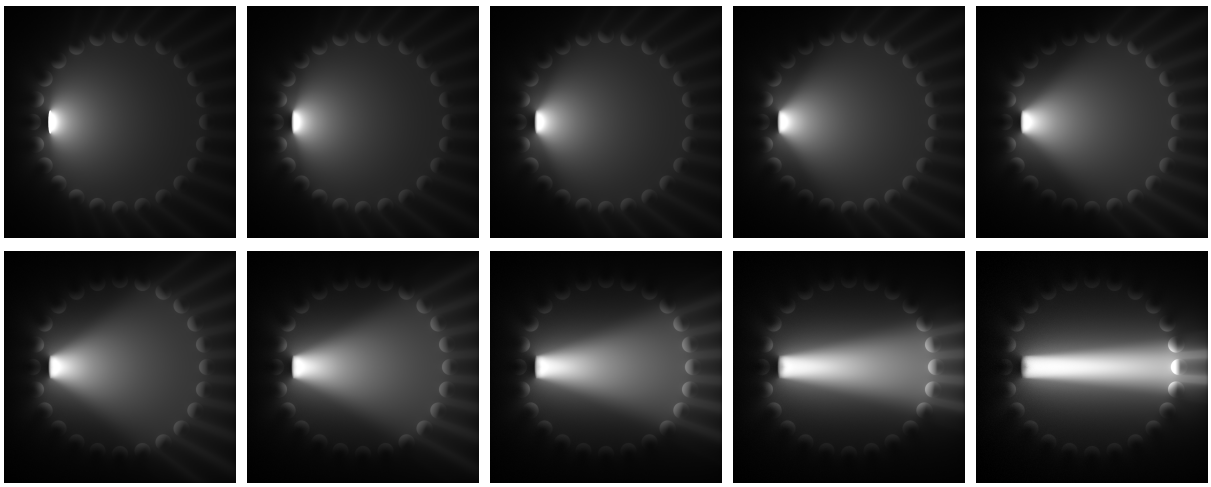


Fig. 5.35 – Using material `spotlight_emission` *with a* `spread_factor` *from 1.0 to 0.1*

Graphical user interfaces for rendering applications typically have many ways of defining light sources in a rendered scene. Some of these features may reflect a legacy of lighting tricks that compensated for the lack of indirect illumination at that time. Combining emissive materials with geometric structures—light containers, bounce cards, rings of small emissive polygons—enable lighting design in a rendering system based on MDL to model traditional photographic practice.

5.13 The removal of obstacles

Without light, objects are not visible, but without objects, the presence of light is unknown. When light interacts with the medium of the environment in MDL, the interaction is not modeled as a series of boundary events, but as an apparently continual modification of light intensity and direction. Now the roles are reversed; the infinitesimal particles give the light form and thereby disappear.

> ...I had a dream which both frightened and encouraged me. It was night in some unknown place, and I was making slow and painful headway against a mighty wind. Dense fog was flying along everywhere. I had my hands cupped around a tiny light which threatened to go out at any moment... Suddenly I had the feeling that something was coming up behind me. I looked back, and saw a gigantic black figure following me... When I awoke I realized at once that the figure was a "specter of the Brocken," my own shadow on the swirling mists, brought into being by the little light I was carrying.
>
> Carl Jung, *Memories, Dreams, Reflections*, 1962.

Part 3 Material combinations

6 Combining distribution functions

The previous chapter presented all the bidirectional scattering distribution functions (BSDFs) defined by MDL. The BSDF value produced by calling a BSDF constructor is an elemental type in the MDL language—as an author of a material, you do not need to be aware of its internal structure, but only of the parameters that control its behavior during the rendering process.

To create appearance models beyond the simple materials of the previous chapter, BSDFs can be combined using *combiner functions*. A BSDF combiner function produces a BSDF, just like the value returned by a BSDF constructor. Both BSDF constructors and combiner functions use the same function-like syntax, with a list of arguments contained within parentheses. However, a BSDF combiner function uses arguments that are themselves BSDFs.

Though the combinations provided by combining functions are only the result of arithmetic operations on numerical values, it is useful to compare these combinations to familiar activities in the physical world:

- *Mixing*—Two or more BSDFs are combined in the manner of mixing different colors of paint, in which the MDL mixing function specifies the proportion of each BSDF to be used in the mix.
- *Layering*—Two BSDFs are combined, with one BSDF considered to be the *base* upon which a *layer* is applied, like the application of a coat of varnish over a painted surface.

Sometimes the arithmetic relationship between two BSDFs can be as easily expressed by mixing as by layering. When simulating an effect in the physical world, you should attempt to mimic its structure in your design of BSDF combinations. Changes—fixing errors, making improvements—are inevitable for a complex combination of BSDFs. Making these changes is easier when you first consider the real-world effect being modeled, which can then suggest how you should change the material. For example, combining two layers evenly or mixing two BSDFs by the same amount will produce equivalent results, but lacquer painted on wood makes much more intuitive sense as a process of layering, and not as one of mixing.

6.1 Layering functions

In the simplest type of BSDF layering, *weighted layering*, a scaling factor, called the *weight*, determines the fraction of the *layer* to be applied over the *base*.

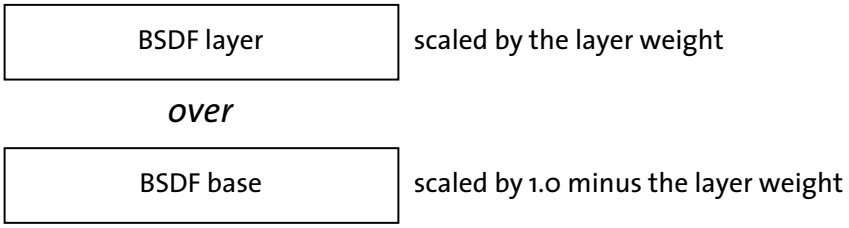


Fig. 6.1 – A layer over a base, scaled by the weighting factor

The other three layering types all similarly define a layer applied over a base based on a weighting factor, but differ in how the fractional amounts of layer and base are calculated.

Layering function	*Effect*
`weighted_layer`	Add BSDF layer based on a weighting factor
`fresnel_layer`	Add BSDF layer based on a weighting factor and a Fresnel term expressed by an index of refraction
`custom_curve_layer`	Add BSDF layer based on weighting factor and directionally dependent curve function
`measured_curve_layer`	Add BSDF layer based on weighting factor and measured reflectance curve

The four layering functions

This chapter first describes layering and mixing two BSDFs. Later sections describe how multiple BSDFs can be combined, as well as suggesting strategies for managing the structural complexity of more elaborate material designs. Later chapters describe the combination of multiple EDFs or VDFs, in which physical principles limit the combination mode to mixing.

6.2 Weighted layering

The syntax of the layering function when it is used in a material is the same as the syntax of the distribution functions—the name of the function, followed by a parenthesized list of arguments, separated by commas.

```
df::weighted_layer (
   weight: layer-fraction,
   layer: BSDF,
   base:  BSDF )
```

For the first example of weighted layering, a `df::simple_glossy_bsdf` will be layered over a `df::diffuse_reflection_bsdf`. These BSDFs can be visualized by using each separately in a minimal material.

To visualize the base argument to `df::weighted_layer`, Figure 6.2 (page 95) is rendered using `df::diffuse_reflection_bsdf`:

```
material red() =
material (
  surface:
    material_surface (
      scattering:
        df::diffuse_reflection_bsdf (
          tint:
            color(0.3, 0.03, 0.05))));
```

Figure 6.2

To visualize the layer argument, Figure 6.3 is rendered using `df::simple_glossy_bsdf`:

```
material shiny() =
material (
  surface:
    material_surface (
      scattering:
        df::simple_glossy_bsdf (
          tint: color(0.15),
          roughness_u: 0.08,
          mode: df::scatter_reflect)));
```

Figure 6.3

In the same way that the two BSDFs were used for the `scattering` argument of the `material_surface` constructor, the BSDFs are now used as arguments to `df::weighted_layer`:

Listing 6.1

```
material shiny_red(
  float shiny_weight = 0.5) =
material (
  surface: material_surface (
      scattering: df::weighted_layer (          Layering function
    weight: shiny_weight,                       Weighting factor
    layer:  df::simple_glossy_bsdf (
      tint: color(.15),
      roughness_u: .08,                         Shiny component
      mode: df::scatter_reflect),
    base: df::diffuse_reflection_bsdf (
      tint: color(0.3, 0.03, 0.05)))));         Diffuse red component
```

The default value of 0.5 for the `shiny_weight` parameter of material `shiny_red` produces an appearance which is composed of equal parts of the two BSDFs in Figure 6.4:

`shiny_red()`

Figure 6.4

Varying the `shiny_weight` argument from 0.1 to 0.9 in increments of 0.1 demonstrates the range of possible effects that the `shiny_red` material can produce.

shiny_weight: 0.1 *shiny_weight: 0.2* *shiny_weight: 0.3*

shiny_weight: 0.4 *shiny_weight: 0.5* *shiny_weight: 0.6*

shiny_weight: 0.7 *shiny_weight: 0.8* *shiny_weight: 0.9*

Fig. 6.5 – Visual effect of varying the weight parameter for the glossy reflection

Note that the simplicity of the arithmetic of the BSDF combination doesn't explain the apparently different substances being rendered—lacquered wood? a metallic finish? It is very hard to predict the perceptual effect of even simple combinations like this example of weighted layering, in which a process that seems to entail matters of degree (the weighting of the layers) become in fact matters of category (the appearance of different substances). More dependable results require materials based on designs that model light interaction in the world—modeling made possible in MDL by the combining functions described in this chapter.

6.2.1 Simplifying a material's structure with temporary variables

Not all of the language features in MDL are concerned with describing appearance. Some features provide means to structure and organize materials to allow for increasing complexity of expression. One such feature is the *let-expression*, a means to simplify larger material definitions using intermediate calculations.

In simpler material definitions, field values in the material struct can only be of four types:

1. Constant values (for example, a number like `0.55`)
2. A value resulting from a constructor (like `color(1,0,0)`)
3. A parameter from the material definition's signature (like argument `shiny_weight` in material `shiny_red`)
4. Combinations of these first three types

In three of these types, the value to be used is expressed directly, either as a constant or a constructor. Only the parameters to the material definition can provide names for values used in the definition of the material.

Listing 6.2

```
material shiny_red (
   float shiny_weight = 0.5) =
material (
    surface: material_surface (
      scattering: df::weighted_layer (
     weight: shiny_weight,
     layer:  df::simple_glossy_bsdf (
       tint: color(.15),
       roughness_u: .08,
       mode: df::scatter_reflect),
     base: df::diffuse_reflection_bsdf (
       tint: color(0.3, 0.03, 0.05)))));
```

Signature with parameters

Material constructor referring to the parameters contained in the signature

A material designed in this way consists of two fundamental parts: the signature containing the material parameters, and the material definition containing field values that can refer to the parameters.

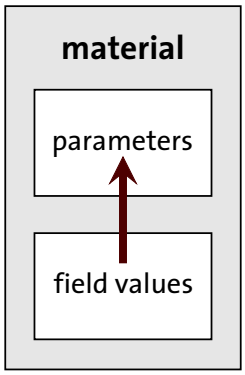


Fig. 6.6 – Field values in the material definition can refer to parameters in the signature

In traditional programming languages, complex calculations can be broken up into more manageable pieces by using symbolic names, called *variables*, to store intermediate values. In MDL, the same simplifying procedure exists through an optional third section to the material definition, the *let-expression*.

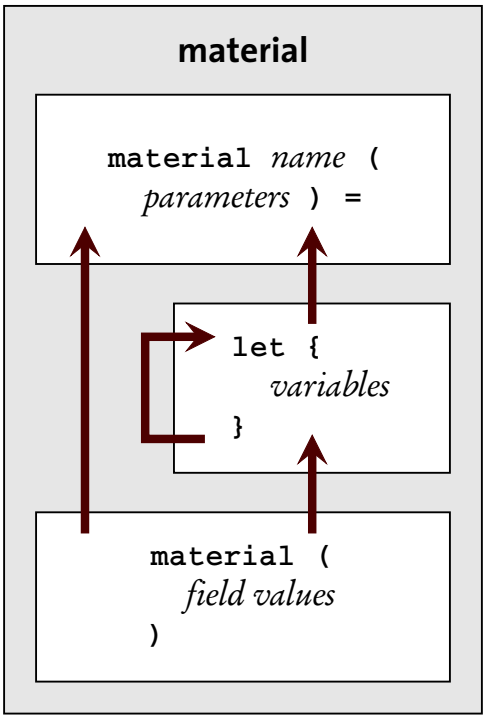


Fig. 6.7 – Temporary variables provide intermediate calculations

The *let-expression variables* can refer to parameters in the signature, as well as to other variables that have been previously defined in the let-expression. Field values can then refer to both these variables as well as the parameters of the material's signature.

Syntactically, the let-expression follows the equals sign in a material definition, and encloses a series of variable definitions separated by semicolons:

```
let {
   data-type variable-name = value-of-type ;
   ...
} in
```

For example, note that the values of the `base` and `layer` fields in the following material, `shiny_red_with_let`, are BSDF constructors. Rather than constructing the BSDF in the material struct itself, the two BSDFs can be defined as let-expression variables of type `bsdf`, and then used as the values of the respective fields. Material `shiny_red_v1` is the result of using let-expression variables in material `shiny_red`:

Listing 6.3

```
material shiny_red_with_let (
   float shiny_weight = 0.5)
=
```

Signature with a single parameter

```
let {
   bsdf shiny_bsdf =
      df::simple_glossy_bsdf (
         tint: color(.15),
         roughness_u: .08,
         mode: df::scatter_reflect);
   bsdf red_bsdf =
      df::diffuse_reflection_bsdf (
         tint: color(0.3, .03, 0.05));
} in
```

Temporary variables defined for use in the material definition

```
material (
   surface: material_surface (
      scattering: df::weighted_layer (
     weight: shiny_weight,
     layer: shiny_bsdf,
     base: red_bsdf)));
```

The material definition using the “shiny_weight” input parameter and the two temporary variables that define BSDFs

These two materials—`shiny_red` and `shiny_red_with_let`—produce the same results in a rendered image. But *how* the material creates that result has been clarified: two BSDFs are constructed and then used as the arguments to `df::weighted_layering`. Better still, the names of the let-expression variables—`shiny_bsdf` and `red_bsdf`—serve to describe the visual intent of those BSDFs. In this simple example, that intent may be easy to see in the version without a let-expression. For more complex materials, however, the naming of intermediate variables can be an important way to find the inevitable errors that complex structures encourage.

6.2.2 Reusing parts of existing materials

The `shiny_red` and `shiny_red_v1` materials use the same BSDF constructors that were shown at the beginning of the chapter, in the materials `red` and `shiny`. Rather than copying an existing field value of a struct, MDL provides the dot operator (described in “Accessing struct components with the dot operator” (page 23)) to refer to the fields within any struct, including the `material` struct.

The dot operator can be used to extract the BSDFs from materials `red` and `shiny`. But unlike the example in Chapter 2, in which field values were extracted from instances saved as variables, the following reimplementation of `shiny_red` extracts BSDFs from instances constructed directly within the field value:

Listing 6.4

```
material shiny_red_extract(
   float shiny_weight = 0.5)
=
let {
   bsdf shiny_bsdf =
      shiny().surface.scattering;

   bsdf red_bsdf =
      red().surface.scattering;
```

Scattering BSDF extracted from material “shiny”

Scattering BSDF extracted from material “red”

```
} in
material (
  surface: material_surface (
    scattering: df::weighted_layer (
   weight: shiny_weight,
    layer: shiny_bsdf,    Using the BSDF of "shiny"

    base: red_bsdf)));    Using the BSDF of "red"
```

At first glance, it appears that the `shiny` and `red` materials have been created only to be taken apart. However, in developing complex materials, being able to visualize components—because they are renderable as complete materials in themselves—can greatly simplify the design and debugging process. But also note that it is not necessary for the original MDL code to be available for those materials—indeed, for the user of those materials to even know how they are defined—to be reused for the BSDFs that they contain.

6.2.3 Parameterizing a layered material

The `shiny_red` material is limited in its design, only allowing the weight of the layering process to be controlled through its `shiny_weight` parameter. Adding more control over a material's behavior by adding additional parameters is called *parameterization*.

To create a more flexible material based on `shiny_red`, the two materials that provide the BSDFs need first to be parameterized. Material `diffuse` is produced by making the `tint` argument of `df::diffuse_reflection_bsdf` a parameter:

Listing 6.5

```
material diffuse (
   color tint = color(0.5))    Parameter "tint" for the diffuse reflection color
=
material (
  surface: material_surface (
    scattering:
      df::diffuse_reflection_bsdf (
          tint: tint)));    Using the "tint" parameter
```

Similarly, the `tint` and `roughness` arguments to `df::simple_glossy_bsdf` are defined to be parameters to create material `glossy`:

Listing 6.6

```
material glossy (
   color tint = color(0.5),    Parameter "tint" for the glossy reflection color

   float roughness = .1)    Parameter "roughness" for the degree of glossy reflection
=
material (
```

```
   surface: material_surface (
      scattering:
         df::simple_glossy_bsdf (
              tint: tint,                    Using the "tint" parameter

              roughness_u: roughness,        Using the "roughness" parameter
            mode: df::scatter_reflect)));
```

Now that the materials used as components to `shiny_red` have been parameterized, a parameterized version of `shiny_red_with_let` (using the dot operator to extract BSDFs) can also be created, here called `glossy_over_diffuse`.

Listing 6.7

```
material glossy_over_diffuse(
    color glossy_tint = color(0.1),
    float glossy_roughness = 0.1,          Glossy parameters
    float glossy_weight = 0.5,

    color diffuse_tint = color(0.5))       Diffuse parameter
=
let {
    bsdf glossy_bsdf = glossy(
      tint: glossy_tint,
      roughness: glossy_roughness          Glossy BSDF value extracted from "glossy" material
        ).surface.scattering;              created here

    bsdf diffuse_bsdf = diffuse(
      tint: diffuse_tint
        ).surface.scattering;              Diffuse BSDF value extracted from "diffuse" material
                                           created here
} in
material (
   surface: material_surface (
       scattering: df::weighted_layer (
           weight: glossy_weight,          Layering
           layer: glossy_bsdf,
           base: diffuse_bsdf)));
```

Now that `shiny_red` has been parameterized to produce `glossy_over_diffuse`, the visual possibilities of this simple layering material can be explored. For example, the `glossy_over_diffuse` parameter values of Figure 6.8 (page 103) create an object that looks like blue porcelain (rather than lacquered and painted wood):

```
glossy_over_diffuse(
  glossy_tint:
    color(0.1, 0.1, 0.08),
  glossy_roughness: 0.2,
  diffuse_tint:
    color(0.7, 0.8, 0.9))
```

Fig. 6.8 – Varying the parameters of material glossy_over_diffuse

Using material `glossy_over_diffuse`—generalized through parameterization—material `shiny_red` can be redefined as a special case of `glossy_over_diffuse`:

Listing 6.8

```
material shiny_red_specialized()
=
glossy_over_diffuse (
   glossy_tint: color(0.15),
   glossy_roughness: 0.08,
   diffuse_tint: color(0.3, 0.03, 0.05));
```

Recall from Chapter 3 that whitespace—the space, tab and newline characters—are ignored in MDL. At first glance, the definition of `shiny_red_v3` may appear to be different in structure than the previous material definitions, but this change in format emphasizes that the syntax for all material definitions corresponds to the variable declaration syntax described in "The structure of a material" (page 17).

data-type variable-name = *variable-value* ;

Similarly, the blue ceramic effect can be defined as a new material called "light_blue_ceramic" by also using `glossy_over_diffuse`.

Listing 6.9

```
material light_blue_ceramic()
=
glossy_over_diffuse (
   glossy_tint: color(0.1, 0.1, 0.08),
   glossy_roughness: 0.2,
   diffuse_tint: color(0.7, 0.8, 0.9));
```

The definitions of `shiny_red_v3` and `light_blue_ceramic` do not have any parameters; a user of those definitions cannot change the internal parameter values of `glossy_over_diffuse` from which these materials are derived. Controlling access to internal state in this manner is called *encapsulation*—the parameters of an existing material have been hidden (encapsulated) in the creation of a new material.

When creating a new material from an existing material, it may be useful to include a subset of the parameters, a technique called *partial encapsulation*. For example, this material partially encapsulates `glossy_over_diffuse`, only allowing the `diffuse_tint` parameter to be modified in the new material:

Listing 6.10

```
material shiny_tint(
    color tint = color(1))    Parameterizing the diffuse color as “tint”
=
glossy_over_diffuse(
    glossy_tint: color(0.15),
    glossy_roughness: 0.08,
    diffuse_tint: tint);      Using the “tint” parameter
```

Partial encapsulations allows complex, highly parameterized materials to serve as the basis of a family of materials, each particularizing the original material for its own purpose. In a production context, a single material may created to implement a complex optical effect—anodized aluminum, car paint with metal flakes and a clear coat, boomerang Formica. Designers are provided with a set of materials that partially encapsulate this complex material, allowing them control over a limited set of the underlying parameters. In an application’s graphical user interfaces, a partially encapsulated material presents the user with a simplified interface, allowing modification of only those aspects of the material that the design should allow.

Section 5 of this chapter develops a complex material and then demonstrates this process of partial encapsulation and the way in which it simplifies the user interface to the material.

6.3 Layering based on the viewing angle

The `df::weighted_layer` function combines the base and layer BSDFs in the same way over the surface of the object, controlled by the weighting factor. In contrast, the `df::fresnel_layer` is *directionally dependent*; it includes the viewing angle at each point on the rendered surface in its calculation.

6.3.1 Background: The refractive index

The directional dependency of `df::fresnel_layer` is based on the *Fresnel equations*, named after Augustin-Jean Fresnel, who derived them in 1823. These equations describe reflection and refraction of light when it strikes a boundary between two transparent media, like air and glass.

When a light strikes such a boundary, the light is typically both reflected and refracted. The proportion of reflection to refraction is determined by two factors: the *refractive index* of both media and the angle at which the light strikes the boundary (the *incident angle*). The higher the ratio of the two refractive indices the more light is bent when it crosses the boundary.

Substance	*Refractive index*
Vacuum	1.0 (by definition)
Air	1.00029
Ice	1.31
Water at 20° Celsius	1.33
Ethyl alcohol	1.36
Fused quartz	1.46
Glycerine	1.473
Acrylic glass	1.490-1.492
Crown (optical) glass	1.52-1.62
Polystyrene	1.55-1.59
Emerald	1.565-1.602
Topaz	1.609-1.643
Sapphire	1.77
Diamond	2.417
Silicon	3.678

Various substances and their refractive indices

Traditionally, computer graphics systems have made the simplifying assumption that objects exist in a vacuum, so that only the refractive index of the object matters (since a number is unchanged by dividing by 1.0). The refractive index of air is close enough to 1.0 so that it, too, can safely be ignored. However, the *ratio* of the refractive indices of an ice cube in water is very close to 1.0, so that the light direction is changed very little (making ice in water harder to see than glass in air).

The Fresnel equations also define the proportion of light that is reflected and refracted at a boundary. In an idealized object (with no absorption or emission), the sum of the reflected and refracted light will equal the incoming light. Both the ratio of refractive indices and the incident angle determine the amount of light that is reflected and the amount that is refracted. The greater the refractive index ratio, the more light will be reflected from the boundary, rather than refracted through it. In addition, the smaller the incident angle, the more light will be reflected. As the angle between the light direction and the surface (the *grazing angle*) approaches zero,

the surface becomes increasingly reflective. At the theoretical limit—when the grazing angle is zero—the surface is only reflective, with no refraction across the boundary.

For example, in this diagram of the reflective and refractive properties of a sphere of optical glass, the length of the two arrows at each incident point represent the relative proportions of reflection and refraction. Very little light is reflected when the light strikes the surface directly (a grazing angle close to 90°). Much more light is reflected when the grazing angle is close to zero (at the top and bottom of the sphere).

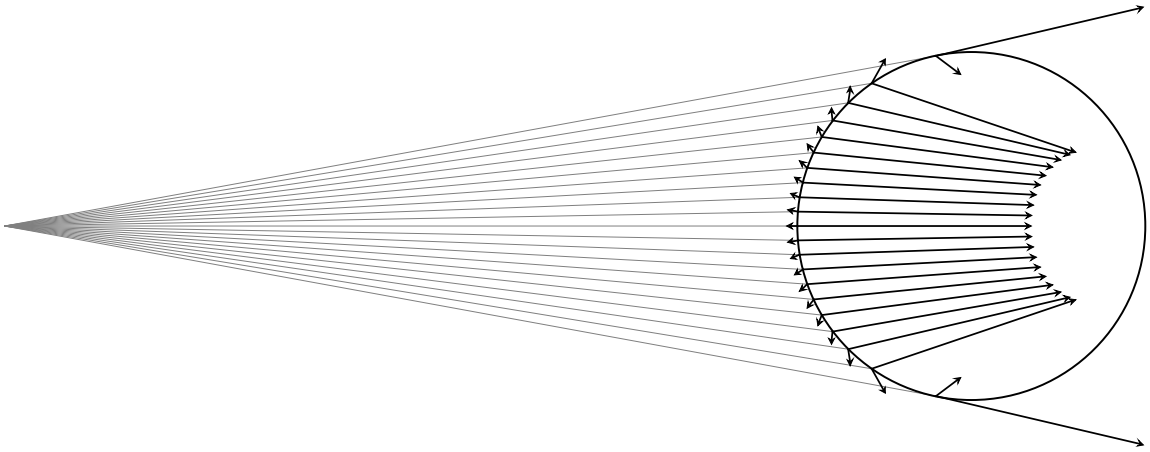


Fig. 6.9 – Reflection and refraction of optical glass in a vacuum, refractive index ratio of 1.52

The same increase in reflection at small grazing angles can be seen in this diagram of a sphere made of diamond. But the higher refractive index ratio causes an increase in reflection for all light directions.

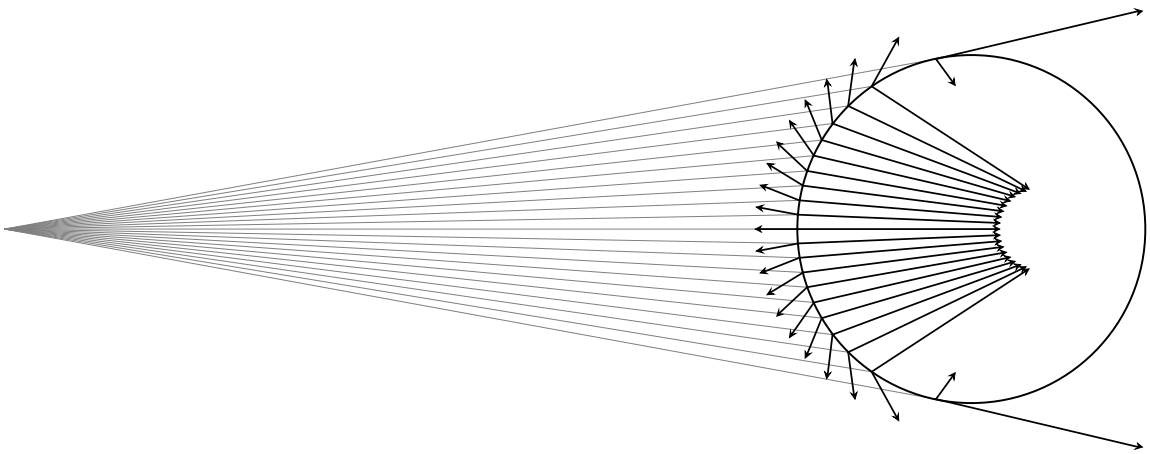


Fig. 6.10 – Reflection and refraction of diamond in a vacuum, refractive index ratio of 2.417

The diagram of diamond shows, in comparison to the diagram of glass, why diamonds sparkle—the surface reflects light at any angle. The Fresnel equations quantify this amount: 17% of the light striking a diamond surface is reflected when the light direction is straight on (when it forms a 90° angle with the surface), but only 4% of the incoming light is reflected by glass at that angle. The facets of a cut diamond make use of this property, providing many surfaces at different angles to increase the chance of reflections reaching the eye of the beholder.

6.3.2 Fresnel layering in a material

The Fresnel equations define light interaction at a boundary: a relationship between refractive indices, angle of light incidence, and the resulting proportion of reflected and transmitted light. But the thin coating of a transparent substance like varnish provides little distance for the change of light direction of the transmitted light—its refraction—to have much of an effect.

MDL models the small influence of refraction in a thin layer with the `df::fresnel_layer` function. The Fresnel calculation of the fractional amount of reflection is used to scale the contribution of the BSDF value supplied as the value of the `layer` argument to `df::fresnel_layer`. To the scaled `layer` value is added the value of the `base` BSDF, scaled by the complement of the reflection factor:

`(layer *` *Fresnel-reflection-factor*`) + (base * (1.0 -` *Fresnel-reflection-factor*`))`

The change of direction from refraction is ignored, so the same point on the surface is used for the evaluation of both the `layer` and `base` BSDFs.

The `df::fresnel_layer` function is similar in structure to `df::weighted_layer`, but adds an *index of refraction* parameter, abbreviated "ior," to the three parameters in `df::weighted_layer`:

```
df::fresnel_layer (
   ior: index-of-refraction-of-layer
   weight: layer-fraction,
   layer: BSDF,
   base: BSDF )
```

Based on the Fresnel equations, the `ior` parameter controls how much the light the `layer` parameter's BSDF reflects, with the incident angle defined by the view from the renderer's virtual camera.

The following two BSDFs will be combined by `df::fresnel_layer` in Figure 6.13 (page 109) at different `ior` values:

```
material diffuse (
  color tint = color(0.5))
=
material (
  surface: material_surface (
    scattering:
      df::diffuse_reflection_bsdf (
        tint: tint)));
```

Figure 6.11

```
material glossy (
  color tint = color(0.5),
  float roughness = .1)
=
material (
  surface: material_surface (
    scattering:
      df::simple_glossy_bsdf (
        tint: tint,
        roughness_u: roughness,
        mode: df::scatter_reflect)));
```

Figure 6.12

Two temporary variables in the let-expression define the glossy BSDF used as a layer and the diffuse BSDF for the base:

Listing 6.11

```
material fresnel_glossy_over_diffuse(
   float glossy_roughness = 0.1,
   color diffuse_tint = color(0.5),
   color ior = color(1.3))                    Index of refraction parameter
=
let {
   bsdf glossy_bsdf = glossy(
      tint: color(0.7),
      roughness: glossy_roughness).surface.scattering;
                                              Glossy layer: the BSDF
                                              from the "glossy" material

   bsdf diffuse_bsdf = diffuse(
      tint: diffuse_tint).surface.scattering;
                                              Diffuse layer: the BSDF from the
                                              "diffuse" material
} in
material (
   surface: material_surface (
       scattering: df::fresnel_layer (        Fresnel layering

          ior: ior,                           Index of refraction to control glossy layer
        weight: 1.0,
          layer: glossy_bsdf,                 Layer affected by the index of refraction layer
        base: diffuse_bsdf)));
```


Rendering a series of images with an increasing `ior` value shows the increasing amount of reflection of the `glossy` layer. For an `ior` value of 1.0, there is no reflection at all. As the `ior` value increases, more and more of the surface becomes increasingly reflective, showing the influence of the glossy layer.

ior 1.0 *ior 1.3* *ior 1.6*

ior 1.9 *ior 2.2* *ior 2.5*

ior 2.8 *ior 3.1* *ior 3.4*

Fig. 6.13 – Varying the index of refraction of the glossy layer

Though the `df::fresnel_layer` ignores the refraction component, the reflection values are consistent with the the Fresnel equations. In a later section, the `df::fresnel_layer` is added as the last layer in a combination of several materials, simulating in a physically based way a clear, protective coating.

6.4 Mixing functions

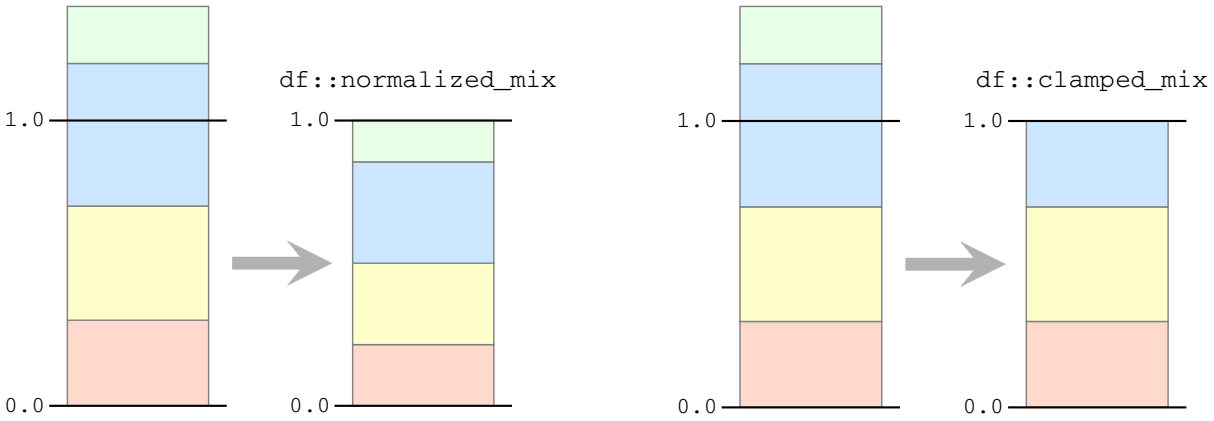
The layering functions define how two BSDFs should be combined with a clear spatial relationship—a layer over a base. In MDL's *mixing functions*, a set of BSDFs are combined in an additive way, like mixing paint. Each component of the mix only defines the contribution of its BSDFs using a weighting factor, a value of type `float` greater than or equal to 0.0.

Because any number of components can be combined in a mixing function and all components can have an arbitrary weighting factor, the two mixing functions are characterized by how they handle a set of components with a sum of weighting factors that exceeds 1.0.

Mixing function	*Modification if weighting factors sum to greater than 1.0*
`normalized_mix`	All weighting factors are scaled proportionally so that their sum is 1.0 before mixing begins.
`clamped_mix`	Components are summed in order until the sum of the weighting factors is greater than 1.0. The weighting factor of the final component that caused the sum to exceed 1.0 is reduced so that the sum is now 1.0. That final component with its reduced weight factor is then used in the mix. All further components are ignored.

The two mixing functions

A picture can clarify the different behavior of the two mixing functions when the sum of their components exceeds 1.0: normalizing retains all the components, but at a different scale; clamping maintains the scale of the components, but does not retain them all.



The normalized mix rescales the values to fit between 0.0 and 1.0

The clamped mix retains the original values but removes the portion above 1.0

Fig. 6.14 – Modifying the sum of mixing components when the sum is greater than 1.0

6.4.1 The syntax of mixing functions

Each component of a mixing function is defined by a struct of two fields: its weight and BSDF.

```
df::bsdf_component (
   weight: fraction-for-component,
   component: BSDF-instance )
```

For example, this `df::bsdf_component` constructor call creates a diffuse reflection component with a weight of 50

```
df::bsdf_component (
   weight: 0.5,
   component: df::diffuse_reflection_bsdf (
      tint: color(0,0,1)))
```

The components are collected in an MDL `array`. An array is an ordered sequence of values of the same data type. A pair of square brackets following a data type defines an array of that type. The constructor function for the array lists the components of the array within parentheses, separated by commas.

array-element-type`[] (`
   array-element-1`,`
   array-element-2`,`
   `...`
   array-element-n `)`

In the two mixing functions, the value of the `components` field is an array of type `df::bsdf_component[]` as in this example of `normalized_mix`:

```
normalized_mix (
   components:
      df::bsdf_component[] (
         component-1,
         component-2,
         ...
         component-n ))
```

For example, this function call of `normalized` mix combines three diffuse reflection BSDF components:

Listing 6.12

```
normalized_mix (
  components:
    df::bsdf_component[] (
        df::bsdf_component (
          weight: 0.3,
          component: df::diffuse_reflection_bsdf (
            tint: color(1,0,0))),
        df::bsdf_component (
          weight: 0.2,
          component: df::diffuse_reflection_bsdf (
            tint: color(0,1,0))),
        df::bsdf_component (
          weight: 0.5,
          component: df::diffuse_reflection_bsdf (
            tint: color(0,0,1)))))
```

Component 1

Component 2

Component 3

Like the layering functions, the `normalized_mix` and `clamped_mix` functions return BSDFs and can therefore be used wherever a BSDF value is required—even as a component in a layering function or in another set of BSDFs combined in a mixing function. An illustration of combinatorial possibilities is provided in the last section of this chapter, where five BSDFs are combined.

6.4.2 Mixing glossy reflections

Chapter 1 described the various types of reflection and transmission at a surface. The shape of the curve that describes relative reflective intensities for glossy reflection from a surface is often described as a "lobe."

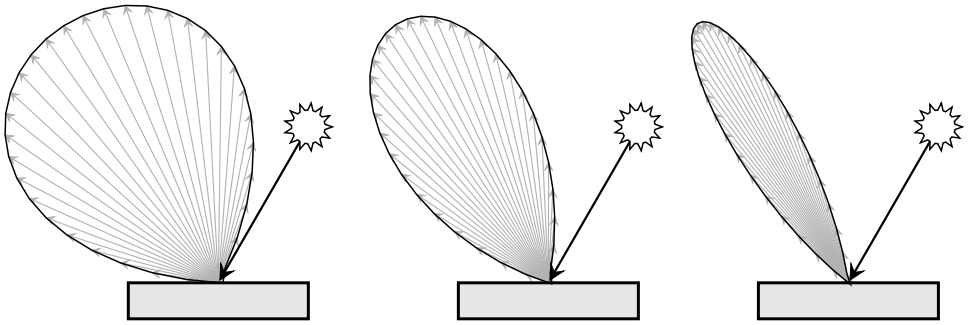


Fig. 6.15 – Characterizing glossy reflection as a "lobe"

As an example of mixing, the material presented below, `two_glossy_lobes`, combines two different types of glossy reflections, with color and roughness for both glossy reflections and their relative weight defined by parameters to the material.

Listing 6.13

```
material two_glossy_lobes (
    float roughness_1 = 0.1,
    float weight_1 = 0.5,
    color tint_1 = color(0.7),                  Parameters for first lobe

    float roughness_2 = .5,
    float weight_2 = 0.5,
    color tint_2 = color(0.7))                  Parameters for second lobe
=
material (
  surface: material_surface (
    scattering: df::clamped_mix (
        components: df::bsdf_component[] (      Array constructor for
                                                "df::bsdf_component"
          df::bsdf_component(
            weight: weight_1,
            component: df::simple_glossy_bsdf (   Component one
              tint: tint_1,
              roughness_u: roughness_1)),
```

```
df::bsdf_component(
  weight: weight_2,
  component: df::simple_glossy_bsdf (   Component two
    tint: tint_2,
    roughness_u: roughness_2))));
```

To demonstrate the effect of mixing two glossy reflections, these parameter values will be used in `two_glossy_lobes`:

Parameter	*Component 1*	*Component 2*
`tint`	`color(0.2, 0.3, 0.25)`	`color(0.6, 0.4, 0.2)`
`roughness`	`0.3`	`0.6`

By setting the weight of one component to 1.0 and the other to 0.0, the individual contribution of both components can be displayed. First, the effect of the first component alone:

```
two_glossy_lobes(
  weight_1: 1.0,
  tint_1:
    color(0.2, 0.3, 0.25),
  roughness_1: 0.3,
  weight_2: 0.0,
  tint_2:
    color(0.6, 0.4, 0.2),
  roughness_2: 0.6)
```

Figure 6.16

Using only the second component in rendering the objects with `two_glossy_lobes` produces Figure 6.17:

```
two_glossy_lobes(
  weight_1: 0.0,
  tint_1:
    color(0.2, 0.3, 0.25),
  roughness_1: 0.3,
  weight_2: 1.0,
  tint_2:
    color(0.6, 0.4, 0.2),
  roughness_2: 0.6)
```

Figure 6.17

The effect of the combination of these two glossy reflections can be shown by systematically varying the relative weights between the two glossy reflection components.

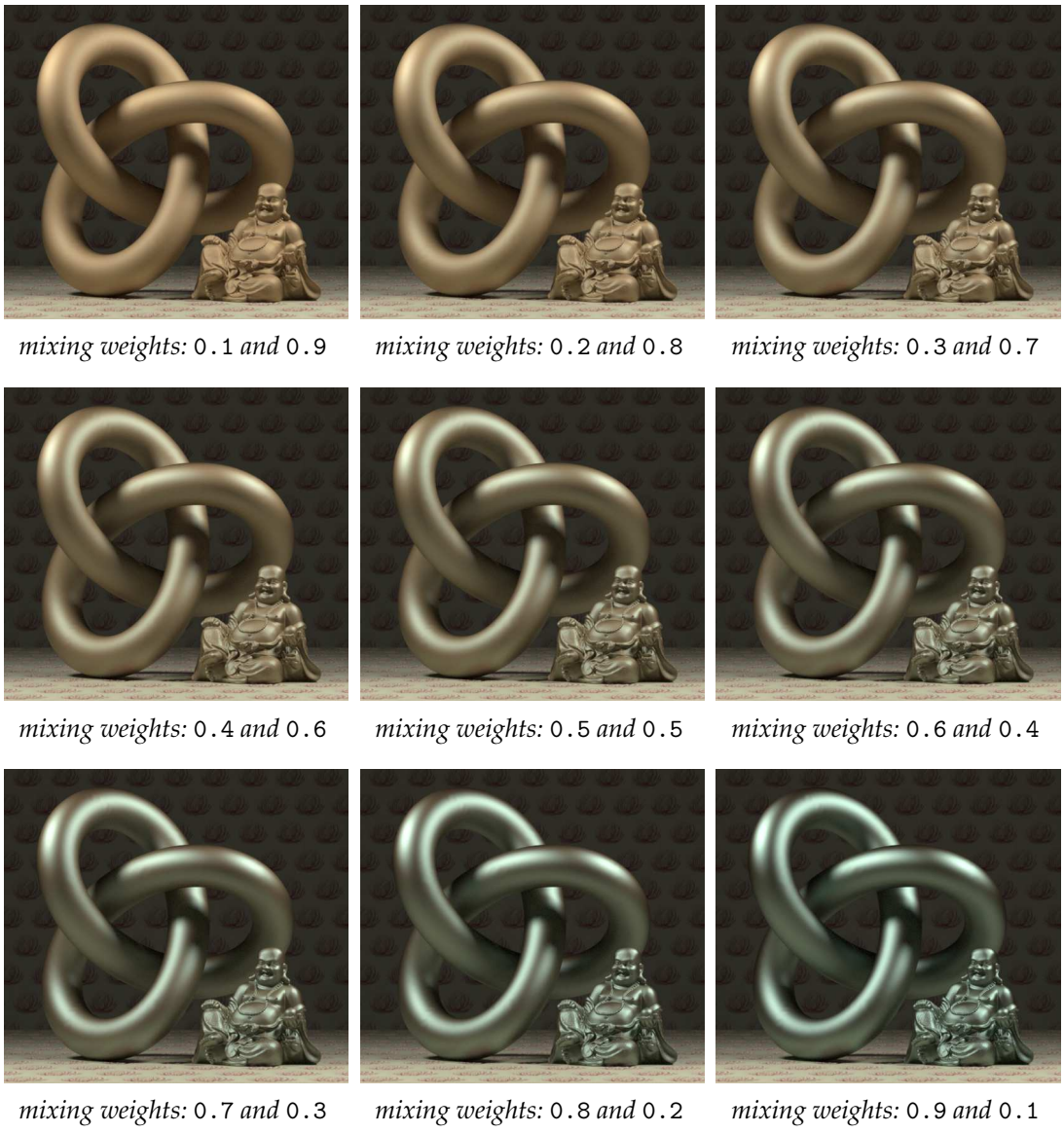


Fig. 6.18 – Varying the relative weights of two glossy reflection components

Controlling the weights of components in a combining function not only provides greater design control over the effect of the material, it also allows—as in the examples above—a way of checking that the individual components are behaving as expected.

6.4.3 An approximation of metal

In the `fresnel_glossy` layering function, the reflective component of the Fresnel equations provides a sense of physical plausibility in the appearance of the resulting image. The design of a successful material may not use actual formulas from physics and yet still base its design on physical principles that provide a higher degree of visual "realism" than are possible in strategies that ignore physics completely. For example, the color of reflection from metals varies based on viewing angle, so that in reality, the reflections from silver aren't simply gray, nor is copper simply brown or gold only yellow.

First, the visually obvious yellow hue of the sharper highlights of the gold reflection, here made visible as in the previous section by setting its component value to 1.0, and the other component to 0.0:

```
two_glossy_lobes(
  weight_1: 0.0,
  tint_1:
    color(0.3, 0.15, 0),
  roughness_1: 0.6,
  weight_2: 1.0,
  tint_2:
    color(0.5, 0.4, 0.1),
  roughness_2: 0.3)
```

Figure 6.19

The other glossy reflection is a darker, orange color, with a greater degree of roughness so that it is visible outside the areas of greatest reflection of the yellow component.

```
two_glossy_lobes(
  weight_1: 1.0,
  tint_1:
    color(0.3, 0.15, 0),
  roughness_1: 0.6,
  weight_2: 0.0,
  tint_2:
    color(0.5, 0.4, 0.1),
  roughness_2: 0.3)
```

Figure 6.20

Mixing the yellow and orange components artistically approximates the complex color behavior of gold reflections:

```
two_glossy_lobes(
  weight_1: 0.5,
  tint_1:
    color(0.3, 0.15, 0),
  roughness_1: 0.6,
  weight_2: 0.5,
  tint_2:
    color(0.5, 0.4, 0.1),
  roughness_2: 0.3)
```

Figure 6.21

By varying the color and roughness of yellow and orange, an artistic simulation of the complex reflective color of gold becomes not just possible, but, with the parameters of `two_glossy_lobes`, easily fine-tuned for artistic purposes. In this example, the solution to the problem of metallic reflection is not so much physically based, as physically inspired.

6.5 Multiple layers

In all the previous examples of this chapter, only two BSDFs were combined, either layered or mixed together in various ways. Because the result of the layering and mixing functions is a value of type BSDF, that result can be used as input to yet another combining function. In this way, layer and mixing can be used to create arbitrarily complex aggregations of simple and compound BSDFs.

For example, the output of `weighted_layer` can serve as the `layer` or `base` of `another` call to the `weighted_layer` function:

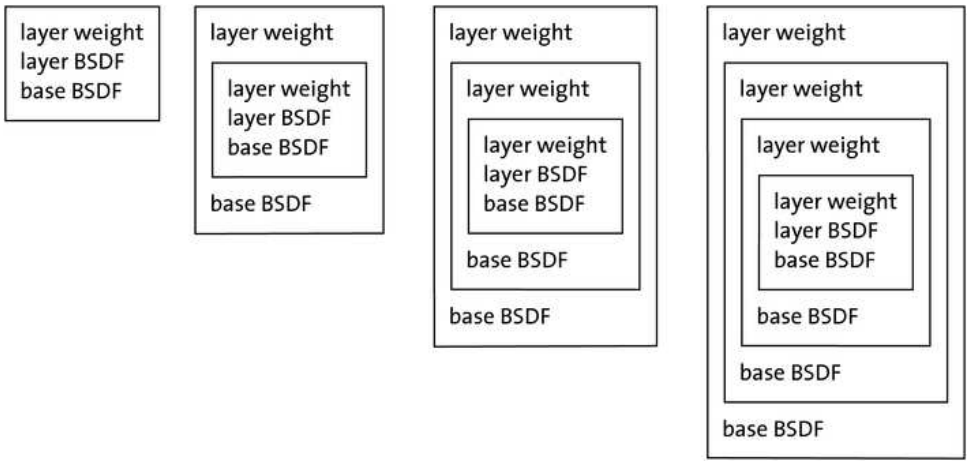


Fig. 6.22 – Nesting layering functions

The nesting of the layering function is also clear from the resulting syntactic structure when the result of `weighted_layer` is used as the value of the `layer` parameter to another call to `weighted_layer` (here made more visible by the symbolic representation of the BSDFs as letters A to E):

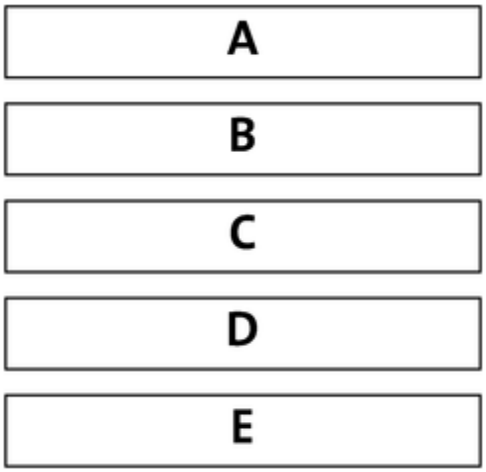


Fig. 6.23 – The vertical relationship of layers is implied by the nesting of distribution functions

```
weighted_layer (
   weight: w4,
   weighted_layer (
      weight: w3,
      weighted_layer (
         weight: w2,
         weighted_layer (
            weight: w1,
            layer: A,
            base: B),
         base: C),
      base: D),
   base: E)
```

The syntactic similarity of the combining functions to the constructor of the `material` struct could obscure other ways of thinking about the relationship of these function calls. Traditional programming languages might favor a more horizontal layout:

```
function(w4, function(w3, function(w2, function(w1, A, B), C), D), E)
```

As the number of combinations of BSDFs increases, part of the material designer's task is to manage this complexity in the way that the materials are organized. The next two sections show two of the simpler organizational principles for building more complex materials.

6.5.1 A linear series of layered materials

To demonstrate a series of layering functions, the following four materials are defined:

1. Diffuse green reflection as base color
2. A dull glossy blue (a glossy with a "wide" lobe)
3. A shiny glossy blue (a glossy with a "narrow" lobe)
4. A mirror reflection for a "clear coat" effect

To simplify the material definitions as well as provide consistency when the same color is desired, three `const` variables of type `color` are defined:

```
const color white = color(0.7);
const color blue = color(0.2, 0.3, 0.7);
const color green = color(0.1, 0.3, 0.05);
```

No parameters are defined in all four of the following materials; the values to be used in layering are explicitly provided as arguments to the BSDF itself.

```
material diffuse_green_base()
= material (
  surface: material_surface (
    scattering:
      df::diffuse_reflection_bsdf (
        tint: green)));
```

Fig. 6.24 – Material 1: Diffuse green reflection as base color

```
material wide_blue_glossy()
= material (
  surface: material_surface (
    scattering:
      df::simple_glossy_bsdf (
        tint: blue,
        roughness_u: 0.5)));
```

Fig. 6.25 – Material 2: A dull glossy blue (a glossy with a "wide" lobe)

```
material narrow_blue_glossy()
= material (
  surface: material_surface (
   scattering:
     df::simple_glossy_bsdf (
       tint: blue,
       roughness_u: 0.2)));
```

Fig. 6.26 – Material 3: A shiny glossy blue (a glossy with a "narrow" lobe)

```
material mirror()
= material (
  surface: material_surface (
    scattering:
      df::specular_bsdf (
        tint: white,
        mode:
          df::scatter_reflect)));
```

Fig. 6.27 – Material 4: A mirror reflection for a "clear coat" effect

four_layers
`fresnel_layer`

three_layers
`weighted_layer`

two_layers
`custom_curve_layer`

In this depiction of BSDF combination, the vertical order of the rendered components suggests different ways of thinking about the process: a thin mirror-like coating over sharp blue highlights, over glossy blue on a base layer of diffuse green. On the other hand, a painter may think in the other direction, with a base layer of matte green, over which blue washes with different degrees of glossiness, finished by a clear-coat layer of varnish.

The first combination layers the wide blue glossy BSDF over the diffuse green base in material `two_layers`:

```
material two_layers () =
   combinations::two_glossy_lobes(
      weight_1: 0.5,
      tint_1: orange,
      roughness_1: 0.6,
      weight_2: 0.5,
      tint_2: yellow,
      roughness_2: 0.3);
```

Figure 6.28

The combination of the first two BSDFs in `two_layers` is used as the `base` value, extracted by the dot operator to create the temporary variable `base` in the let expression of material `three_layers`:

```
material three_layers ()
= let {
  bsdf layer =
    two_layers ()
      .surface.scattering;
  bsdf base =
    diffuse_green_base()
      .surface.scattering;
} in
material (
  surface: material_surface(
    scattering:
      df::custom_curve_layer(
        normal_reflectivity: 0,
        grazing_reflectivity: 1.0,
        exponent: 0.5,
        weight: 1,
        layer: layer,
        base: base)));
```

Figure 6.29

Finally, a mirror-like layer is combined over the BSDF extracted in the let from material `three_layers`:

```
material four_layers ()
= let {
  bsdf layer =
    narrow_blue_glossy()
      .surface.scattering;
  bsdf base =
    three_layers ()
      .surface.scattering;
} in
material (
  surface: material_surface(
    scattering:
      df::weighted_layer(
        weight: 0.3,
        layer: layer,
        base: base)));
```

Figure 6.30

6.5.2 Using combined materials as layers

To combine mixing and layering, the result of the mixing function within `two_glossy_lobes` will be used as a `layer` argument to `df::custom_curve_layer`.

As before, the `tint` parameter values for the components of `two_glossy_lobes` are defined for clarity as `const` variables of type `color`:

```
const color orange = color(0.3, 0.15, 0.0);
const color yellow = color(0.5, 0.4, 0.1);
```

Adjusting the weights of `two_glossy_lobes` to display the orange layer alone produces Figure 6.31 (page 123):

```
two_glossy_lobes(
  weight_1: 1,
  tint_1:
    color(orange),
  roughness_1: 0.6,
  weight_2: 0,
  tint_2:
    color(yellow),
  roughness_2: 0.3)
```

Figure 6.31

Adjusting the weights to display the yellow layer alone produces Figure 6.32:

```
two_glossy_lobes(
  weight_1: 0,
  tint_1:
    color(orange),
  roughness_1: 0.6,
  weight_2: 1,
  tint_2:
    color(yellow),
  roughness_2: 0.3)
```

Figure 6.32

five_layers
`fresnel_layer`

four_layers_v1
`weighted_layer`

three_layers_v1
`custom_curve_layer`

two_layers_v1
`clamped_mix`

The orange and yellow components are combined and used for the `layer` parameter of `custom_curve_layer`, with the diffuse green component as a base. The other two components from the previous example are layered above as before.

The orange and yellow glossy reflections are combined in `two_glossy_lobes`:

```
material two_layers () =
   combinations::two_glossy_lobes(
      weight_1: 0.5,
      tint_1: orange,
      roughness_1: 0.6,
      weight_2: 0.5,
      tint_2: yellow,
      roughness_2: 0.3);
```

Figure 6.33

The BSDF of `two_glossy_lobes` is used as a layer over the diffuse green BSDF, which is used as a base:

```
material three_layers ()
= let {
  bsdf layer =
    two_layers ()
      .surface.scattering;
  bsdf base =
    diffuse_green_base()
      .surface.scattering;
} in
material (
  surface: material_surface(
    scattering:
      df::custom_curve_layer(
        normal_reflectivity: 0,
        grazing_reflectivity: 1.0,
        exponent: 0.5,
        weight: 1,
        layer: layer,
        base: base)));
```

Figure 6.34

The BSDF of `narrow_blue_glossy` is used as a layer over the BSDF of `three_layers`:

```
material four_layers ()
= let {
  bsdf layer =
    narrow_blue_glossy()
      .surface.scattering;
  bsdf base =
    three_layers ()
      .surface.scattering;
} in
material (
  surface: material_surface(
    scattering:
      df::weighted_layer(
        weight: 0.3,
        layer: layer,
        base: base)));
```

Figure 6.35

Finally, the combination of the four components is the `base` for BSDF of `mirror` used as the uppermost layer in Figure 6.36:

```
material five_layers()
= let {
   bsdf layer =
      mirror()
         .surface.scattering;
   bsdf base =
      four_layers ()
         .surface.scattering;
} in
material (
   surface: material_surface(
      scattering:
         df::fresnel_layer(
            ior: color(1.6),
            weight: 1.0,
            layer: layer,
            base: base)));
```

Figure 6.36

Creating separate materials for each incremental combination of the separate BSDFs is very helpful in designing complex materials. Once the basic structure of a complex material has been designed in this manner, further refinements can be made, including the selection of internal parameter values that should be exposed in the signature of the top-level material.

Fig. 6.37 – Beneath a layer, another layer.

7 Plastic

> In the past decade, we have witnessed the development of a number of systems for the rendering of solid objects by computer. The two principal problems encountered in the design of these systems are the elimination of the hidden parts and the shading of the objects. Until now, most effort has been spent in the search for fast hidden surface removal algorithms. With the development of these algorithms, the programs that produce pictures are becoming remarkably fast, and we may now turn to the search for algorithms to enhance the quality of these pictures.
>
> In trying to improve the quality of the synthetic images, we do not expect to be able to display the object exactly as it would appear in reality, with texture, overcast shadows, etc. We hope only to display an image that approximates the real object closely enough to provide a certain degree of realism. This involves some understanding of the fundamental properties of the human visual system. Unlike a photograph of a real world scene, a computer generated shaded picture is made from a numerical model, which is stored in the computer as an objective description. When an image is then generated from this model, the human visual system makes the final subjective analysis. Obtaining a close image correspondence to the eye's subjective interpretation of the real object is then the goal. The computer system can be compared to an artist who paints an object from its description and not from direct observation of the object. But unlike the artist, who can correct the painting if it does not look right to him, the computer that generates the picture does not receive feedback about the quality of the synthetic images, because the human visual system is the final receptor.
>
> Bui Tuong Phong, "Illumination for Computer Generated Pictures,[6]" Communications of the ACM, Volume 18, Number 6, June, 1975.

MDL represents a change of paradigm that requires a reevaluation of many traditional computer graphics ideas, and suggests a return to first principles. One of the venerable lighting models that came to define the "plastic" appearance of early computer-generated imagery was published by Bui Tuong Phong in 1975. This chapter considers the assumptions made by Phong in the development of his lighting model and how they correspond to the simulation of plastic in MDL.

7.1 Background: The Phong model

In his landmark paper of 1975, Bui Tuong Phong made two fundamental contributions to "the rendering of solid objects by computer." The first was an improved *shading function*, defined by Phong in the paper as

> a function which yields the intensity value of each point on the body of an object from the characteristics of the light source, the object, and the position of the viewer. (Bui Tuong Phong, ibid.)

6. `http://dl.acm.org/citation.cfm?id=360839&CFID=597616054`

At that time, the typical shading function depended upon the weighted average of the color calculated at the vertices of a polygon, a method published by Henri Gouraud in 1971 and called *Gouraud shading*. Instead of averaging the colors at vertices, Phong's technique averaged the normal vectors at the vertices of the polygon to calculate the average normal vector of the point to be shaded. *Phong shading*, as this method is now called, produced much better results for "transparency and highlight effects," as shown in this illustration from Phong's 1975 paper:

Fig. 2. An example of the use of Newell, Newell, and Sancha's shading technique, showing transparency and highlight effects.

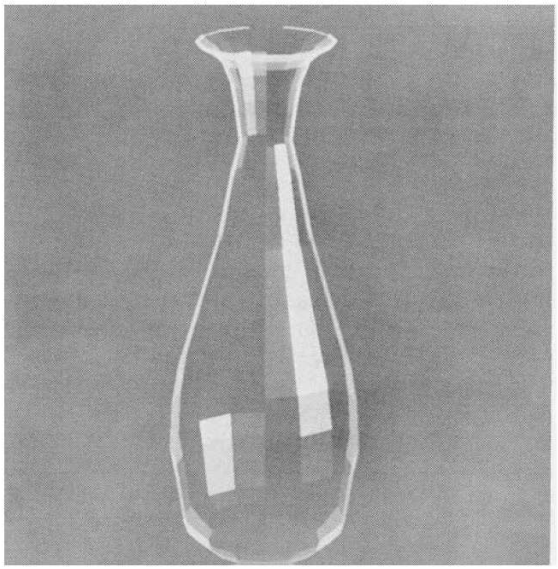


Fig. 9. Improved shading, applied to the example of Figure 2.

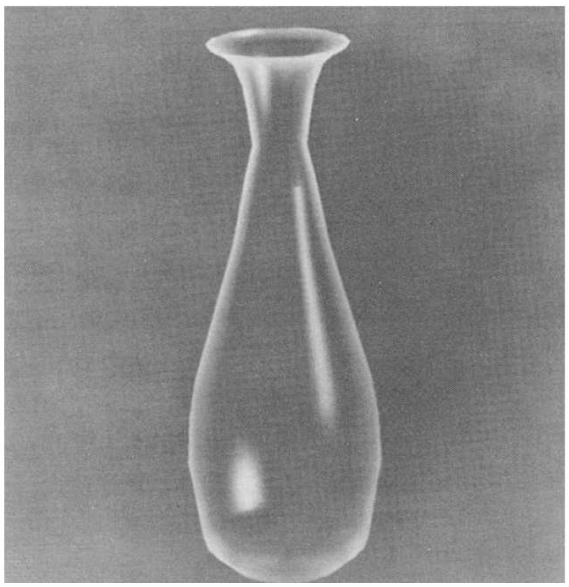


Fig. 7.1 – Bui Tuong Phong,"Illumination for Computer Generated Pictures," 1975

The second contribution of Phong's paper was a method of adding "specular highlights" to the calculation of diffuse reflection. These brighter areas of a rendered object simulate the blurred reflections of light sources modeled as geometric points in space. (Later illumination models, including the model implemented by MDL, reserve the term "specular" for perfect mirror reflections and instead call blurred reflections due to surface roughness "glossy." See "Specular interaction at a surface" (page 45) and and "Glossy interaction" (page 52) for a description of specular and glossy components in MDL.)

Here is Phong's formula from the original paper:

$$S_p = C_p[\cos(i)(1-d)+d] + W(i)[\cos(s)]^n$$

where (quoting from the paper):

- C_p is the reflection coefficient of the object at point P for a certain wavelength.
- i is the incident angle.
- d is the environmental diffuse reflection coefficient.
- $W(i)$ is a function which gives the ratio of the specular reflected light and the incident light as a function of the incident angle i.
- s is the angle between the direction of the reflected light and the line of sight.
- n is a power which models the specular reflected light for each material.

The terms of Phong's equation can be grouped into five categories that define the diffuse and specular components.

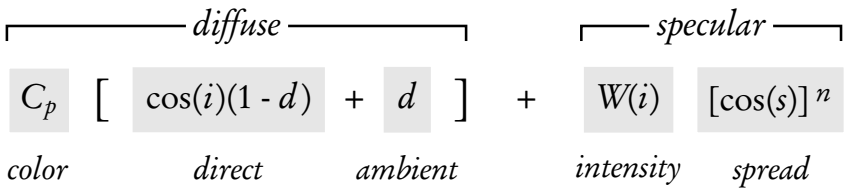


Fig. 7.2 – Categorizing the terms of the Phong equation

The term d, the "environmental diffuse reflection coefficient" is a directionally *independent* scalar value and not a color. Early commercial rendering applications typically called this value the "ambient light"—the light that is present because of environment interreflections—and implemented it as a color value. In Phong's equation, d only increases the intensity of the object's color without regard to the incident angle i, simulating a base level of light striking all surfaces.

A second interesting feature of this equation is that the specular component is a *scalar* value, and, like the environmental term d, is not a color. Adding this scalar value was typically implemented in early rendering systems as a scaling factor of the incoming light color. Coincidentally, highlights that duplicate the color of the light source on a diffuse reflective surface simulate the visual appearance of plastic. Perhaps the unintended look of plastic due to the scalar specular term of the Phong equation helped contribute to the early criticism of computer-generated imagery as cheap and inauthentic.

7.2 The Phong model and MDL

The Phong model makes several assumptions about light-surface interaction that provide a strong contrast with the physically based approach of MDL.

Direct illumination only

The diffuse and specular terms of the Phong equation are calculated as the surface interaction of a source of light, typically modeled as a geometric point without area. Only the direct contribution of the light from the light objects is considered in calculating the diffuse and specular terms. To provide for light reflected from other surfaces, the Phong model adds a constant value—the "environmental diffuse reflection" term d in the Phong equation—that represents perfect diffuse reflection. This term d approximates light reflected from other surfaces as an increase that is constant throughout the scene. This constant value is simply added to the calculation based on the light objects. The interreflections between two objects that are spatially close to each other are not represented by the approximation.

Light objects and the light loop

To calculate the effect of all light objects in the scene, the Phong model assumes an iteration through them to accumulate their individual contributions to the shading of the surface point being rendered. This iteration is often called the *light loop*. MDL does not define a special type of object that emits light nor does it provide a language stucture for iterating through a set of such light objects. Instead, a surface can emit light, defined as a property in the material struct (page 40) that is similar in its MDL representation to the properties of reflection and transmission. Distribution functions in the material properties provide a single mechanism for all light interaction, eliminating both the explicit

accumulation of light object contributions and the need for a compensating approximation of light from other surfaces.

Highlights as the only reflections

The "specular" component in the Phong model is a scalar value that is added to the diffuse color calculation for each light object. As a scalar value, the specular component can only serve to brighten the diffuse color, simulating an additional white light striking that surface point. Though this appears as a reflection of a light source on the surface of the object, no other objects in the scene are reflected.

The final term in the Phong equation, a cosine function raised to a power, controls the spatial area over which the highlight is distributed. The exponential term n, often called the "Phong exponent" in rendering applications, affects the degree to which the light is spread across the surface. Because the blurriness of a reflection can be due to the roughness of the surface, the Phong exponent value is often available in the user interface as a means of artistic control of the surface's apparent roughness.

Empirical derivation of a non-physically based formula

In the paper, Phong describes the calculation of the specular component of the model:

> The function *W(i)* and the power n express the specular reflection characteristics of a material. For a highly reflective material, the values of both *W(i)* and n are large. The range of *W(i)* is between 10 and 80 percent, and n varies from 1 to 10. These numbers are empirically adjusted for the picture, and no physical justifications are made.
> (Bui Tuong Phong, ibid.)

As the parameters of the Phong equation are adjusted, the amount of reflected light at a point can easily be made greater than the total amount of light striking that point. For rendering systems that only calculate direct illumination, arbitrary adjustments made to the reflection parameters of individual objects have no effect on the light received by other objects. (It is in this sense that direct illumination is also called "local illumination.") However, in rendering systems that are based on the physics of light interaction between objects, an object that reflects more light than it receives is adding light energy to the scene, violating the physical principle of energy conservation.

The following sections of this chapter begin with an imitation of the Phong model in MDL, followed by a set of strategies for better physical simulations of the interaction of light and plastic.

7.3 A Phong-like plastic model

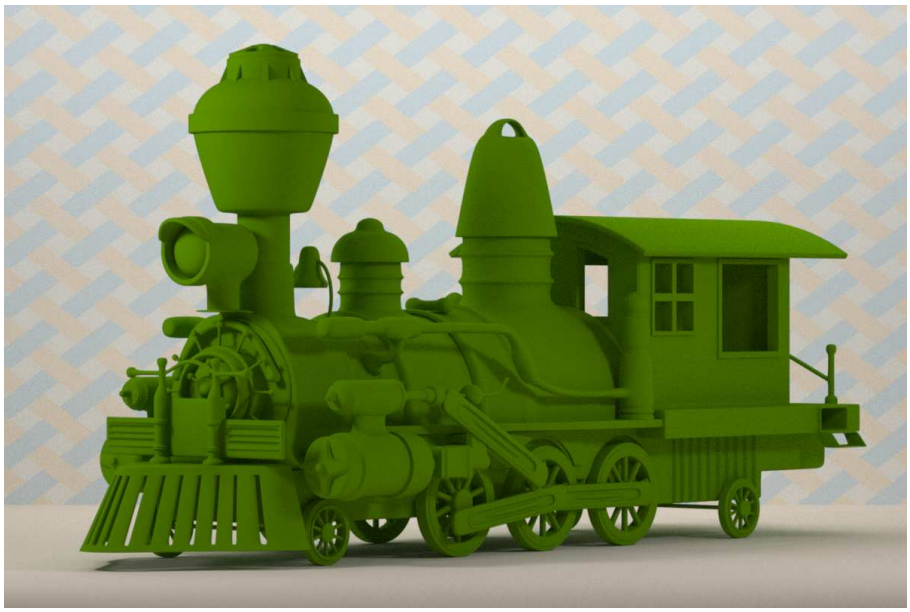
The diffuse and specular components of the Phong model are represented in MDL by the distribution functions `df::diffuse_reflection_bsdf` (page 34) and `df::simple_glossy_bsdf` (page 52). To simulate a "Phong BSDF," these two components need to be combined but in a manner that preserves the principle of energy conservation. The control of the spread of the reflection by the exponent in the Phong equation is represented in the `df::simple_glossy_bsdf` by the relatively more intuitive parameter of `roughness`.

In the following material, called `phonglike_plastic`, intensity parameters provide scaling factors for the diffuse and specular color parameters. The `df::normalized_mix` (page 110) BSDF combining function proportionally scales these factors if they exceed 1.0, providing an implementation of the desired limitation of the reflected light.

Listing 7.1

```
material phonglike_plastic (
    color diffuse_color = color(0.7),                Diffuse color of plastic body
    float diffuse_intensity = 0.8,                   Intensity of the diffuse color
    color highlight_color = color(1),                Color of the highlight reflection
    float highlight_intensity = 0.2,                 Intensity of the highlight color
    float highlight_roughness = 0.2) =               Amount of spread of the highlight
material (
  surface: material_surface (
      scattering: df::normalized_mix (               Mixing function for diffuse color and highlight
        components: df::bsdf_component[] (
            df::bsdf_component (
              weight: diffuse_intensity,
              component: df::diffuse_reflection_bsdf (
                tint: diffuse_color )),              Diffuse component
            df::bsdf_component (
              weight: highlight_intensity,
              component: df::simple_glossy_bsdf (
                tint: highlight_color,
                roughness_u: highlight_roughness ))))));   Highlight component
```

Using the `phonglike_plastic` material with the highlight intensity set to 0.0 produces an image of the diffuse component alone:



```
phonglike_plastic(
  diffuse_color:
    color(.15, .4, 0),
  diffuse_intensity: 0.8,
  highlight_intensity: 0.0)
```

Figure 7.3

Setting the diffuse intensity to 0.0 produces an image that only contains the "highlights" created by the `df::simple_glossy_bsdf` distribution function:

```
phonglike_plastic(
  diffuse_color:
    color(.15, .4, 0),
  diffuse_intensity: 0.0,
  highlight_intensity: 0.2)
```

Figure 7.4

Combining the two components with the `df::normalized_mix` function produces Figure 7.5:

```
phonglike_plastic(
  diffuse_color:
    color(.15, .4, 0),
  highlight_roughness: 0.2)
```

Figure 7.5

The roughness of the surface is controlled by the material's `highlight_roughness` parameter, passed within the material as an argument to the `df::simple_glossy_bsdf` distribution function:

```
phonglike_plastic(
  diffuse_color:
    color(.15, .4, 0),
  highlight_roughness: .05)
```

Figure 7.6

```
phonglike_plastic(
  diffuse_color:
    color(.15, .4, 0),
  highlight_roughness: .3)
```

Figure 7.7

The `phonglike_plastic` material will behave very much like the Phong illumination model. However, function `df::normalized_mix` will adjust the intensity parameters so that their sum is never greater than 1.0. If an application user sets one intensity parameter to a value that would create an intensity sum greater than 1.0, the other component will become darker in a potentially unexpected way.

7.4 Layering a Phong-like plastic model

The previous material, `phonglike_plastic`, implemented the Phong equation's addition of the diffuse and "specular" components by using the `df::normalized_mix` function. A combination of diffuse and glossy components can also be implemented by defining an order for their evalution using the `df::weighted_layer` (page 110) function.

Listing 7.2

```
material phonglike_plastic_layered (
  color diffuse_color = color(0.7),
  color highlight_color = color(1.0),
  float highlight_intensity = 0.2,          Fractional contribution of the highlight component
  float highlight_roughness = 0.2 ) =
material (
  surface: material_surface (
    scattering: df::weighted_layer (        Layering function
      weight: highlight_intensity,          Highlight contribution fraction
      base: df::diffuse_reflection_bsdf (   Diffuse component
        tint: diffuse_color ),
      layer: df::simple_glossy_bsdf (       Highlight component
        tint: highlight_color,
        roughness_u: highlight_roughness ))));
```

Rendering with the `phonglike_plastic_layered` material produces Figure 7.8:

```
phonglike_plastic_layered(
  diffuse_color:
    color(.15, .4, 0),
  highlight_roughness: .25)
```

Figure 7.8

The appearance of objects in the real world is often due to layers of substances with different reflective properties, for example, a coat of varnish over an oil painting or shellac on wood. The changes to the light as it moves through these layers determines the object's ultimate appearance. The explicit order of the `df::weighted_layer` function enables this real-world layering to be intuitively expressed directly in MDL. The highlights of plastic are modeled in `phonglike_plastic_layered` as a glossy layer above the diffuse reflection of a colored base. As the contribution of the highlight layer is increased by increasing the `weight` parameter of

`df::weighted_layer`, the contribution of the diffuse base component is decreased, thereby maintaining energy conservation by keeping the sum of their weights below or equal to 1.0.

7.5 Modeling the dielectric properties of plastic

Plastic is an example of a class of physical materials called *dielectrics*. As a dielectric, the increase in plastic's reflectivity at shallow grazing angles is pronounced. This increased reflectivity of plastic can be modeled with the `df::fresnel_layer` (page 93) combining function. Increased reflectivity from the Fresnel effect is proportional to the index of refraction.

Listing 7.3

```
material dielectric_plastic (
  color diffuse_color = color(0.5, 0.04, 0.04),
  color glossy_color = color(1.0),
  float glossy_weight = 1.0,          Relative weight of the glossy component
  float glossy_roughness = 0.05,
  float ior = 1.54 ) =
material (
  surface: material_surface (
    scattering: df::fresnel_layer (   Fresnel-based layering function
      ior: color(ior),                Index of refraction
      weight: glossy_weight,          Weight parameter to control layer proportions
      layer: df::simple_glossy_bsdf (
        tint: glossy_color,
        roughness_u: glossy_roughness),   Glossy layer
      base: df::diffuse_reflection_bsdf (
        tint: diffuse_color ))));         Diffuse layer
```

Rendering with the `dielectric_plastic` material produces Figure 7.9 (page 138) and Figure 7.10 (page 138):

```
dielectric_plastic(
  diffuse_color:
    color(.15, .4, 0),
  ior: 1.35,
  glossy_roughness: 0.1)
```

Figure 7.9

```
dielectric_plastic(
  diffuse_color:
    color(.15, .4, 0),
  ior: 1.575,
  glossy_roughness: 0.1)
```

Figure 7.10

Real-world measurements for industrial plastics include their index of refraction. An MDL material can accurately simulate the reflective properties of a particular type of plastic by including its measured index of refraction in the `material` struct.

7.6 Modeling plastic with two glossy lobes

The "highlights" of the original Phong model simulate the reflections of small light sources, blurred across the surface by a cosine function raised to a power. Increasing the exponential term creates the appearance of increasingly rougher surfaces and the blurrier reflections they produce. The reflection characteristics of actual plastic may not be so simple, however. A sharp highlight could be surrounded by a blurred reflection that could not be simulated by the exponential property of the Phong specular term.

Complex highlights for plastic can be created by combining more than one glossy distribution function to produce multiple "lobes" as described in section 4.4.2 (page 112). In the following `dielectric_plastic_two_lobe` material, two `df::simple_glossy_bsdf` instances are combined using `df::normalize_mix`. The result is then combined with the diffuse component using `df::fresnel_layer` to create the increased intensity of reflection at shallow grazing angles.

Listing 7.4

```
material dielectric_plastic_two_lobe (
  color diffuse_color = color(0.5, 0.04, 0.04),
  color glossy_color = color(1.0),
  float glossy_roughness_1 = 0.05,
  float glossy_weight_1 = 0.5,
  float glossy_roughness_2 = 0.25,
  float glossy_weight_2 = 0.5,
  float glossy_weight = 1.0,
  float ior = 1.54 ) =
material (
  surface: material_surface (
    scattering: df::fresnel_layer (
      ior: color(ior),
      weight: glossy_weight,
      layer:  df::normalized_mix (
        components: df::bsdf_component[] (
          df::bsdf_component (
            weight: glossy_weight_1,
            component: df::simple_glossy_bsdf (
              tint: glossy_color,
              roughness_u: glossy_roughness_1 )),
          df::bsdf_component (
            weight: glossy_weight_2,
            component: df::simple_glossy_bsdf (
              tint: glossy_color,
              roughness_u: glossy_roughness_2 )))),
      base: df::diffuse_reflection_bsdf (
        tint: diffuse_color ))));
```

Parameters of the first lobe

Parameters of the second lobe

Fresnel layering to combine glossy and diffuse components

Overall glossy component created from two glossy BSDFs combined using "df::normalize_mix"

Diffuse component

Rendering with the `dielectric_plastic_two_lobe` material produces Figure 7.11 (page 140) and Figure 7.12 (page 140).

```
dielectric_plastic_two_lobe(
  diffuse_color:
    color(.15, .4, 0),
  ior: 1.575,
  glossy_roughness_1: 0.05,
  glossy_roughness_2: .2)
```

Figure 7.11

```
dielectric_plastic_two_lobe(
  diffuse_color:
    color(.15, .4, 0),
  ior: 1.5894,
  glossy_roughness_1: 0.1,
  glossy_roughness_2: .35)
```

Figure 7.12

Extending the `dielectric_plastic_two_lobe` material to more than two lobes is straightforward: for each additional component in the `df::normalized_mix`, two additional parameters for roughness and weight are added to the parameters of the material. The parameter adjustments of multiple lobes in order to achieve a particular appearance might require tweaking by eye in an interactive application, in some ways a return to the empirical origins of the Phong model.

7.7 Adding translucency to plastic

All the previous materials of this chapter have simulated the interaction of light with plastic as purely a *surface* effect. Based on the chemical constitution of the plastic, some amount of light may penetrate the surface, creating an effect of *translucency*. The appearance of translucency

can be added to the `dielectric_plastic_two_lobe` material of the last section by developing a material that implements diffuse transmission (as in section 3.4 (page 38)) and combining it with the two-lobe material.

The following material uses `df::diffuse_transmission_bsdf` to simulate translucency:

Listing 7.5

```
material translucency(
  color transmission_color = color(1)) =
material (
  thin_walled: true,          Refraction disabled with thin-wall model
  surface: material_surface (
    scattering: df::diffuse_transmission_bsdf (
      tint: transmission_color)));          Diffuse transmission BSDF
```

Rendering with this simple material produces Figure 7.13:

```
translucency(
  transmission_color:
    color(0.95, 1.0, 0.9))
```

Figure 7.13

To combine the visual properties of `dielectric_plastic_two_lobes` with `translucency`, a new material must combine the "scattering" BSDFs of the materials defined by their `surface` properties. To extract the BSDF of a material instance, the dot operator is used in combination with the field names (as described in "Reusing parts of existing materials" (page 100)).

For example, the following material, which combines the opaque appearance of `dielectric_plastic_two_lobes` with the translucent appearance of `translucency`, uses the dot operator in this way:

```
bsdf name = material-instance .surface.scattering;
```

For clarity, the dot operator's extraction of each material's BSDF is placed on a line after the closing parenthesis for clarity; this is possible because whitespace is ignored (page 33) in MDL.

Listing 7.6

```
material translucent_plastic (
  color diffuse_color = color(0.5, 0.04, 0.04),
  color glossy_color = color(1.0),
  float glossy_roughness_1 = 0.05,
  float glossy_weight_1 = 0.5,
```

Parameters of the first lobe

```
  float glossy_roughness_2 = 0.25,
  float glossy_weight_2 = 0.5,
```

Parameters of the second lobe

```
  float glossy_weight = 1.0,
  float ior = 1.54,
  float transmission_weight = 0.2,
  color transmission_color = color(0.9),
  float transmission_exponent = 3.0) =
```

Parameters of the translucency component

```
let {
  bsdf opaque = dielectric_plastic_two_lobe(
    diffuse_color: diffuse_color,
    glossy_color: glossy_color,
    glossy_roughness_1: glossy_roughness_1,
    glossy_weight_1: glossy_weight_1,
    glossy_roughness_2: glossy_roughness_2,
    glossy_weight_2: glossy_weight_2,
    glossy_weight: glossy_weight,
    ior: ior)
    .surface.scattering;
```

BSDF of opaque (non-transmissive) surfaces extracted from material “dielectric_plastic_two_lobe”

```
  bsdf translucent = translucency(
    transmission_color: transmission_color)
    .surface.scattering;
```

Translucency BSDF extracted from material “translucency”

```
} in
material (
  thin_walled: true,
  surface: material_surface (
    scattering: df::custom_curve_layer (
      weight: 1.0,
      normal_reflectivity: 1.0 - transmission_weight,
      grazing_reflectivity: 1.0,
      exponent: transmission_exponent,
      layer: opaque,
      base: translucent)));
```

Opaque and translucent components combined with “df::custom_curve_layer”

Using the `translucent_plastic` material produces Figure 7.14 (page 143):

```
translucent_plastic(
  diffuse_color:
    color(.15, .4, 0),
  ior: 1.5894,
  glossy_roughness_1: 0.1,
  glossy_roughness_2: .35,
  transmission_weight: 0.5,
  transmission_color:
    color(0.95, 1.0, 0.9))
```

Figure 7.14

Using the dot operator to extract a BSDF from a fully developed and independently useful material demonstrates the value of material libraries and the recombinations they promote.

In the next two chapters, topics introduced in the exploration of the Phong model will be developed further. Glass (page 59) is another dielectric material with similar surface properties that can be augmented by volume distribution functions. The technique of material combination through the extraction of BSDFs will provide translucency to a model of woven fabric (page 145).

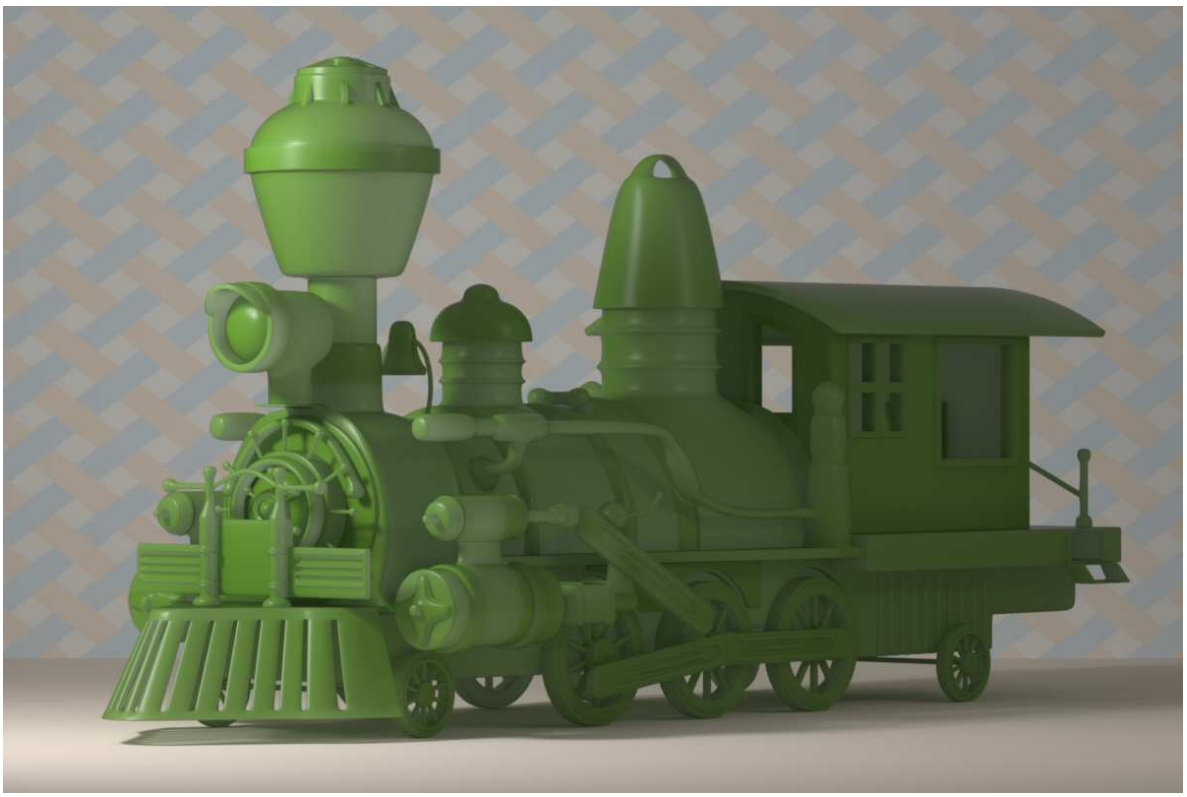


Fig. 7.15 – Material `translucent plastic`

8 Fabric

The previous two chapters presented the set of BSDFs defined by MDL and the functions that can combine them. With these ideas in place, the development of an MDL material can proceed in the other conceptual direction: rather than starting with elemental MDL building blocks to see what can be produced, the material designer can begin with a visual effect in the physical world, analyze it, break it into simpler parts, and investigate how an MDL material could simulate that visual effect.

This chapter develops a material to simulate the appearance of silk-like fabric that displays *iridescence*—the fabric appears to change color when viewed from different directions. This MDL material is inspired by research done by Iman Sadeghi and others at the University of California, San Diego, and published in a 2013 paper entitled "A Practical Microcylinder Appearance Model for Cloth Rendering.[7]"

8.1 A strategy for an MDL material

Many problems can be simplified by decomposing them into a set of smaller problems. Modeling the appearance of fabric lends itself to this approach. Weaving intertwines two sets strings at right angles; the strategy in this chapter is to first consider these two sets separately.

The traditional names for the two sets of strings are *warp* and *weft*. The word "warp" is related to the Old Norse *varp*, the throw of a dragging net. It is the warp strings, fixed to the loom like a net, that catch the strings of the *weft*—related to English "weave"—as they are woven through the warp. For the purposes of an MDL material, we simplify the structure of the warp and the weft by assuming that the warp strings are effectively in line with the surface of the fabric.

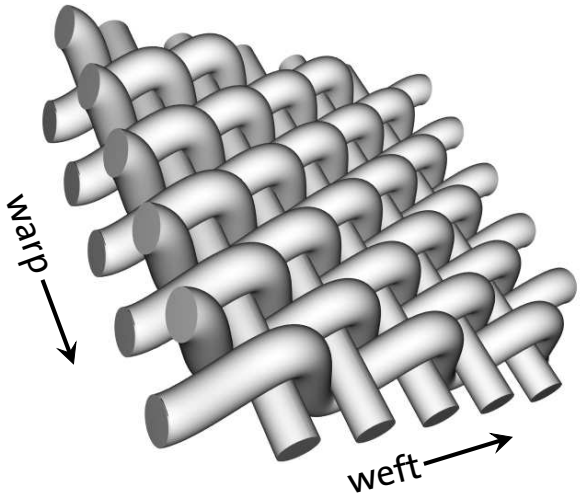


Fig. 8.1 – The names of the two thread directions

7. `http://graphics.ucsd.edu/~iman/a_practical_microcylinder_appearance_model_for_cloth_rendering.php`

Besides the separate shape and reflectivity of the warp and the weft, the thickness of the material will affect how much light is transmitted through the fabric, its *translucency*. By designing separate materials for the warp and the weft—to be combined using the techniques of the last chapter—the fabric model can be separated into three parts:

1. The appearance of the warp: a set of straight strings
2. The appearance of the weft: a set of periodically curving strings
3. The appearance of the fabric: a combination of the warp and weft with iridescence and translucency

To suggest the relationship of the code with the real-world model, the terms "warp" and "weft" will be used in the names of materials and their parameters. Students of the cinema may find that remembering the difference between the two directions is simplified by considering the Elmer Fudd Mnemonic: "The weft goes from weft to wight."

8.2 Simplifying assumptions

The research paper mentioned previously treats the threads of the warp and weft as *microcylinders*, cylinders small enough that they cannot be seen individually, but which when considered together display an appearance that can be described in a statistical way, without regard to the individual cylinders themselves. The microcylinder model suggests two properties that can be implemented in a material: directional reflections and multiple highlights.

8.2.1 An isotropic cylinder appears to be anisotropic

Section 3.7 described *anisotropic* glossy reflection, in which reflection is oriented with respect to the surface due to its geometric properties, like brushed metal. If the threads in a fabric are made of a glossily reflective material, like silk, the cylindrical model suggests that the reflection will appear to be oriented in the direction of the thread, even if the reflection is not explicitly defined to be anisotropic. For example, both the cylinders and sphere in Figure 8.2 are rendered with equal glossy roughness in both the u and v directions, though the cylinders' reflections appear anisotropic.

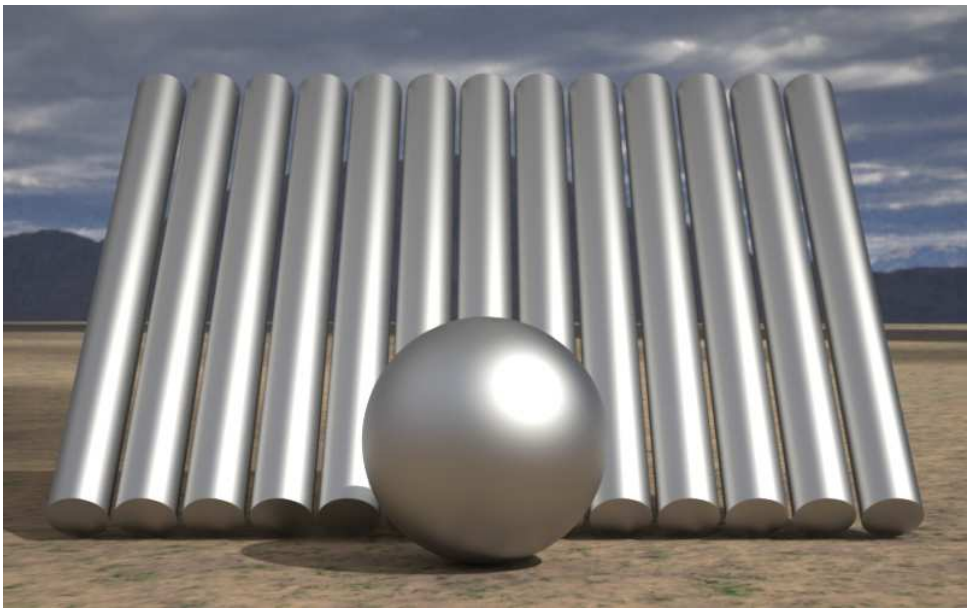


Fig. 8.2 – Appearance of aniostropic reflection on cylinders aligned together

Turning this around, a single surface with oriented reflections can look like the result of the oriented reflections of a fabric's threads. Both the warp and weft materials in this chapter use this anisotropic property, but with their reflections oriented at right angles, like the threads themselves.

8.2.2 Multiple warp threads can create multiple highlights on the weft

The initial illustration of the warp and weft depicted only the simplest possible weaving pattern. When multiple warp threads are taken together for the path of the weft, the small structure of the weft as it goes in and out of the warp may not be so symmetrical. For example, one weft thread may go under a single warp thread, then over two warp threads, then under one warp thread again, and so on—1, 2, 1, 2, 1....

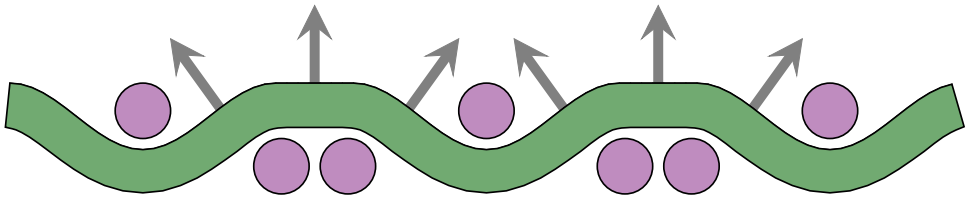


Fig. 8.3 – Doubled threads in the warp create three reflective surfaces in the weft

Modeling the weft thread as a flexible cylinder that is under tension as it courses through the warp suggests that the curvature may not be the same throughout—that some areas may be relatively flat enough to create separately reflecting surfaces. The weft material in this chapter simulates three highlights from the flattening of the thread path suggested by the diagram. An analysis of the structure of other weaving patterns could inspire similar types of complex reflections.

8.3 The structural design of the fabric material

The combining functions of the previous chapter provide the structure for the fabric material. Weighted layering creates three highlights for the weft from three glossy reflection BSDFs. The warp, with a single highlight, is defined by a single glossy reflection BSDF. The warp and weft BSDFs are combined in a directionally dependent manner by the `custom_curve_layer` function to create the iridescent shift of color across the surface. Finally, a translucent layer is added, also controlled by `custom_curve_layer` so that the fabric seen straight on adds a greater degree of translucency than at the edges.

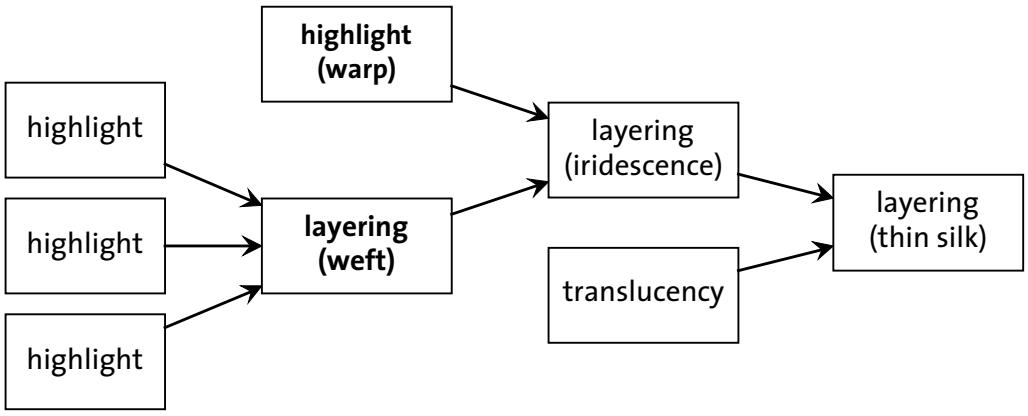


Fig. 8.4 – Relationship of the components in the design of the fabric material

8.4 The directional sheen of the warp

The anisotropic glossy reflection of the warp's highlight can be defined by `df::simple_glossy_bsdf`. The `anisotropic_glossy` material makes both the roughness in the u and in the v direction available as parameters.

Listing 8.1

```
material anisotropic_glossy (
   color tint = color(0.7),
   float roughness_u = 1.0,
   float roughness_v = 0.15) =
material (
   surface: material_surface (
       scattering: df::simple_glossy_bsdf (      Glossy reflection distribution function
         tint: tint,
           roughness_u: roughness_u,
           roughness_v: roughness_v,              Control of anisotropic reflection
         mode: df::scatter_reflect )));
```

In Figure 8.5, the spheres show the single anisotropic highlight from the warp. The variable shape of the fabric creates a much more complex set of reflections, showing the importance of geometry in displaying the appearance of a material.

```
anisotropic_glossy(
  tint:
    color(.3, .22, .3),
  roughness_v: .3)
```

Figure 8.5

The `anisotropic` material, used by itself for the warp component, will also define the three highlights of the weft in the next section.

8.5 Creating multiple highlights

Now a simplification from a previous chapter reveals itself. The BSDF layering functions all include another parameter not described there, the surface normal vector to be used for that layering operation. For the weft material, the `df::weighted_layer` function will include the normal vector in this way:

```
df::weighted_layer (
   weight: layer-fraction,
   normal: vector,
   layer: BSDF,
   base:  BSDF)
```

By default, the normal vector used for lighting calculations is the normal defined by the geometric surface. However, the geometrically derived normal can be replaced by an arbitrary vector. For the weft material, one highlight will be produced by the default normal vector, the two others from modifying, or *perturbing*, the normal vector.

Based on the previous diagram of doubled warp threads, the two modified normals will be "bent" to the right and to the left, respectively. In this case, "left" and "right" will mean moving in the *u* direction in the texture space on the surface using standard MDL functions. Texture spaces and MDL functions are described in the next chapter; the important point here is that the apparent orientation of the surface—and thereby the reflection direction—can be manipulated by modifying the description of surface orientation, the surface normal.

Listing 8.2

`math::normalize(` Normalize the result of:

`state:normal()` Acquiring the surface normal

`+ state::texture_tangent_u()` ...and lengthening it in the *u* direction

`* bend_factor)` ...scaled by some amount.

In the material for the weft definition, `anisotropic_glossy_three_lobes`, this expression is used twice within a let-expression to define the two different normal vectors (left and right).

Listing 8.3

```
material anisotropic_glossy_three_lobes (
   color tint = color(0.7),
   float roughness = 0.1,
   uniform float bend_factor = 1.5 )
=
let {
   float3 left_normal = math::normalize(
      state::normal()
      + state::texture_tangent_u(0)
      * -bend_factor);
```

Bend the normal vector one way

```
    float3 right_normal = math::normalize(
       state::normal()
       + state::texture_tangent_u(0)
       * bend_factor);                                  Bend it the other way
   bsdf glossy_component = anisotropic_glossy(
     tint: tint,
     roughness_v: 1,
     roughness_u: roughness).surface.scattering;
} in
material (
   surface: material_surface (
     scattering: df::weighted_layer (
         weight: 0.5,                                   Mix half and half with combined layers below
         normal: state::normal(),                       The unmodified normal vector
       layer: glossy_component,
       base: df::weighted_layer (
           weight: 0.5,                                 Mix half and half with layer below
           normal: right_normal,                        The modified normal vector
         layer: glossy_component,
         base: df::weighted_layer (
             weight: 1.0,                               Full intensity for bottom layer
             normal: left_normal,                       The modified normal vector
           layer: glossy_component )))));
```

Rendering the fabric and spherical objects with `anisotropic_glossy_three_lobes` produces the image. As with the warp image, the spheres help demonstrate the anisotropic effect: in this case for the three highlights from the use of `df::weighted_layer`.

```
anisotropic_glossy_three_lobes(
  tint:
    color(.2, .3, .2),
  roughness: .15)
```

Figure 8.6

8.6 Combining the warp and the weft

With the warp and weft components defined, the structure of the combining material requires that the internal parameters of the warp and weft can still be controlled. The relative weight of the warp and the weft should also be modifiable.

Default colors that very clearly demonstrate the warp and weft combination are defined as `const` variables of type `color`.

```
const color magenta = color(.3, .22, .3);
const color green = color(.20, .3, .2);
```

As the material becomes more complex, naming conventions in its parameters and the use of a let-expression help keep its intent clear.

Listing 8.4

```
material warp_and_weft(
    color warp_color = magenta,
    float warp_roughness = 0.3,                    Warp control

    color weft_color = green,
    float weft_roughness = 0.15,
    uniform float weft_bend_factor = 1.5,          Weft control

    uniform float warp_to_weft = 3,                The proportional warp contribution

    float shadowing = 0.0 )                        Control of the amount of self-shadowing by the fabric
=
let {
```

```
    bsdf warp = anisotropic_glossy (
       tint: warp_color,
       roughness_u: 1.0,
       roughness_v: warp_roughness).surface.scattering;
```

Calculate the warp layer

```
    bsdf weft = anisotropic_glossy_three_lobes (
       tint: weft_color,
       roughness: weft_roughness,
       bend_factor: weft_bend_factor).surface.scattering;
```

Calculate the weft layer

```
    float warp_weight =
      warp_to_weft > 1 ? 1.0 - 1.0 / warp_to_weft : warp_to_weft;
```

Calculate the warp factor

```
} in
material (
   surface: material_surface (
        scattering: df::custom_curve_layer (
```

Use default black layer to add self-shadowing effect

```
           weight: shadowing,
```

Contribution of default black layer

```
         normal_reflectivity: 0.05,
         grazing_reflectivity: 1.0,
         exponent: 3,
           base: df::weighted_layer (
              weight: warp_weight,
              layer: weft,
              base: warp))));
```

Combine warp and weft layers

The strong color contrast between the magenta warp and green weft components helps clarify their contribution in the more complex geometry of the fabric.

```
warp_and_weft(
  warp_color:
    color(.3, .22, .3),
  weft_color:
    color(.2, .3, .2))
```

Figure 8.7

8.7 A translucency component for thinner fabric

The `warp_and_weft` material is fully opaque, though thin silk-like materials will allow some light to pass through. For this contribution of light transmission, a separate material for translucency uses `df::diffuse_transmission_bsdf`.

Listing 8.5

```
material translucency(
   color transmission_color = color(1) )
=
material(
   thin_walled: true,
      surface: material_surface (
          scattering: df::diffuse_transmission_bsdf (
            tint: transmission_color )));
```

Light scattering by transmission

`translucency()`

Figure 8.8

Note that this is a general material that could readily be used as a building block in any material that requires such a translucent component. Often in developing complex materials, such generic building blocks will suggest themselves, and can be collected in libraries of materials that will simplify future material development.

8.8 Combining iridescence and translucency

Adding translucency to the `warp_and_weft` material requires two changes:

1. Add parameters to control the degree of translucency
2. Combine the BSDF of the `warp_and_weft` with the BSDF of `translucency`

The BSDF of `warp_and_weft` and `translucency` are defined as temporary variables in a let-expression. The body of the material combines them with function `df::custom_curve_layer`.

Listing 8.6

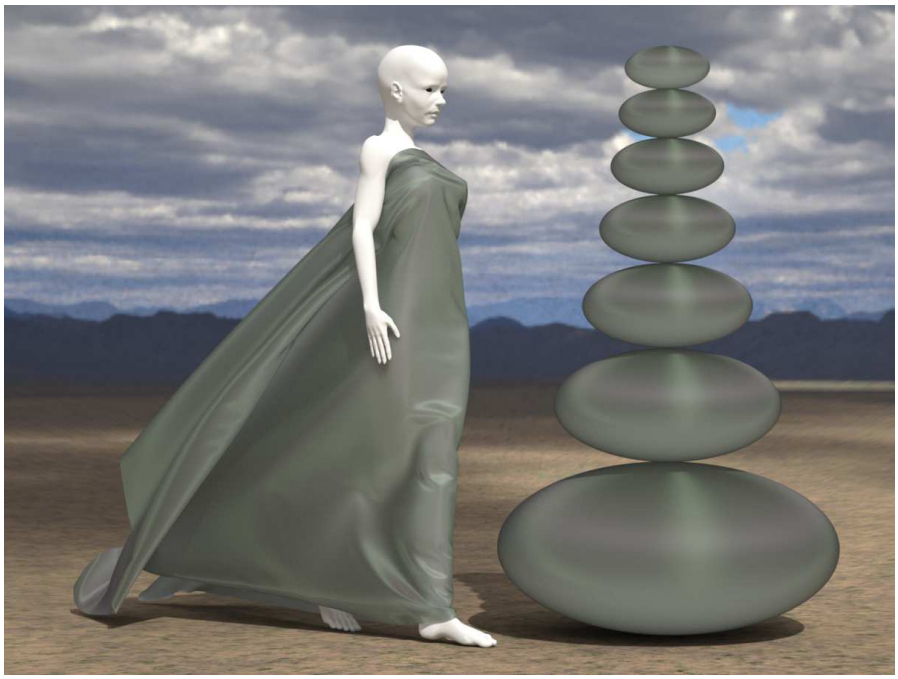
```
material iridescent_silk (
    color warp_color = magenta,
    float warp_roughness = 0.3,                          Warp control

    color weft_color = green,
    float weft_roughness = 0.15,
    uniform float weft_bend_factor = 1.5,                Weft control

    uniform float warp_to_weft = 3,                      Proportional contribution of warp

    float transmission_weight = 0.2,
    color transmission_color = color(0.95, 1.0, 0.95),
    float transmission_exponent = 3.0,                   Transmission control

    float shadowing = 0.0)                               Self-shadowing control
=
let {
    bsdf glossy = warp_and_weft(
      warp_color: warp_color,
      warp_roughness: warp_roughness,
      weft_color: weft_color,
      weft_roughness: weft_roughness,
      weft_bend_factor: weft_bend_factor,
      warp_to_weft: warp_to_weft,
      shadowing: shadowing).surface.scattering;          Glossiness calculation

    bsdf translucent = translucency(
      transmission_color:                                Translucency calculation
        transmission_color).surface.scattering;
} in
material (
  thin_walled: true,
  surface: material_surface (
      scattering: df::custom_curve_layer (               Layering function
        weight: 1.0,
        normal_reflectivity: 1.0 - transmission_weight,
        grazing_reflectivity: 1.0,
        exponent: transmission_exponent,
          layer: glossy,                                 Glossy layer

          base: translucent ) ));                        Translucent layer
```

The degree of translucency is controlled by the `transmission_weight` parameter to the material. The default value of 0.2 produces Figure 8.9:

```
iridescent_silk()
```

Figure 8.9

By increasing the `transmission_weight` to 0.5, the fabric appears to be thinner and the shadows have become lighter.



```
iridescent_silk(
  transmission_weight: .5
)
```

Figure 8.10

The warp and weft color parameters in the previous examples clearly show their individual contributions. However, a variety of effects are possible that suggest other types of woven silk.

In Figure 8.11, the single blue highlight of the warp predominates, with wide highlights of dark red providing a subtle sheen in the flatter surfaces.

```
iridescent_silk(
  warp_color:
    color(.2, .3, .6),
  weft_color:
    color(.07, .05, .05),
  transmission_color:
    color(.95, .95, 1),
  transmission_weight: .05,
  shadowing: .3,
  warp_roughness: .3,
  weft_roughness: .15
)
```

Figure 8.11

Part 4 Defining functions

9 Function calls as arguments

MDL employs two different programming language paradigms, each appropriate for the type of information it needs to describe:

- In the first paradigm, a material is defined by a set of *properties*. Creating a new material consists of specifying values for those properties, for example, the emission distribution function for a light. The distribution functions themselves have properties so that defining the intensity of emission consists of providing a value for the intensity property of the emission distribution function. This type of programming, in which the manner of execution is not defined, but the end result (for example, an emission intensity of 100 watts), is called *declarative* programming.
- In the second paradigm, an MDL material can specify values like emission intensity. It does this by calling functions defined in the MDL standard function library in possible combination with custom functions defined by the author of the material. The functions of the standard library functions and the ability to define new ones provide MDL with an *imperative* programming language model that describes *how* a value should be calculated.

This chapter explores the possibilities in MDL for the procedural definition of material struct parameters that are provided by the imperative paradigm. For more information on specific language features and for additional examples of MDL syntax in function definition, see the MDL language specification: NVIDIA Material Definition Language (version 1.6.2, August 5, 2020).[8]

9.1 The syntax of imperative MDL

The general syntax of user-defined functions will be familiar to programmers using any of the wide variety of languages that have been influenced by the C and C++ languages. Readers familiar with C-style syntax may consider skipping to the next section; control structures like `for` and `if` are identical in form to C. However, there are important limitations to MDL with regard to standard C programming techniques, as well as useful extensions. This section will briefly present the language features required to understand the example functions developed in the following sections. Where there are important differences between MDL's language for function definition and C/C++, these will be noted in the examples in this and later chapters.

9.1.1 Data types and variables

The previous chapters have already presented the high-level data types of MDL that are specific to the domain of appearance definition: distribution function objects (`bsdf`, `edf`, `vdf`) and the various `struct` types for material definition (for example, `material` and `material_surface`). The `struct` is a *compound* type; it combines in one datum a set of other data values. In user-defined functions, the simple (non-compound) types are frequently used: `int` (integer), `float` and `double` (approximations of the mathematical concept of real numbers), and `bool` (the Boolean values of `true` and `false`). The `color` type is also a primitive type in MDL

8. `http://mdlhandbook.com/pdf/MDL_spec_1.6.2_05Aug2020.pdf`

and will often be the value that is produced by a function. A few programming elements were introduced in "The struct type" (page 20). This section provides a more general description of the fundamentals required for writing MDL functions.

A name that refers to a value is called a *variable*. To use a variable in an MDL function, the variable must first be *declared*, which associates the variable's name with a data type. The data that the variable represents is called the variable's *value*. If a value is specified at the time the variable is declared, this specification *initializes* the variable.

To declare a variable and give it a value, the *constructor* for that data type is used to create the value. The constructor syntax specifies the initial value or values for the variable as required by the variable's data type. The full form of a variable declaration uses the equal sign to associate the name with its initial value:

data-type variable-name `=` *data-type* `(` *arguments* `);`

For example, the following line declares a variable of type `float` named `total` and initializes the variable to 0.0:

```
float total = float(0.0);
```

For simple (non-compound) types, explicitly specifying the constructor is unnecessary:

```
float total = 0.0;
```

An abbreviated form removes the redundant use of the type name:

```
float total(0.0);
```

This abbreviated form is most useful for compound types:

```
color blue(0.0, 0.0, 1.0);
```

A variable must have a value; if it is not initialized when it is declared, the default value for the data type is used as the initial value. Default values are typically a value that can be thought of as "zero" for the type; numeric zero for `int`, `float` and `double`, or "black" for the `color` type (`color(0.0, 0.0, 0.0)`, also abbreviated as `color(0.0)`). This idea of a "zero equivalent" for the default value is also used by the components of the `material` struct; if no `emission` value is defined for the `material_surface` struct, then no light is emitted.

The equals sign (=) is used in variable declaration in a different sense here than in mathematics; `total` is not *equal to* 0.0, `total` currently has a *value of* 0.0, a value that can be changed later in the function. However, in talking about variables, it would be typical to say something like, "at the beginning, the total is zero," with the variable name representing the concept of "total" throughout the function.

Variable declarations are also provided by the let-expression, as described in "Simplifying a material's structure with temporary variables" (page 97).

9.1.2 Control flow

The declaration of variables is one example of a `statement` in MDL; it is the simplest unit of a user-defined function. Other statements define when and how many times a set of statements should be executed. The syntax of the looping and conditional statements—*control flow* – will be very familiar to C/C++ programmers.

The most flexible loop structure is the `for` statement:

```
for ( variable-declaration ; termination-condition ; end-of-loop-action ) {
    statements
}
```

The *curly braces* that surround the statements of the `for` loop create a *block*. Wherever a single statement can appear, a sequence of statements can be provided instead by surrounding the sequence with curly braces.

For example, this code fragment will sum up all the integers from 1 to 42:

```
int sum = 0;
for (int i = 1; i < 43; i = i + 1) {
    sum = sum + i;
}
```

This can be read as, "Start with `i` equal to 0, and while `i` is less than 43, add `i` to `sum` and add 1 to `i`."

In this case, the curly braces were not strictly necessary because only a single statement was executed by the loop. The `for` loop could have therefore been written without the braces:

```
int sum = 0;
for (int i = 1; i < 43; i = i + 1)
    sum = sum + i;
```

Because they occur so frequently in structures like loops, MDL also provides C's set of increment and decrement abbreviations, for example:

Abbreviation	*Equivalent to*
`++i`	`i = i + 1`
`--i`	`i = i - 1`
`i += 10`	`i = i + 10`
`i -= 10`	`i = i - 10`

The previous example can be rewritten using the increment abbreviations:

```
int sum = 0;
for (int i = 1; i < 43; ++i)
    sum += i;
```

The `if` statement is the other important control flow structure and enables *conditional execution* of sequences of one or more statements. The `if` statement has two forms. In the first form, a Boolean expression is evaluated—the *condition*—to determine whether it is true or false. If the statement is true, the next statement (or block) is executed. If not, no statements are executed.

```
if ( Boolean-expression ) {
    statements-if-expression-is-true...
}
```

In the second form of the `if` statement, a second statement/block is preceded by `else`. This second statement/block is executed only if the Boolean expression is false.

```
if ( Boolean-expression ) {
    statements-if-expression-is-true...
} else {
    statements-if-expression-is-false...
}
```

MDL also provides a *conditional expression* that uses the characters "?" and ":".

Boolean-expression `?` *value-if-expression-is-true* `:` *value-if-expression-is-false*

This example initializes `color` variable `height_color` to be white if the y coordinate of the current position is greater than 0.0 and black if it is not.

```
float3 position(0.0);
...
color height_color = position.y > 0.0 ? color(1) : color(0);
```

The "`...`" in the example would include the calculation of the current point using MDL's *renderer state functions*, which are described in the next section.

9.1.3 Standard functions and MDL modules

Several of the user-defined functions developed later in this chapter depend upon the *rendering state*, data used by the rendering system in calculations that depend upon attributes of the geometric structures that are being rendered. These values of the rendering state are available through functions that are not part of the MDL language, but are defined in a separate software component called a *module*. The module for renderer state functions, called the `state` module, is one of a set of *standard modules* that are defined by the MDL specification. In addition to the `state` module, the examples in this chapter will also use the `math` module, which provides a set of mathematical functions (for minimum, maximum, cosine, sine, etc.).

The current point in space that is being rendered—the point on a surface or in a volume for which the `material` struct is being evaluated—is acquired by using the `position` state function. The module name `state` is used as a prefix to the function name and is separated from it by two colon characters (`::`).

```
float3 pos = state::position();
```

The `float3` type defines the x, y, and z coordinates of the point in 3D space. The `float3` value returned by a call to `state::position` is a point in *internal space*, an implementation-dependent coordinate system. Two other spaces are defined by MDL. *Object space* is the coordinate system of the object being rendered, a *local* coordinate system for the geometric structures of the object. *World space* defines the *global* coordinate system within which all objects are positioned.

The state function `transform_point` can be used to convert the space in which a point is defined to one of the other two spaces.

`state::transform_point(`*original-space*`,` *target-space*`,` *point-to-transform*`)`

To specify the coordinate spaces in the arguments of `state::transform_point`, MDL defines the `coordinate_space` enum type in the `state` module. The type definition contains the names that can be used as arguments to `transform_point`:

```
enum coordinate_space {
  coordinate_internal,
  coordinate_object,
  coordinate_world
};
```

Combining these elements of the `state` module, a `float3` variable that provides the current position in object space can be declared and initialized as follows:

```
float3 object_position = state::transform_point(
    state::coordinate_internal,
    state::coordinate_object,
    state::position());
```

Note that the state names are prefixed by the `state` module name when they are used in a user-defined functions. An enum type is also used to define the scatter mode for BSDFs in "Specular interaction at a surface" (page 45), but are defined in the `df` (distribution function) module and are therefore prefixed with the `df::` module name.

Another type of "space" is *texture coordinate space*, which defines a coordinate system for the surface of a polygon in a modeling system. Associating a point on the surface of an object with the value of a function or a color in an image are examples of the traditional computer graphics technique called *texture mapping*. By convention, the two-dimensional texture coordinate axes are called u and v (rather than x and y). MDL extends the texture coordinates to three dimensions, where the coordinates are called u, v, and w.

MDL supports texture mapping through the state function `texture_coordinate`. Several examples in this chapter will use the result of `texture_coordinate` in defining the surface color as part of the definition of a material.

These various components of MDL—variable declarations, control flow, functions provided by the standard MDL modules—can be combined in *user-defined functions*, described in the next section.

9.1.4 User-defined functions

To augment the set of functions defined by the MDL standard modules, the author of a material can define custom functions to be used by the arguments of the material struct. A function consists of one or more statements (the function's *body*) that calculate a value that is considered to be the value of the function. The value that the function calculates is said to be "returned" from the function when the function is executed. Executing a function is also described as *calling the function*.

type-of-return-value function-name `(` *parameters* `) {`
    statements
`}`

One of the statements in the function defines the value returned by the function using the word `return`.

`return` *expression* `;`

For example, the following function returns the color black if the y position of the object point is equal or below a value (defined by an argument), but returns white if the point is above that value:

Listing 9.1

```
color white_above_value(float y_value = 0.0) {
   float3 object_position = state::transform_point(
      state::coordinate_internal,
      state::coordinate_object,
      state::position());
   color result;
```

Declare the variable “result” to be of type “color”

```
   if (object_position.y > y_value)
      result = color(1);
   else
      result = color(0);
```

Calculate the value of “result”

```
   return result;
```

Provide “result” as the value of the function

```
}
```

A function may contain more than one `return` statement, but the function terminates as soon as the first `return` statement is executed. Given this property of the `return` statement, the previous function could be rewritten as:

Listing 9.2

```
color white_above_value(float y_value = 0.0) {
   float3 object_position = state::transform_point(
      state::coordinate_internal,
      state::coordinate_object,
      state::position());
   if (object_position.y > y_value)
      return color(1);
   else
      return color(0);
}
```

Alternate “return” statements

More than one `return` statement is considered by many programmers as an example of bad style. With multiple `return` statements, the value returned by the function is not immediately apparent, especially in larger functions. However, many programmers also believe that large functions are also an example of bad style, so that functions should be small enough that their behavior can be understood and easily debugged.

Like material constructors, the parameters of a user-defined function can be assigned default values. For example, calling function `white_above_value`, the default value for parameter `y_value` is 0.0.

`white_above_value()` *is equivalent to* `white_above_value(0.0)`

The `float3` type of `object_position` is a compound type. As used in the preceding `white_above_value` function, it represents a three-dimensional point. The three components of the

`float3` type represent the x, y, and z coordinates of the point. To use one of these three values in an expression, the dot operator (`.`) is used. In this case, the y coordinate is specified with `object_position.y`.

Using the variable `result` to define the function's return value is not strictly necessary in this case. Using a conditional expression allows the function to be more compactly written.

Listing 9.3

```
color white_above_value(float y_value = 0.0) {
   float3 object_position = state::transform_point(
      state::coordinate_internal,
      state::coordinate_object,
      state::position());
   return
      object_position.y > y_value
        ? color(1.0)
        : color(0.0);
}
```

Conditional expression

Note that an MDL compiler will typically perform optimizations to produce more efficient code for execution by the rendering system.

9.2 Displaying spatial parameters as colors

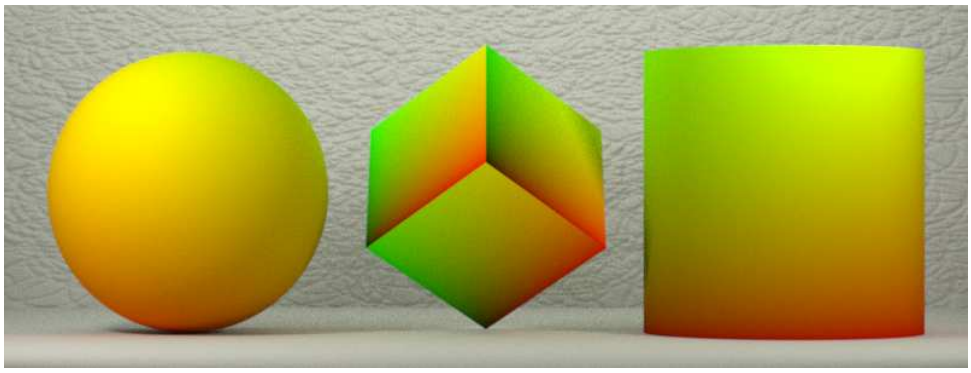
As the first example of using standard and user-defined functions, this section will treat a surface property of an object as a color. The state function `texture_coordinate` returns a `float3` value that identifies a point in the *texture coordinate space*. By using a `float3` value in the `color` type constructor, the u, v and w values are converted to values of red, green, and blue.

Listing 9.4

```
material uv_as_color_material() =
let {
   color diffuse_color =
     color(state::texture_coordinate(0));
   bsdf diffuse_bsdf = df::diffuse_reflection_bsdf(
     tint: diffuse_color);
} in material(
   surface: material_surface(
     scattering: diffuse_bsdf));
```

Calling the state function "texture_coordinate"

Typically, the values of an object's texture coordinates are in the range of 0.0 to 1.0, producing colors that are combinations of red and green. The objects in the example scene all have texture coordinates in the 0.0 to 1.0 range.



`uv_as_color_material()`

Figure 9.1

To demonstrate the use of a user-defined function as the value of a parameter in a material, the call to `state::texture_coordinate` can be wrapped in a function with a name that describes its intended use as a color.

Listing 9.5

```
color uv_as_color() {
    return color(state::texture_coordinate(0));
}
```

Calling the state function

Now the user-defined function takes the place of the color constructor in the let-expression.

Listing 9.6

```
material uv_as_color () =
let {
    color diffuse_color = uv_as_color();
    bsdf diffuse_bsdf = df::diffuse_reflection_bsdf(
      tint: diffuse_color);
} in
material(
  surface: material_surface(
    scattering: diffuse_bsdf));
```

Call the user-defined function “uv_as_color”

Though the body of the `uv_as_color` function only contains a `return` statement, the name provides an explanation of the value produced by the function.

9.3 Mapping from spatial parameters to an image

The texture coordinates of an object’s surface are frequently used in mapping the pixel values of a digital image to the surface of an object. An image to be used in this way is represented in MDL as an instance of the type `texture_2d`. A constructor for a `texture_2d` can take the filename of an image file as an argument.

`texture_2d(` *image-filename* `)`

A coordinate system is defined for `texture_2d` images in which the origin is at the lower left corner and the two axes, called *u* and *v*, range from 0.0 to 1.0. This range is used for u and v for images of any aspect ratio.

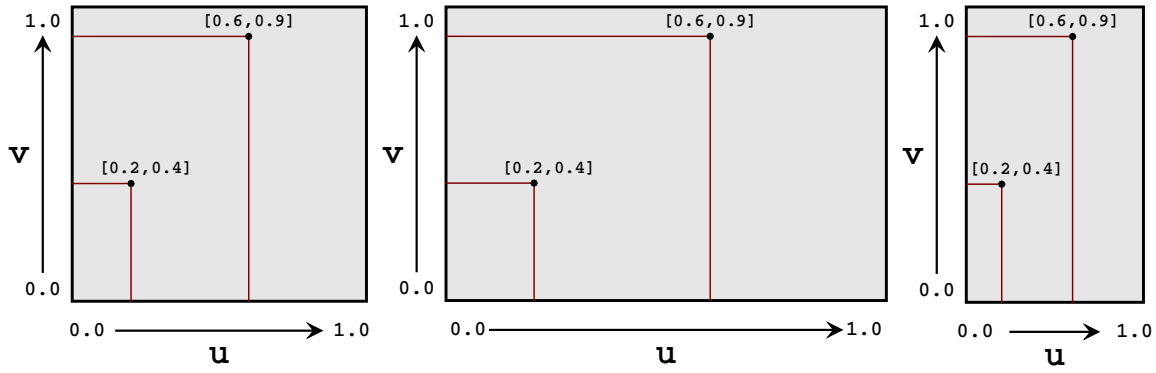


Fig. 9.2 – The uv coordinate system scales to fit the image

Given an instance of `texture_2d`, a color value can be *sampled* from the image with the state function `lookup_color` in the `tex` module.

`tex::lookup_color(` *instance-of-texture_2d*`,` *uv-coordinates* `)`

For example, the following code example initializes the variable `center_color` with the color of the center of the image file `background.png`.

```
color center_color =
    tex::lookup_color(texture_2d("background.png"), float2(0.5, 0.5));
```

In the previous section, the texture coordinate in the renderer state as provided by function `state::texture_coordinate` was displayed as a color. In the more typical use of `state::texture_coordinate` it provides the mapping from colors in an image to colors on the surface of a geometric object. This mapping can be encapsulated in a function:

Listing 9.7

```
color texture_2d_lookup(uniform texture_2d texture) {
    float3 uvw = state::texture_coordinate(0);          uv coordinates

    return tex::lookup_color(texture, float2(uvw.x, uvw.y));   Color lookup in texture
}
```

To use this texture function in a material, consider this material definition:

Listing 9.8

```
material generic_diffuse_material(
    color diffuse_color = color(1.0)) =     Defining a material parameter of type “color”
let {
  bsdf diffuse_bsdf = df::diffuse_reflection_bsdf(
      tint: diffuse_color);     Using the color parameter for the “tint” field value
} in
```

```
material(
  surface: material_surface(
    scattering: diffuse_bsdf));
```

The material's single parameter defines the color for the diffuse reflection BSDF. For example, assume that Figure 9.3 is available in the file system as a file named "uv.png".

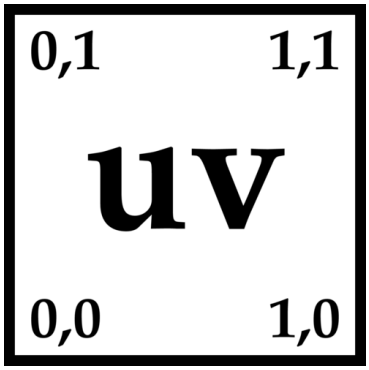


Fig. 9.3 – Image file `uv.png`

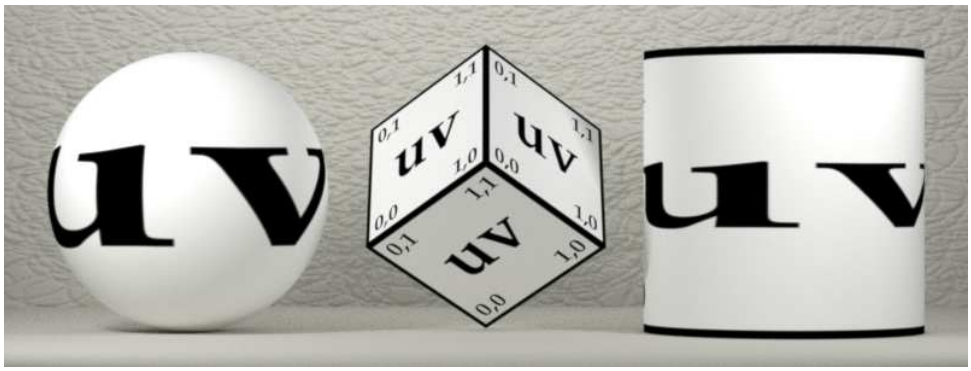
The image in file `uv.png` can be used as the value of the diffuse reflection color in the following way:

Listing 9.9

```
material tex2d_diffuse() =
  generic_diffuse_material(
    diffuse_color:
      texture_2d_lookup(texture_2d("uv.png")));
```

Color argument value from a user-defined function

Treating the uv coordinates as colors made possible a visual representation of their change across the surface of the object. Using the `uv.png` image as a texture map displays the pattern of uv coordinates even more clearly.



`tex2d_diffuse()`

Figure 9.4

The position and size of the texture map image across the surface of the objects can be modified by using the uv coordinate values as factors in an arithmetic expression. For example, the following function, `texture_2d_lookup_scaled`, adds u and v scaling factors as arguments to the `texture_2d_lookup` function. To keep the u and v coordinates in the range of 0.0 to 1.0,

the `frac` function in the `math` standard module is called on the scaled value. The `frac` function returns the fractional component of its input, keeping the values in the 0.0 to 1.0 range.

Listing 9.10

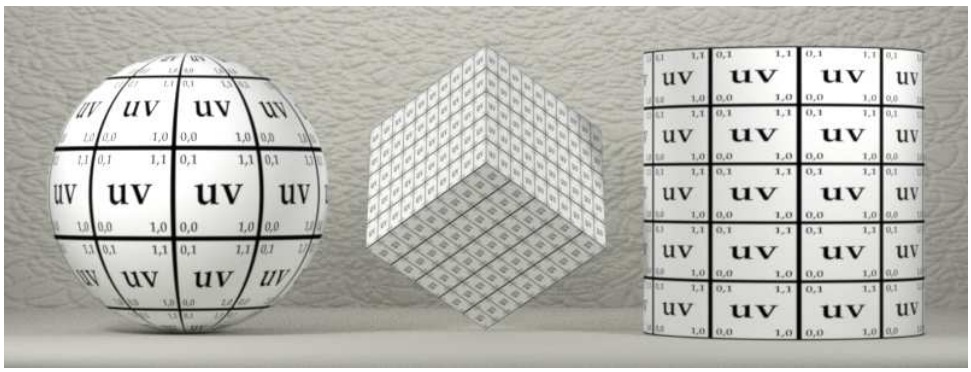
```
color texture_2d_lookup_scaled(
  uniform texture_2d texture,
  float u_scale = 1.0,      Scaling factor in u direction
  float v_scale = 1.0)      Scaling factor in v direction
{
  float3 uvw = state::texture_coordinate(0);
  color result = tex::lookup_color(
    texture,
    float2(
      math::frac(uvw.x * u_scale),      Scaling of u coordinate
      math::frac(uvw.y * v_scale)));    Scaling of v coordinate
  return result;
}
```

The u and v scaling factors can be added as parameters to a material which are in turn passed as arguments to the `texture_2d_lookup_scaled` function:

Listing 9.11

```
material tex2d_diffuse_scaled(
  float u_scale = 1.0,      Scaling factor in u direction
  float v_scale = 1.0) =    Scaling factor in v direction
  generic_diffuse_material(
    diffuse_color: texture_2d_lookup_scaled(
      texture_2d("uv.png"), u_scale, v_scale));   Arguments to user-defined function
```

Rendering with scale factors of 10 for the u coordinate and 5 for the v coordinate produces Figure 9.5:



```
tex2d_diffuse_scaled(
  u_scale: 10,
  v_scale: 5)
```

Figure 9.5

Rendering with materials that display non-visual surface attributes as colors can be an important step in a production pipeline to verify assumptions about those attributes. The next section describes the visualization of other attributes of the rendering state.

9.4 Coordinate spaces

The section "Standard functions and MDL modules" (page 162) described the world and object coordinate spaces that are part of the rendering state. Wrapping `state::transform_point` in a *utility function* can simplify acquiring the current position in those spaces. For example, the following function returns the current object position:

Listing 9.12

```
float3 object_position() {
  return state::transform_point(
      state::coordinate_internal,
      state::coordinate_object,
    state::position());
}
```

Transform the original point from this coordinate system...

...to a resulting point in this coordinate system.

Similarly, the following function returns the current world position:

Listing 9.13

```
float3 world_position() {
  return state::transform_point(
      state::coordinate_internal,
      state::coordinate_world,
    state::position());
}
```

Original point's coordinate system

New coordinate system for the point

For symmetry, the following function for texture space complete the set of coordinate space utility functions:

Listing 9.14

```
float3 uvw_position() {
  return state::texture_coordinate(0);
}
```

Now these transforming functions can be combined in a single function that includes how color values should be mapped to object as a parameter to the function. When a function requires a parameter that specifies a selection among a set of possible choices, a new enum data type can be defined that describes the set. The following code defines an enum data type called `mapping_mode` with three possible values: `uvw`, `object`, and `world`.

Listing 9.15

```
enum mapping_mode {
  uvw,
  object,
  world
};
```

The new `mapping_mode` type can be used as the type of a parameter to a user-defined function. Note that a user-defined enum can declare the type of a function's parameter in the same way as a native MDL type (like `float` or `int`). The enum type is useful for selecting among a set of alternatives using the MDL `switch` statement.

```
switch ( expression ) {
   case value-1 :
       statements
       break;
   case value-2 :
       statements
       break;
   ...
   case value-n :
       statements
       break;
}
```

The *value* of each case statement is compared in turn to the *switch expression*. If the two are equal, the statements following the case statement, the *case block*, are executed. If only that case block should be executed, the case block ends with a `break` statement, which terminates execution of the entire `switch` statement. Without the `break` statement, the statements of the next case block will also be executed and so forth for the entire `switch` statement.

For a generalized function that returns a point in one of the three spaces—texture, object, and world—the `mapping_mode` enum can be used as the expression in a case statement:

Listing 9.16

```
float3 position(mapping_mode mode) {
  float3 result;
  switch (mode) {
    case uvw:
      result = uvw_position();
      break;
    case world:
      result = world_position();
      break;
    case object:
      result = object_position();
      break;
  }
```

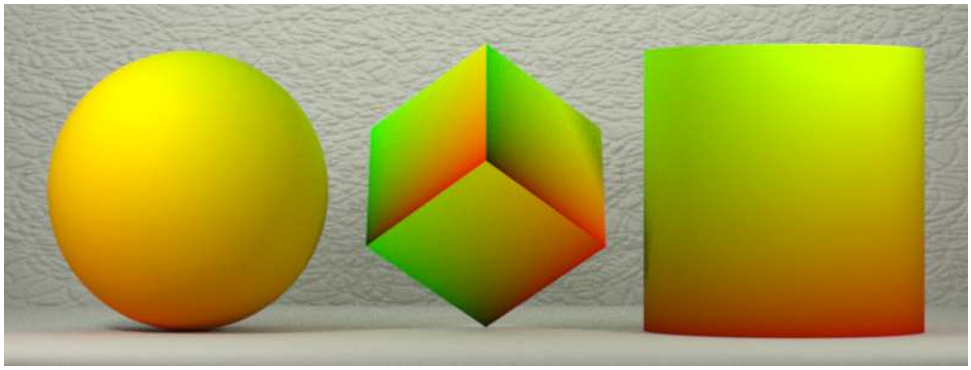
```
  return result;
}
```

Using the `position` function, the material `uv_as_color` can be rewritten as follows to produce a new material called `uvw_space_as_color`.

Listing 9.17

```
material uvw_space_as_color() =
  generic_diffuse_material(
    diffuse_color: color(position(uvw)));
```

Rendering with `uvw_space_as_color` produces Figure 9.6:



`uvw_space_as_color()`

Figure 9.6

However, converting the u and v coordinates to a color assumes that the coordinate values will be in the range of 0.0 to 1.0. Coordinate values in in object and world space will be outside this range, with negative numbers and numbers greater than 1.0. To address this, the `math::frac` function can be used (as in the texture image example) to convert all numbers into the 0.0 to 1.0 range. (The fractional component of a negative number returned by `math::frac` is positive.) The new function `position_to_color` also includes a scaling parameter.

Listing 9.18

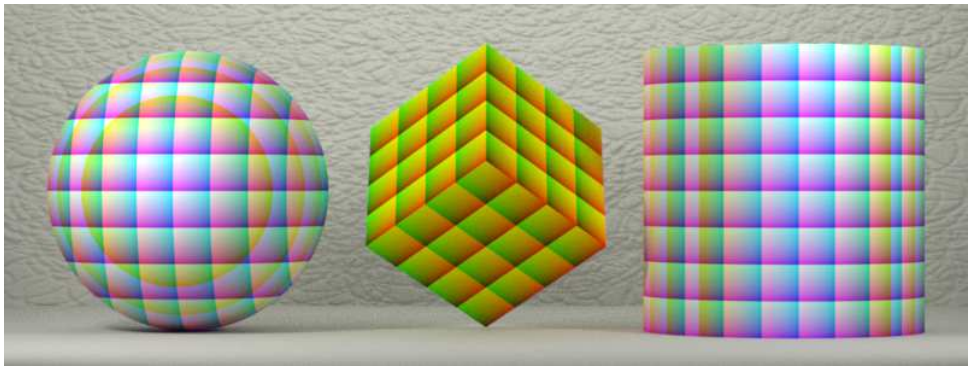
```
color position_to_color(mapping_mode mode, float scale) {
  return color(math::frac(scale * position(mode)));
}
```

The `mapping_mode` enum can be used as a parameter to a material, as well as a scaling factor. These material parameters are passed to the `position_to_color` function that provides the value for the diffuse color:

Listing 9.19

```
material space_as_color(
  mapping_mode mode = uvw,
  float scale = 1.0) =
generic_diffuse_material(
  diffuse_color: position_to_color(mode, scale));
```

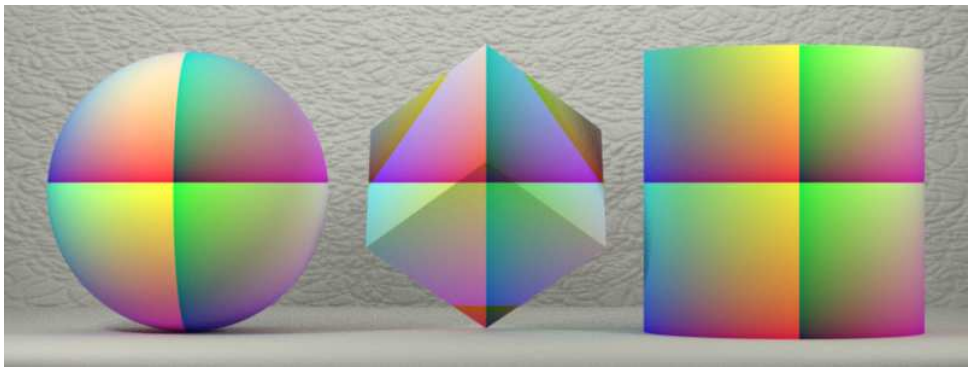
Rendering object space coordinates as colors with a scaling factor of 4 produces Figure 9.7:



```
space_as_color(
  mode: "object",
  scale: 4)
```

Figure 9.7

Rendering with world space colors displays the same coordinate system for all three of the objects:



```
space_as_color(
  mode: "world",
  scale: 4)
```

Figure 9.8

In the examples of this section, the coordinates of the three spaces were treated as colors, a useful visualization tool for visualizing the non-visual attributes of geometric objects. In the next section, coordinate values are used as the initial arguments for calculations that are more complex than simply converting between data types.

9.5 State functions and conditional expressions

In this section, functions use the state functions of the coordinate spaces to select between two colors for stripe and checkerboard patterns.

9.5.1 Stripes

A "stripe" is defined as the regular repetition of a region at right angles to either the x or y axis. To create a function to determine if a point is within a stripe, the geometry of the stripe can be described in the following way:

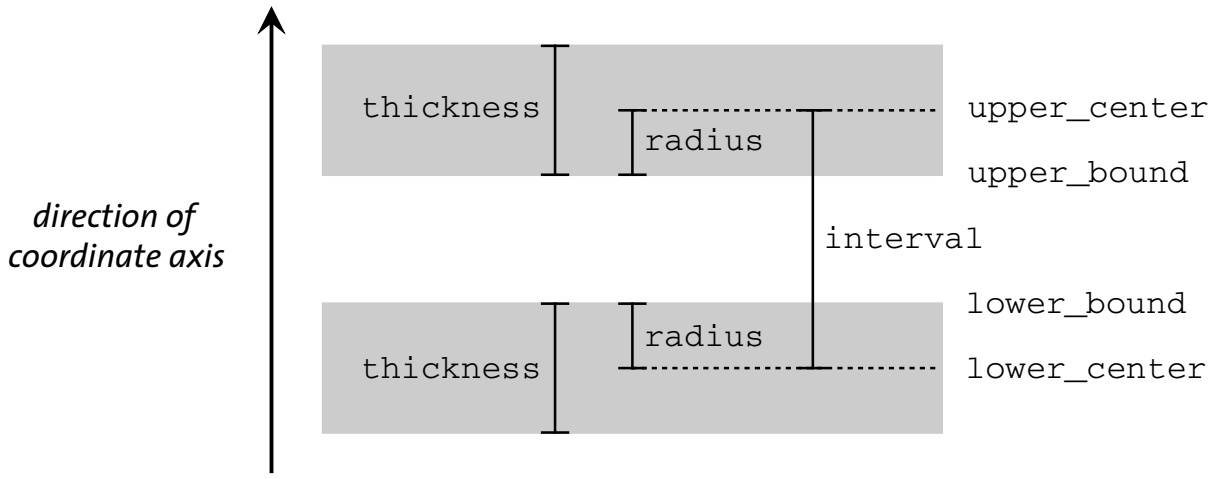


Figure 9.9

The parameters `interval` and `thickness` are used to calculate the lower and upper boundaries of the stripe near the input value `f`.

Listing 9.20

```
bool stripe(float f, float interval, float thickness) {
  float position = math::abs(f);
  float radius = thickness / 2.0;
  float lower_center = int(position / interval) * interval;
  float upper_center = lower_center + interval;
  float lower_bound = lower_center + radius;
  float upper_bound = upper_center - radius;
  return (position < lower_bound) || (position > upper_bound);

}
```

Function `horizontal_stripe_pattern` uses `stripe` to determine if the y coordinate of a point is within the region of a stripe, and returns the appropriate color that was supplied as an input argument.

Listing 9.21

```
color horizontal_stripe_pattern(
  mapping_mode mode=uvw,
  float interval=0.2,
  float thickness=0.1,
  color stripe_color = color(0.0),
  color bg_color = color(1.0))
{
  return
```

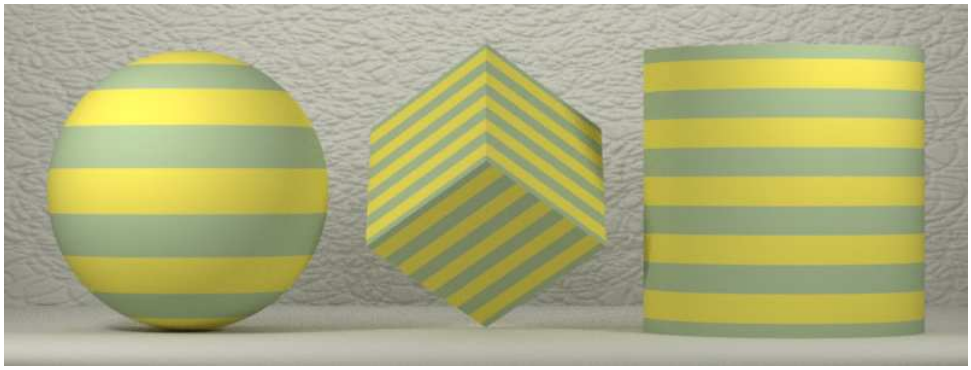
```
      stripe(position(mode).y, interval, thickness)
        ? stripe_color
        : bg_color;
}
```

Function `horizontal_stripe_pattern` provides the diffuse color in material `horizontal_stripes`:

Listing 9.22

```
material horizontal_stripes(
   color stripe_color = color(0.0),
   color bg_color = color(1.0)) =
generic_diffuse_material(
   diffuse_color: horizontal_stripe_pattern(
      stripe_color: stripe_color,
      bg_color: bg_color));
```

Rendering with material `horizontal_stripes` produces Figure 9.10:



```
horizontal_stripes(
  stripe_color:
    color(.4, .5, .3),
  bg_color:
    color(.8, .7, .1))
```

Figure 9.10

Note that material `horizontal_stripes` is an example of a material that hides several of the parameters of the function it contains. All the parameters of a function can be exposed in the material that uses it. More specific materials can be made from the generalized material in a manner similar to the encapsulation of parameters in the creation of layered materials.

For example, the following material makes all the parameters of function `horizontal_stripe_pattern` available as material parameters:

Listing 9.23

```
material horizontal_stripes_component(
   color stripe_color = color(0.0),
   color bg_color = color(1.0),
   mapping_mode mode = uvw,
   float interval = 0.2,
   float thickness = 0.1) =
generic_diffuse_material(
   diffuse_color: horizontal_stripe_pattern(
      stripe_color: stripe_color,
```

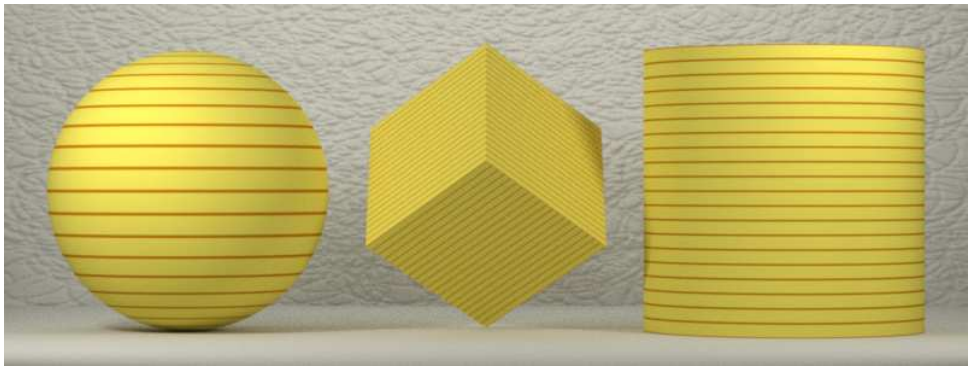
```
      bg_color: bg_color,
      mode: mode,
      interval: interval,
      thickness: thickness));
```

More specific materials can then be built from this fully parameterized component material. For example, the following material hides all the parameters of the `horizontal_stripes_component` material:

Listing 9.24

```
material red_stripes_on_yellow() =
  horizontal_stripes_component(
     stripe_color: color(.5, .2, .0),
     bg_color: color(0.8, 0.7, 0.1),
     interval: 0.05,
     thickness: 0.0075);
```

Rendering with the `red_stripes_on_yellow` material produces Figure 9.11:



`red_stripes_on_yellow()`

Figure 9.11

9.5.2 Checkerboards

To construct a checkerboard pattern of two colors, A and B, the `stripe` function can be used to create two stripes at right angles to each other. If a point is either in one stripe or the other, but not both, then it is assigned color A, else it is assigned color B.

The logical condition "one or the other but not both" is called *exclusive or*. MDL's logical operators can be used to define a general exclusive or function for two Boolean inputs:

Listing 9.25

```
bool exclusive_or(bool a, bool b) {
  return (a || b) && !(a && b);
}
```

Implementing a checkboard pattern then simply requires determining whether a point is in the vertical stripes (by using the x coordinate of the point) or in the horizontal stripes (by using the y coordinate).

Listing 9.26

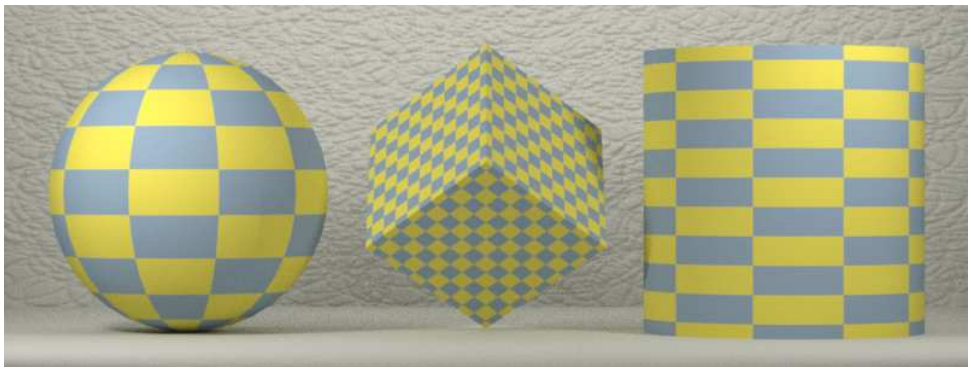
```
color checkerboard_pattern(
  float interval=0.2,
  mapping_mode mode = uvw,
  color color_1 = color(1.0),
  color color_2 = color(0.0))
{
  float3 point = position(mode);
  float thickness = interval / 2.0;
  bool vertical_stripe =
    stripe(point.x, interval, thickness);
  bool horizontal_stripe =
    stripe(point.y, interval, thickness);
  return
    exclusive_or(horizontal_stripe, vertical_stripe)
      ? color_1
      : color_2;
}
```

The two colors of the checkerboard function are the arguments of the `checkerboard` material.

Listing 9.27

```
material checkerboard(
  color color_1 = color(0.0),
  color color_2 = color(1.0),
  mapping_mode mode = uvw) =
generic_diffuse_material(
  diffuse_color: checkerboard_pattern(
    color_1: color_1,
    color_2: color_2,
    mode: mode));
```

Using the `checkerboard` material produces Figure 9.12:



```
checkerboard(
  color_1:
    color(.3, .4, .5),
  color_2:
    color(.8, .7, .1))
```

Figure 9.12

The next chapter demonstrates the use of functions to implement the widely used Perlin noise algorithm.

10 Noise

Specifying the values of material arguments with user-defined functions allows traditional techniques to be incorporated into MDL's paradigm of physically based rendering. As a practical example of incorporating traditional techniques, this section implements the traditional "noise" function first developed by Ken Perlin. The functions in the following sections are a translation from Java into MDL based on Perlin's reference implementation in:
http://mrl.nyu.edu/~perlin/noise/

10.1 Utility functions

The linear interpolation function in the MDL `math` module, `lerp`, has a different order for its arguments than the `lerp` function used in the Java code. Rather than change the code and risk errors that would be difficult to detect, the following function `plerp`—with arguments in the same order as in the Perlin code—is used instead in the converted noise function.

Listing 10.1

```
double plerp(double t, double a, double b) {
  return a + t * (b - a);
}
```

Two other utility functions in the Java code can be translated directly:

Listing 10.2

```
double fade(double t) {
  return t * t * t * (t * (t * 6.0 - 15.0) + 10.0);
}
```

Listing 10.3

```
double grad(int hash, double x, double y, double z) {
  int h = hash & 15;
  double u, v;
  if (h < 8)
    u = x;
  else
    u = y;
  if (h < 4)
    v = y;
  else {
    if (h == 12 || h == 14)
      v = x;
    else
      v = z;
```

```
    }
    return (((h & 1) == 0) ? u : -u) + (((h & 2) == 0) ? v : -v);
}
```

10.2 The noise function

Like the utility functions, the main noise function is also a straightforward port of the Java code. The Java `static final int` permutation table is implemented as a local variable in the MDL function. As an optimization to avoid a modulus operation, the Java code duplicates the table of 256 permutation values. This duplication is done explicitly in the MDL local variable.

```
 double perlin_noise(double3 pt) {
// The first 256 values of the table are duplicated explicitly here,
// but done at run-time in the Java implementation:
   int[512] p(
      151,160,137,91,90,15,
      131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
      190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
      88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
      77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
      102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
      135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
      5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
      223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
      129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
      251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
      49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
      138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180,
      151,160,137,91,90,15,
      131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
      190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
      88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
      77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
      102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
      135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
      5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
      223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
      129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
      251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
      49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
      138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180);

   // Find unit cube that contains point:
   int
      X = int(math::floor(pt.x)) & 255,
      Y = int(math::floor(pt.y)) & 255,
      Z = int(math::floor(pt.z)) & 255;

   // Find relative x,y,z of point in cube and compute fade curves:
   double
      x = pt.x - math::floor(pt.x),
      y = pt.y - math::floor(pt.y),
      z = pt.z - math::floor(pt.z),
      u = fade(x),
      v = fade(y),
      w = fade(z);
```

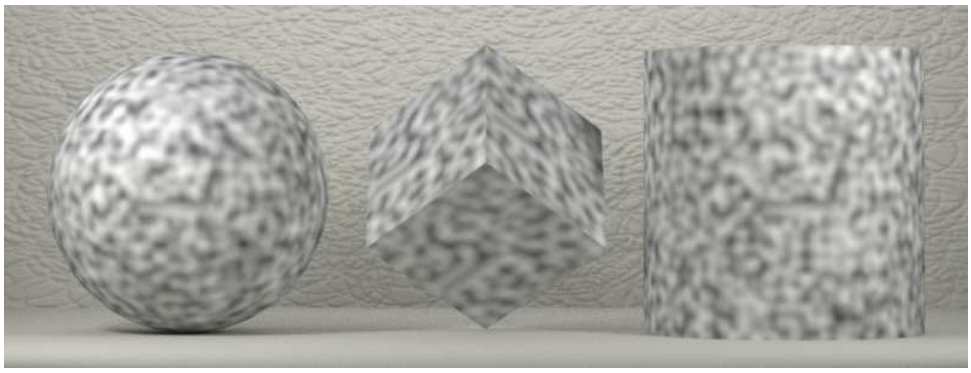
```
    // The hash coordinates of the eight cube corners:
    int
       A = p[X]+Y,   AA = p[A]+Z, AB = p[A+1]+Z,
       B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;

    // Add blended results from eight corners of the cube:
    return
       plerp(w,
             plerp(v,
                   plerp(u,
                         grad(p[AA], x, y, z),
                         grad(p[BA], x-1.0, y, z)),
                   plerp(u,
                         grad(p[AB], x, y-1.0, z),
                         grad(p[BB], x-1.0, y-1.0, z))),
             plerp(v,
                   plerp(u,
                         grad(p[AA+1], x, y, z-1.0),
                         grad(p[BA+1], x-1.0, y, z-1.0)),
                   plerp(u,
                         grad(p[AB+1], x, y-1.0, z-1),
                         grad(p[BB+1], x-1.0, y-1.0, z-1.0))));
}
```

To use the `perlin_noise` function in a material, the `point` argument for the noise function is created by the `position` function described above.

Listing 10.4

```
material perlin_noise_material(
   color color_1 = color(0.0),
   color color_2 = color(1.0),
   double scale = 1.0,
   functions::mapping_mode     Type for parameter “space” from section "Coordinate spaces"
      space = functions::object) =
let {
   double3 scaled_point =
      scale * functions::position(space);
                               Function position() from section "Coordinate spaces"
   double noise_value =
      0.5 + 0.5 * perlin_noise(scaled_point);
                               Rescale noise value from [-1.0,1.0] range to [0.0,1.0]
   color noise_color =
      math::lerp(color_1, color_2, float(noise_value));
                               Interpolate between the two input colors based on the noise value
} in
material(
   surface: material_surface(
      scattering: df::diffuse_reflection_bsdf(
         tint: noise_color)));
```

Rendering with the `perlin_noise_material` produces Figure 10.1 (page 182):



```
perlin_noise_material(
  scale: 10)
```

Figure 10.1

10.3 Combining calls of the noise function

The `perlin_noise` function developed above can be used as a component in the development of more complex noise-based functions. The following function adds a series of noise calculations at different scales of the base noise function.

Listing 10.5

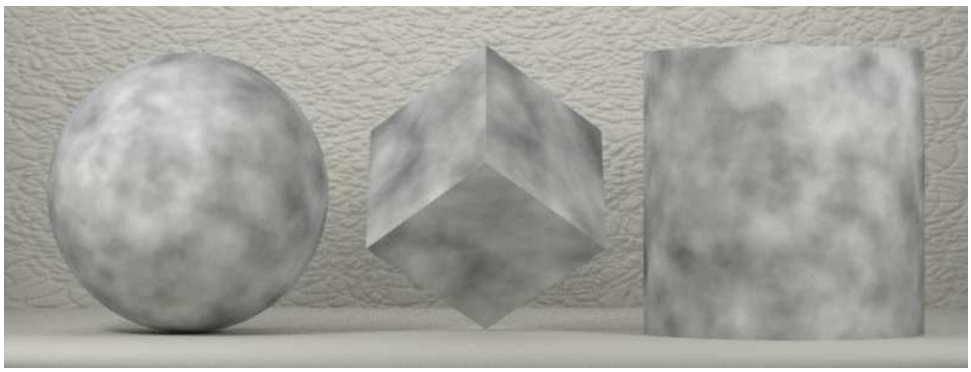
```
double summed_perlin_noise (
  double3 point,
  int level_count=4,
  double level_scale=0.5,
  double point_scale=2.0,
  bool turbulence=false)
{
  double scale = 1.0, noise_sum = 0.0;
  double3 level_point = point;
  for (int i = 0; i < level_count; ++i) {
    double noise_value = perlin_noise(level_point);
    noise_sum += noise_value * scale;
    scale *= level_scale;
    level_point *= point_scale;
  }
  if (turbulence)
    noise_sum = math::abs(noise_sum);
  else
    noise_sum =  0.5 + 0.5 * noise_sum;

  return noise_sum;
}
```

The noise function returns values in the range of -1.0 to 1.0. Traditionally, taking the absolute value of the noise result produces an effect that has been called "turbulence." In `summed_perlin_noise`, the absolute value calculation is parameterized. If false, the noise value is set to the range of 0.0 to 1.0.

Listing 10.6

```
material summed_noise_material (
  color color_1 = color(0.0),
  color color_2 = color(1.0),
  double scale = 1.0,
  int level_count = 4,
  bool turbulence = false,
  functions::mapping_mode space = functions::object) =
let {
  double3 scaled_point = scale * functions::position(space);
  double noise_value = summed_perlin_noise(
    scaled_point, level_count, turbulence: turbulence);
  color noise_color = math::lerp(color_1, color_2, float(noise_value));
} in
material(
  surface: material_surface(
    scattering: df::diffuse_reflection_bsdf(
      tint: noise_color)));
```

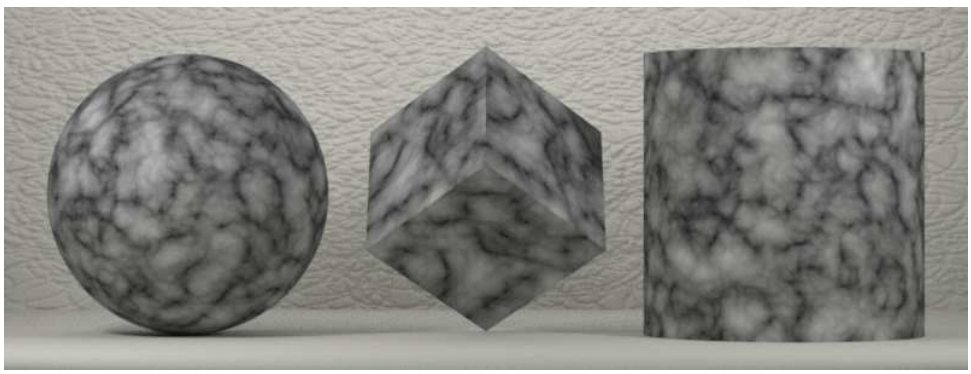
Rendering with the summed_perlin_noise function in the range 0.0 to 1.0 produces Figure 10.2:



```
summed_noise_material(
  scale: 3)
```

Figure 10.2

Rendering with the absolute value of the noise function (turbulence) produces Figure 10.3:



```
summed_noise_material(
  scale: 3,
  turbulence: true)
```

Figure 10.3

Because of the C language heritage of MDL's imperative side, translating Perlin's Java reference code to MDL can be done in a relatively literal way. A later chapter will explore writing functions like Perlin's noise using the extended language features of MDL itself.

Part 5 Modifying geometry

11 Geometry in a material

The appearance of an object is not simply the result of the lighting environment and the light interaction properties of its surface. The geometric structure of a surface—no matter what the physical constituents of the underlying object may be—also plays a fundamental role in how we see and categorize that object. From an early point in the history of computer graphics, researchers understood the importance of geometric form in improving the physical plausibility of a rendered image. This recognition has typically resulted in geometric properties being included in the parameters of a rendering system. This chapter introduces the two methods MDL provides for modifying the geometric properties of a surface.

11.1 Background: Rendering as modeling

In the previous chapter about user-defined functions, the color of a surface was modified by associating, or *mapping*, positions on that surface to color values. These colors were defined by a mathematical function or derived from the pixels of a digital image. Instead of associating a color to a position, *displacement mapping* associates a distance to each surface position. The surface point is moved, or *displaced*, by this distance, typically in the direction of the surface normal at that point, thus changing the geometry of the surface.

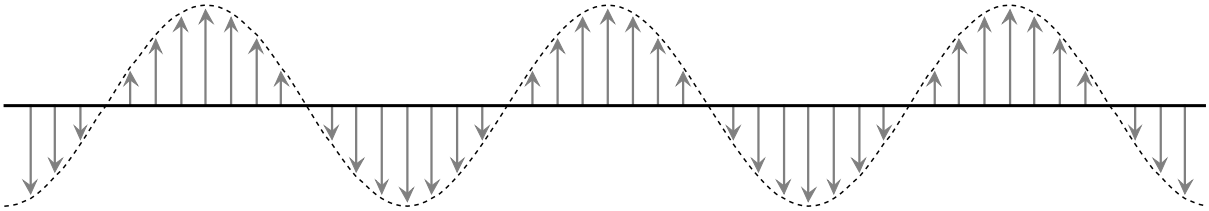


Fig. 11.1 – Changing the position of a surface point

In a rendering system, the displacement of a surface occurs *before* the calculation of the surface color that is dependent on the spatial orientation of that surface. The displaced position of each point may require the calculation of a new normal vector because the surface's orientation has also changed.

A second technique for modifying the geometric properties of a surface does not actually change its geometric data. Instead, the normal vector that a displaced surface *would have* is used for the associated point on the original surface. Because the normal vector is a critical parameter in the calculation of light interaction, the surface, though geometrically unchanged, has the appearance of a surface that has been displaced. The illusion that a geometrically smooth surface is "bumpy" gave this technique the name *bump mapping*.

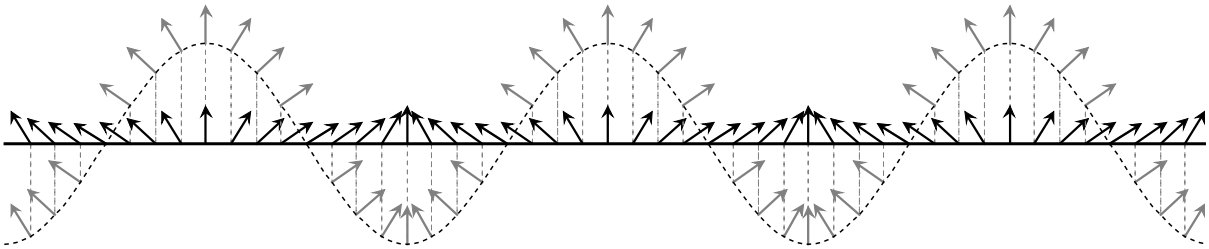


Fig. 11.2 – Modifying the normal vector defined at a surface point

Because bump mapping does not modify the geometry, the silhouette of a bump mapped object may not agree with the shape of the surface suggested by the lighting on that surface. However, when the scale of the detail provided by the bump mapping is small, this inconsistency of the silhouette may not be visible. In that case, bump mapping may be preferred over displacement mapping for reasons of efficiency. The additional geometric data created during the displacement mapping process (typically a mesh of small triangles) will require more processing time than the original, simpler surface.

11.1.1 Historical development of the two techniques

The reshaping of a surface through displacement mapping can be explained to artists new to rendering systems using an analogy of pushing and pulling of the surface of a moldable material like clay. Some commercial applications base their graphical interfaces on the idea that the original geometric object can be further modified in a sculptural way. However, mapping new normal vectors from an idealized surface may not be quite so intuitive. In the previous section, displacement mapping is described first because it is simpler to understand than bump mapping, and provides an introduction to a mapping from an idealized to an actual surface.

However, the research paper describing bump mapping preceded the later paper about displacement mapping by six years. In James Blinn's 1978 paper, "Simulation of wrinkled surfaces,[9]" he proposes a solution to the problem of surfaces that "sometimes look too smooth."

> To best generate images of macroscopic surface wrinkles and irregularities we must actually model them as such. Modelling each surface wrinkle as a separate [geometric] patch would probably be prohibitively expensive. We are saved from this fate by the realization that the effect of wrinkles on the perceived intensity is primarily due to the effect on the direction of the surface normal (and thus the light reflected) rather than their effect on the position of the surface. We can expect, therefore, to get a good effect from having a texturing function which performs a small perturbation on the direction of the surface normal before using it in the intensity formula.
>
> James F. Blinn, "Simulation of wrinkled surfaces," *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM, 1978, pp. 286-292.

Bump mapping may have been developed earlier than displacement mapping simply because it is considerably easier to implement in a rendering system. Only the three components of the normal vector for each point to be shaded had to be modified; creating additional geometric detail during rendering of sufficient resolution for fine detail would have often required computational resources unavailable at the time.

9. `https://www.microsoft.com/en-us/research/publication/simulation-of-wrinkled-surfaces/`

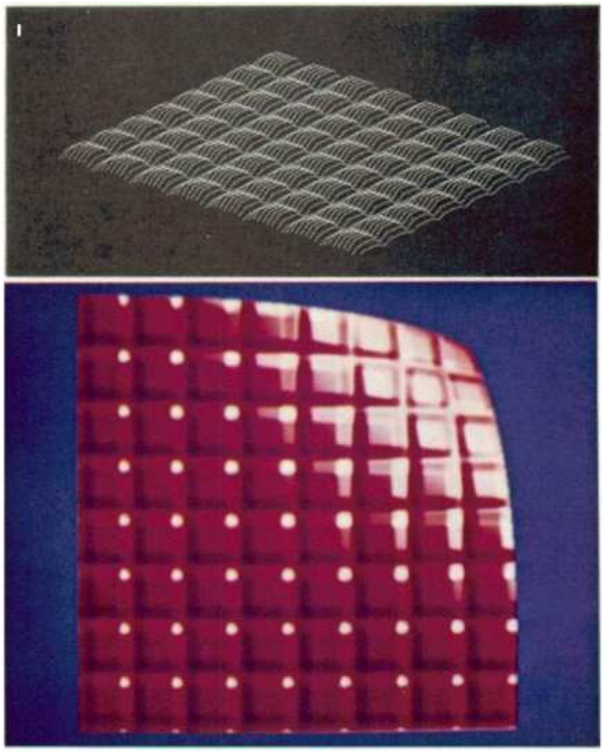


Fig. 11.3 – From James F. Blinn, "Simulation of wrinkled surfaces," 1978.

Figure 11.3, an illustration from Blinn's 1978 paper, demonstrates the problem with silhouettes in bump mapping at larger scales. The vector display image in the upper part of the illustration was used to visualize (and, undoubtedly, to debug) the function that defines the surface from which the new normal vectors should be calculated.

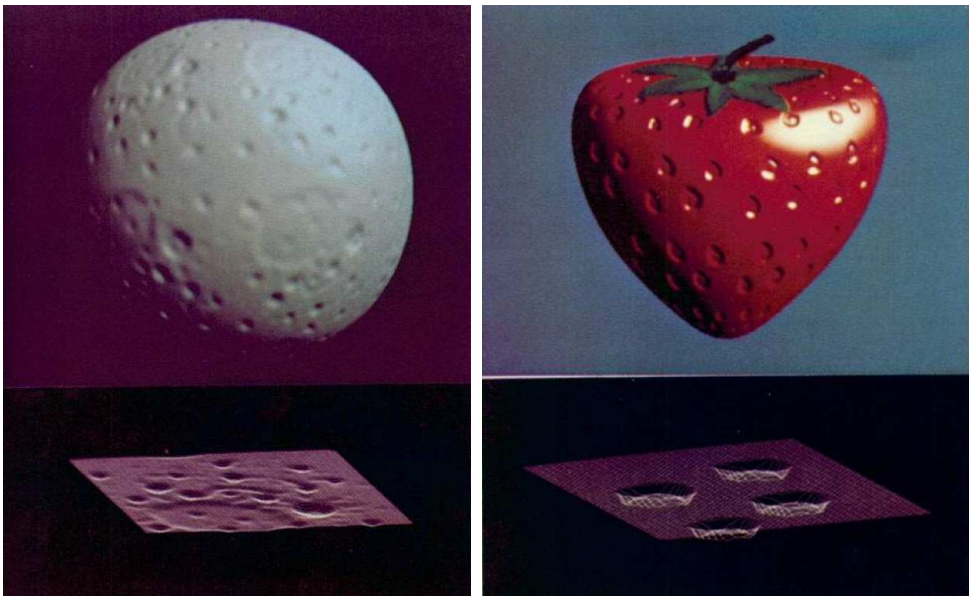


Fig. 11.4 – Illustrations from "Simulation of wrinkled surfaces"

Though the title of Blinn's paper uses the word "wrinkle," by the second page the term "bump function" is used to describe the calculation of the modified (or *perturbed*) surface normal, and this usage is consistent throughout the rest of the paper.

In 1984, Rob Cook published a paper about a hierarchical method for creating shading functions from modular components. In it, he mentions "an extension to bump maps."

> One of the more exotic uses of shade trees is an extension to bump maps called *displacement maps*. Since the location is an appearance parameter, we can actually move the location as well as perturbing the surface normal. Displacement maps began as a solution to the silhouette problems that arise when bump maps are used to make beveled edges. They are useful in many situations and can almost be considered a type of modeling.
>
> R. L. Cook, "Shade trees," *Computer Graphics (SIGGRAPH '84 Proceedings)*, ACM, 1984, pp. 223-231.

The development by Cook and others of a rendering system that first divided an object into triangles that were smaller than a pixel (*micropolygons*) provided numerous improvements in rendering capability, including an efficient implementation of displacement mapping.

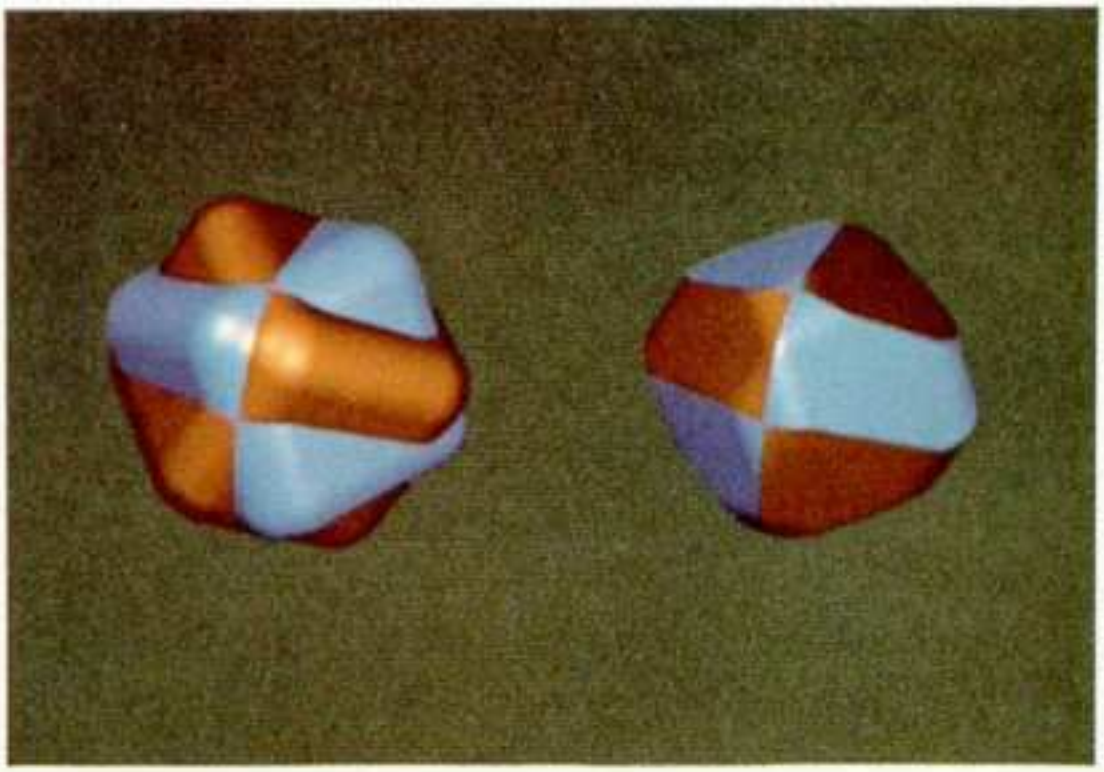


Figure 2. Union and Intersection of Two Cubes Beveled With Displacement Maps.

Fig. 11.5 – From R. L. Cook, "Shade trees," 1984.

Most professional rendering systems today provide both displacement and bump mapping. Artists, still mindful of efficiency, will keep the silhouette problem in mind when choosing between the two techniques. The following sections show how they are implemented in MDL.

11.2 The material_geometry struct

The previous chapters have used five of the six fields of the `material` struct. Control of displacement and bump mapping effects is provided in MDL by the sixth field of the struct, the `geometry` field:

Listing 11.1

```
struct material {
  uniform bool      thin_walled = false;
  material_surface  surface     = material_surface();
  material_surface  backface    = material_surface();
                                                        Light and surface interaction

  uniform color     ior         = color(1.0);
  material_volume   volume      = material_volume();
                                                        Light and interior interaction

  material_geometry geometry    = material_geometry();  Surface structure
};
```

The value of the `geometry` field is an instance of the `material_geometry` struct. The default values of its fields produce no change to the geometric structure or opacity of the object to which the material has been assigned.

Listing 11.2

```
struct material_geometry {
  float3 displacement = float3(0.0);   Modify surface position

  float  cutout_opacity = 1.0;         Modify surface opacity

  float3 normal = state::normal();     Modify apparent surface orientation
};
```

The modifications defined by the `material` struct's `geometry` field can change the renderer state that is used to calculate surface, volume and emissive properties, so geometric changes occur before those calculations begin. The surface is modified in the order of the fields of the `material_geometry` struct. If the surface is displaced, the value of `state::normal()` therefore becomes the normal vector of the displaced surface. It is this modified value that `state::normal()` will return when it is used in an expression to define a new value for the `normal` field of the `material_geometry` struct.

The individual fields modify the renderer state in the following ways:

`displacement`
: The `displacement` field defines a change to the spatial position of the surface. Note that the displacement parameter is a vector (a value of type `float3`). In early rendering systems, the displacement value was typically a single numeric value that defined how far along the normal vector the surface point should be moved. Instead of a scalar value, the `displacement` field is a *vector*. The direction of vector specifies the direction of displacement; the length of the vector defines the displacement distance. The use of vector length as a displacement control implies that the `displacement` field will not be nor-

malized (with a length of 1.0), unlike other typical vector parameters. The significance of vector length also clarifies why the default value of `displacement` is `float3(0.0)`—its length is zero, signifying no displacement.

`cutout_opacity`

The effect of `cutout_opacity` is not "opacity" or "transparency" in the sense of light interaction with transparent or translucent surfaces. Instead, `cutout_opacity` provides the type of control required for combining rendered elements in a traditional image-compositing framework.

The `cutout_opacity` field is a scalar value from 0.0 to 1.0, in which a value of 0.0 defines a material with no opacity at all, resulting in the apparent removal of the object from the scene. A value of 1.0 (the default) does not affect the rendering calculations defined by the other fields of the `material` struct. Values between 0.0 and 1.0 combine the surface color as calculated by the other fields of the `material` struct with the background, proportional to the `cutout_opacity` value. When there are no background objects, the material's color is combined with black. If the rendered image includes an alpha channel, the alpha value is the result of the accumulation of opacity values of all surfaces in that viewing direction.

Rather than interpreting `cutout_opacity` as defining continuous opacity values, a rendering system could instead treat it effectively as a Boolean value by comparing it to a given threshold. The rendering system could, for example, remove the object from the scene if the `cutout_opacity` value does not exceed the threshold.

`normal`

The vector value of the `normal` field replaces the value of `state::normal()` as calculated from the orientation of the surface. The expression can include a call to `state::normal()` itself, which may have been modified by a `displacement` value. Bump mapping is implemented in MDL by supplying the modified normal vector for the surface as the value of this field.

The next two chapters show the effect of the three fields of the `material_geometry` property in combination with the other properties of the `material` struct using the objects in Figure 11.6, rendered here with simple diffuse reflection.

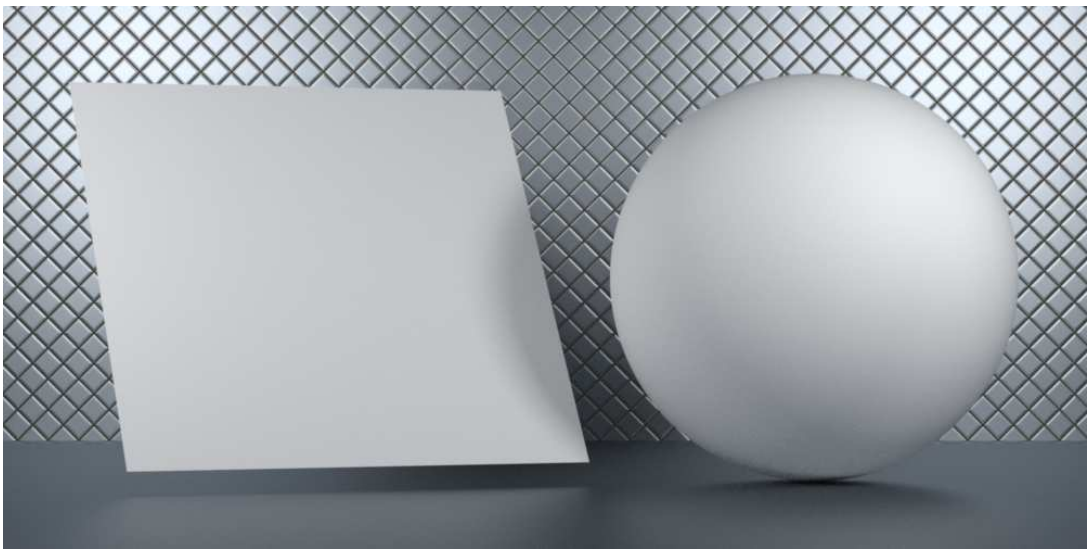


Fig. 11.6 – The objects used in the displacement mapping and normal perturbation examples

12 Displacement mapping: moving a surface point

Displacing a point is implemented in a straightforward way in the `material_geometry` struct: the `displacement` field is assigned a `float3` value that defines the direction and distance in which the surface point should be moved before the calculation of surface color begins. *How* this value is determined is typically dependent upon a association of some feature of surface position to the resulting value—the "mapping" that gives this technique its name.

12.1 Defining displacement distance with a function

The initial displacement examples of this section define a relationship between surface position and the sine function. To map values from the sine function to a useful range for the following materials, three custom functions will be useful.

First, the `fit` function takes the proportional relationship of a number between a minimum and a maximum value and returns the corresponding number that has the same proportional relationship between a different minimum and maximum.

Listing 12.1

```
float fit(
    float v,                                Original value
    float old_min, float old_max,           Range of original value
    float new_min, float new_max)           Range of new value
{
   float scale = (v - old_min) / (old_max - old_min);
   return new_min + scale * (new_max - new_min);
}
```

The `fit` function can be used to map the output of the sine function—which varies from -1.0 to 1.0—to a new minimum and maximum. This second utility function, `sinusoid`, maps the fractional part of the original value to a sine function. An argument defines the frequency of the sine wave.

Listing 12.2

```
float sinusoid(
   float f,
   float frequency=1.0,
   float minval=-1.0,
   float maxval=1.0)
{
```

```
    float fraction = math::frac(f);   Fractional component

    float angle = fraction * math::TWO_PI;   Scale input to single period

    float scaled = angle * frequency;   Multiple periods

    float sine_value = math::sin(scaled);   Value of sine function
   return fit(
      sine_value,
       -1.0, 1.0,   Range of sine function

       minval, maxval);   Range of result
}
```

Displaying the output of the `sinusoid` function (through a translation to a graphing package) shows the intended effects of the parametric modification of the sine function.

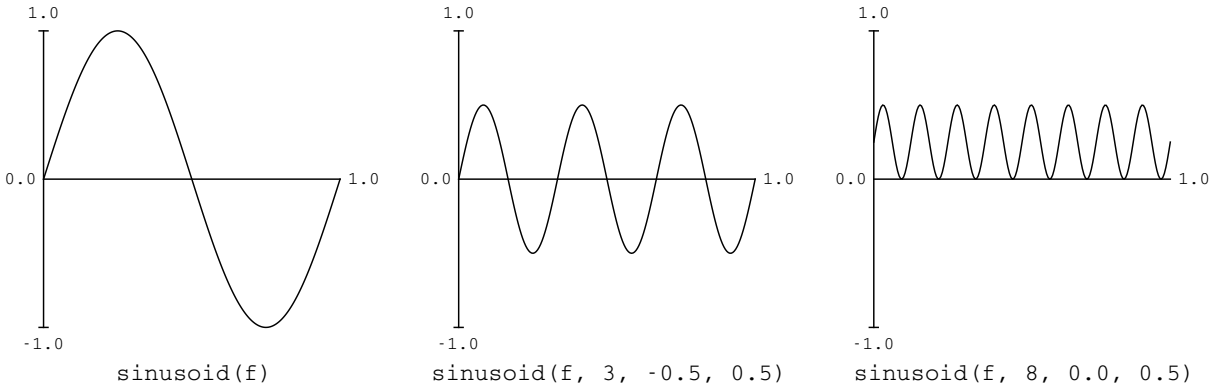


Fig. 12.1 – A utility function for mapping values from 0.0 to 1.0 to a sinusoid

The `fit` and `sinusoid` utilities do not call any functions specific to rendering. The following function, `uv_wave`, uses as input the *uv* coordinates of an object. The result of `state::texture_coordinate(0)` provides the inputs to two calls of function `sinusoid` for both the *u* and *v* component values, stored as the first and second components of local variable `p`. The to calls to `sinusoid` are added together for the result. Control of the separate calls to `sinusoids` provided through three `float2` parameters: `frequency`, `minval` and `maxval`.

Listing 12.3

```
float uv_wave(
   float2 frequency = float2(1.0),
   float2 minval = float2(-1.0),
   float2 maxval = float2(1.0))
{
```

```
    float3 p = state::texture_coordinate(0);
```
Current uvw coordinate

```
    float u_component =
      sinusoid(p[0], frequency[0], minval[0], maxval[0]);
```
u contribution

```
    float v_component =
      sinusoid(p[1], frequency[1], minval[1], maxval[1]);
```
v contribution

```
    return u_component + v_component;
```
Combined u and v

```
}
```

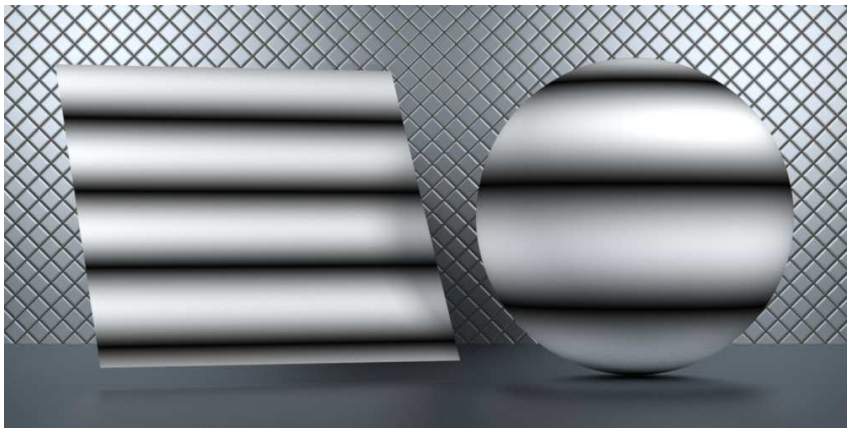
Before using function `uv_wave` for displacement, its intended behavior can be further verified by treating the result of `uv_wave` as the `tint` parameter in `df::diffuse_reflection_bsdf`.

Listing 12.4

```
material waves_as_grayscale (
  uniform float2 frequency = float2(1.0),
  uniform float2 gray_max = float2(0.5)) =
let {
    color tint = color(
      uv_wave(
        frequency, float2(0.0), gray_max));
```
Grayscale value based on uv coordinates

```
} in material (
  surface: material_surface (
    scattering: df::diffuse_reflection_bsdf (
        tint: tint)));
```
Using the uv-dependent grayscale value for tint

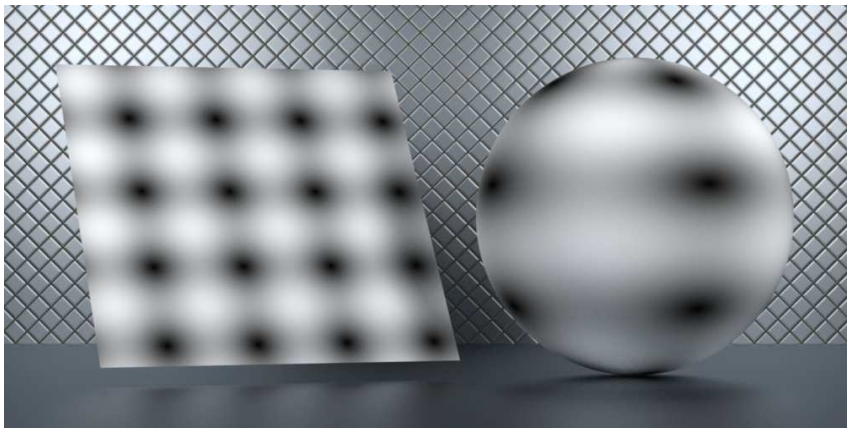
The components of a `color` parameter should range from 0.0 to 1.0, so the `minval` argument of `uv_wave` is set to `float2(0.0)`. With this value of zero, the `maxval` argument then determines the intensity of the `tint` color. Rendering with only the *v* value defining the color variation produces Figure 12.2:



```
waves_as_grayscale(
  frequency:
    float2(0.0, 4.0),
  gray_max:
    float2(0.0, 1.0))
```

Figure 12.2

Using both the *u* and *v* components, the `gray_max` parameter is adjusted to `0.5, 0.5` so that the total of the two components never exceeds 1.0.



```
waves_as_grayscale(
  frequency:
    float2(4.0, 4.0),
  gray_max:
    float2(0.5, 0.5))
```

Figure 12.3

Material `waves_as_displacement` uses function `uv_waves` for displacement values. The `distance` parameter defines the maximum amount of displacement from the original surface, so the arguments to the `minval` and `maxval` parameters of `uv_wave` are `-distance` and `distance`.

Listing 12.5

```
material waves_as_displacement (
   uniform float2 frequency = float2(1.0),
   uniform float2 distance = float2(0.1),
   color tint = color(0.7)) =
let {
   float displacement_distance =
      uv_wave(frequency, -distance, distance);

   float3 displaced_normal =
      state::normal() * displacement_distance;
} in material (
   surface: material_surface (
      scattering: df::diffuse_reflection_bsdf (
         tint: tint)),
   geometry: material_geometry(
      displacement: displaced_normal));
```

Calculate uv-based displacement distance

Scale surface normal by displacement distance

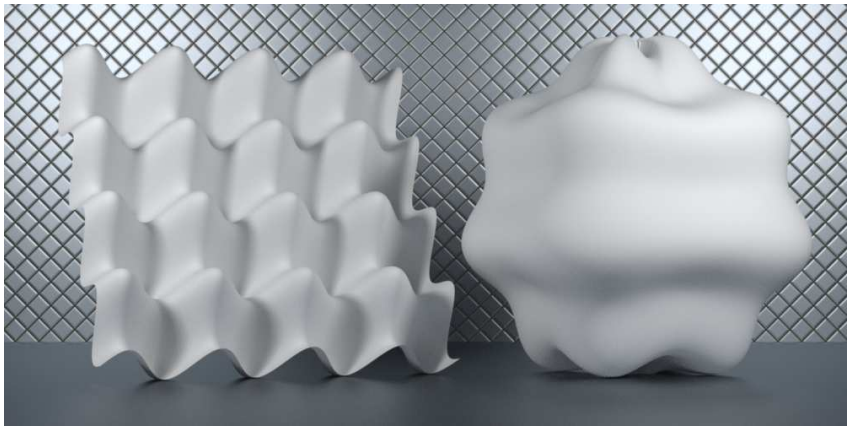
Use modified surface normal in material_geometry instance

Duplicating the frequency parameters from the previous renderings of `waves_as_grayscale` and using 0.1 for the non-zero distance produces Figure 12.4 (page 197) and Figure 12.5 (page 197).

```
waves_as_displacement(
  frequency:
    float2(0.0, 4.0),
  distance:
    float2(0.0, 0.1))
```

Figure 12.4

In Figure 12.5, both the *u* and *v* components contribute to the calculation of the displacement distance:



```
waves_as_displacement(
  frequency:
    float2(4.0, 4.0),
  distance:
    float2(0.075, 0.075))
```

Figure 12.5

Note that a value of 0.0 for either `frequency` or `distance` for the *u* or *v* component effectively disables the effect of that component in the resulting displacement. However, a `frequency` value of 0.0 still requires the execution of the `math::frac` and `math::sin` functions in function `sinusoid`. It is therefore tempting to extend `sinusoid` to handle that special case (and removing the pedagogical local variables):

Listing 12.6

```
float sinusoid2(float f,
                float frequency=1.0,
                float minval=-1.0,
                float maxval=1.0) {
  return
      frequency == 0.0
```

Check if the calculation is unnecessary

```
        ? 0.0
```

If not (the frequency is zero), return 0.0

```
        : fit(math::sin(
              math::frac(f) * math::TWO_PI * frequency),
             -1.0, 1.0, minval, maxval);
}
```

...else call the "fit" function

Rendering systems can integrate MDL for appearance definition in a variety of ways. In the rendering of these images, the check for a zero value in `sinusoid2` only reduced the rendering time to 98% of the original version. This suggests that the calculation of the diffuse surface color requires most of the rendering time. Performing the test again with images that are four times larger in resolution reinforces this assumption: `sinusoid2` provides even less of an improvement, taking 99% of the time of the original version. For higher resolution images, the increased complexity of `sinusoid2` provides no real benefit in decreasing rendering time.

The pitfalls in any attempt to improve the performance of MDL materials are the same as those in traditional programming. It may not be obvious which part of a rendering system will benefit from a proposed optimization of a calculation in a material. For any material that will be part of a shared library which will be maintained and extended over time, the possible lack of clarity in optimized material code—and the subsequent difficulty in the addition of new features and the debugging of those features—can be too high a cost for the benefit of a potentially limited decreased in rendering time.

12.2 Level-of-detail considerations

The last section cautioned against premature optimization—the "root of all evil" according to Donald Knuth's famous warning.[10] However, the rendering of the `waves_as_grayscale` version only took 57% of the time required by `wave_as_displacement`. During production, animation tests of composition and timing may not require the fine detail that displacement mapping provides.

The following material, `waves`, uses the `uv_wave` function for either displacement or shading based on a Boolean parameter. If `use_displacement` is `false`, the `tint` argument is scaled based on the `uv_wave` function. This allows the intended displacement of an object to be visible during tests that do not benefit from displacement's increase in complexity and the resulting increase in rendering time.

Listing 12.7

```
material waves (
   uniform float2 distance = float2(0.1),
   uniform float2 frequency = float2(1.0),
   color tint = color(0.7),
   bool use_displacement = true) =          Displacement or color choice
let {
   color modified_tint =
     use_displacement ?
     tint :
     tint * uv_wave(frequency, float2(0.1), float2(0.5));
                                            Surface color
```

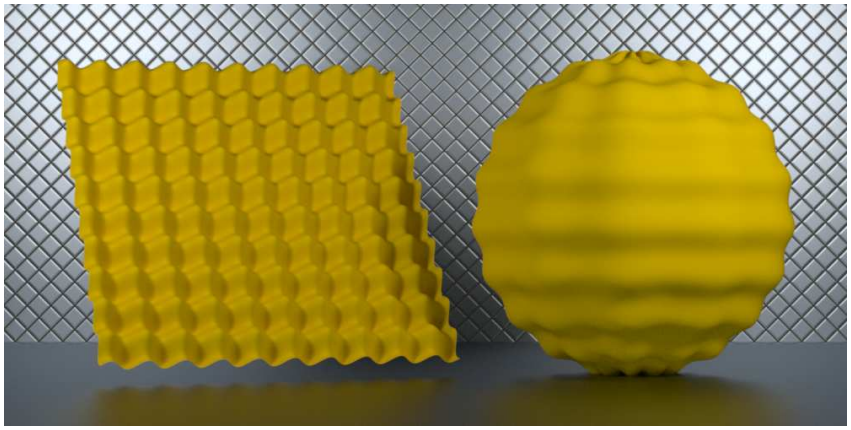
10. http://en.wikiquote.org/wiki/Donald_Knuth#Computer_Programming_as_an_Art_.281974.29

```
    float displacement_distance =
      use_displacement ?
      uv_wave(frequency, -distance, distance) :
      0.0;
    float3 displaced_normal =
      state::normal() * displacement_distance;
} in material (
  surface: material_surface (
    scattering: df::diffuse_reflection_bsdf (
        tint: modified_tint)),
  geometry: material_geometry(
    displacement: displaced_normal));
```

Modification of normal vector

Color based on value of “use_displacement” parameter

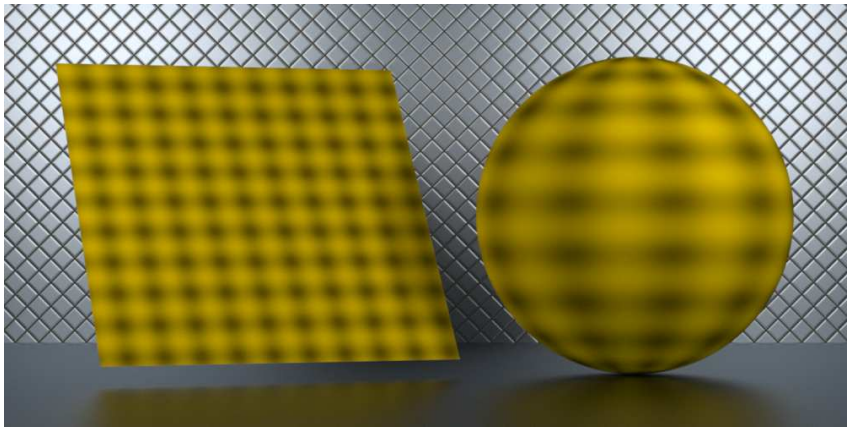
In this first rendering using material `waves`, the `displace` parameter is set to `true` (the default).



```
waves(
  distance:
    float2(0.02, 0.02),
  frequency:
    float2(10.0, 10.0),
  tint:
    color(0.6, 0.4, 0.0),
  use_displacement: true)
```

Figure 12.6

Setting the `displace` parameter to `false`, the pattern of displacement is still visible by a modification of the `surface` value in the material, providing compositional information without the overhead of the actual displacement.



```
waves(
  distance:
    float2(0.02, 0.02),
  frequency:
    float2(10.0, 10.0),
  tint:
    color(0.6, 0.4, 0.0),
  use_displacement: false)
```

Figure 12.7

In material `waves`, displacements are shown by scaling the color value proportional to the distance of the displacement. In a production context, a representation of displacement should clearly be a stand-in for displacement and not the intended surface color. The `wave` material demonstrates the utility of such multiple behaviors encapsulated in one material by using a Boolean parameter to specify the parametric changes required by those behaviors.

12.3 Separating displacement from the control of light interaction

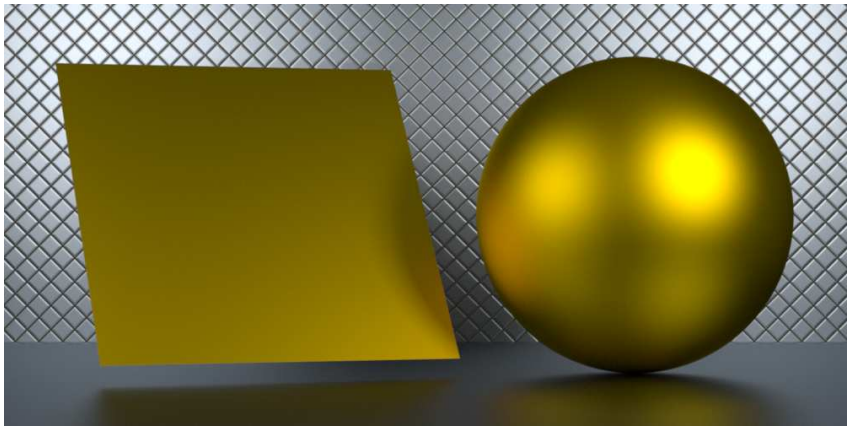
In the previous materials, the type of light interaction of the surface is a constant, defined within the material. This section demonstrates how material a material that defines surface and volume properties can be combined with a material that defines displacement.

Material `glossy` defines the `surface` property but does not specify any modification in the `geometry` property.

Listing 12.8

```
material glossy (
  color tint = color(0.7),
  float roughness = 0.3) =
material (
  surface: material_surface (
      scattering: df::simple_glossy_bsdf (
        tint: tint,                          Glossy BSDF
        roughness_u: roughness)));
```

Rendering the scene with `glossy` produces Figure 12.8:



```
glossy(
  tint:
    color(0.6, 0.4, 0.0))
```

Figure 12.8

In material `waves`, the value of the `surface_property` is an explicit constructor for the `material-surface` struct. As described in "Reusing parts of existing materials" (page 100), components of a material can be used as the field values of other materials. Using the dot operator, the `surface` property of material `glossy_waves` is assigned the `surface` value of an instance of material `glossy`.

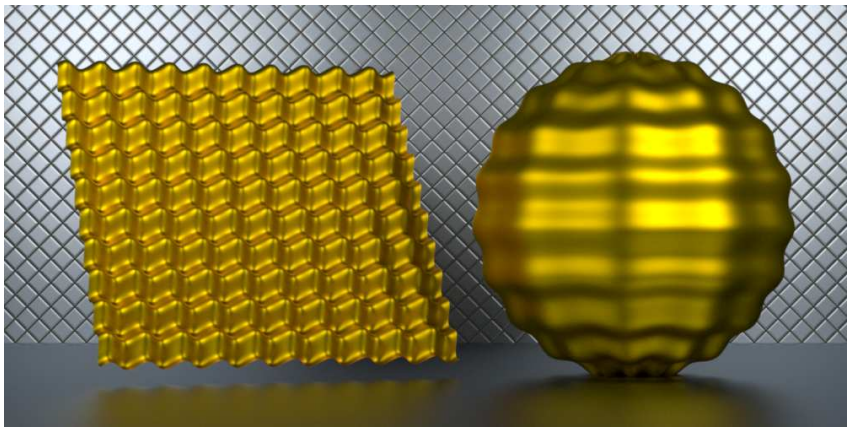
Listing 12.9

```
material glossy_waves (
  uniform float2 distance = float2(0.1),
  uniform float2 frequency = float2(1.0),
  color tint = color(0.7),
  float roughness = 0.3,
  bool use_displacement = true) =
let {
  color modified_tint =
    use_displacement ?
    tint :
    tint * uv_wave(frequency, float2(0.1), float2(0.5));
  float displacement_distance =
    use_displacement ?
    uv_wave(frequency, -distance, distance) :
    0.0;
  float3 displaced_normal =
    state::normal() * displacement_distance;
} in material (
  surface: glossy(
    modified_tint, roughness).surface,
  geometry: material_geometry(
    displacement: displaced_normal));
```

Duplicated structure from material “waves”

The “surface” property of an instance of “glossy”

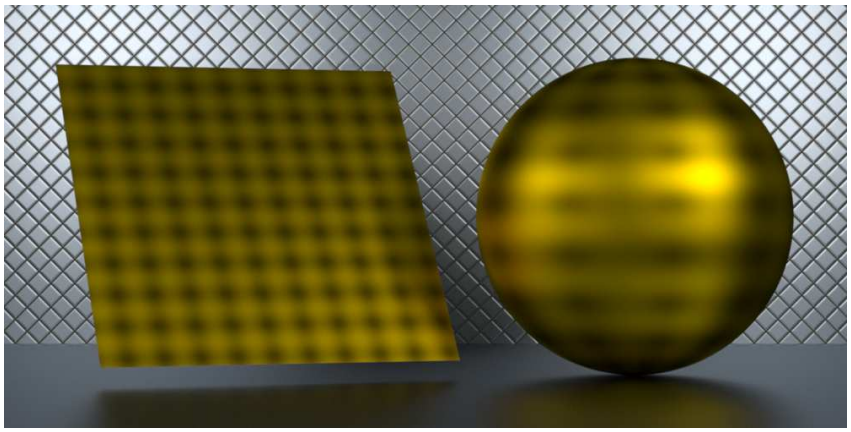
The Boolean `displace` parameter from `waves` has been duplicated in material `glossy_waves`. Rendering with displacement produces Figure 12.9:



```
glossy_waves(
  distance:
    float2(0.02, 0.02),
  frequency:
    float2(10.0, 10.0),
  tint:
    color(0.6, 0.4, 0.0),
  use_displacement: true)
```

Figure 12.9

Turning `displacement` off—using the color modification behavior instead—produces Figure 12.10 (page 202). The non-displaced surface is also rendered with the `glossy` material, but scaled by the displacement value.



```
glossy_waves(
  distance:
    float2(0.02, 0.02),
  frequency:
    float2(10, 10),
  tint:
    color(0.6, 0.4, 0.0),
  use_displacement: false)
```

Figure 12.10

In material `glossy_waves`, the intuitive similarity between creating a material instance and calling a function without named parameters has been emphasized syntactically. The `surface` value could also have been defined in this way:

Listing 12.10

```
surface: glossy(
  tint: modified_tint,
  roughness: roughness).surface
```

The typical advice about consistency also applies in the implementation of materials, especially during library design. Even when positional arguments would be possible, this book typically uses named arguments to clarify the field names of the struct for which argument values are being provided. (See material `glossy_round_bumps` (page 220) for another example of positional arguments.)

12.4 A parameterized displacement material

A further separation of the calculation of light interaction from displacement is possible. A parameter of a material can be an instance of another material. Like material `glossy_waves`, material `waver` uses the dot operator to extract components of a struct instance. However, `waver` defines a parameter of type `material`, named `surface_and_volume`, from which the dot operator is used to extract the values of its `ior`, `surface` and `volume` fields.

Listing 12.11

```
material waver (
    material surface_and_volume,          Material as an argument
    uniform float2 distance = float2(0.1),
    uniform float2 frequency = float2(1.0)) =
let {
    float displacement_distance =
      uv_wave(frequency, -distance, distance);          Calculate displacement distance

    float3 displaced_normal =
      state::normal() * displacement_distance;          Scale surface normal by displacement distance
```

```
} in material (
   ior: surface_and_volume.ior,
   surface: surface_and_volume.surface,
   volume: surface_and_volume.volume,
   geometry: material_geometry(
       displacement: displaced_normal));
```

Use surface component of argument

Use volume component of argument

Define displacement using the scaled surface normal

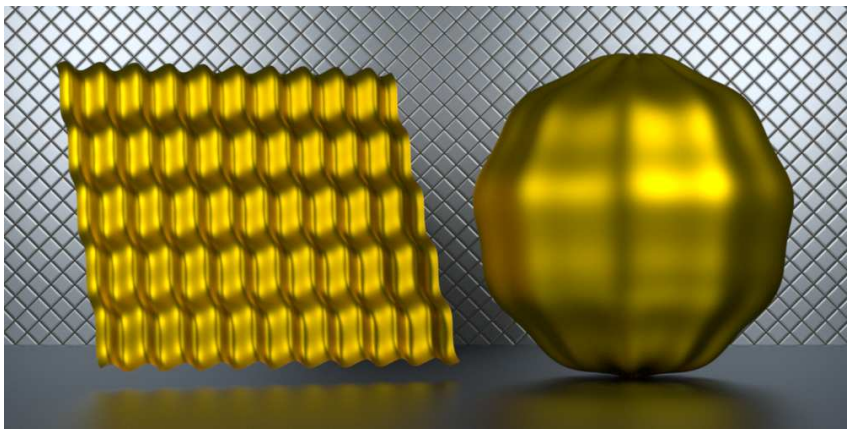
Material `glossy_yellow_waves` uses `material waver`; an instance of the `glossy` material is the value of the `surface_and_volume` parameter of `waver`.

Listing 12.12

```
material glossy_yellow_waves(
   uniform float2 distance = float2(0.1),
   uniform float2 frequency = float2(1.0)) =
   waver(
       surface_and_volume:
          glossy(color(0.6, 0.4, 0.0), 0.3),
      distance: distance,
      frequency: frequency);
```

Create an instance of “glossy” material for use by “waver” material

Rendering with `glossy_yellow_waves` produces Figure 12.11:



```
glossy_yellow_waves(
  distance:
    float2(0.02, 0.02),
  frequency:
    float2(10, 5))
```

Figure 12.11

Material `glass waves` uses the material `subsurface_scattering` from “Subsurface scattering as a model of glass” (page 69) but with the same displacement parameters as `glossy_yellow_waves`.

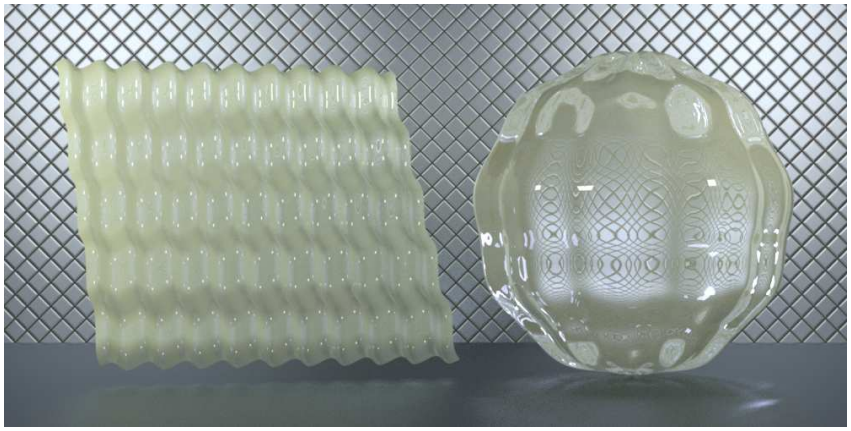
Listing 12.13

```
material glass_waves (
  uniform float2 distance = float2(0.1),
  uniform float2 frequency = float2(1.0)) =
waver(
  surface_and_volume:
    subsurface_scattering(
      transmission_color: color(0.8, 0.9, 1.0),
      scattering_color: color(0.1, 0.1, 0.3),
      transmission_distance: 2.0),
  distance: distance,
  frequency: frequency);
```

An instance of material “waver” is the entire body of material “glass_waves”

Instance of “subsurface_scattering” material that provides the surface and volume properties to material “waver”

Rendering with `glass_waves` produces Figure 12.12:



```
glass_waves(
  distance:
    float2(0.02, 0.02),
  frequency:
    float2(10, 5))
```

Figure 12.12

A further level of abstraction is possible, however. The following material, `shaper`, only takes two arguments of type `material`. One argument provides surface and volume properties; the other argument provides the geometric property.

Listing 12.14

```
material shaper (
  material surface_and_volume,
  material geometry) =
material (
  ior: surface_and_volume.ior,
  surface: surface_and_volume.surface,
  volume: surface_and_volume.volume,
  geometry: geometry.geometry);
```

Material argument for surface and volume properties

Material argument for geometry property

Surface and volume properties from material argument

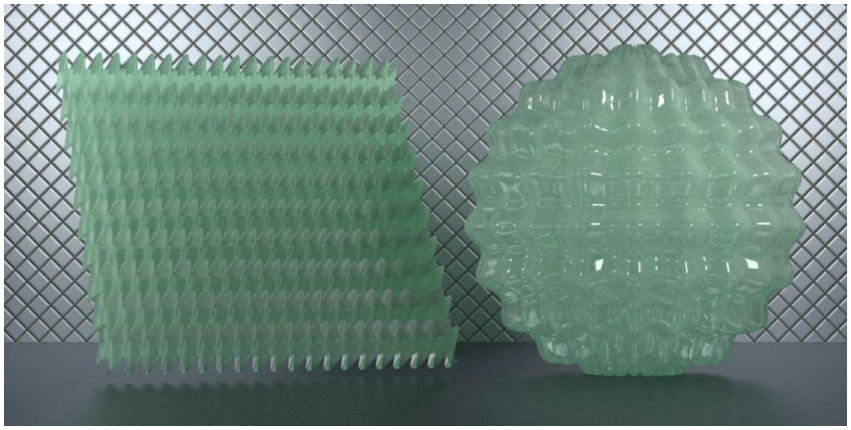
Geometry property from material argument

Using material `shaper`, the `subsurface_scattering` and `waves_as_displacement` materials can be combined.

Listing 12.15

```
material green_glass_waves (
  uniform float2 distance = float2(0.1),
  uniform float2 frequency = float2(1.0)) =
shaper (                    An instance of material "shaper" is the entire body of material "green_glass_waves"
  surface_and_volume:
    subsurface_scattering (
      transmission_color: color(0.1, 0.4, 0.2),
      scattering_color: color(0.1, 0.1, 0.1),
      transmission_distance: 1.0),          Volume and color properties
  geometry:
    waves_as_displacement (
      frequency: frequency,
      distance: distance));                 Geometry property
```

Rendering with material `green_glass_waves` produces Figure 12.13:



```
green_glass_waves(
  distance:
    float2(0.04, 0.04),
  frequency:
    float2(20, 10))
```

Figure 12.13

Materials that can combine the components of other materials can be used as the framework for graphical applications in which artists use preexisting materials in the creation of new appearance models.

12.5 Defining displacement distance with an image

In the materials introduced in "Mapping from spatial parameters to an image" (page 166), the texture images provided color data for use as the `tint` parameter of `df::diffuse_reflection_bsdf`. The pixel data contained in an image can also be used for other types of material input, for example, to describe displacement distance. The association of sampled pixel values as a displacement parameter uses the same *uv* coordinate structure and lookup function as the last chapter did for an image's color values.

To use an image for displacement values, Figure 12.14 shows the intensities of a one-dimensional slice of pixel values interpreted as distance, rather than as color. An image which represents distance in this way is often called a *height map*.

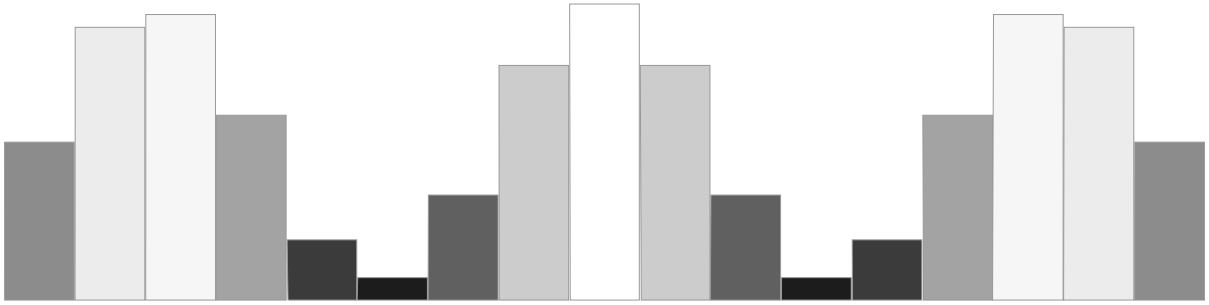


Fig. 12.14 – Visualizing pixel intensity as height

Given the geometric interpretation of pixel intensity, Figure 12.15 suggests an approximation of a curve that passes through the center of each pixel.

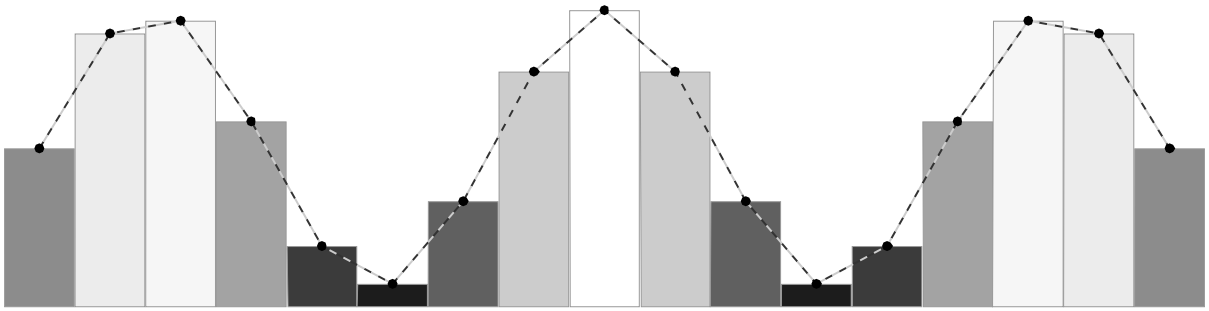


Fig. 12.15 – Defining an approximation to a curve using the pixel intensity interpreted as height

Images of higher resolution can represent a greater level of geometric detail, as in Figure 12.16. Usually the the *uv* coordinates are not exactly centered on a pixel; in this case, the renderer's mechanism for sampling digital images provide an intermediate value in the same way an image is sampled when a color value is required.

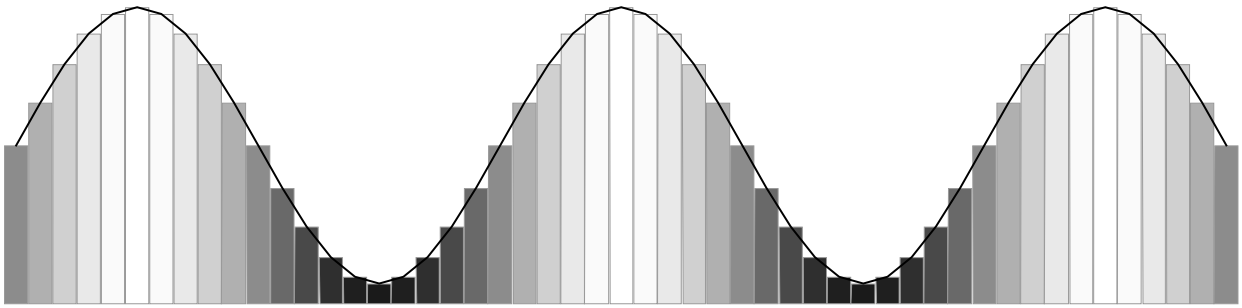


Fig. 12.16 – Increased resolution in the height map enables finer geometric detail

Figure 12.17 (page 207) is a rendering in perspective of a geometric model of woven fibers. The gray values in the image are proportional to the distance of each point to the plane of the weave.

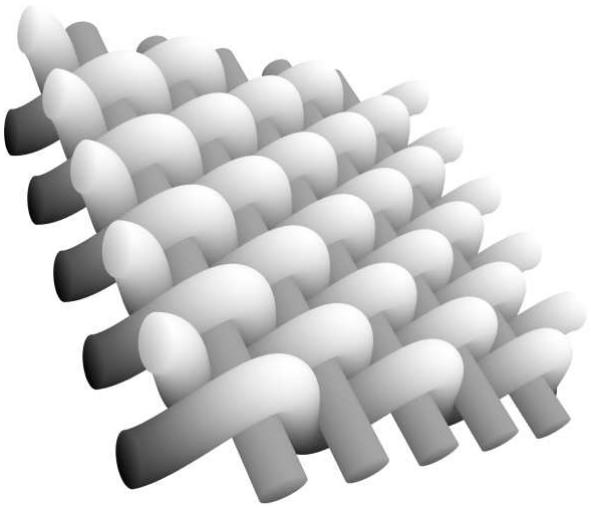


Fig. 12.17 – Rendering using depth as a grayscale value

In Figure 12.18, the fiber model was rendered with the camera oriented at 90° to the plane of the weave to produce a height map. No perspective transformation was used in rendering, and the image has been cropped so that it can can be repeated without any visible seams at the borders.

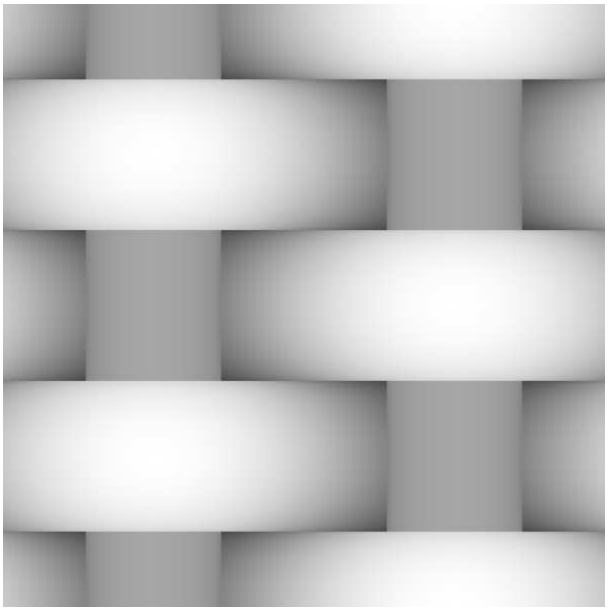


Fig. 12.18 – Orthographic rendering for a tiled displacement map (filename: weave_depth_hires.png)

Figure 12.18 is used in material `diffuse_weave` as the source for the `let` variable `texture_color`:

Listing 12.16

```
material diffuse_weave (
  uniform float2 scale = float2(1.0),
  uniform float displace = 0.0,
  color tint = color(0.7)) =
```

```
let {
   color texture_color =
     functions::texture_2d_lookup_scaled(
       texture_2d("weave_depth.png"),
       scale[0], scale[1]);
```

Get color value from texture map using a function developed in the last chapter

```
   float scaled_displace =
     displace * math::average(texture_color);
```

Scale displacement length by the average of the color's component

```
   float3 displaced_normal =
     state::normal() * scaled_displace;
```

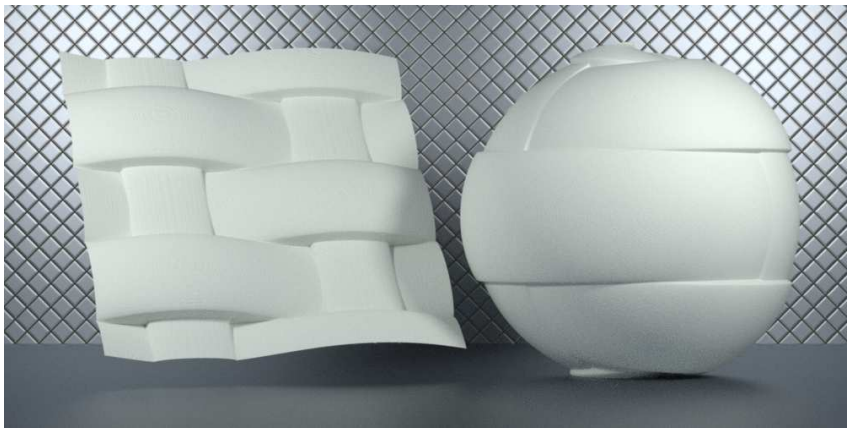
Modify the normal vector by the scaled displacement length

```
} in material (
   surface: material_surface (
     scattering: df::diffuse_reflection_bsdf (
       tint: tint)),
   geometry: material_geometry(
       displacement: displaced_normal));
```

Use modified normal in the geometry property

The function `texture_2d_lookup_scaled` developed in "Mapping from spatial parameters to an image" (page 166) can also be used in a material that will use the color value for displacement, instead of as a color value. Images for use in mapping functions are handled in a consistent manner; *how* that value is used will vary from material to material.

Rendering with material `diffuse_weave` produces Figure 12.19.



```
diffuse_weave(
  displace: 0.1,
  scale:
    float2(1, 1),
  tint:
    color(0.72, 0.75, 0.7))
```

Figure 12.19

The displacement map has been designed so that it can be used as a tile. The `scale` parameter of material is passed to the `u_scale` and `v_scale` parameters of `functions::texture_2d_lookup_scaled`. Scaling by 8 in the u direction and 4 in the v direction produces Figure 12.20 (page 209).

```
diffuse_weave(
  displace: 0.05,
  scale:
    float2(8, 4),
  tint:
    color(0.72, 0.75, 0.7))
```

Figure 12.20

The `diffuse_weave` material uses `df::diffuse_reflection_bsdf` for its `surface` value. Like the `glossy_waves` (page 201) and `waver` (page 202) materials, the reflection property of `diffuse_weave` can be generalized by using a material instance as one of its parameters.

Listing 12.17

```
material weave (
  uniform float2 scale = float2(1.0),
  uniform float displace = 0.0,
  material surface_property = material()) =
let {
  color texture_color =
    functions::texture_2d_lookup_scaled(
      texture_2d("weave_depth.png"),
      scale[0], scale[1]);
  float scaled_displace =
    displace * math::average(texture_color);
  float3 displaced_normal =
    state::normal() * scaled_displace;
} in material (
  surface: surface_property.surface,
  geometry: material_geometry(
    displacement: displaced_normal));
```

Material instance to provide the surface property

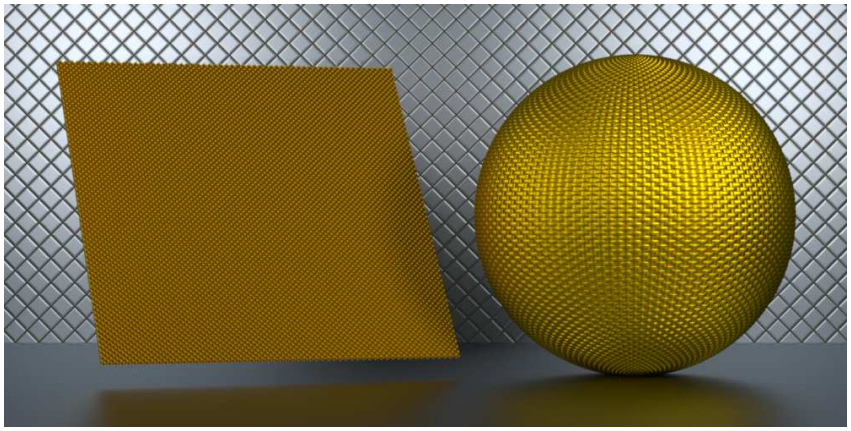
Use the surface property of the material instance

Now that the surface property has been parameterized, a material can reuse the `glossy` material from "Separating displacement from the control of light interaction" (page 200). The `gold_weave` material uses the `weave` material for its implementation, for which the the `glossy` material provides its surface property.

Listing 12.18

```
material gold_weave(
  uniform float2 scale = float2(1.0),
  uniform float displace = 0.0) =
  weave(                    An instance of the “weave” material
    scale: scale,
    displace: displace,
    surface_property:
      glossy(               An instance of the “glossy” material
        tint: color(0.6, 0.4, 0.0),
        roughness: 0.3));
```

Using the `gold_weave` material produces Figure 12.21.



```
gold_weave(
  displace: 0.015,
  scale:
    float2(60, 30))
```

Figure 12.21

13 Bump mapping: perturbing a surface normal

So far, the features of the MDL material struct have been easily related to intuitions from everyday experience—the interaction of light and objects in MDL is modeled on the physical world. Even the displacement mapping of the previous sections can be thought of as the manipulation of a pliable substance like clay.

However, with bump mapping the intuitive world will be left behind. In bump mapping, an artifact of the rendering system—the representation of surface orientation by a vector—is the entity being manipulated, a programming construct for which there is no corresponding object or process in the physical world.

Without an intuitive base, an incremental approach with visual verification—similar to the color replacement of displacement values in the previous sections—will be helpful in developing the new intuitions required for materials that bend the normal vector.

13.1 Background: Working with a local coordinate system

In the previous sections, displacement mapping only required scaling the normal vector—simply by multiplying the normal vector by a number—to produce the `displacement` value for the `material_geometry` struct. No other information was required. However, to bend the normal vector in a controlled way requires that it be rotated, but around which axis? How can the normal vector be consistently manipulated across the varying orientation of the surface?

To define a consistent context for normal vector manipulations, renderings systems typically define a *local coordinate system* using some parameter of the surface. A typical choice is the *uv* coordinate system, which defines a *u* direction and a *v* direction everywhere throughout the texture space. For any point on the surface, three vectors are therefore available: the normal vector, the *u* vector and the *v* vector. These three vectors can define a coordinate system local to that point. Bump mapping calculations can use this consistent framework everywhere across the surface.

A rendering system using this mechanism for the definition of a local coordinate system will provide some way of accessing the texture vectors. In MDL, the *u* and *v* vectors are defined by two state functions: `state::texture_tangent_u()` and `state::texture_tangent_v`. These two vectors define a two-dimensional coordinate system for any tangent vector that passes through that point, called a *tangent space*. Combining the tangent vectors with the normal vector produces a three-dimensional coordinate system defined for every point on the surface.

Figure 13.1 (page 212) displays a representation of the local coordinate system across the surface of a sphere.

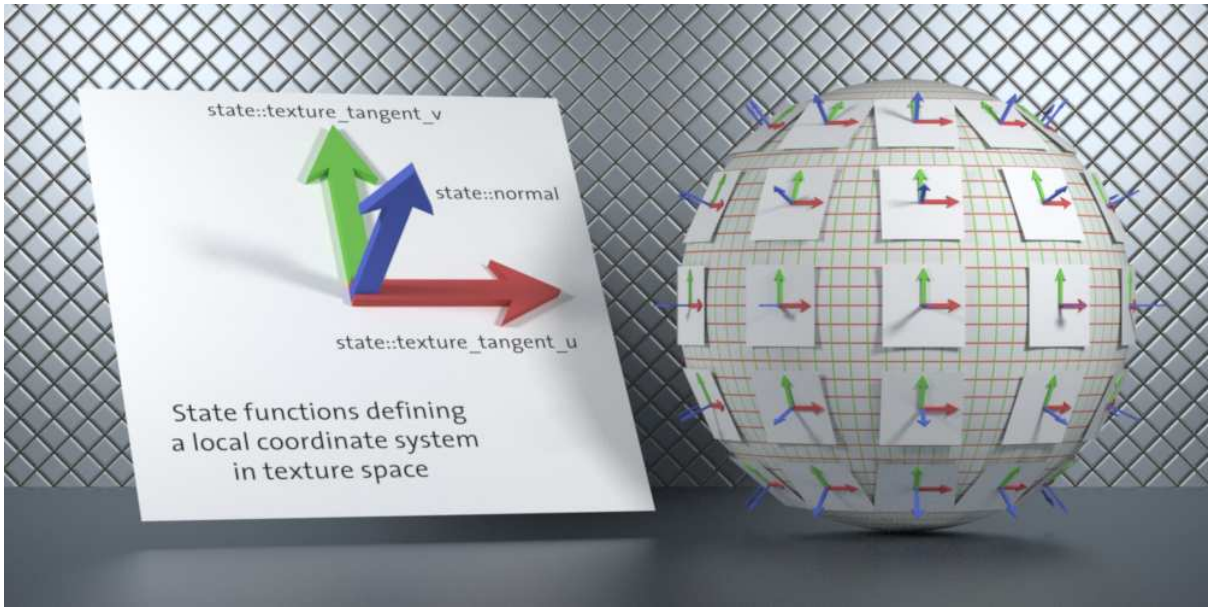
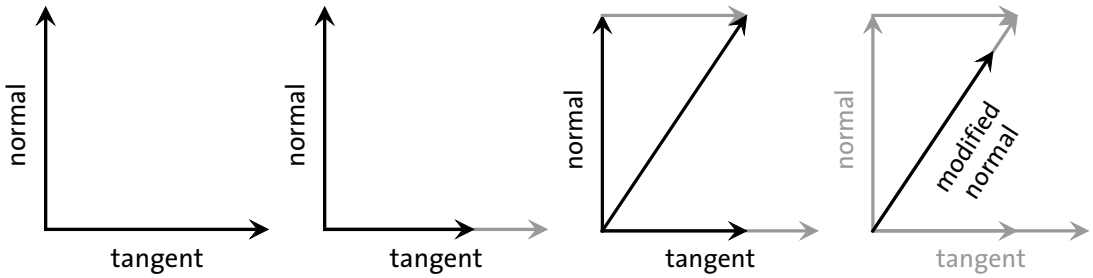


Fig. 13.1 – A local coordinate system based on texture space is defined everywhere on the object's surface

As Figure 13.1 implies, the tangent vectors allow any bump mapping calculations to assume that there is consistent coordinate system everywhere across the surface. In essence, considerations of surface curvature have been removed.

A further simplification is possible. Rather that considering the three-dimensional space defined by the three local coordinate vectors and calculating rotations in that three-dimensional space, the u and v vectors can be considered separately. The u vector and the normal vector form a plane; the normal vector can be modified in this plane by adding it to the result of scaling the tangent. This modification is consistent across the surface, given that it is based on the plane formed by components of the local coordinate system.



1. Initial state of normal and tangent vectors

2. Scale the tangent vector

3. Add the scaled tangent vector to the normal

4. Normalize the normal vector

Fig. 13.2 – Modifying the normal vector by scalar multiplication and vector addition

Modifying the normal vector in this way using the two tangents produces two vectors. Adding these two vectors together and normalizing the result produces the modified normal vector.

Other methods for modifying the surface normal are possible (for example, rotating the normal vector around both tangent vectors and adding the result), but this planar technique will allow the development of a relatively simple utility function in "Modifying the normal vector within the circle" (page 217). First, however, it will be useful to simplify the texture space as well.

13.2 Tiling the texture space

Bump mapping is often used for patterns of fine detail repeated across a surface. Repetition can be implemented as a modification of the *uv* texture space.

Function `fractional_uv` multiples the *u* and *v* coordinates by the scale factor arguments; function `math::frac` truncates these values so they remain the range 0.0 to 1.0.

Listing 13.1

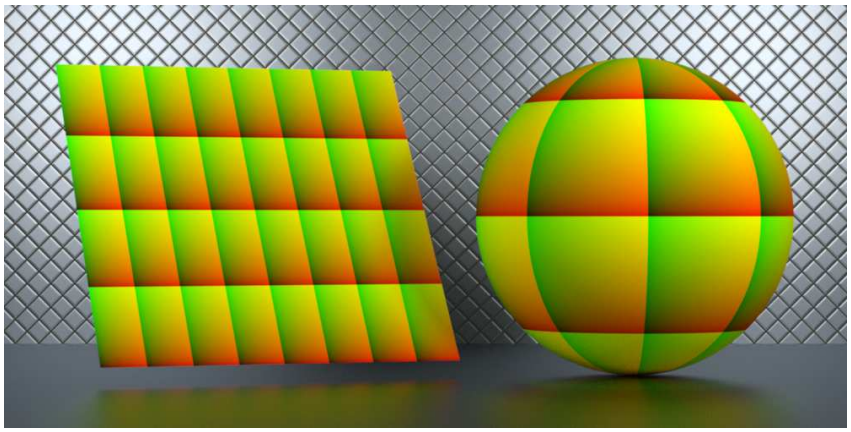
```
float2 fractional_uv(float u_scale, float v_scale) {
  float3 uvw = state::texture_coordinate(0);
  return float2(
      math::frac(uvw.x * u_scale),    Scale u coordinate

      math::frac(uvw.y * v_scale));   Scale v coordinate
}
```

In a manner similar to the tests of displacement values, the scaling of the *uv* coordinates can be implemented as a material:

Listing 13.2

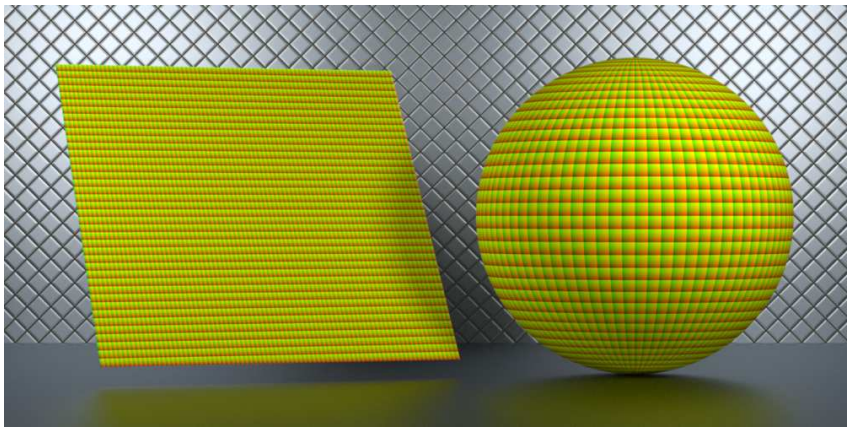
```
material scaled_uv(
  float u_scale = 1.0,
  float v_scale = 1.0) =
let {
  float2 uv = fractional_uv(u_scale, v_scale);    Scaled uv coordinates

  color tint = color(uv[0], uv[1], 0.0);    Create color from scaled uv coordinates
} in
material (
  surface: material_surface (
    scattering: df::diffuse_reflection_bsdf (
        tint: tint)));    Use color derived from uv coordinates
```

Figure 13.3 (page 214) and Figure 13.4 (page 214) show the results of different scaling factors for the `fractional_uv` function.



```
scaled_uv(
  u_scale: 8,
  v_scale: 4)
```

Figure 13.3



```
scaled_uv(
  u_scale: 80,
  v_scale: 40)
```

Figure 13.4

The simple arithmetic involved in the `fractional_uv` function can hide what it in fact produces: a tiling of the surface in which each tile defines a unit square in *uv* space. The next step is to put a circle in each of those tiles.

13.3 Defining a circle within a tile

The virtual tiles in texture space created by the `fraction_uv` function provide a frame of reference to define a circle centered in the tile. The circle in Figure 13.5 (page 215) has a radius of 0.5. Any *uv* point is inside the circle if the distance to the center is less than or equal to 0.5.

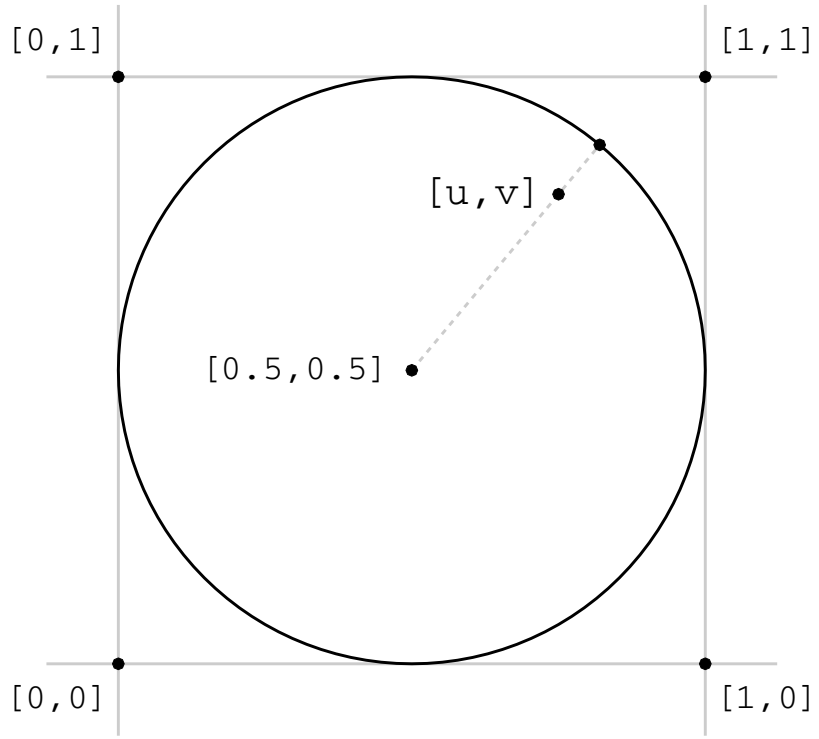


Fig. 13.5 – The uv coordinate in a circle within the texture tile

In the spirit of the previous tests that converted the *uv* coordinate to a color, the `scaled_uv_circle_color` function displays the *uv* value inside the circle. On the outside of the circle the `outside_color` argument is used instead.

Listing 13.3

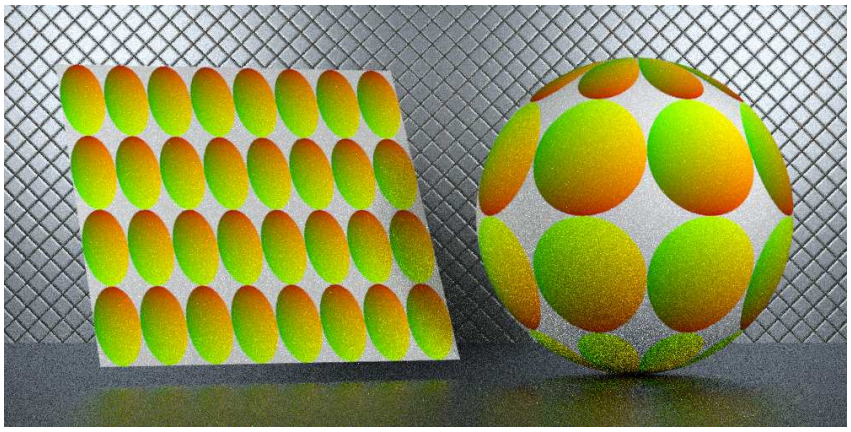
```
color scaled_uv_circle_color(
  float u_scale, float v_scale, color outside_color)
{
  float2 uv = fractional_uv(u_scale, v_scale);
  float2 center = float2(0.5, 0.5);   Center of circle

  if (math::distance(uv, center) < 0.5) {   Is distance from center greater than radius?

    return color(uv[0], uv[1], 0.0);   If so, use uv color inside circle...
  } else {
    return outside_color;   ...else use value of outside_color parameter.
  }
}
```

The `scaled_uv_circle` material uses `scaled_uv_circle_color` for the definition of the `tint` field:

Listing 13.4

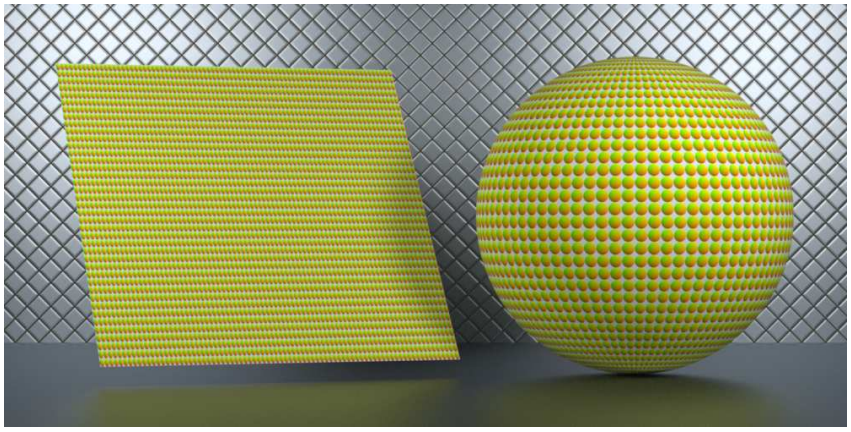
```
material scaled_uv_circle(
  float u_scale = 1.0,
  float v_scale = 1.0,
  color outside_color = color(0.7)) =
let {
  color tint = scaled_uv_circle_color(     Calculate uv color based on circle
    u_scale, v_scale, outside_color);
} in
material (
  surface: material_surface (
    scattering: df::diffuse_reflection_bsdf (
      tint: tint)));                       Use color derived from the uv position
```

Figure 13.6 and Figure 13.7 show the results of rendering with `scale_uv_circle`.



```
scaled_uv_circle(
  u_scale: 8,
  v_scale: 4)
```

Figure 13.6



```
scaled_uv_circle(
  u_scale: 80,
  v_scale: 40)
```

Figure 13.7

13.4 Modifying the normal vector within the circle

The examples of the previous section visualize a technique for determining *uv* values within a repeated circular pattern. This technique can now be used to determine where and how the normal vector should be modified for an example of bump mapping.

The geometric structure to be simulated with bump mapping will be a section of a sphere, called a *spherical cap*. Figure 13.8 shows the normal vectors at a cross section of a cap.

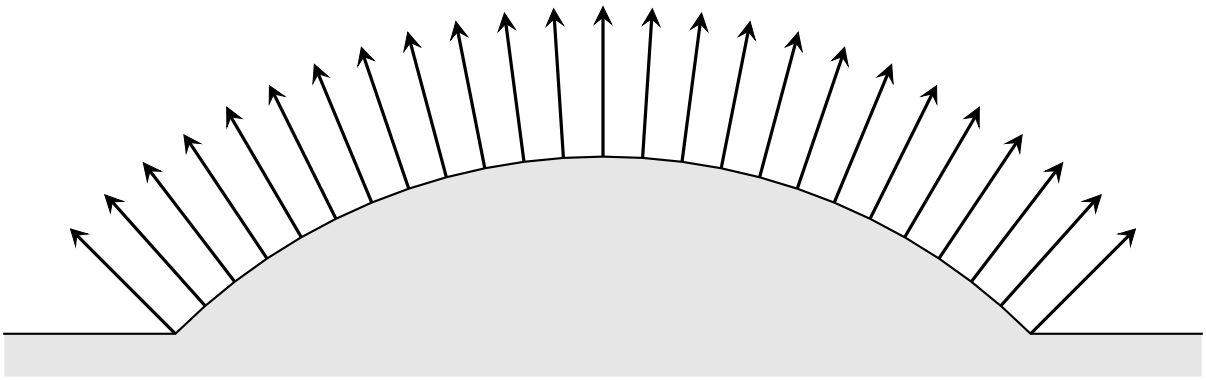


Fig. 13.8 – The normal vectors for a spherical cap

To simulate a spherical cap with bump mapping, the normal vectors of the cap can be associated with a scalar value. Figure 13.9 shows one possible strategy, in which a value from -0.5 to 0.5 is associated with the normal vectors from one side of the cap to the other.

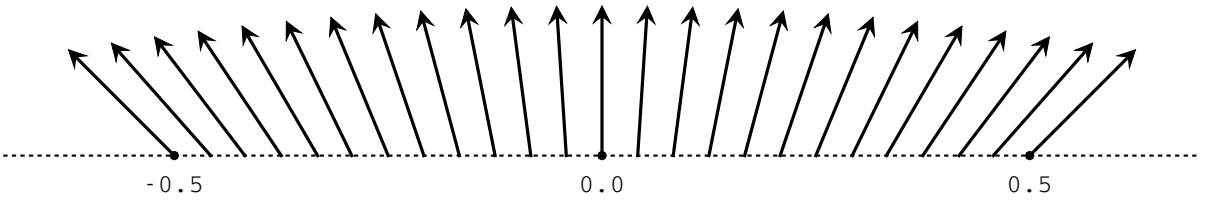


Fig. 13.9 – Normal vectors modified across the range of 0.0 to 1.0

How can these vectors be constructed? Figure 13.10 shows the varying components of the spherical cap's normal vectors. Note that the horizontal component increases in size in proportion to its distance from 0.0.

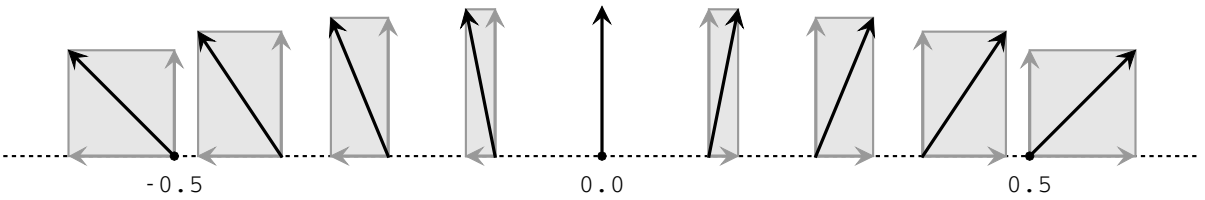


Fig. 13.10 – Components of the vectors representing the surface normal vectors of the spherical cap

The *u* and *v* tangent vectors, both of length 1.0, can be scaled by the *u* and *v* components of the current point. Their sum is a vector from the center of the circle to the current *uv* point.

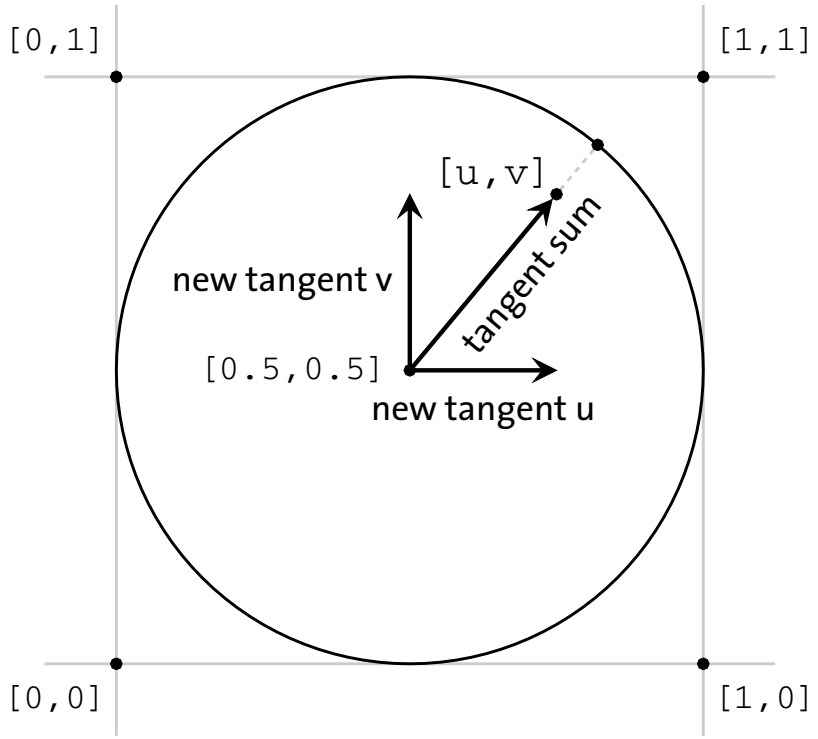


Fig. 13.11 – Sum of tangents scaled by uv components

Adding the tangent sum vector to the normal vector will resulting in bending the normal in a manner that approximates a spherical surface. The following function, `round_bump_normal`, uses the technique of Figure 1.2 (page 212), to modify `state::normal()`, but only if the *uv* point is within or on the boundary of the circle.

Listing 13.5

```
float3 round_bump_normal(
  float bump_scale, float u_scale, float v_scale)
{
  float2 uv = fractional_uv(u_scale, v_scale);
  float2 center = float2(0.5, 0.5);
  float3 result;
  if (math::distance(uv, center) <= 0.5) {
```

Is the point inside the circle?

```
    float2 center_offset = uv - center;
```

Then: Make directional offset from center

```
    float3 new_tangent_u =
      state::texture_tangent_u(0) * center_offset[0];
```

Scale tangent in u direction

```
    float3 new_tangent_v =
      state::texture_tangent_v(0) * center_offset[1];
```

Scale tangent in v direction

```
        float3 tangent_sum =
          new_tangent_u + new_tangent_v;          Sum the two scaled tangent vectors

        float3 scaled_tangent_sum =
          tangent_sum * bump_scale;               Scale tangent sum by "bump_scale" argument

        float3 bent_normal =
          state::normal() + scaled_tangent_sum;   Add tangent vector to normal vector

        float3 normalized =
          math::normalize(bent_normal);           Normalize vector

        result = normalized;                      The result for points inside the circle

    } else {                                      Is the point outside the circle?

        result = state::normal();                 Then: The result is the unmodified surface normal
    }
    return result;
}
```

In the following material, `round_bumps`, the `round_bump_normal` function defines the modified normal in the `material_geometry` property. Like the `waver` material in "A parameterized displacement material" (page 202), the surface property of `round_bumps` has also been parameterized.

Listing 13.6

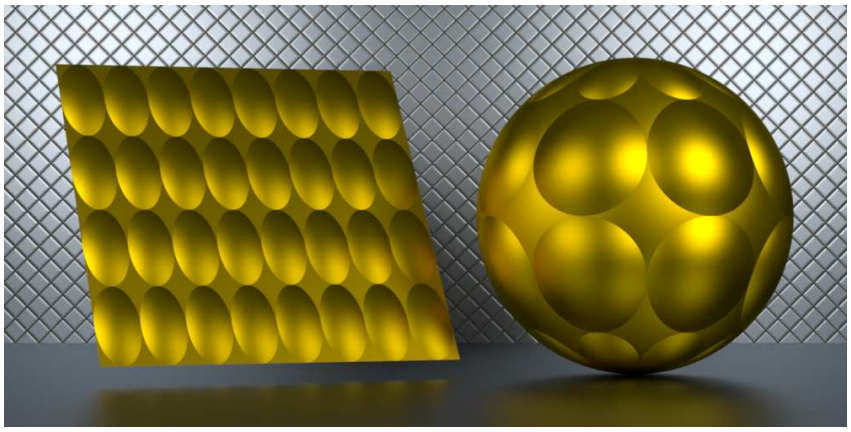
```
material round_bumps(
    material surface_material,                Material providing "material_surface" property
    float bump_scale = 1.0,
    float u_scale = 1.0,
    float v_scale = 1.0) =
let {
    float3 modified_normal = round_bump_normal(
      bump_scale, u_scale, v_scale);          Modified normal created by function
                                              "round_bump_normal"
} in
material(
    surface: surface_material.surface,        Extract the "surface" property from the
                                              "surface_material" argument
    geometry: material_geometry (
        normal: modified_normal));            Using the modified normal in "material_geometry"
```

The body of the following material, `glossy_round_bumps`, is the material `round_bumps`, using the `glossy` material (from "Separating displacement from the control of light interaction" (page 200)) as the `surface_material` argument.

Listing 13.7

```
material glossy_round_bumps(
  float bump_scale = 1.0,
  float u_scale = 1.0,
  float v_scale = 1.0) =
round_bumps(
  glossy(color(0.6, 0.4, 0.0)),     Material providing “material_surface” component
  bump_scale, u_scale, v_scale);    Parameters to “round_bump_normal” function
```

Rendering with `glossy_round_bumps` produces Figure 13.3 (page 214).

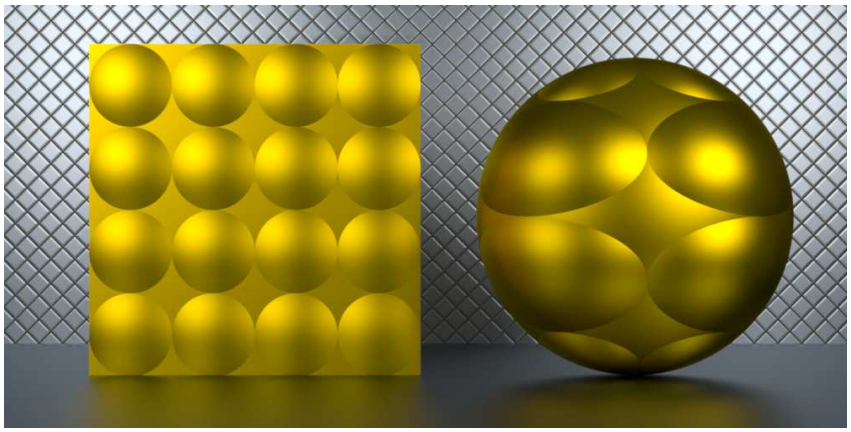


```
glossy_round_bumps(
  u_scale: 8,
  v_scale: 4)
```

Figure 13.12

Though the glossy reflections of the sphere in Figure 13.12 makes apparent visual sense, there is something wrong with the square. The lighting environment consists of two light sources, one above and to the left, the other above and to the right. What is the source of the light reflected from the lower part of the spherical caps?

As another test of `glossy_round_bumps`, Figure 13.13 orients the square directly toward the virtual camera and distributes the circular regions evenly across its face.

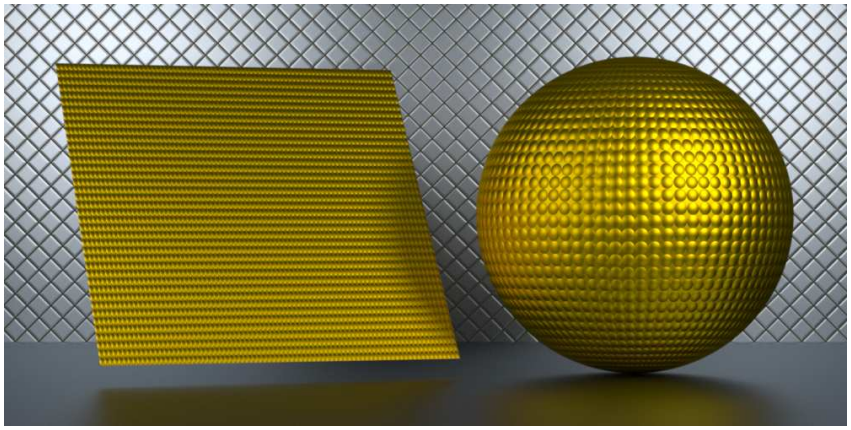


```
glossy_round_bumps(
  u_scale: 4,
  v_scale: 4)
```

Figure 13.13

Now the glossy reflections on the square in Figure 13.13 (page 220) make visual sense. The orientation of the square in Figure 13.12 (page 220) is deceptive; it is tilted over much more than the image suggests. Because the geometry is not modified—only the normal vectors have changed—the self-shadowing of the surface that would be expected does not exist. Figure 13.12 (page 220) demonstrates the primary problem with bump mapping at larger scales, when geometric modification is necessary.

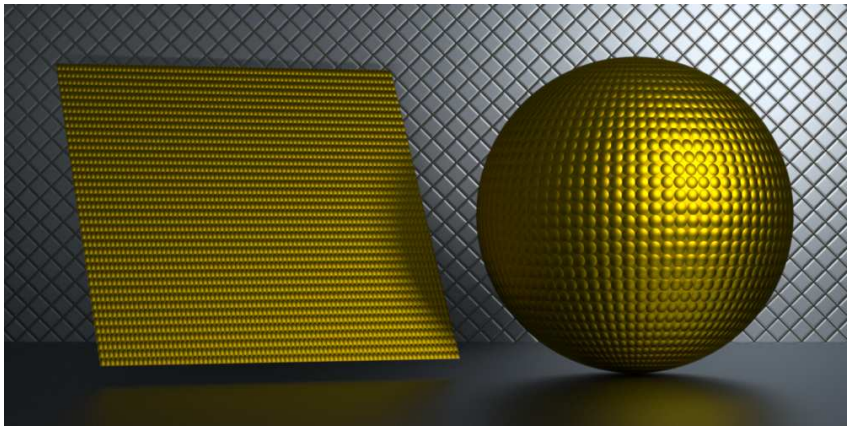
Figure 13.14 shows a more typical use of bump mapping at a finer level of detail.



```
glossy_round_bumps(
  u_scale: 80,
  v_scale: 40)
```

Figure 13.14

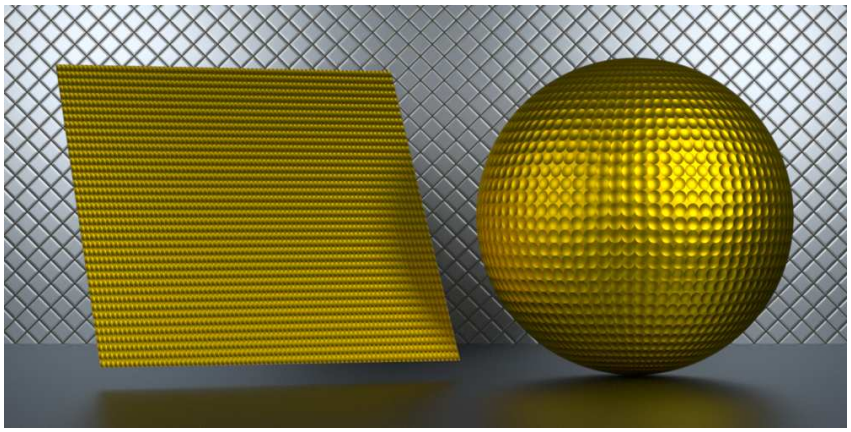
As with many rendered effects, the lighting environment is crucial for convincing bump mapping. Figure 13.15 removes the light source from the left; in various regions of the sphere the effect may appear to be more convincing.



```
glossy_round_bumps(
  u_scale: 80,
  v_scale: 40)
```

Figure 13.15

The `bump_scale` argument provides a way of inverting the apparent geometry of the bumps. For example, Figure 13.16 (page 222) multiples the computed normal vector by -1.0—in effect, turning the bumps inside out.



```
glossy_round_bumps(
  u_scale: 80,
  v_scale: 40,
  bump_scale: -1.0)
```

Figure 13.16

13.5 Deriving normals from a height map image

Using an image to define bump mapping requires a more complicated calculation than displacement mapping. In displacement mapping, each point in an image defines, for that corresponding point in the *uv* space on the surface, the distance that point is to be moved from its original position, implemented as a scalar multiplication of the surface normal.

However, bump mapping depends upon the orientation of the surface—a three-dimensional problem that cannot be defined by a scalar value derived from an image. Instead, a triangle of three nearby points in an image—again treating pixel intensity as height—can define a plane that a new normal vector will represent.

13.5.1 A two-dimensional simplification of surface orientation

The same decomposition into the separate calculations of the *u* and *v* vectors applied in the previous sections can be used image-based bump mapping. Figure 13.17 shows the two-dimensional slice of the surface implied by pixel values, including the normal vectors of each segment.

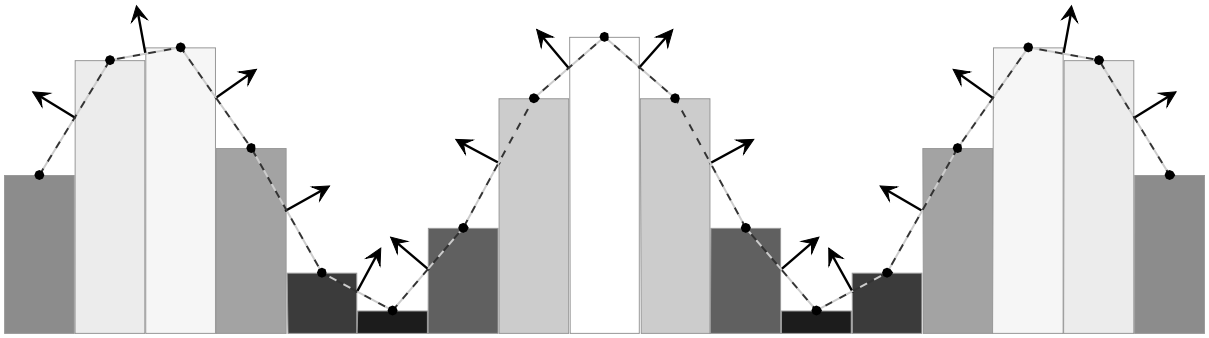


Fig. 13.17 – Defining normal vectors by interpreting pixel intensity as height

As in the case of displacement mapping, increasing the number of pixels in bump-map image improves the rendered detail of the surface—not by the increase of geometric detail, but by improved representation of surface that the normal vectors represent.

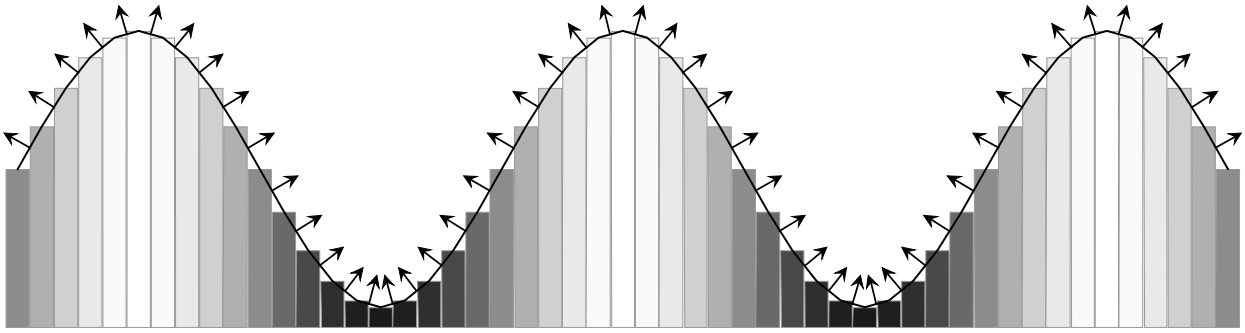


Fig. 13.18 – Increased image resolution improves the detail implied by the derived normal vectors

13.5.2 The normal vector of a line segment

Figure 13.18 suggests that the curve implied by a set of samples in an image can be treated as a series of line segments that connect the sample points. A simple geometric property of the slope of a line will be useful here. Slope is defined as the ratio of the change in the y direction to the change in the x direction. If the slope of a line is `y/x`, then the slope of the line 90° to the given line is `x/-y`. Figure 13.19 suggest how that relationship results in a 90° rotation of the rectangle, and therefore the diagonal within it.

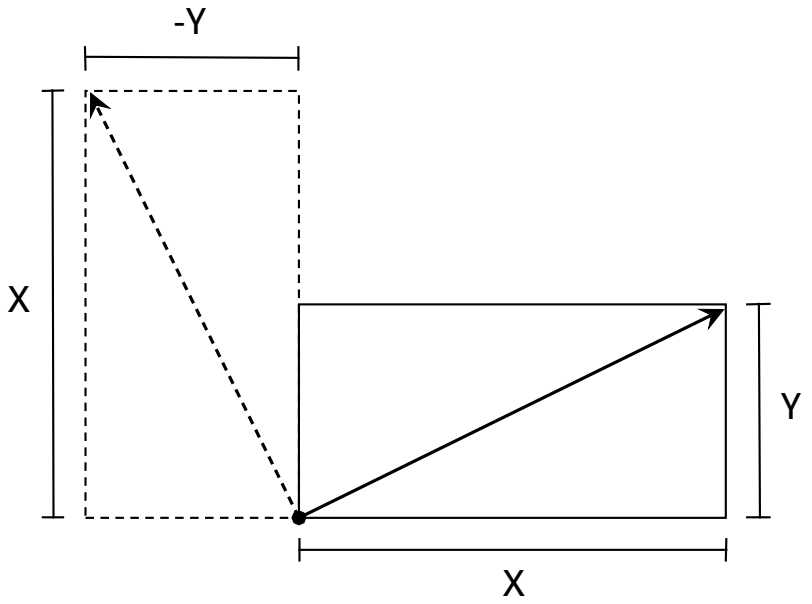


Fig. 13.19 – The negative reciprocal of the slope of a line is a line 90° to it

This simple relationship will permit the efficient creation of normal vectors from line segments. Creating the rectangle from image pixels is the problem to solve next.

13.5.3 Constructing the normal vector

As in the case of displacement mapping with an image, pixels will be interpreted as rectangles in which their height is proportional to intensity. The "center" of each pixel will be the endpoints of the line segment.

Given these two points, the `x` and `y` lengths of Figure 13.18 (page 223) can be calculated. Given the variable aspect ratios of texture images, it is useful to normalize the inter-pixel length between 0.0 and 1.0, or within the range of texture space.

For an image with width `w`, the width of a pixel in texture space is `1/w`. Similarly, for an image of height `h`, the height of a pixel in texture space is `1/h`. In this case, however, the difference between "width" and "height" of an image can be ignored; like the previous bump mapping examples, the two components of the modified normal will be calculated separately. The distance in Figure 13.20 will be handled in the same way in both the u and v directions.

The `y` component of Figure 13.18 (page 223) is calculated by sampling the texture at the two points and taking the difference of the values. Figure 13.20 implies that the exact value of a pixel is used to determine the difference, but the distance is arbitrary (and will become a parameter in the final bump-mapping material).

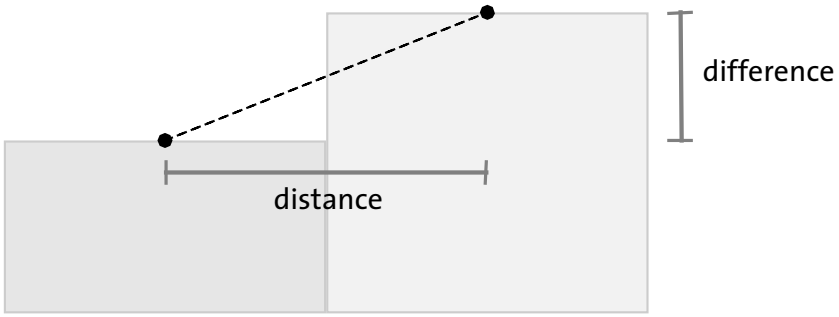


Fig. 13.20 – The distance between the sample points and the difference between the value of the samples, considered geometrically

The distance and difference values define the lengths of the sides of a rectangle in Figure 13.21.

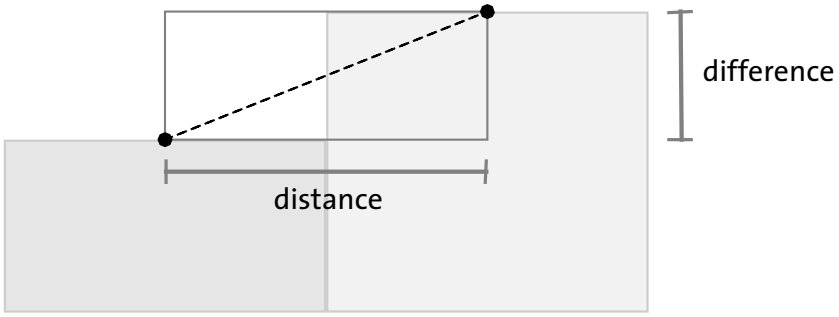


Fig. 13.21 – The rectangular region defined by the extent of the implicit edge between sample points

Given the calculation of the distance and difference and the position of the sample point on the left, Figure 13.22 (page 225) shows the construction of the corresponding rectangle rotated 90°.

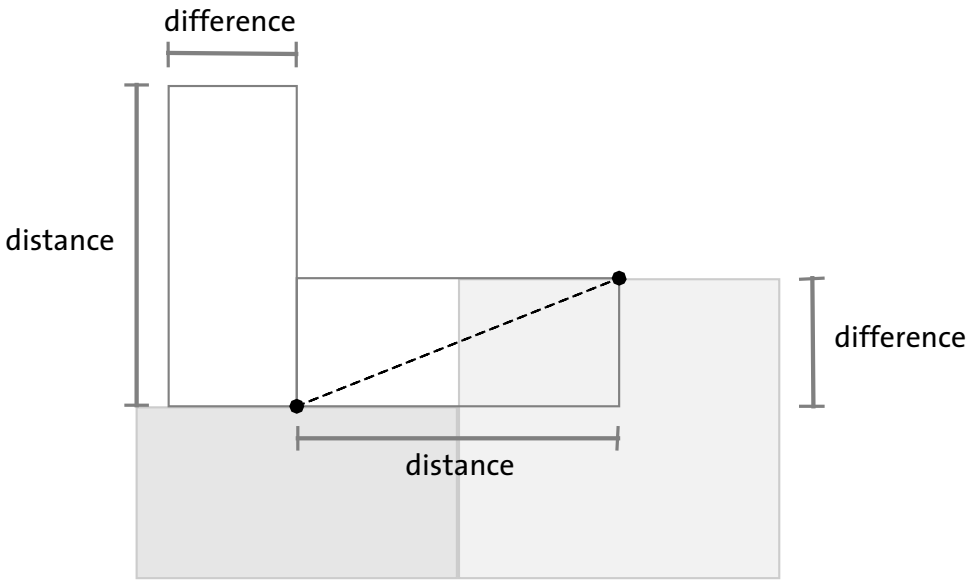


Fig. 13.22 – Constructing the rotated rectangle as suggested by Figure 13.20 (page 224)

The diagonal of the second rectangle in Figure 13.23 is the normal vector of the original segment.

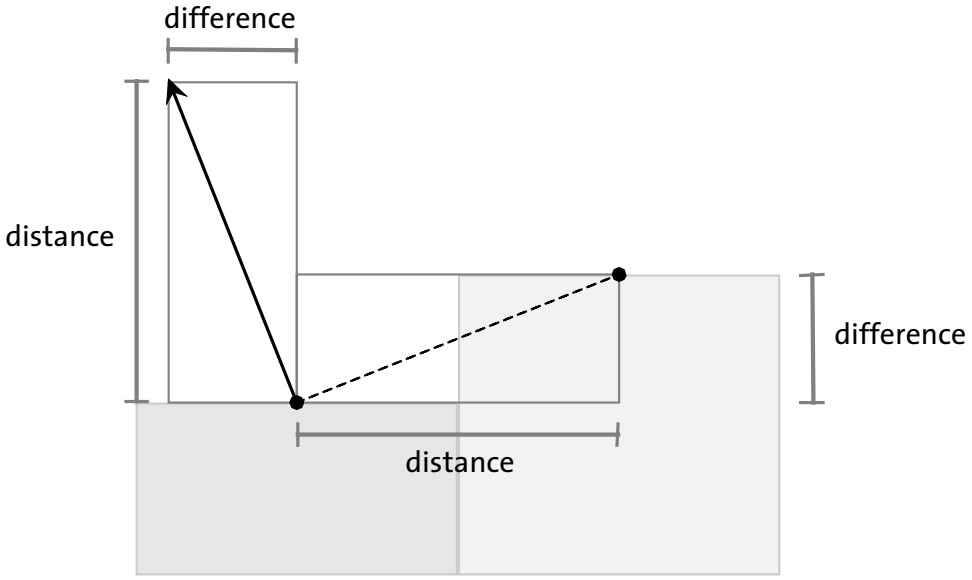


Fig. 13.23 – The vector that is normal to the implicit edge between samples

The surface normal and tangent vectors shown in Figure 13.24 (page 226) will provide the necessary framework to apply this construction method to the arbitrary orientation of points on a three-dimensional surface.

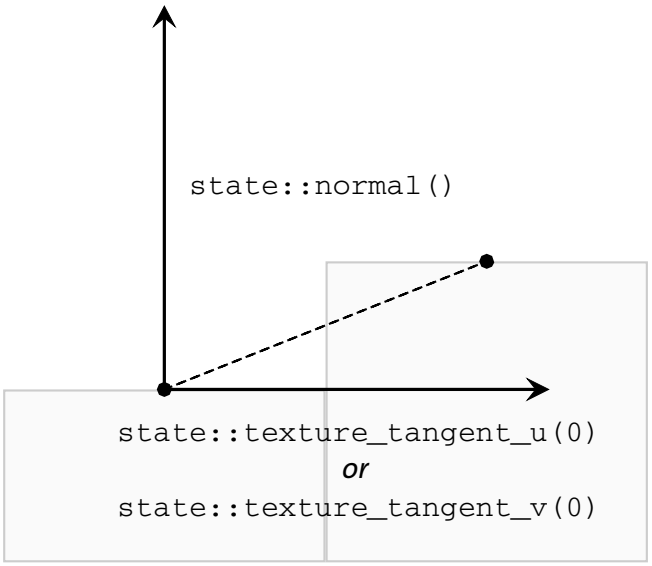


Fig. 13.24 – The state functions that provide the vectors for the construction of the new normal vector

13.5.4 Implementing the construction method

This section develops four functions for use in two materials that implement bump mapping based on an image. These functions follow the logic of the previous section and serve as an example of translating geometric intuitions into and MDL material.

In "Defining displacement distance with an image" (page 205), material `diffuse_weave` used the function `texture_2d_lookup_scaled`. This function returned the color value of the current point in texture space as defined by `state::texture_coordinate(0)`. However, creating the new normal vector for bump mapping also requires that texture values *near* the current point can be found. Acquiring values from a texture is also called *sampling* the texture. Function `texture_2d_sample` allows arbitrary u and v values to define the location in the texture to be sampled.

Listing 13.8

```
color texture_2d_sample(
  uniform texture_2d texture,
  float2 texture_scale = float2(1.0),   Scaling factor for texture repetition

  float2 uv = float(0.0))               Explicit uv coordinates as parameter
{
  float2 scaled_uv = math::frac(uv * texture_scale);
  return tex::lookup_color(texture, scaled_uv);
}
```

The next functions, `nearby_texture_differences` uses `texture_2d_sample` to acquire the values for the current (center) point and the points that are near it, as defined by the `distance` parameter.

Listing 13.9

```
float2 nearby_texture_differences(
  uniform texture_2d texture,
  float2 distance,
  float2 texture_scale)
{
  float3 uvw = state::texture_coordinate(0);

  float2 center_point = float2(uvw.x, uvw.y);
  float2 u_point = center_point + float2(distance[0], 0);
  float2 v_point = center_point + float2(0, distance[1]);

  float center_value = math::average(
    texture_2d_sample(texture, texture_scale, center_point));
  float u_value = math::average(
    texture_2d_sample(texture, texture_scale, u_point));
  float v_value = math::average(
    texture_2d_sample(texture, texture_scale, v_point));

  float u_difference = u_value - center_value;
  float v_difference = v_value - center_value;

  return float2(u_difference, v_difference);
}
```

Distance from current point in u and v directions

Current point in texture space

uv coordinates for current and nearby points

Lookup texture values at the three points

Find differences

Combine two differences as a value of type “float2”

With the distance between the sample points and the difference between the values of the samples, the normal vector and tangent vectors can be modified to produce the new normal vector at right angles to the segment. Figure 13.25 shows the relationship of the distance and difference to the normal and tangent vectors.

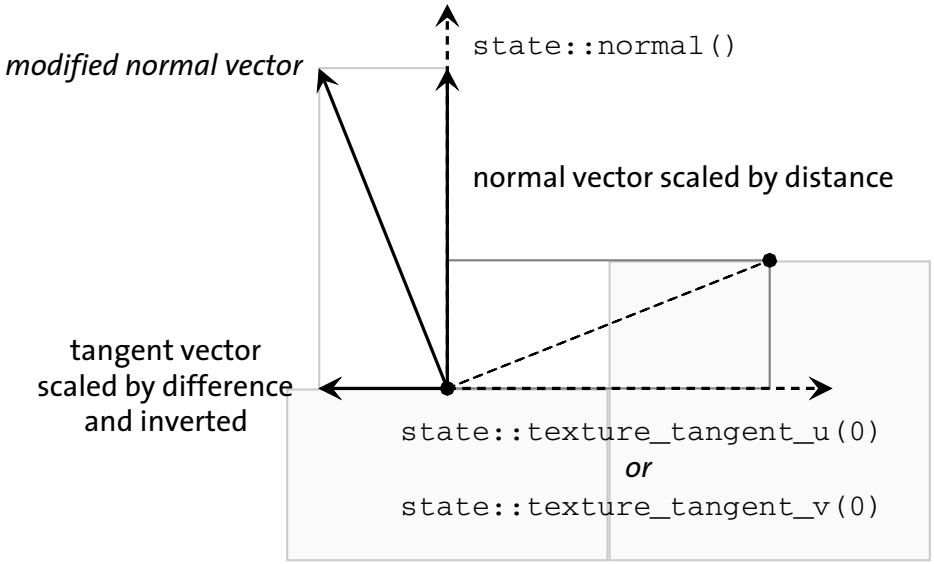


Fig. 13.25 – Use of distance and difference to scale the normal and tangent functions

The `bend_normal` function implements the vector scaling and addition shown in Figure 13.25 (page 227).

Listing 13.10

```
float3 bend_normal(
    float3 tangent,                    Tangent vector in u or v
    float distance,                    Distance between sample points
    float difference)                  Difference between the values of the sample points
{
    float3 scaled_normal =
      state::normal() * distance;      Scale vertical vector by horizontal factor
    float3 bend_direction =
      tangent * -difference;           Scale horizontal vector by negated vertical factor
    float3 result =
      math::normalize(
        scaled_normal + bend_direction);   Add the modified vectors and normalize
    return result;
}
```

With all the component functions in place, the `modify_normal_from_texture` function combines the normal bending from the u and v directions to define the new normal.

Listing 13.11

```
float3 modify_normal_from_texture(
    uniform texture_2d texture,
    float2 pixel_distance,
    float2 texture_scale)
{
    float2 dimensions =
      float2(tex::width(texture), tex::height(texture));   Width and height of texture
    float2 texture_distance =
      pixel_distance / (dimensions * texture_scale);       Convert from pixels to distance in texture space
    float2 difference =
      nearby_texture_differences(
        texture, texture_distance, texture_scale);         Find the difference of values in u and v directions
    float3 u_bend =
      bend_normal(
        state::texture_tangent_u(0),
        texture_distance[0], difference[0]);               Result of bending normal in u direction
```

```
    float3 v_bend =
      bend_normal(
        state::texture_tangent_v(0),
        texture_distance[1], difference[1]);
```

Result of bending normal in v direction

```
    float3 result =
      math::normalize(u_bend + v_bend);
```

Add and normalize the bent vector components

```
    return result;
}
```

The `texture_bump` material uses `modify_normal_from_texture` in the `material_geometry` property, parameterizing the material to be used for the surface property and the texture that defines the bump mapping.

Listing 13.12

```
material texture_bump (
    material surface_material,
```

Parameterize surface property

```
    uniform texture_2d texture,
```

Parameterize texture for bump mapping

```
    float2 pixel_distance = float2(1.0),
    float2 texture_scale = float2(1.0)) =
material(
    surface: surface_material.surface,
    geometry: material_geometry (
        normal: modify_normal_from_texture(
            texture, pixel_distance, texture_scale)));
```

Call function to calculate modified normal

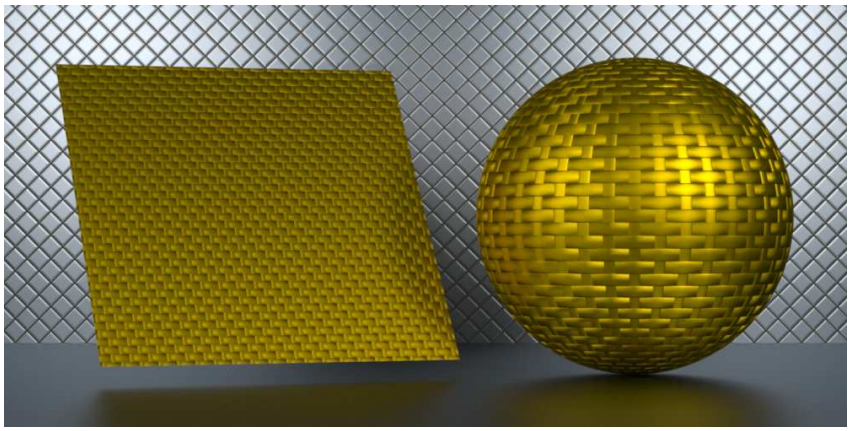
The `glossy_texture_weave_bump` material uses an instance of the `texture_bump` material that encapsulates the image from "Defining displacement distance with an image" (page 205) for the bump-mapping texture with the the `glossy` material for the surface property.

Listing 13.13

```
material glossy_texture_weave_bump(
    color tint = color(0.7),
    float roughness = 0.5,
    float2 pixel_distance = float2(1.0),
    float2 texture_scale = float2(1.0)) =
    texture_bump(
        glossy(tint, roughness),
        texture_2d("weave_depth.png"),
        pixel_distance,
        texture_scale);
```

Instance of material "texture_bump"

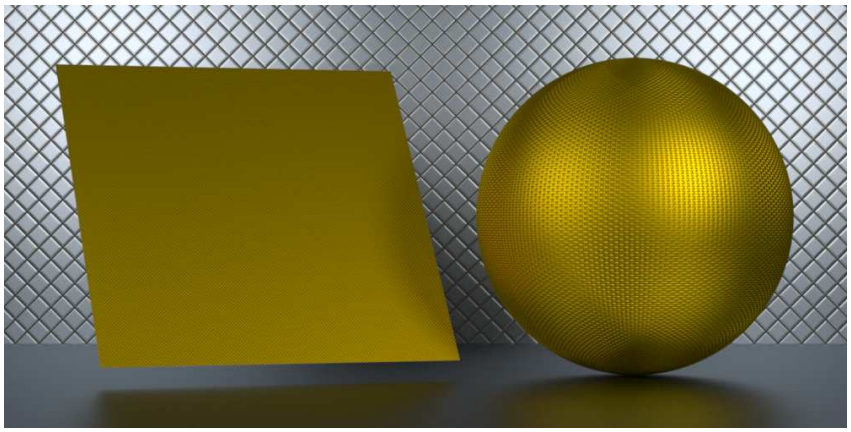
Rendering with `glossy_texture_weave_bump` produces Figure 13.26.



```
glossy_texture_weave_bump(
  tint:
    color(0.6, 0.4, 0.0),
  roughness: 0.3,
  texture_scale:
    float2(20, 10))
```

Figure 13.26

A larger repetition of the texture map produces Figure 13.27.



```
glossy_texture_weave_bump(
  tint:
    color(0.6, 0.4, 0.0),
  roughness: 0.3,
  texture_scale:
    float2(100, 50))
```

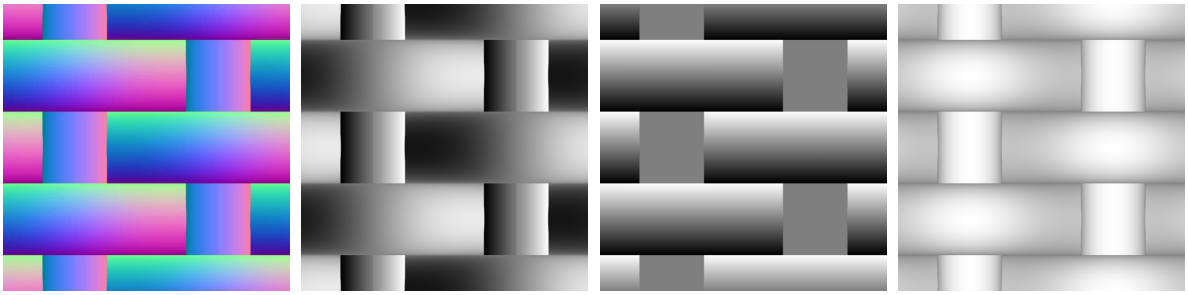
Figure 13.27

13.6 Using normal vectors encoded in an image

In the previous section, new normal vectors were derived from the surface implied by treating pixel intensities as geometric height. In a technique called *normal mapping*, the red, green, and blue components of image pixels encode the x, y, and z components of normal vectors directly. The required calculations are simple enough—interpret the color as a vector—that this section also explores extended ways of visualizing normal vectors and useful ways to modify the vectors encoded by the image.

13.6.1 The format of a normal map

No obviously ideal way to encode vectors in an image suggests itself; rendering applications vary in how a normal map image should be structured. A popular representation uses red for x, green for y, and blue for z. The range of the components of a normalized vector—from -1.0 to 1.0—is represented by values that vary from zero to the maximum value of the integer channel types (255 for 8-bit images, 65535 for 16-bit images). For images containing floating-point data, a value of 1.0 will typically be used as the maximum. This encoding allows the image be readily displayed with a variety of computer graphics applications, but requires that the channel values are scaled to -1.0 to 1.0 before the vector is constructed.



Normal map as an image *Red as x component* *Green as y component* *Blue as z component*

Fig. 13.28 – A widely used format for storing normal vector data in an image

The red and green components show the orientation of the vector component—increasing to the right for x and upward for y. The blue component represents z, the vector component pointing outward from the plane of the image. Geometrically, this component cannot be less than 0.0 or the surface would not be visible, lying beyond the "horizon" of what the virtual camera can "see." The blue component therefore has no value less that 0.5 (which will become 0.0 when converted), resulting in an image with no dark values, unlike the full intensity range of the red and green channels.

The three vector components of the texture space are all normalized—have a length of 1.0—so applying the normal map consists of scaling the three components by the color channels and adding these scaled components together to create the new normal vector.

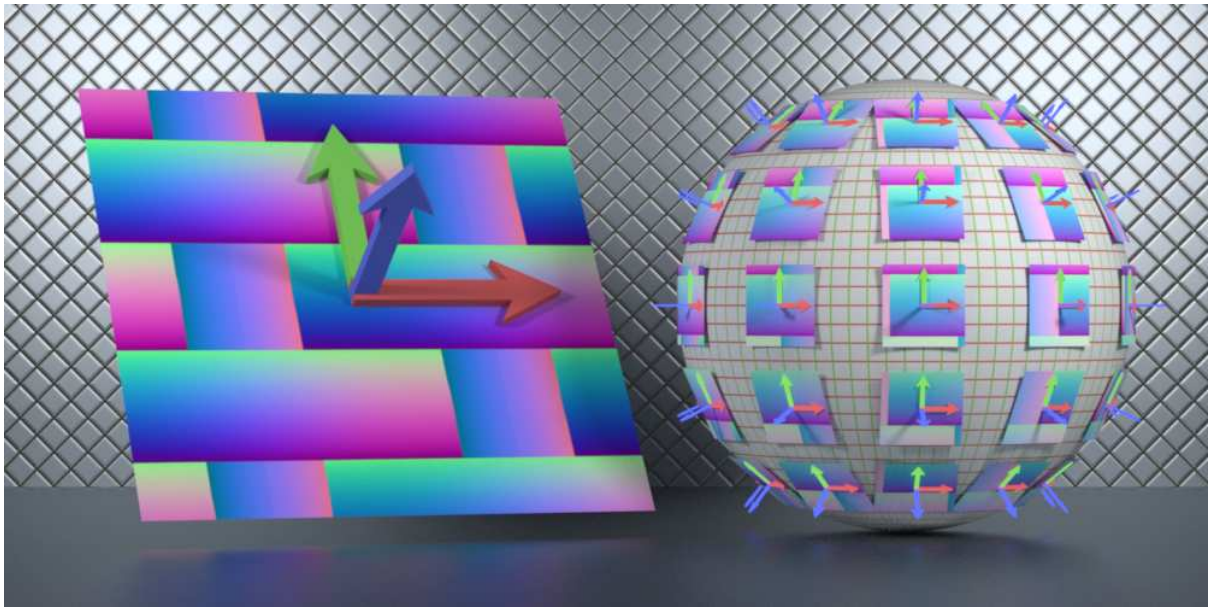


Fig. 13.29 – The samples from a normal map define new normal vectors

13.6.2 Visualizing the vectors of the normal map

The previous sections used the `math::average` function to acquire a single value from the colors of a texture image. A texture can also be read with other *lookup functions* based on the type of data to be sampled from the texture. To acquire the three components of the vectors defined by a normal map image, the `tex::lookup_float3` function will return values of type `float3`. If the renderer supports reading negative values from a texture and the normal map has stored the vector components directly (as an image of floating-point components in the

range of -1.0 to 1.0), then little further processing must be done beyond reading texture values and assigning them to the `normal` field of the `material_geometry` property.

In the case of the typical normal map format described above, the pixel components must be converted from the range of the data type's minimum and maximum values. However, the texture lookup functions transform, or *normalize*, the texture values to the range of 0.0 to 1.0 as defined by the data type.

For values in the range 0.0 to 1.0, the conversion to -1.0 to 1.0 is trivial. Given `c` as the value of an image channel, converting `c` to the -1.0 to 1.0 range is simply:

```
(2.0 * c) - 1.0
```

However, a more general solution would not make assumptions about the range of the original pixel data. The `fit` function (page 193), developed for displacement mapping, can also be used to parameterize the original range of the image as it is transformed to -1.0 to 1.0:

`fit(`*channel-value*`,` *channel-minimum*`,` *channel-maximum*`, -1.0, 1.0)`

However, the conversion of an RGB value to a vector is not an operation on single values, but on triplets—in terms of MDL's type system, a conversion from a `float3` to a `float3`. Another `fit` function can define the same transformation for `float3` values:

Listing 13.14

```
double fit(
   double v,                                  Original value

   double old_min, double old_max,            Range of original value

   double new_min, double new_max)            Range of new value
{
   double scale = (v - old_min) / (old_max - old_min);
   return new_min + scale * (new_max - new_min);
}
```

There are now two functions named "fit," but no ambiguity results when the `fit` function is called. The identify of a function is determined not simply by its name, but also by the types of the function's arguments and the type of the value the function returns.

Using the `float3` version of function fit, a new texture lookup function, `texture_2d_lookup_float3`, includes rescaling the sample values based on parameters of the function.

Listing 13.15

```
float3 texture_2d_lookup_float3(
   uniform texture_2d texture,
   float2 texture_scale = float2(1.0),
   float sample_min = 0.0,
   float sample_max = 1.0,                    Range of values in texture
```

```
    float new_min = 0.0,
    float new_max = 1.0)                    Rescaled range for result
{
    float3 uvw = state::texture_coordinate(0);
    float2 uv = float2(uvw.x, uvw.y) * texture_scale;   Lookup sample in texture
    float3 sample = tex::lookup_float3(texture, uv);

    float3 result = fit(
       sample,
       float3(sample_min), float3(sample_max),
       float3(new_min), float3(new_max));   Rescale the sample value, using "float3" constructors with the "float" parameters
    return result;
}
```

Function `bump_normal` includes the original range of values in the normal map as parameters `sample_min` and `sample_max`. Their default values implement the common normal map format of the 0.0 to 1.0 range. The red, green, and blue values scale the u tangent, the v tangent and the surface normal vector. However, this function also provides for optional scaling of the normal vector by the blue channel. (The visual effect this provides is shown in "Modifying normal map components" (page 238).)

Listing 13.16

```
float3 bump_normal(
   uniform texture_2d texture,
   float bump_scale = 1.0,
   bool scale_vector_z = true,          Allow control of scaling by the z component
   float2 texture_scale = float2(1.0),
   float sample_min = 0.0,
   float sample_max = 1.0) {
   float3 sample =
      texture_2d_lookup_float3(
         texture, texture_scale,        Lookup sample in texture
         sample_min, sample_max,
         -1.0, 1.0);

   float3 normal =
      state::normal() * (scale_vector_z ? sample.z : 1.0);   Surface normal vector, optionally scaled

   float3 tangent_u =
      state::texture_tangent_u(0) * sample.x * bump_scale;   Scaled u tangent vector

   float3 tangent_v =
      state::texture_tangent_v(0) * sample.y * bump_scale;   Scaled v tangent vector

   float3 result =
      math::normalize(
         normal + tangent_u + tangent_v);   Vector components combined and normalized
```

```
  return result;
}
```

Given the possible variations in the encoding of a normal map (and the need to debug an implementation), it can be very useful to visually verify its interpretation as a field of vectors. The utility function `color_from_normal` transforms the normal to world coordinates and scales the vector components to the 0.0 to 1.0 range. The function also provides for using its output as a value for light intensity, so that the visibility of the vector representation does not depend upon lighting in the scene. However, the light emission value is the *radiant exitance* of the surface, so it must be scaled by π, provided by the MDL constant `math::PI`.

Listing 13.17

```
color color_from_normal(
  float3 normal = float3(1.0),
  bool normal_color_as_emission = true)
{
  float3 object_normal = state::transform_normal(
    state::coordinate_internal,
    state::coordinate_world,
    normal);
```

Transform normal to world coordinate system

```
  color result = color(0.5 + 0.5 * object_normal);
```

Rescale normal components from [-1.0,1.0] to [0.0,1.0]

```
  if (normal_color_as_emission)
    result /= math::PI;
```

Convert color to radiant exitance

```
  return result;
}
```

The `color_from_normal` function provides the value of a temporary variable in the `normal_as_color` material. The Boolean parameter `normal_color_as_emission` defines the internal parameters of the `scattering` and `emission` values. If emission is used, there is not reflectance; if reflectance is used, there is no emission.

Listing 13.18

```
material normal_as_color(
  float3 normal = float3(1.0),
  uniform bool normal_color_as_emission = true) =
```

A switch to define the vector to be interpreted as reflectance or emission

```
let {
  color normal_color = color_from_normal(
    normal, normal_color_as_emission);
```

Normal converted to color

```
} in
material(
  surface: material_surface(
```

```
scattering: df::diffuse_reflection_bsdf(
   tint: normal_color_as_emission
      ? color(0.0)
      : normal_color),
```

If the material is emissive, there is no diffuse color

```
emission: material_emission (
   emission: df::diffuse_edf(),
   intensity: normal_color_as_emission
      ? normal_color
      : color(0.0))));
```

If the material is reflective, there is no emission

Finally, this seemingly endless march of nested utility functions is at an end. The next section will define materials and use them to make some pictures.

13.6.3 Using the normal map

The parameters of the `normal_map_bump` material control the behavior of its assembly of all the component functions developed in the previous section. This is a general-purpose material, in which the `surface` property is provided as an input of type `material`, the parameter `surface_material`.

Listing 13.19

```
material normal_map_bump(
   float bump_scale = 1.0,
   float2 texture_scale = float2(1.0),
   bool scale_vector_z = true,
   material surface_material = material(),
   uniform texture_2d texture = texture_2d(),
   float sample_min = 0.0,
   float sample_max = 1.0,
   uniform bool use_normal_for_color = false,
   uniform bool normal_color_is_emission = true) =
```

Control of the display of the normal vector as a color for testing purposes

```
let {
   float3 modified_normal = bump_normal(
      texture, bump_scale, scale_vector_z, texture_scale,
      sample_min, sample_max);
} in
material(
   surface:
      use_normal_for_color
         ? normal_as_color(
            modified_normal,
            normal_color_is_emission).surface
         : surface_material.surface,
   geometry:
      material_geometry (
```

New normal vector based on texture map

Choose between the “normal_as_color” function or “surface” property of the “surface_material” parameter

```
        normal:
            use_normal_for_color
              ? state::normal()
              : modified_normal));
```

Choose between the default or modified surface normal

By providing explicit values to some of its parameters, the following `normal_map_bump` material can be *partially instantiated* to define the more specialized material `glossy_normal_map_weave`. This specialization resembles the mechanism in functional programming whereby a new function is created from an existing one by providing explicit arguments to a subset of the parameters of the original function, a process known as *currying*.

Listing 13.20

```
material glossy_normal_map_weave(
  float bump_scale = 1.0,
  float2 texture_scale = float2(1.0),
  bool scale_vector_z = true,
  uniform bool use_normal_for_color = false,
  uniform bool normal_color_is_emission = true) =
```

Control of the display of the normal vector as a color for testing purposes

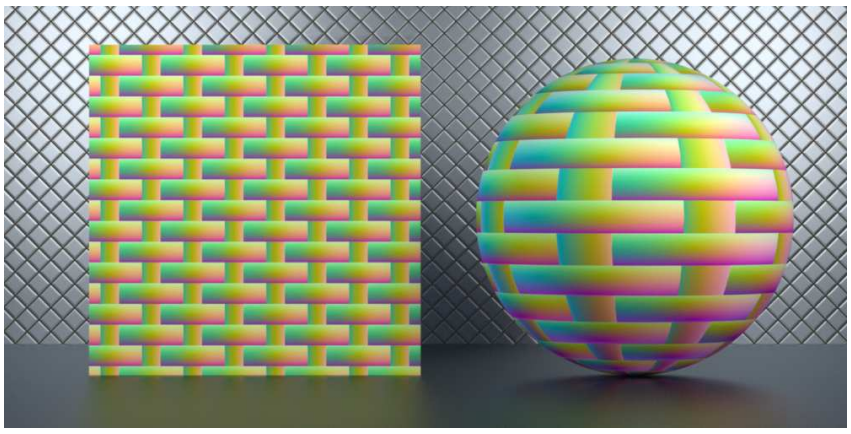
```
normal_map_bump(
```

The material body consists only of a call to the “normal_map_bump” material

```
  bump_scale: bump_scale,
  texture_scale: texture_scale,
  scale_vector_z: scale_vector_z,
  surface_material: glossy_yellow(),
  texture: texture_2d("weave_normal_map.tif"),
  sample_min: 0.0,
  sample_max: 1.0,
  use_normal_for_color: use_normal_for_color,
  normal_color_is_emission: normal_color_is_emission);
```

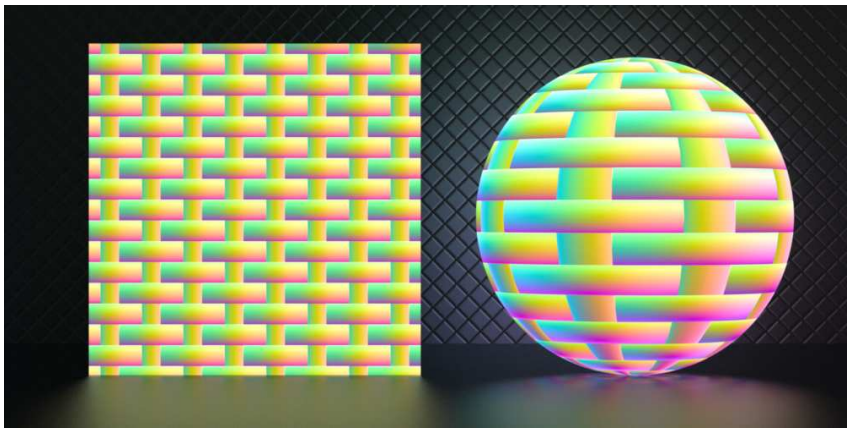
Explicit parameters that specialize “normal_map_bump”

Figure 13.30 and Figure 13.31 (page 237) show the use of the two modes of visually verifying the values of the normal map—as reflectance or emission.



```
glossy_normal_map_weave(
  use_normal_for_color: true,
  normal_color_is_emission: false,
  texture_scale:
    float2(4, 4))
```

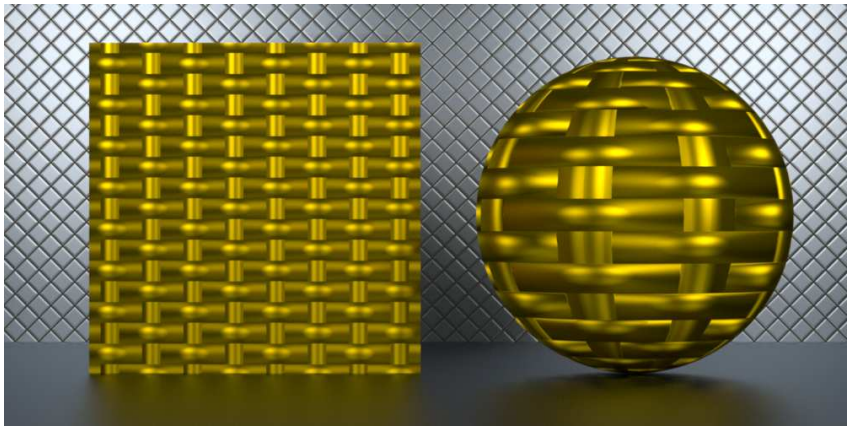
Figure 13.30



```
glossy_normal_map_weave(
  use_normal_for_color: true,
  normal_color_is_emission: true,
  texture_scale:
    float2(4, 4))
```

Figure 13.31

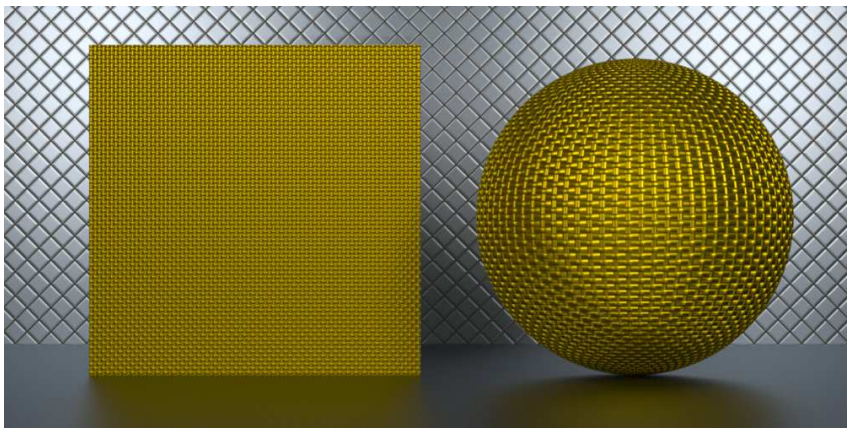
The explicit parameter values provided by `glossy_normal_map_weave` to `normal_map_weave` provide its specific color and glossy reflection effect.



```
glossy_normal_map_weave(
  texture_scale:
    float2(4, 4))
```

Figure 13.32

Figure 13.33 shows another example of the more typical use case of a bump map in which a finer level of detail will hide the geometric inconsistency of the profile.

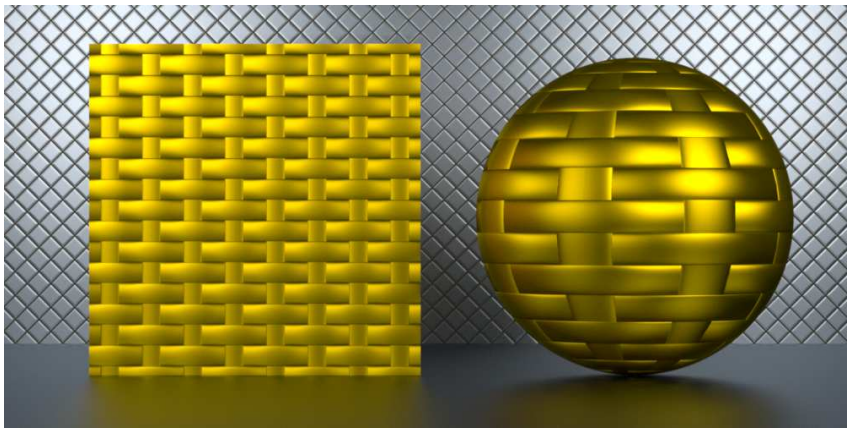


```
glossy_normal_map_weave(
  texture_scale:
    float2(40, 20))
```

Figure 13.33

13.6.4 Modifying normal map components

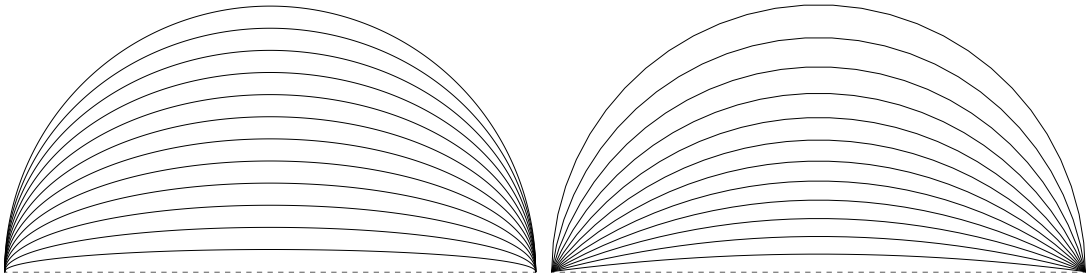
One of the parameters passed through the chain of component functions in the `glossy_normal_map_weave` provides an intuitive method of controlling how high above the surface its apparent geometric variation extends.



```
glossy_normal_map_weave(
  texture_scale:
    float2(4, 4),
  bump_scale: 0.2)
```

Figure 13.34

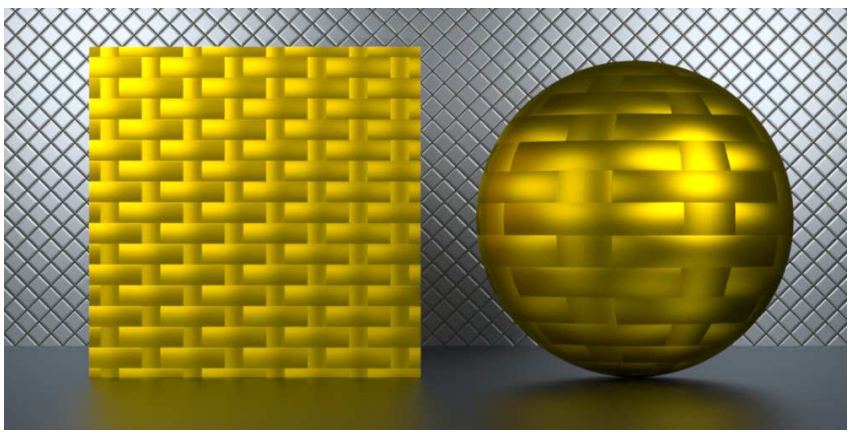
The Boolean parameter `scale_vector_z` controls whether or not the scale of the z vector component (blue in the normal map) is used in the calculation of the new normal. Figure 13.35 shows variation as the vector components are scaled, most noticeable at either ends of the arc.



Using all three components of the normal vector *Scaling only by x (red) and y (green) components*

Fig. 13.35 – The simulated geometry with and without using the z (blue) component of the normal map

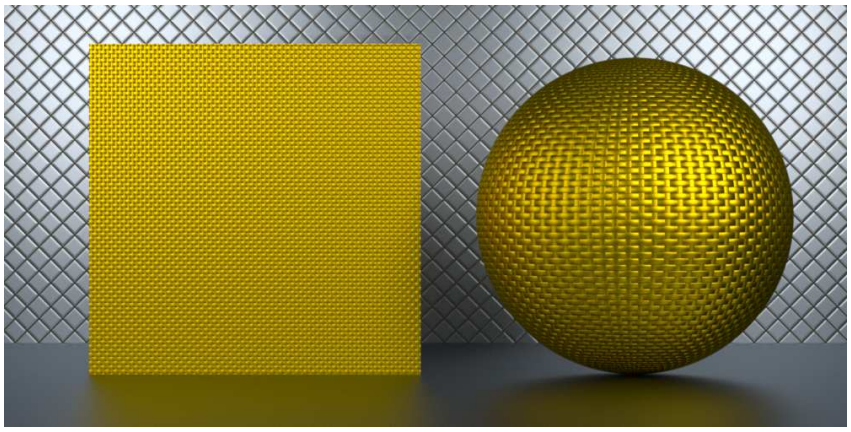
Removing the effect of the z component—providing a value of `false` for `scale_vector_z`—reduces the height at the edges of the rounded areas, implying a flatter surface than in Figure 13.35.



```
glossy_normal_map_weave(
  texture_scale:
    float2(4, 4),
  bump_scale: 0.25,
  scale_vector_z: false)
```

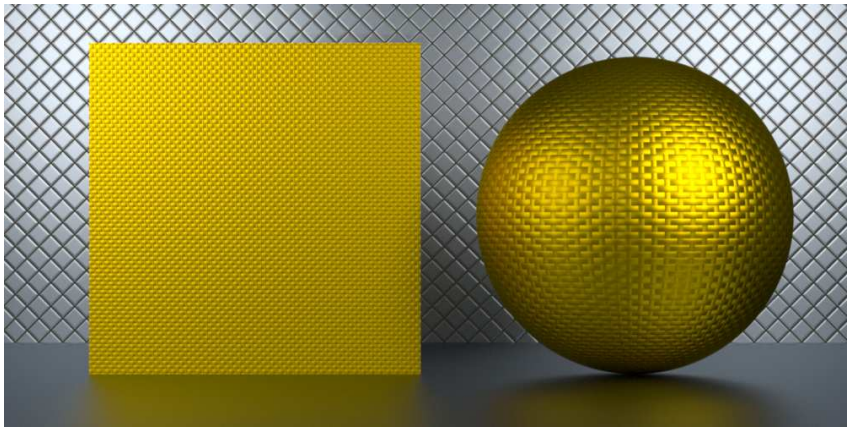
Figure 13.36

The inclusion of scaling by z varies in its visual effect based on scale. Figure 13.37 and Figure 13.38 vary only by the value of `scale_vector_z`.



```
glossy_normal_map_weave(
  texture_scale:
    float2(40, 20),
  bump_scale: 0.25,
  scale_vector_z: true)
```

Figure 13.37



```
glossy_normal_map_weave(
  texture_scale:
    float2(40, 20),
  bump_scale: 0.25,
  scale_vector_z: false)
```

Figure 13.38

As with the other visual effects produced by bump and normal mapping, but unlike the careful calculations of light inter-reflection, the parameterization of scaling by z has no actual correlation in the physical world. Techniques like this can be used in MDL to explore design possibilities; the later practical business of creating plans for the fabrication of the objects being designed will require a conversion to some geometric expression that is acceptable in the world of milling machines and 3D printing.

14 Geometric profiles

The description of the `material_geometry` struct (page 191) mentioned that early rendering systems only allowed displacement of the surface in the direction of the normal vector. In these systems, the length of the normal vector was either ignored or the vector was *normalized* to have a length of 1.0. An additional scalar value defined how far the surface should be displaced in the direction of the normal vector.

In MDL, the position of a displaced surface point takes both the direction of the modified surface normal as well as its length into account. Because displacements are not limited to movement along the surface normal, concave shapes can be developed as displacements of simple surfaces.

This chapter develops the idea of a *geometric profile*, a two-dimensional curve that defines surface displacement in a material. Profiles are a useful way to explore the flexible capabilities of MDL's displacement mechanism as well as other aspects of material development.

14.1 A framework for vector displacement

Before developing a material for displacement, it will first be useful to define a material to assist in testing and debugging the modification of a surface that displacements make possible.

14.1.1 Utility functions and their names

Previous chapters have developed functions that serve as components in the development of more complex functions and as field values in the properties of materials. The following functions simplify references to the u and v coordinates of a given texture space. Function `u` returns the first coordinate of the texture space:

Listing 14.1

```
float u(int texture_space=0) {
  return state::texture_coordinate(texture_space).x;
}
```

Function `v` only varies from `u` in the component of the texture coordinate it returns:

Listing 14.2

```
float v(int texture_space=0) {
  return state::texture_coordinate(texture_space).y;
}
```

Note the dangerously simple names of these two functions. It is easy to imagine another material for which such functions would be useful and for which the material author provided functions of the same name. This redundancy would result in an error when the new functions were defined.

So far, the materials and functions of the previous chapters have, for simplicity, ignored the MDL feature that organizes names of user-defined components. *Modules* and *packages* provide a hierarchical scheme for MDL's implementation of the traditional programming language notions of *namespaces* and the *scope* of variables. Modules and packages enable an author of one MDL source file to use a name without fear that the same name defines a different MDL component in another source file—or, at least, that such a collision occurs at the more readily corrected level of the module and packages names themselves.

14.1.2 A material for drawing grids

The strategy for the grid-drawing material developed by this section depends upon the distance from the current point in texture space to a grid line. The function `near_grid_value` determines the one-dimensional case—when a value falls within a certain distance from a multiple of the grid cell size. The distance determines the thickness of the lines in the grid.

Listing 14.3

```
bool near_grid_value(
    float f, float cell_count, float thickness, float edge=1)
{
   float thick = thickness == 0.0
     ? 0.05 / cell_count
     : thickness;
   thick *= edge;
   float nearest_value = math::floor((f + thick) * cell_count) / cell_count;
   return (f >= nearest_value - thick) && (f <= nearest_value + thick);
}
```

A thickness value of 0.0 triggers the calculation of a default based on cell_count

The combination of `near_grid_value` in both the *u* and *v* directions determines the location of grid lines.

Listing 14.4

```
bool in_grid_line(
   float2 cell_count, float2 thickness, int texture_space, float edge=1)
{
   return
      near_grid_value(u(texture_space),
                   cell_count[0], thickness[0], edge)
      || near_grid_value(v(texture_space),
                      cell_count[1], thickness[1], edge);
}
```

The function `grid_color` uses `in_grid_line` to determine if a point is part of the grid or the background, with the two colors provided as parameters to the function.

Listing 14.5

```
color grid_color(
  float2 cell_count = float2(4.0, 4.0),

  color line_color = color(0.0),
  //color line_color = color(u(), v(), 0.0),

  color bg_color = color(1.0),
  float2 thickness = float2(0.0),
  int texture_space = 0)
{
  /*
  color result = bg_color;
  if (in_grid_line(cell_count, thickness, texture_space))
    result = line_color;
  else if (in_grid_line(cell_count, thickness, texture_space, 1.5))
    result = color(0);
  return result;
  */
  return in_grid_line(cell_count, thickness, texture_space)
    ? line_color
    : bg_color;
}
```

The following material, `grid`, is simply a wrapper for the calculation of the grid, providing the resulting color as the `tint` field of `df::diffuse_reflection_bsdf`.

Listing 14.6

```
material grid(
  float2 cell_count = float2(4.0, 4.0),
  color line_color = color(0.0),
  //color line_color = color(u(), v(), 0.0),
  color bg_color = color(0.7),
  float2 thickness = float2(0.0),
  int texture_space = 0) =
let {
    color tint = grid_color(
      cell_count, line_color, bg_color, thickness,
      texture_space);
} in material(
  surface: material_surface(
    scattering:
        df::diffuse_reflection_bsdf(tint: tint)));
```

Calculate tint using the grid_color function

Simple diffuse reflection using the grid color

14.1.3 A template material for vector displacement

The `grid` material demonstrates a frequent strategy in recent chapters. A function is developed (for material `grid`, function `grid_color`) that defines all the calculations required for a field in a material's property or distribution function.

Similarly, a template material for testing the effects of displacement can make the calculation of the displacement vector external to the material, in other words, provided as the value of a function. The surface property is extracted from the material instance that is an argument to the vector displacement material.

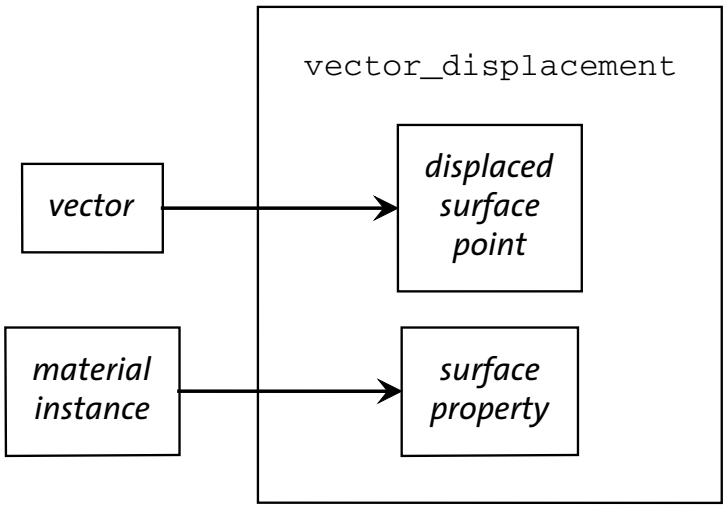


Fig. 14.1 – The `vector_displacement` *material provides arguments that define a modified surface point and a surface property*

In material `vector_displacement`, the default value for the displaced surface point is `float(0.0)` – that is, no displacement. As a default, the `grid` profile serves as a good default for testing the way in which the displacement alters the texture space of the surface.

Listing 14.7

```
material vector_displacement(
    float3 displacement = float3(0.0),      Displacement vector for the surface point

    material material_for_surface = grid()) =   Material providing the surface property
material(
  thin_walled: true,
    surface:                                Extract surface property from the material passed as
      material_for_surface.surface,         an argument

    geometry:                               Displace surface based on the vector passed as
      material_geometry(displacement));     an argument
```

As an example of the use of the the `vector_displacement` material, imagine that each surface point is moved by the same amount in the direction of the surface normal vector.

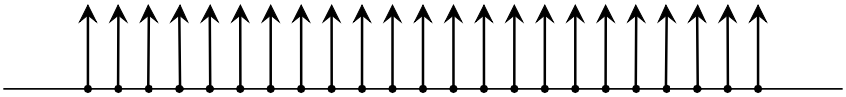


Fig. 14.2 – All surface normal vectors are scaled by the same amount

Function `move_up` is the simplest possible modification of the surface normal vector—it only scales the length of `state::normal()`.

Listing 14.8

```
float3 move_up(float scale) {
  return scale * state::normal();
}
```

Material `raise` provides arguments to an instance of the `vector_displacement` material. Function `move_up` provides the new normal vector; the surface property of the `grid` material provides `raise` with its surface property.

Listing 14.9

```
material raise(
  float scale = 0.0,
  float2 grid_count = float2(4.0)) =
  vector_displacement(
    move_up(scale),
    grid(grid_count));
```

Instance of the vector_displacement material

Figure 14.3 and Figure 14.4 use the `grid` material for the square on the left and the `raise` material for the square on the right. In Figure 14.3, the `raise` material is used with a `scale` value of 0.0, resulting in no displacement of the square. In Figure 14.4, the `raise` material uses a `scale` value of 0.2.

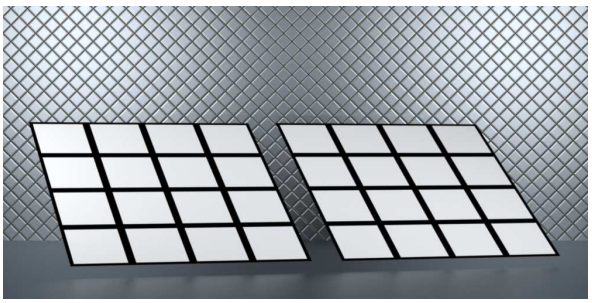


Fig. 14.3 – Material `raise` *with a* `scale` *value of 0.0*

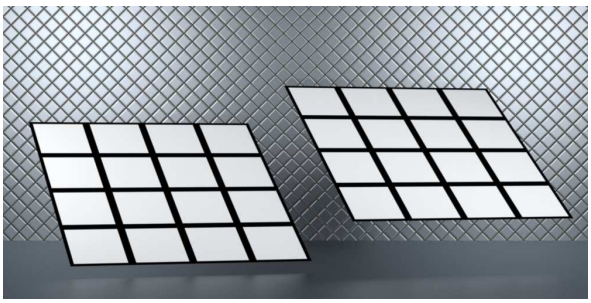


Fig. 14.4 – Material `raise` *with a* `scale` *value of 0.2*

Figure 14.3 and Figure 14.4 serve as a model for the rendered images in this chapter. In each rendered image, the square on the left is rendered with the `grid` material and shows the texture map scaling used for both objects. The square on the right is rendered with a material based on the `vector_displacement` material.

14.2 Maintaining a linear texture space

As a surface point is transformed by a displacement vector, the texture space—the u and v coordinates—is maintained. For certain transformations, the resulting texture space will only be stretched or shrunk.

For example, an isosceles triangle (a triangle with two sides of equal length) can be defined by scaling the surface normal vector.

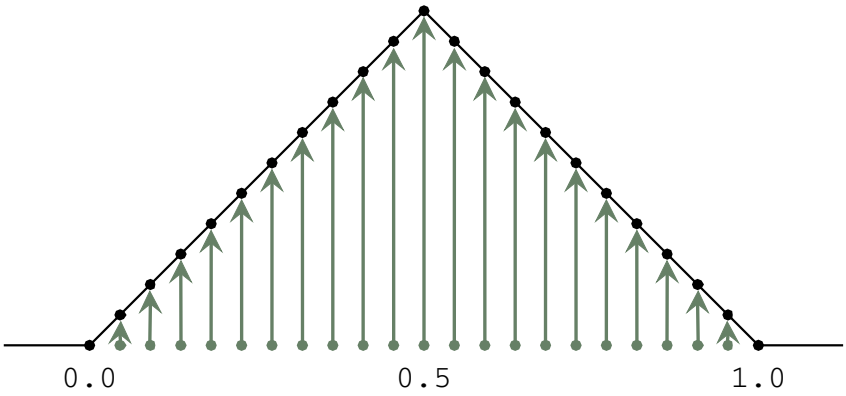


Fig. 14.5 – Scaling the surface normal vector to form a triangular surface

Assuming the triangle fits within a range of 0.0 to 1.0, the `triangle_displacement` function defines the height of the triangle for a given point between 0.0 and 1.0 as a function of the triangle's height.

Listing 14.10

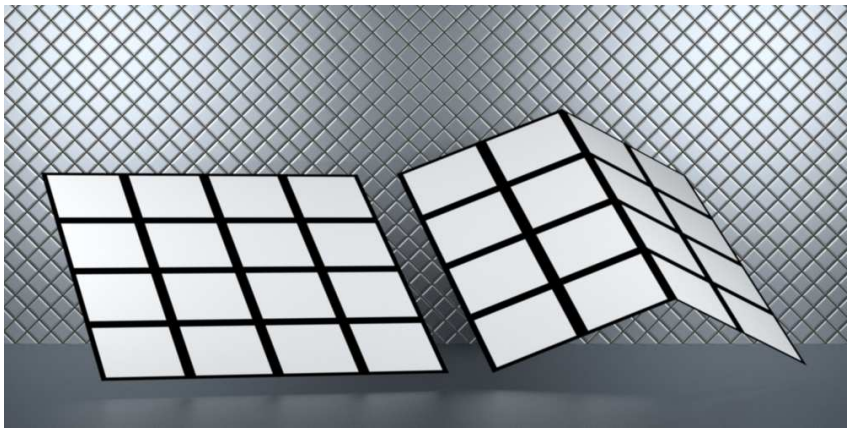
```
float3 triangle_displacement(float f, float height) {
  float length =
    (f <= 0.5)
    ? fit(f, 0.0, 0.5, 0.0, height)
    : fit(f, 0.5, 1.0, height, 0.0);
  return length * state::normal();
}
```

The `triangle_displacement` function defines the value of the `displaced_normal` argument for the `triangle` material.

Listing 14.11

```
material triangle(float height=0.5) =
  vector_displacement(
    triangle_displacement(u(), height));
```

In Figure 14.6 (page 247), the square on the left is rendered with the `grid` material. The square on the right is rendered with the `triangle` material using the `grid` material (with the same parameters) for the surface property and the `triangle_displacement` function to define the displacement. (In the renderings in this chapter, the material listed to the right of the image is used by the square on the right; the `grid` material is used by the square on the left.)



```
triangle(
  height: 0.25)
```

Fig. 14.6 – A linear relationship between the default and displaced texture space of a square

However, a shape like an isosceles triangle is a special case. Though the texture space is stretched in the triangular object, the stretching is proportional (or *linear*) throughout—the width of a grid cell is the same in the displaced surface. In other simple shapes, this proportional relationship will not be produced by a simple scaling of the surface normal vector.

For example, a circular profile can be defined in a range of 0.0 to 1.0 by scaling by the *y* value determined from the Pythagorean theorem.

Listing 14.12

```
float3 circle_linear_vector(float f, float radius = 0.5) {
    float x = fit(f, 0.0, 1.0, -0.5, 0.5);   x between -0.5 and 0.5 for the unit circle

    float y_2 = radius * radius - x * x;   y^2 from the Pythagorean theorem

    float y = (y_2 >= 0.0) ? math::sqrt(y_2) : 0.0;   y clamped to a positive value

    return y * state::normal();   y scales the surface normal vector
}
```

Figure 14.7 shows how the default value of 0.5 for the `radius` argument maps points on the segment to points on the circle:

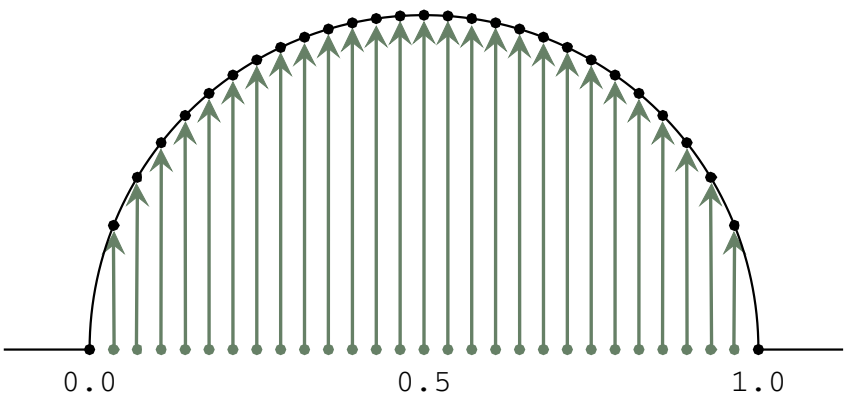


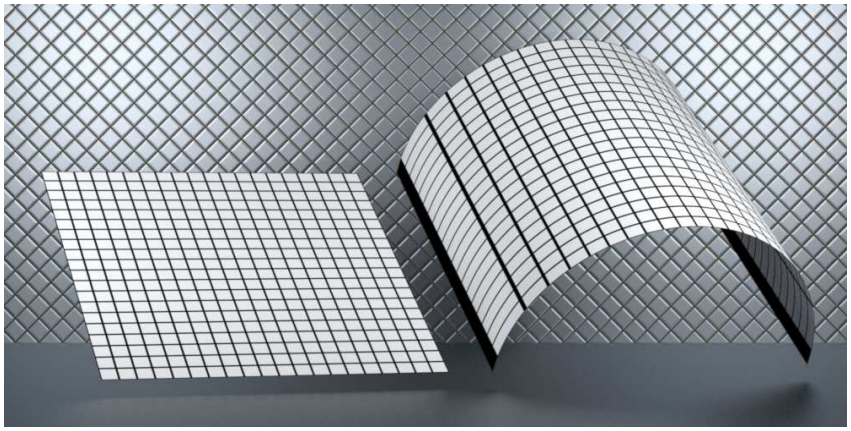
Fig. 14.7 – Scaling the surface normal vector to form a semicircular shape

In material `circle_linear`, the `circle_linear_vector` function defines the scaling of the surface normal vector. A 20 by 20 `grid` material defines the surface property.

Listing 14.13

```
material circle_linear(float height=0.5)
= vector_displacement(
   circle_linear_vector(u(), height),
   grid(float2(20, 20)));
```

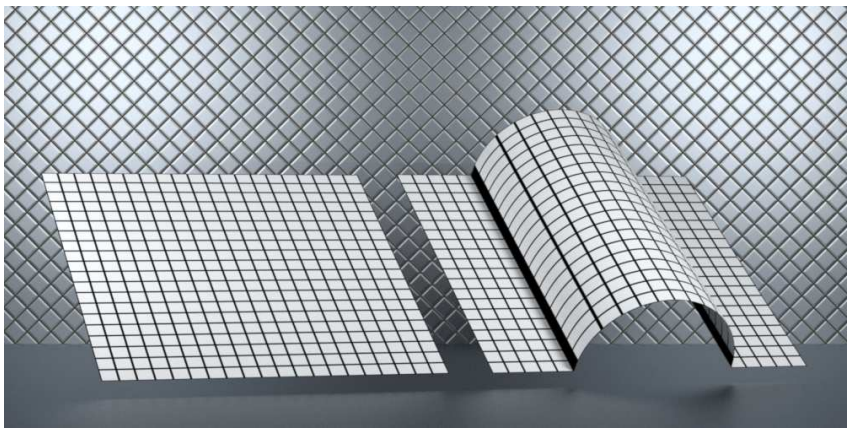
Rendering with material `circle_linear` produces the shape on the right in Figure 14.8. The grid is stretched unevenly across the surface.



`circle_linear()`

Fig. 14.8 – A distorted texture space produced by scaling the normal vector

A `radius` value of 0.25 for material `circle_linear` produces Figure 14.9.



```
circle_linear(
  height: 0.25)
```

Fig. 14.9 – Inconsistent mapping between linear and non-linear texture spaces

In order to create an undistorted mapping of the texture space, a different relationship between distances in texture space and distance along the curve must be defined. New surface points must be produced by both scaling and rotating the surface normal vector. For a semicircle, distances in the 0.0 to 1.0 range of the texture space must be mapped to proportional distances along the semicircular curve.

Listing 14.14

```
float3 circle_nonlinear_vector(float f, float height) {
   float x1 = fit(f, 0, 1, -1, 1);
```
Position in -1.0 → 1.0 range for f

```
   float angle = fit(f, 0, 1, math::PI, 0);
```
Proportional angle on a semicircle

```
   float x2 = math::cos(angle);
   float y2 = math::sin(angle);
```
Determine the coordinates of the point on the semicircle

```
   float3 displacement_vector =
      y2 * state::normal()
      + (x2 - x1) * state::texture_tangent_u(0);
```
Map the vector to the plane of the normal and the tangent in u

```
   return height * displacement_vector;
```
Scale the resulting vector by the height

```
}
```

The mapping of points by the `circle_nonlinear_vector` function is demonstrated in Figure 14.10:

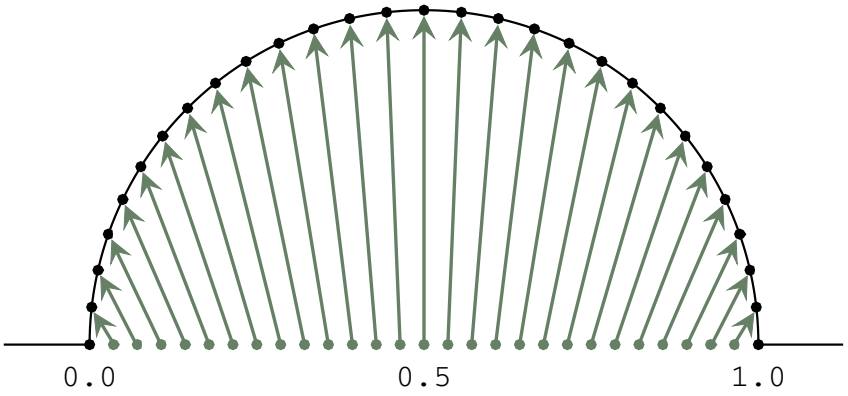


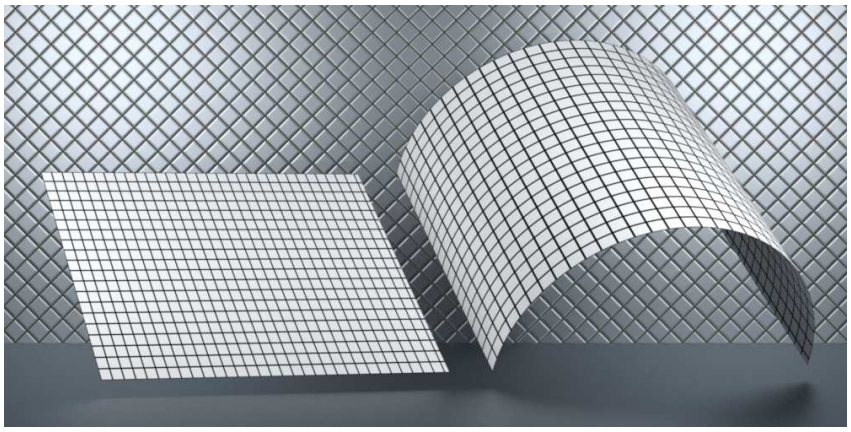
Fig. 14.10 – Proportional mapping of points on a line to a semicircle

The `circle_nonlinear` material uses the `circle_nonlinear_vector` to define the displacement vector.

Listing 14.15

```
material circle_nonlinear(
   float height = 0.5,
   float2 count = float2(20, 20))
= vector_displacement(
   circle_nonlinear_vector(u(), height),
   grid(count));
```

Figure 14.11 uses material `circle_nonlinear` for the object on the right. Note that the scaling of the grid in u is $\pi/2$ times the scaling in v, resulting in grid cells of equal width and height (as measured along the circle with a radius of 0.5).



```
circle_nonlinear(
  height: 0.5,
  count:
    float2(31.416, 20))
```

Fig. 14.11 – Adjusting the uv mapping for a circular shape

Careful calculations of the mapping of points from a geometric primitive to a new shape produced by displacement can maintain the linearity of the texture space. Determining this relationship is similar to the process of *parameterizing* a curve, so that a value from 0.0 to 1.0—the curve's *parametric value*—defines proportional points along the curve. The previous materials of this chapter are all dependent on a particular geometry and texture space to enable that parameterization. However, the same process underlies a variety of commercial applications, in which the apparent clay-like modeling in the user interface is simply the calculation of a set of displacement vectors that modify a simpler surface.

14.3 Concavities and overhangs

The last section showed how limiting the modification of the surface normal vector to scaling alone could distort the texture space. Simple scaling also restricts the types of geometric structures that displacements can create. For example, the normal vector intersects the cross section of the surface in Figure 14.12 twice.

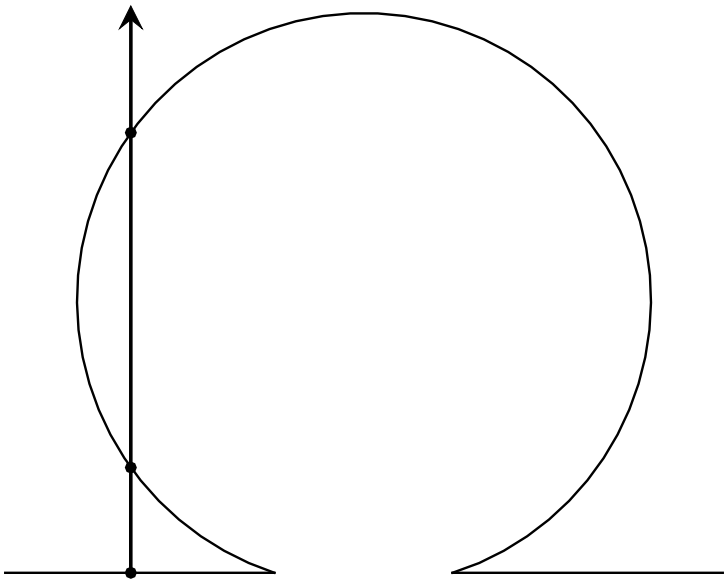


Fig. 14.12 – A scaled normal vector can intersect some shapes more than once

Happily, developing a parameterization for a two-dimensional curve can also create surfaces that cannot be defined by only scaling the normal vector.

14.3.1 The bubble profile

To demonstrate the combined problem of texture distortion and the representation of overhanging parts, this section presents a curve nicknamed a *bubble profile*, defined by a circle intersected by a line segment. Given a segment length of 1.0, a relationship between the circular arc and the segment can be defined by specifying the radius of the circle and the circle's "height," the distance of the circle's center to the segment:

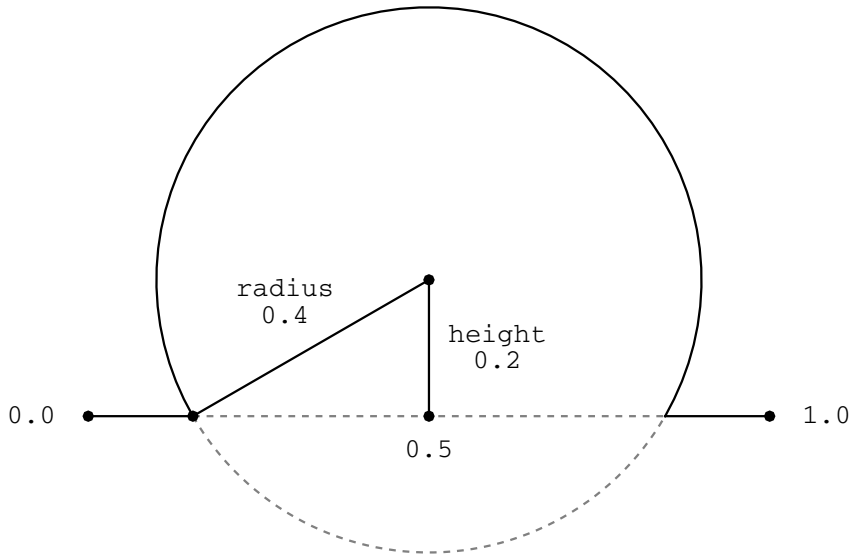


Fig. 14.13 – A bubble profile is defined by the circle's radius and the height of the circle's center

A radius of 0.5 and a height of 0.0 produces a semicircle:

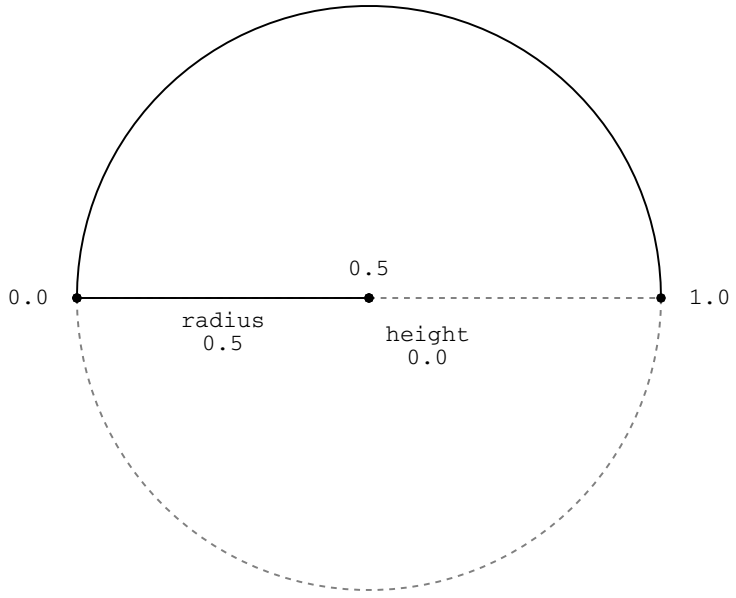


Fig. 14.14 – A bubble radius of 0.5 and a height of 0.0 creates a semicircle

Negative values for the height parameter produce smaller arcs. Varying the height and radius from 0.0 to 0.5 produces a family of circular arcs and segments at the arc's endpoints.

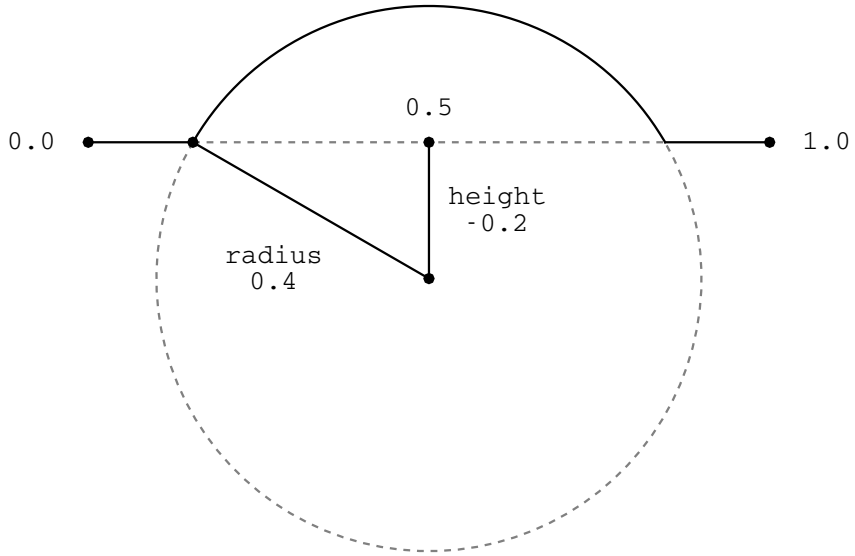


Fig. 14.15 – A negative bubble height produces a circular arc smaller than a semicircle

A material can create a bubble shape with displacement vectors by defining a proportional relationship between points on a planar polygon and points on the bubble:

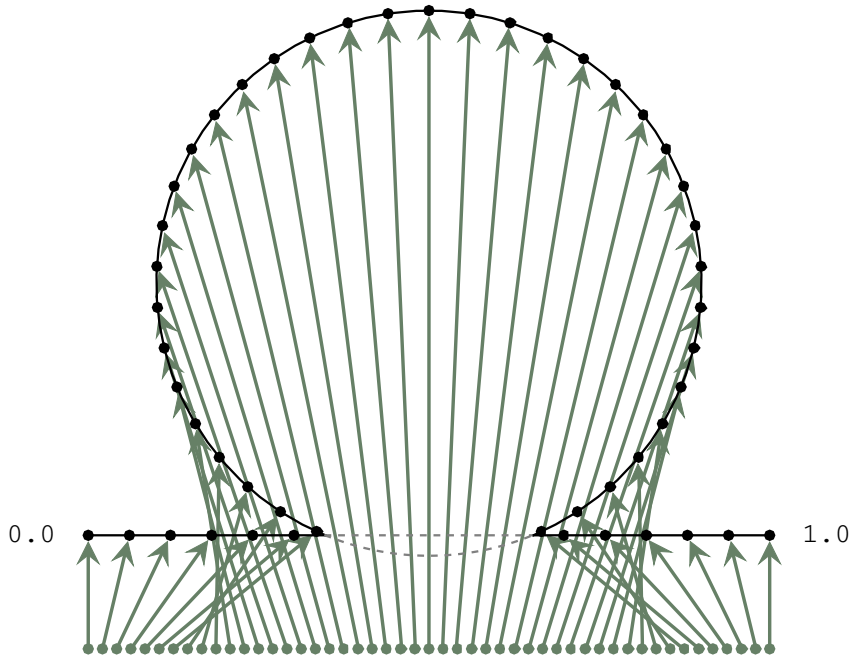


Fig. 14.16 – Mapping points in a line segment to evenly spaced points in the bubble

In the material, some points will remain on the polygon while others will form the circular fraction:

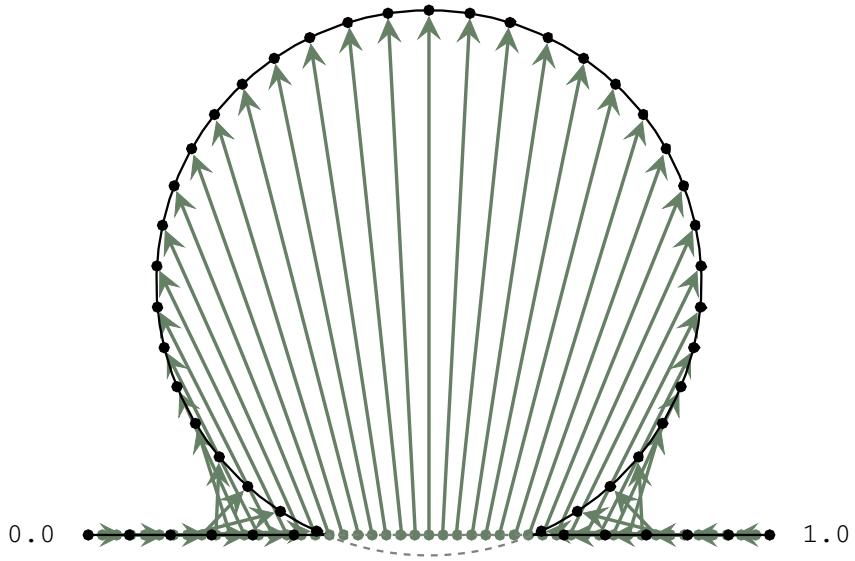


Fig. 14.17 – The transformation of surface normal vectors to displacement vectors that form a bubble

The following section implements this idea of a bubble profile in the same spirit as the previous materials in this chapter, reducing the problem to the construction of a displacement vector based on the bubble's radius and height parameters.

14.3.2 Bubble calculation

The radius and height parameters define the angles and lengths that are necessary to map a value from 0.0 to 1.0 to a point along the bubble shape. Two geometric assumptions simplify the problem: the length of the base line is 1.0, and the center of the circle is located above the midpoint of the base, perpendicular to it. Figure 14.18 shows the definitions of `base_angle` and `half_angle` based on these assumptions.

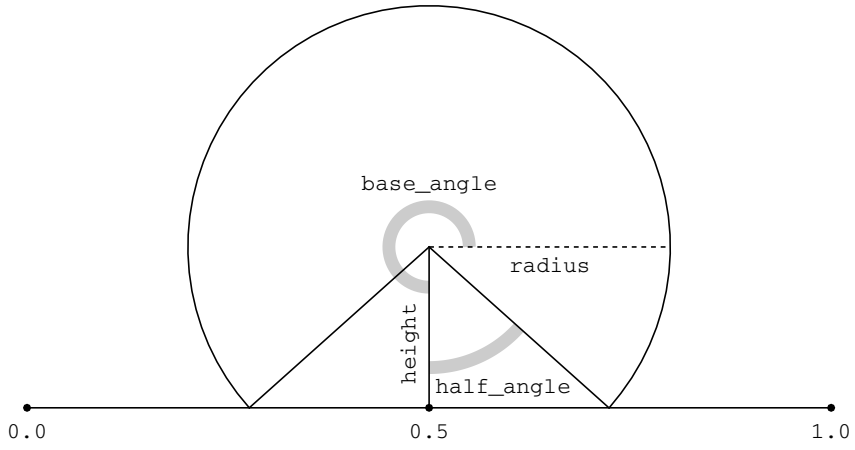


Fig. 14.18 – Initial calculations: `base_angle` $= \frac{3\pi}{2}$, `half_angle` $= \arccos \frac{height}{radius}$

From these two angles the starting and ending angles of the circular arc can be defined, as well as the length of the portion of the base that cuts the circle (a *chord*) and the arc's circumference.

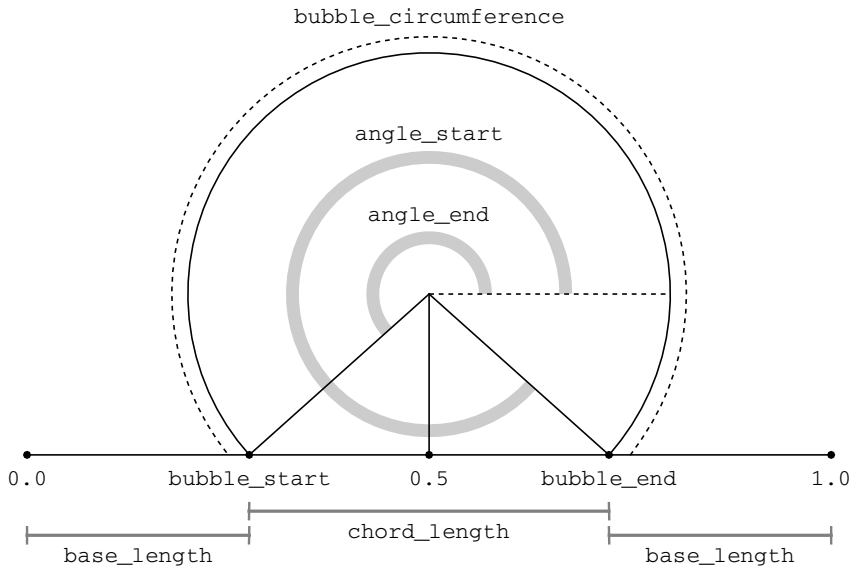


Fig. 14.19 – Intermediate and final angles and lengths for mapping calculations

For a given pair of radius and height arguments, the angles and lengths necessary to map a value from 0.0 to 1.0 to a point on the bubble profile only need to be calculated once. To clarify the difference between those constant arguments and the varying points mapped on the profile, it is convenient to group the related calculations in a function. However, such a function needs to return multiple values. A user-defined struct provides such a mechanism. The custom struct `bubble_args` defines seven fields that are used in the bubble displacement and the material:

Listing 14.16

```
struct bubble_args {
  float radius;
  float height;
  float angle_start;
  float angle_end;
  float bubble_start;
  float bubble_end;
  float total_length;
};
```

The seven arguments are used to calculate the varying displacement of a surface. However, the radius and height arguments of the bubble calculation are constant during the entire rendering process. This implies that the five derived values will also be constant and only need to be calculated once by a rendering system. Providing this information to a rendering system can enable it to avoid redundant calculations. In MDL, this signal is performed by the keyword `uniform` placed before the data type of a function or variable.

For example, the function `get_bubble_args` calculates the values for the seven fields of the `bubble_args` struct. The `uniform` keyword defines a *contract* with the rendering system that the input arguments and the value returned by the function will not change at any point during the rendering process.

Listing 14.17

```
uniform bubble_args get_bubble_args(                Return uniform instance of the bubble_args struct
    uniform float radius, uniform float height)     Input parameters are uniform
{
    float base_angle =
      math::PI * 3 / 2;
    float half_angle =
      math::acos(height / radius);
    float angle_start =
      base_angle + half_angle;
    float angle_end =
      base_angle - half_angle + 2 * math::PI;
                                                    Angle calculations
    float bubble_circumference =
      (2 * math::PI - 2 * half_angle) * radius;
    float chord_length =
      2 * radius * math::sin(half_angle);
    float total_length =
      bubble_circumference + (1 - chord_length);
                                                    Length calculations
    float circumference_fraction =
      bubble_circumference / total_length;
    float base_length =
      (1.0 - chord_length) / 2;
    float bubble_start =
      base_length / total_length;
    float bubble_end =
      bubble_start + circumference_fraction;
                                                    Fractional length calculations
    return bubble_args (
      radius,
      height,
      angle_start,
      angle_end,
      bubble_start,
      bubble_end,
      total_length);                                Construct and return the bubble_args struct
}
```

The following function, `bubble_point`, calculates the two-dimensional coordinates of the bubble profile corresponding to the `fraction` arguments value in the range 0.0 to 1.0. The `bubble_args` struct, declared as `uniform`, is passed as an argument to `bubble_point`.

Three components of the profile are calculated separately: the base to the left of the bubble, the bubble's arc, and the base to the right of the bubble. After the point is determined, a repetition that is not equal to 1.0 scales the x and y components of the point. An offset value is added to the x component after the scaling.

Listing 14.18

```
float2 bubble_point(
  float fraction, float repeat,
  uniform bubble_args ba)                    Custom struct as a function argument
{
  float y = 0, x = 0;                        Components of the float2 result

  float f = math::frac(fraction * repeat);   Truncate fraction within the 0.0 → 1.0 range

  if (f <= ba.bubble_start) {
    x = f * ba.total_length;                 Base at the left side of the bubble

  } else if (f <= ba.bubble_end) {
    float pt_angle =
      fit(f,
          ba.bubble_start, ba.bubble_end,
          ba.angle_end, ba.angle_start);
    x = math::cos(pt_angle) * ba.radius + 0.5;
    y = math::sin(pt_angle) * ba.radius + ba.height;
                                             Bubble

  } else {
    x =  1.0 - (1.0 - f) * ba.total_length;  Base at the right side of the bubble
  }

  if (repeat != 1) {
    float offset =
      math::floor(fraction * repeat) / repeat;
    x = x / repeat + offset;
    y /= repeat;
  }                                          Modify for the repetition of the bubble
  return float2(x, y);
}
```

The two-dimensional point returned by `bubble_point` determines a new point in the plane defined by the surface normal and a tangent vector.

Listing 14.19

```
float3 modify_normal(
  float f, float2 target, float3 tangent)
{
  float3 result =
      (tangent * (target[0] - f)) +          The profile's x components scale the tangent vector
      (state::normal() * target[1]);         The profile's y components scale the surface normal vector
  return result;
}
```


Putting the pieces together, function `displace_with_bubble` returns a vector based on the tangent vector selected by a non-zero value of the `repeat` parameter.

Listing 14.20

```
float3 displace_with_bubble(
  float2 repeat, uniform bubble_args ba)
{
  float3 uvw = math::frac(state::texture_coordinate(0));
  float3 result = float3(0.0);
```

The default value is a null vector (no change)

```
  if (repeat[0] > 0)
    result = modify_normal(
      uvw[0],
      bubble_point(uvw[0], repeat[0], ba),
      state::texture_tangent_u(0));
  //6 Displace in the v direction
  else if (repeat[1] > 0)
    result = modify_normal(
      uvw[1],
      bubble_point(uvw[1], repeat[1], ba),
      state::texture_tangent_v(0));
  return result;
}
```

Displace in the u direction

The vector constructed by `displace_with_bubble` can now be used for displacement in the material developed in the next section.

14.3.3 Bubble material

The original proportions of the texture will be distorted by the lengthening of the surface created by its displacement. The separate calculation of the arguments for bubble displacement provides a means to restore the correct proportions as the texture is applied to the surface.

The values of the `bubble_args` struct is defined as a variable in the material's `let` clause. The `total_length` field of the struct is used to compensate for the lengthening of the geometry due to displacement. As in the previous materials, `vector_displacement` takes a displacement vector and a surface color, defined by the `grid` function. The `grid_point` value, calculated in the `let` clause, uses the modified length of the surface if `normalize_grid` is `true`.

Listing 14.21

```
material bubble(
  uniform float radius = 0.25,
  uniform float height = 0.25,
  float2 repeat = float2(1.0),
  float2 grid_count = float2(4.0),
  bool normalize_grid = false) =
let {
```

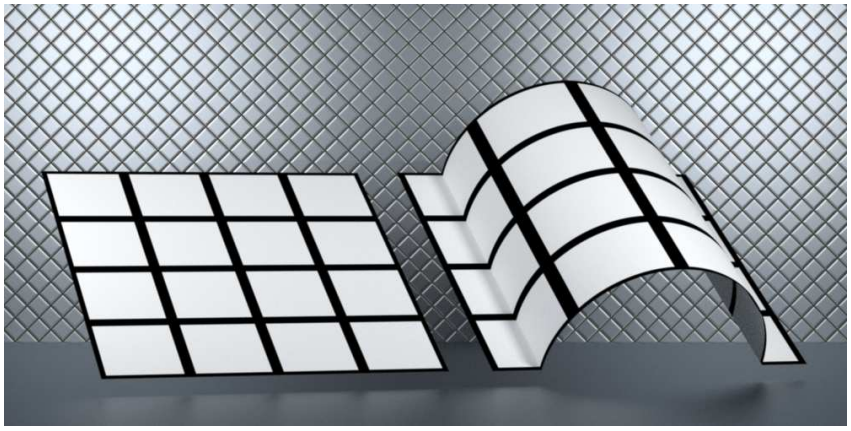
Adjust u and v texture scaling to match

```
    uniform bubble_args ba =
       get_bubble_args(radius, height);
```
Parameters for both the surface and the displacement

```
    float2 grid_point(
        grid_count[0] *
           (normalize_grid && repeat[0] != 1.0
            ? ba.total_length
            : 1),
```
Grid coordinate x

```
        grid_count[1] *
           (normalize_grid && repeat[1] != 1.0
            ? ba.total_length
            : 1));
} in vector_displacement(
    displace_with_bubble(repeat, ba),
    grid(grid_point));
```
Grid coordinate y

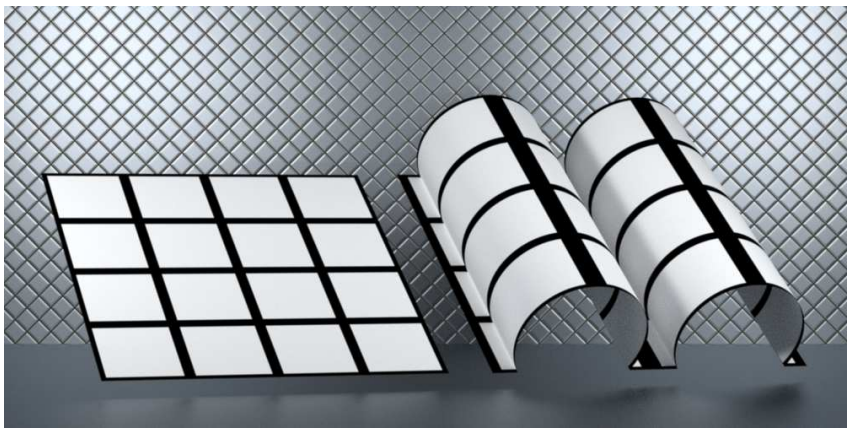
In Figure 14.20, the bubble profile is rendered with a single repetition and without adjusting the texture scaling. The texture is stretched in the *u* direction.



```
bubble(
  radius: 0.35,
  height: 0.0,
  repeat:
    float2(1, 0))
```

Fig. 14.20 – A single bubble profile without texture adjustment

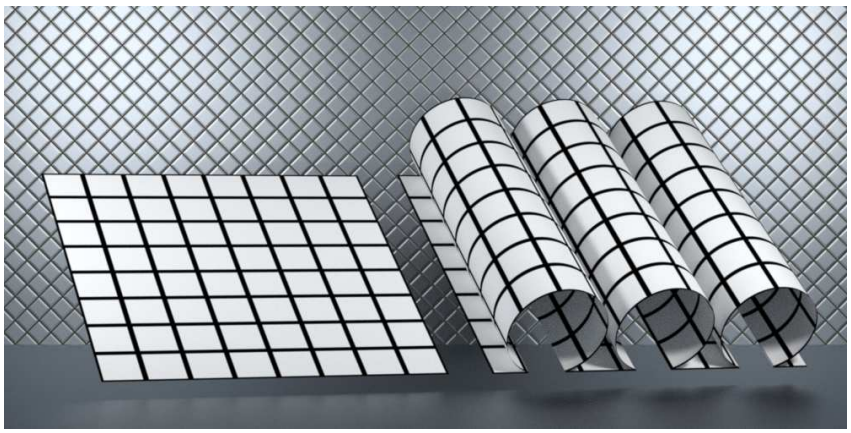
Figure 14.21 shows the result of repeating the bubble profile.



```
bubble(
  radius: 0.4,
  height: 0.2,
  repeat:
    float2(2, 0),
  grid_count:
    float2(4, 4))
```

Fig. 14.21 – Repeating the bubble profile

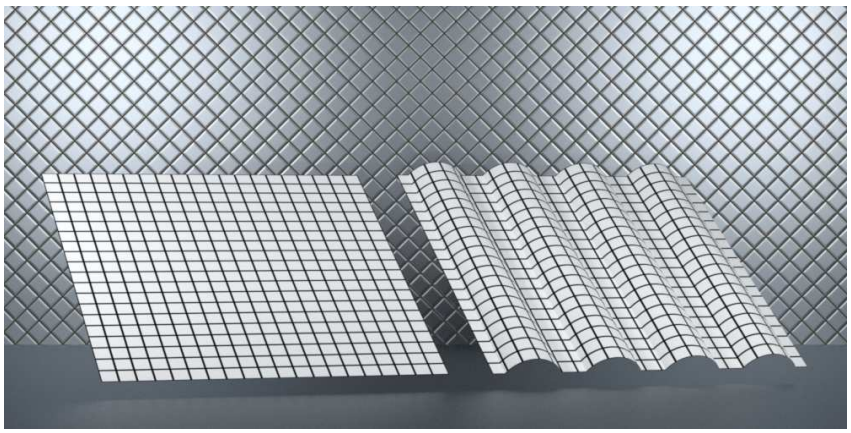
In Figure 14.22, the texture scaling is adjusted to restore the proportions in the u and v directions.



```
bubble(
  radius: 0.45,
  height: 0.42,
  repeat:
    float2(3, 0),
  grid_count:
    float2(8, 8),
  normalize_grid: true)
```

Fig. 14.22 – Restoring the proportions of the texture

Figure 14.23 demonstrates the result of a height value less that 0.0, with proportional adjustment of the texture.



```
bubble(
  radius: 0.4,
  height: -0.2,
  repeat:
    float2(4, 0),
  grid_count:
    float2(20, 20),
  normalize_grid: true)
```

Fig. 14.23 – A height value less that 0.0

The calculation of the angle and length arguments of the bubble profile are consolidated in a function that returns a custom struct. This modularization allows values to be shared between the geometry and surface properties. The bubble arguments are used in the calculation of the displacement vector, but the total length of the displaced surface also allows the texture to be rescaled to its original proportions.

14.4 Displacement defined by segments

The previous section shows the utility of custom structs as a means for sharing values between the surface and geometry properties of a material. This section develops the use of arrays for a more general vector displacement method.

14.4.1 Defining line segments

Consider a shape made of a series of connected points. Pairs of points define a curve consisting of a series of line segments. Such a curve has a *piece-wise* definition. Like the bubble profile of the previous section, a series of line segments can define the vector displacement of a surface.

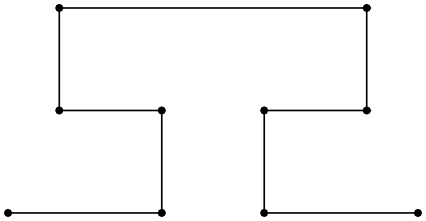


Fig. 14.24 – A profile of nine line segments

Unfolding the segments of Figure 14.24 into a straight line in Figure 14.25 produces a series of points separated by the same proportional relationships as in the original curve.

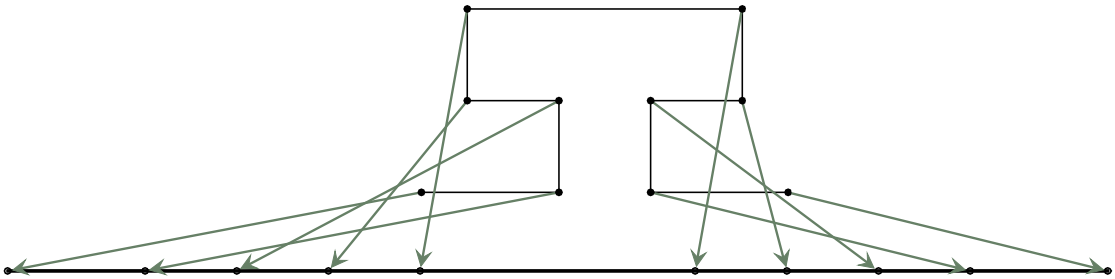


Fig. 14.25 – Mapping profile vertices to proportional positions on a line

Inverting the mapping produces Figure 14.26, with evenly distributed points on a segment associated with points evenly distributed along the profile.

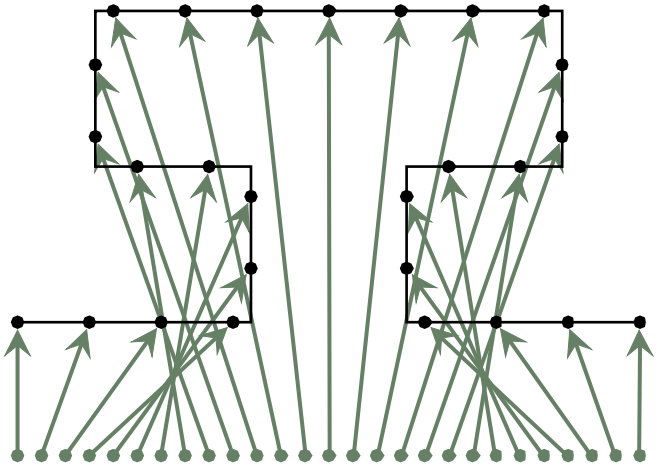


Fig. 14.26 – Mapping evenly spaced positions along a line to a profile

For a displacement mapping developed from line segments, an arbitrary relationship is defined between the position of the profile and the surface of the polygon that it displaces. In Figure 14.27 (page 261), the texture coordinate range of 0.0 to 1.0 is defined as the base of the figure. Some points will therefore only move between two positions on the polygon itself.

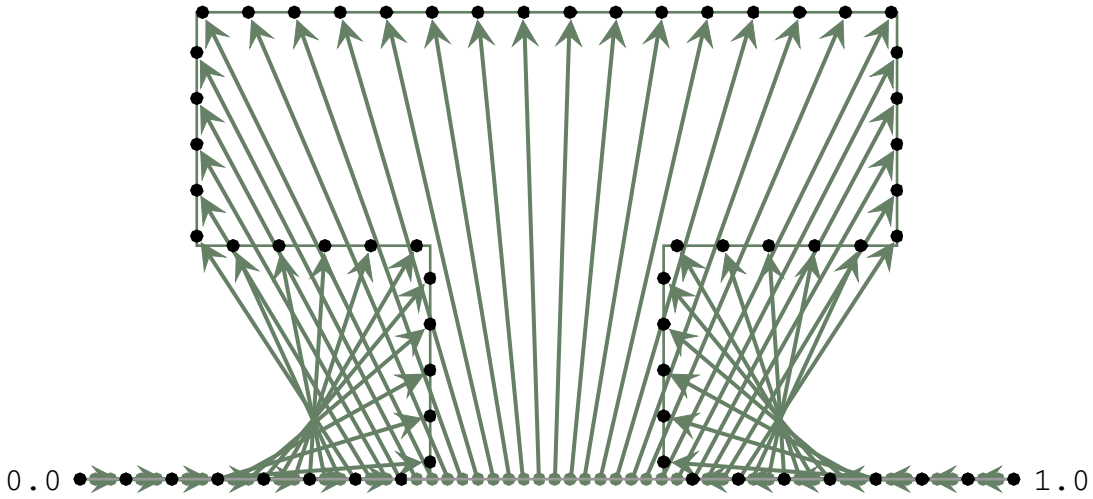


Fig. 14.27 – Points in texture space mapped to displacement vectors that create the profile

Like the bubble profile, a displacement material based on a profile can be divided into the functions that define the displacement vector and the use of those functions in the `material_geometry` property of the material.

14.5 Implementing a segmented profile

This section develops a general implementation of a profile based on a series of line segments, a *segmented profile*.

14.5.1 Segmented profile calculations

A segmented profile can be defined as `const` MDL array of `float2` values.

Listing 14.22

```
const float2[10] tee(
  float2(0.000, 0.000),
  float2(0.375, 0.000),
  float2(0.375, 0.250),
  float2(0.125, 0.250),
  float2(0.125, 0.500),
  float2(0.875, 0.500),
  float2(0.875, 0.250),
  float2(0.625, 0.250),
  float2(0.625, 0.000),
  float2(1.000, 0.000));
```

To be able to determine the proportional distance along the segmented profile, the accumulated absolute distance from the first point to each point must be calculated. In Listing 14.23 (page 262), function `profile_lengths_10` takes an array of ten `float2` values as an argument. The size of the `result` array must also be explicitly defined, here an array of ten elements. (As a tiny nod to efficiency, the distance from the first point to itself is set to zero before the subsequent distance calculations begin.)

Listing 14.23

```
uniform float[10] profile_lengths_10(
  uniform float2[10] pts)
{
  float[10] result;
  result[0] = 0.0;
  for (int i = 1; i < 10; i++) {
    result[i] =
      result[i-1]
      + math::distance(pts[i-1], pts[i]);
  }
  return result;
}
```

Result array has the same size as the argument array

Each element is the sum of the previous segment lengths

However, MDL also allows the size of an array to be specified in its declaration by a variable called a *size identifier*. The size identifier takes the place of the explicit array size defined by an integer. It is identified by being surrounded by angle brackets, so that `<N>` declares `N` to be a size identifier. Later evaluations of `N` will use the value specified in the defining expression.

An array declared with a size identifier is called a *size-deferred* array. When the execution time of size definition should be emphasized, an array with an explicit size is called a *size-immediate* array.

Listing 14.24

```
float[10] ten_values;
```

Size-immediate array

```
float[<N>] values;
```

Size-deferred array; size identifier `N` surrounded by angle brackets

```
float[N] sums;
```

Array `sums` contains the same number of elements as the `values` array

Because the value of the size identifier is defined at the time the actual array size is known when the function or material is called, functions and materials can be generalized with regards to the size of the arrays that they accept as arguments.

Rewriting function `profile_lengths_10` using size-deferred arrays, the variable `N` is initialized in the argument list, and then used for the return type as well as in the body of the function.

Listing 14.25

```
uniform float[N] profile_lengths(
  uniform float2[<N>] pts)
{
  float[N] result;
```

Variable N initialized in the argument declaration and used for the return array

Use N to define result array


```
  result[0] = 0.0;
  for (int i = 1; i < N; i++) {   Use N for loop termination
    result[i] =
      result[i-1]
      + math::distance(pts[i-1], pts[i]);
  }
  return result;
}
```

The function `profile_lengths` returns the length of the profile at each point. Function `normalized_lengths` returns an array that scales the length values into the range of 0.0 to 1.0, also using deferred-size arrays.

Listing 14.26

```
uniform float[N] normalized_lengths(
  uniform float[<N>] lengths)          Input and output arrays have the same size
{
  float total_length = lengths[N-1];   Last element of the input array is the total length
  float[N] result;
  result[0] = 0.0;
  for (int i = 1; i < N; i++) {
    result[i] = lengths[i] / total_length;   New value is a fraction of the length (0.0 → 1.0)
  }
  return result;
}
```

Given a fractional value of 0.0 to 1.0, a point at that position on a segmented profile can be found in two steps. First, the segment that contains the point is located. Second, the fractional position *within* the segment is determined to find the point.

Listing 14.27

```
float2 point_along_profile(
  float distance,                      Distance along the profile
  float2[<N>] profile,                 Profile array of N float2 values
  uniform float[N] t,                  Normalized profile length array of N float values
  float repeat)                        Repetition of the profile curve
{
```

```
  float dist =
    math::frac(distance * repeat);
```
Total distance with the repetition taken into account

```
  int i = 1;
  while (i < N) {
    if ((t[i-1] <= dist) && (dist <= t[i]))
      break;
    i += 1;
  }
```
Find segment that contains the point that is "dist" units from the start of the segment, using N for the loop termination

```
  float segment_fraction =
    (dist - t[i-1]) / (t[i] - t[i-1]);
```
Fractional position of the point in the segment

```
  float2 result = math::lerp(
    profile[i-1], profile[i], segment_fraction);
```
Point on the segment at the fractional position

```
  if (repeat != 1.0) {
    float offset =
      math::floor(distance * repeat) / repeat;
    result /= repeat;
    result[0] += offset;
  }
```
Scale profile point for the repeat parameter if necessary

```
  return result;
}
```

Like the bubble profile calculation, once the profile point is found, the `modify_normal` function creates a vector in three dimensions, based on the surface normal and the selected tangent vector.

Listing 14.28

```
float3 displace_with_profile(
  uniform float2[<N>] profile, uniform float[N] t,
  int uv_index, float repeat)
{
  float f = math::frac(
    state::texture_coordinate(0)[uv_index]);
```
Fractional value along the profile from the texture coordinate

```
  float3 tangent = uv_index == 0
    ? state::texture_tangent_u(0)
    : state::texture_tangent_v(0);
```
Select the tangent vector to be scaled by the x component of profile

```
  float2 profile_point =
    point_along_profile(f, profile, t, repeat);
```
Point on the profile for normal vector modification

```
  return modify_normal(f, profile_point, tangent);
}
```


14.5.2 Segmented profile material

The vector displacement of the segmented profile will, like the bubble profile, stretch the texture space. To compensate, one of the two scale factors for the `grid` material can be scaled to match.

Listing 14.29

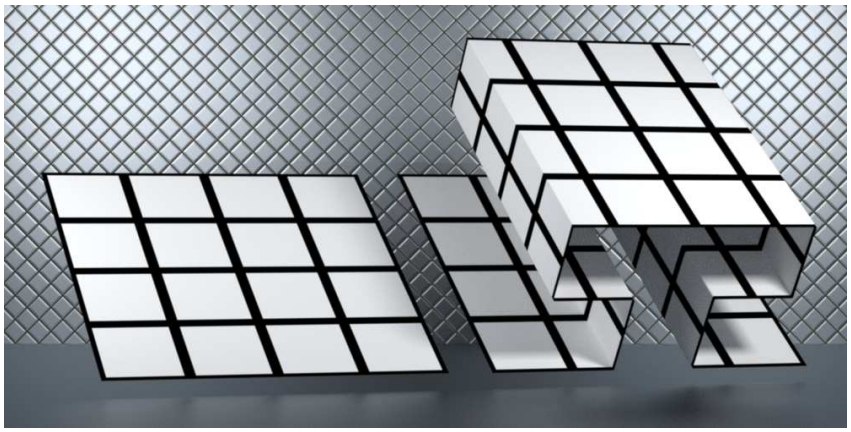
```
uniform float2 scale_component(
  uniform float2 v, uniform float[<N>] scale, uniform int index)
{
  float2 result = v;
  result[index] *= scale[N-1];   Scaling compensation selected by the index
  return result;
}
```

Calculating the segmented profile's incremental length vector in a `let` statement allows the total length—the value of the last element of the array of segment lengths—to be used in scaling the grid to maintain the proportions in u and v.

Listing 14.30

```
material tee_displacement(
  uniform int uv_index = 0,
  uniform float repeat = 1.0,
  uniform float2 grid_count = float2(4.0))
= let {
  uniform float[] lengths =
    profile_lengths(profile_arrays::tee);   Incremental length of the profile

  uniform float[] t =
    normalized_lengths(lengths);   Profile lengths scaled from 0.0 → 1.0

  uniform float2 scaled_grid_count =
    scale_component(
      grid_count, lengths, uv_index);   Total length used to adjust the grid scaling
} in
  vector_displacement(
    displace_with_profile(
      profile_arrays::tee, t, uv_index, repeat),   Replacement of the normal vector
    grid(scaled_grid_count));   Grid scaled to compensate for texture stretching
```

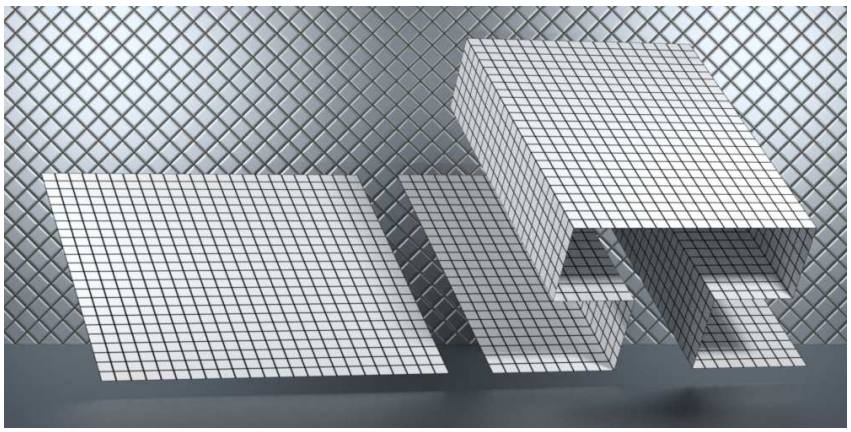
Figure 14.28 (page 266) shows the result of the default parameter values for the `tee_displacement` material.



```
tee_displacement()
```

Fig. 14.28 – The linear uv coordinate space maintained during the mapping of a segmented profile

Figure 14.29 shows that increasing the `grid_count` still maintains the proportional relationship of the texture coordinates.



```
tee_displacement(
  grid_count:
    float2(24, 24))
```

Fig. 14.29 – Scaling the uv coordinate space for a profile object

14.5.3 Profile repetition

A single segmented profile can be designed to repeat in a symmetrical manner. For example, Figure 14.30 shows the symmetry of three tee profiles attached end to end.

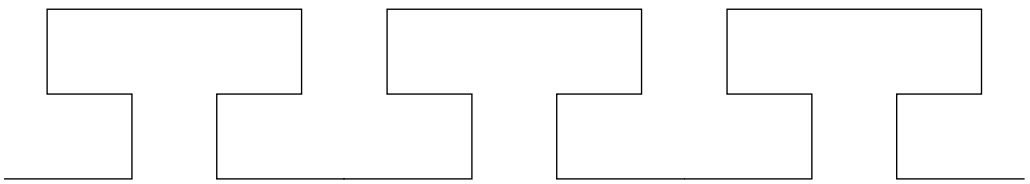


Fig. 14.30 – Three repetitions of the tee profile

The repetition of a segmented profile across the surface only requires the scaling of the *uv* space of the surface.

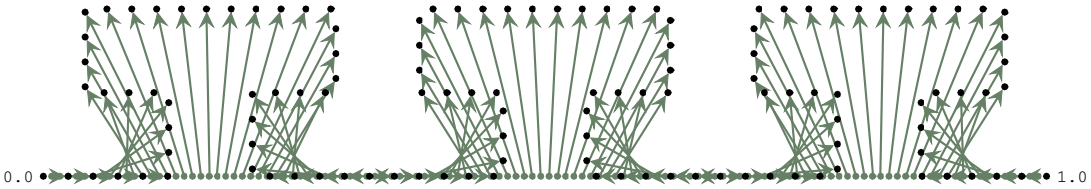
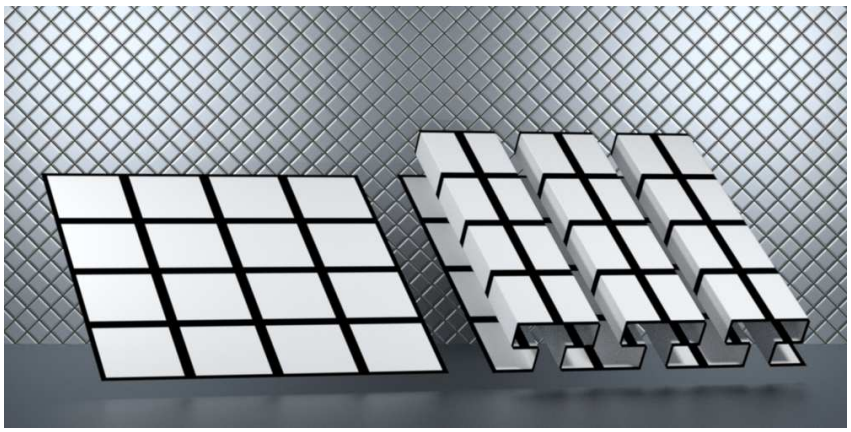


Fig. 14.31 – Repeating a profile in texture space

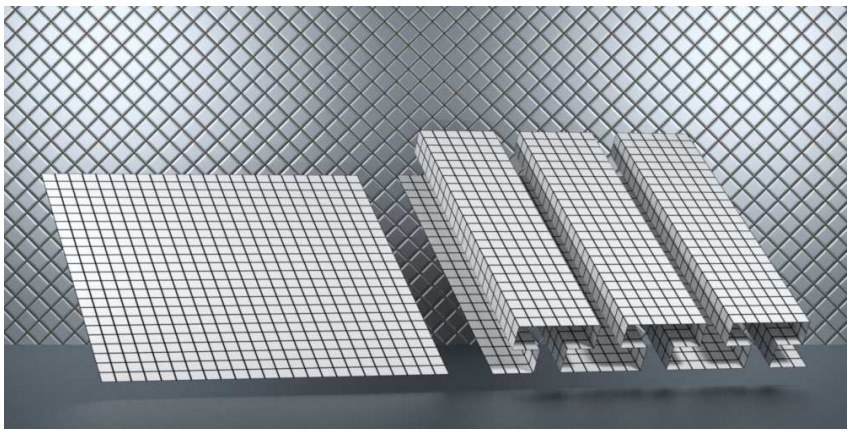
Unmentioned in the previous description of `tee_displacement` was its `repeat` parameter. This scales the *uv* space to create a repetition of the displacement caused by the profile.



```
tee_displacement(
  repeat: 3)
```

Fig. 14.32 – Repeating the profile three times in the u direction

The proportion of the texture is also maintained when both the texture and the profile are repeated.



```
tee_displacement(
  repeat: 3,
  grid_count:
    float2(24, 24))
```

Fig. 14.33 – Repeating the profile and scaling the texture

14.5.4 Profiles as rendering resources

The careful reader (or frustrated material user who finds that `tee_displacement` (page 265) will not compile) will notice that the array that forms the tee shape was included in the material as `profile_arrays::tee`. In other words, array `tee` must be defined in a module called `profile_arrays`, produced from the file `profile_arrays.mdl`. For profile arrays, the use of a separate module helps separate data (profile arrays) from the processes that use those data (material calculations during rendering). This separation supports applications that provide a

choice of complex inputs to materials, as well as allowing the ready addition of new profiles by users of the application.

The next chapter describes how this machinery of profiles as input data for materials can be extended to a feature of traditional architectural design.

15 Architectural details

The examples of geometric manipulation by MDL's `material_geometry` property in the previous chapters blur the traditional line between rendering and modeling. Displacements during rendering—and the resulting creation of additional triangles—are not based on the absolute size of the geometric model, but on the number of rays that strike the object. This concern for the appropriate amount of geometric detail with respect to screen space is called *level of detail* (LOD) modeling.

This chapter shows how vector displacements in MDL can provide the efficiencies of level-of-detail methods. A small, domain-specific language simplifies the definition of geometric shapes, providing further efficiencies in the human, expressive realm.

15.1 A neoclassical handbook

Neoclassical building style in the eighteenth century copied and expanded upon the elaborate architectural designs of classical Greece and Rome. Published in 1767, *Palladio Londinensis*[11] by William Salmon explains the economic details of construction along with practical lessons in geometry. Salmon also presents diagrams including numerical proportions for various architectural elements. For example, Figure 15.1 shows the proportional measurements for the imposts (arch supports) and arches of three classical orders:

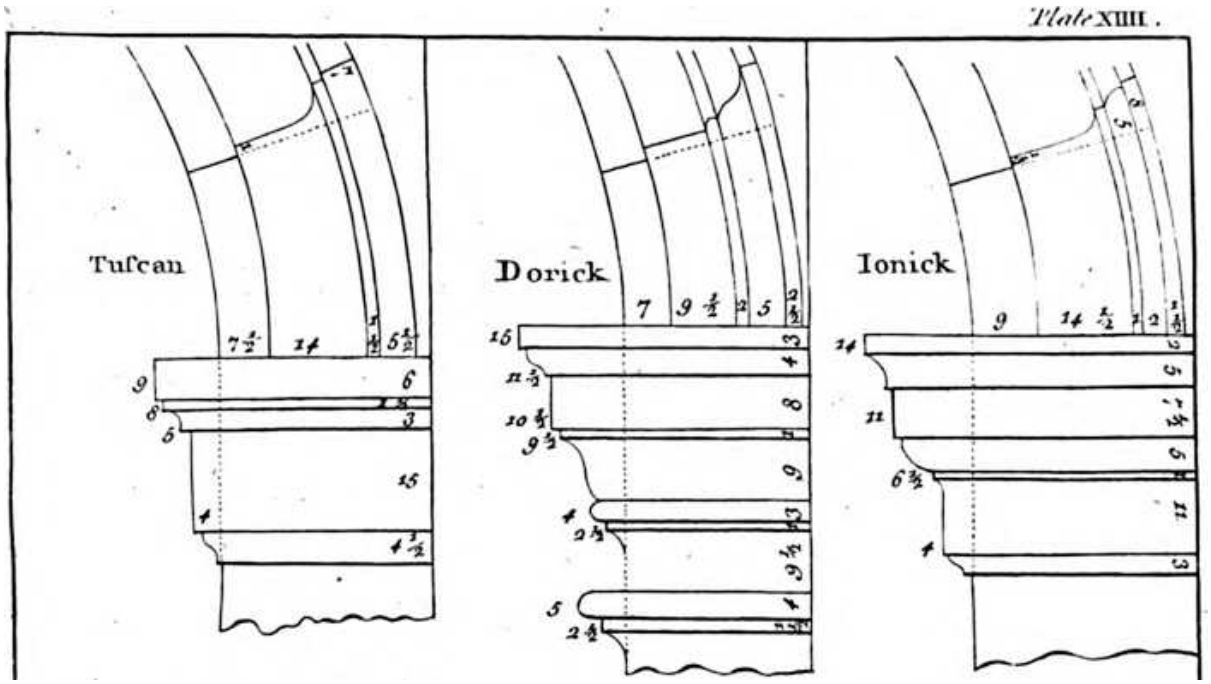


Fig. 15.1 – Diagrams of imposts and the arches they support for the Tuscan, Doric, and Ionic orders in Palladio Londinensis *from Plate XIIII*

11. `http://www.mdz-nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:bvb:12-bsb10048482-2`

In Plate XXIX (Figure 15.2) Salmon provides the numerical proportions of two examples of door and window ornaments that "may be made in divers Ways."

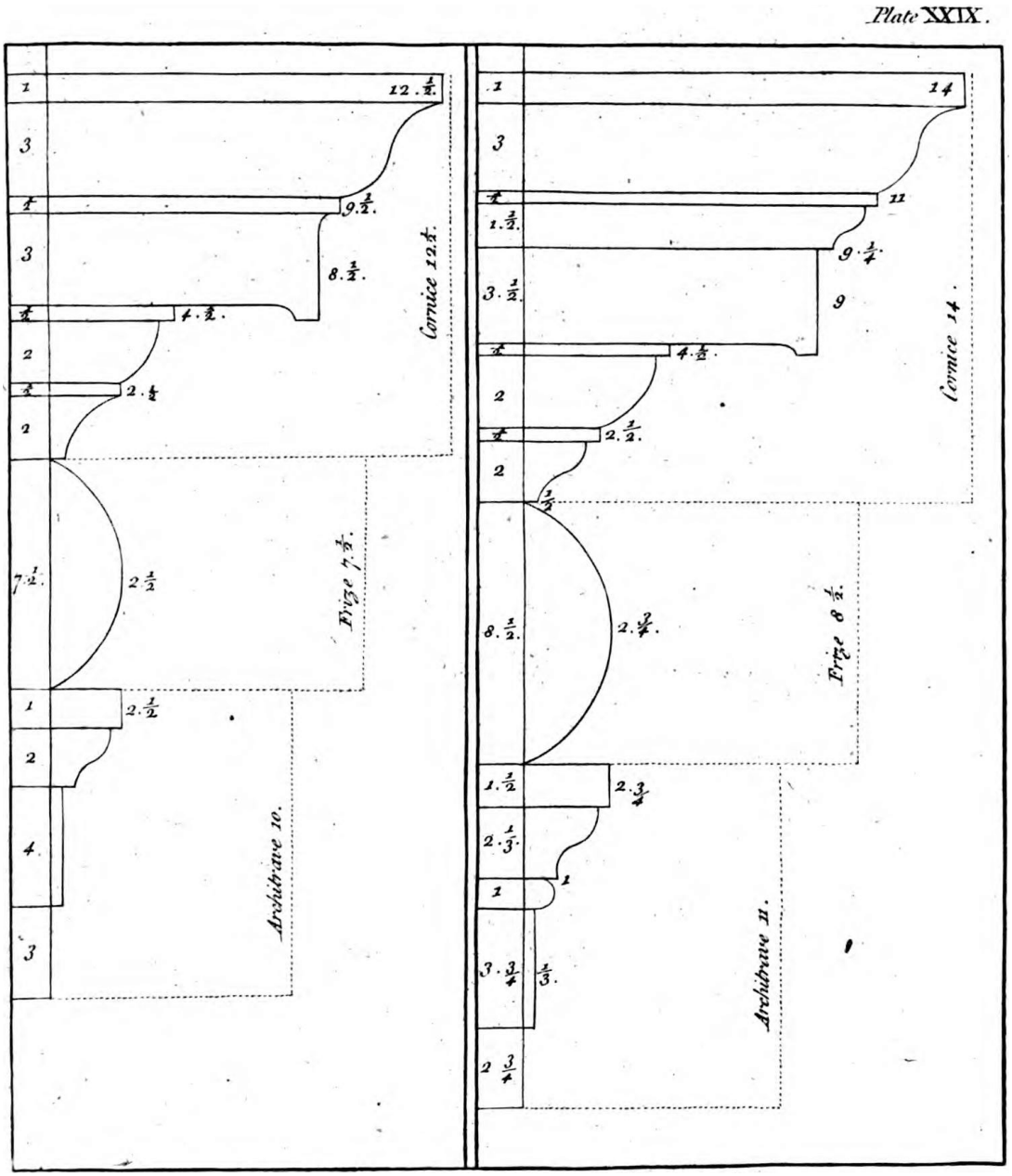


Fig. 15.2 – Proportional measurements for window and door ornaments, Palladio Londinensis, *Plate XXIX*

In the last chapter, "Geometric profiles" (page 241), vector displacement was defined by a `const` array of type `float2` (page 261). To describe the form of Salmon's designs, circular arcs as well as straight line segments will be needed. In this chapter, a description of a *piece-wise curve* in a simple language is translated into an MDL array to model surface structures by using vector displacements.[12]

12. Architectural historians will, it is hoped, forgive any technical naiveté displayed in their domain for the benefit of better explaining MDL.

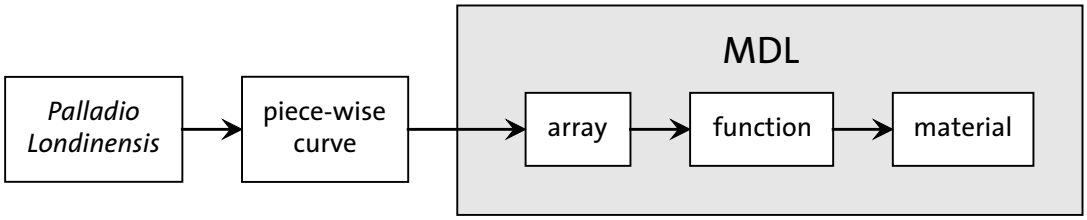


Fig. 15.3 – Encoding a geometric shape described in the Palladio Londinensis *and making it available to a material*

15.2 A grammar for ornamental form

This section defines a method for representing a two-dimensional piece-wise curve. This method will describe the profiles of three-dimensional shapes presented in *Palladio Londinensis* for use as vector displacements in MDL.

For this simple implementation, the piece-wise curve will consist only of line segments and circular arcs. A point on the curve is called a *pw-point*. A series of pw-points is called a *pw-curve*. Each pw-point defines how it is connected to the previous pw-point in the pw-curve.

A pw-point specifies an x and a y coordinate value. By default, two successive pw-points are connected by a line segment. An optional *pw-point property* defines other ways of connecting two pw-points.

A pw-point is specified in a file on a single line and has one of three forms:

1. *x y*
2. *x y property-name*
3. *x y property-name property-arguments*

To connect two pw-points by a circular arc, one of the following properties are added to the second point:

`:angle` *degrees*
: This pw-point and the previous one in the pw-curve are connected by a circular arc of angle *degrees*. The sign of *degrees* controls whether the arc is convex or concave in the context of the pw-curve.

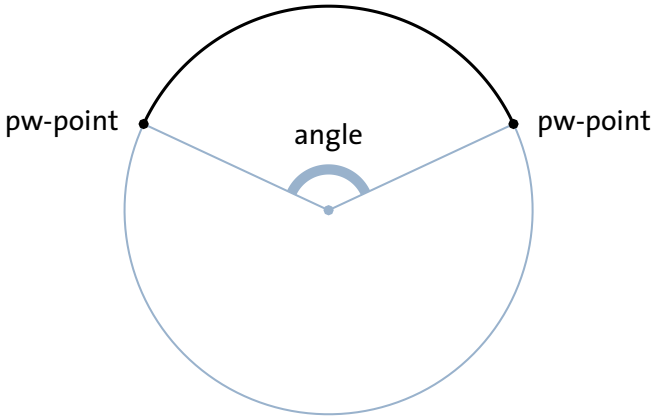


Fig. 15.4 – Two pw-points are connected by a circular arc

`:sag` *distance*

The *sagitta* is the perpendicular distance from the midpoint of a chord to the midpoint of the chord's circular arc. For the `:sag` property, the two pw-points are connected by a circular arc such that the sagitta of the chord formed by the two pw-points is *distance*.

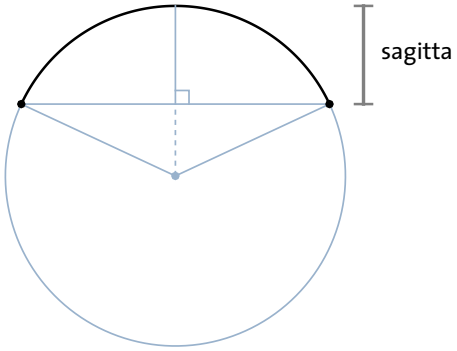


Fig. 15.5 – The definition of the sagitta distance

`:cyma_recta`
`:cyma_reversa`

The `cyma_reversa` and `:cyma_recta` arguments connect two pw-points by two circular arcs. In the *Palladio Londinensis*, drawing with a compass constructs the centers of the two circles of the connected arcs (Figure 15.6)

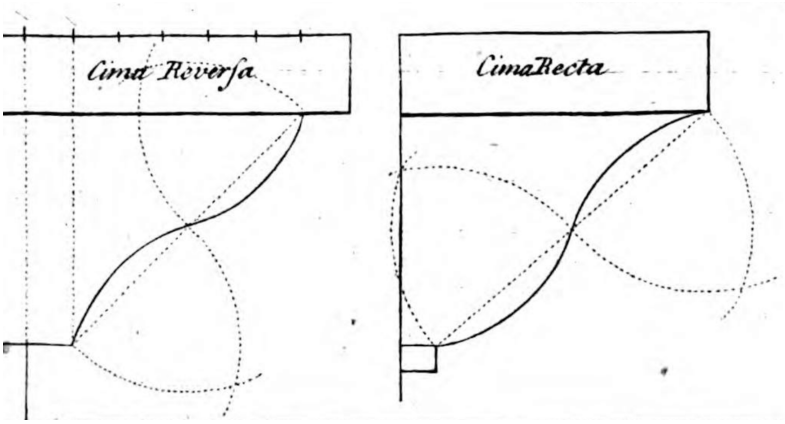


Fig. 15.6 – The geometric description of the cyma reversa and cyma recta in the Palladio Londinensis

The two pw-points are vertices in equilateral triangles formed from their midpoint and the center of the circle on which the pw-point lies. At the midpoint, the slopes of the two arcs are identical, forming a smooth curve at that point.

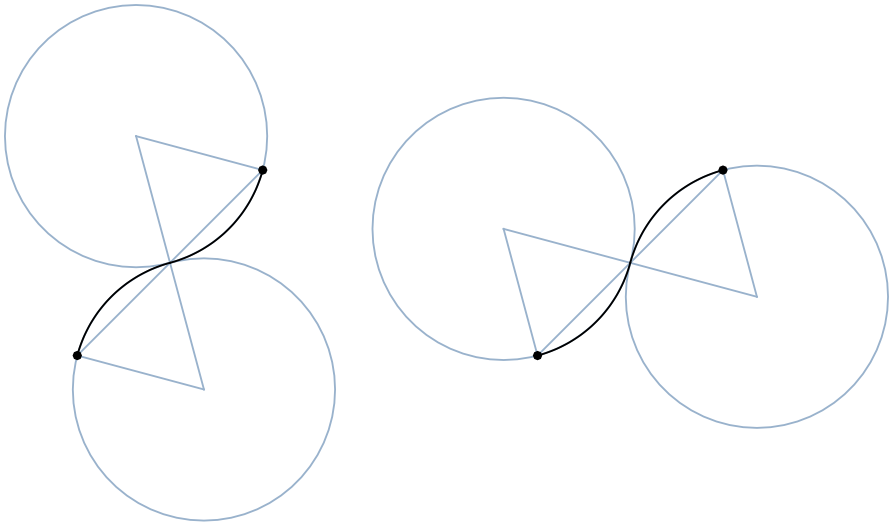


Fig. 15.7 – The cyma reversa and cyma recta are formed by circles of the same size intersecting at a single point

`:flare`
: A *flare* joins two points with an arc. The beginning part of the arc is tangent to the previous segment. This constraint determines where on the circle the second point lies.

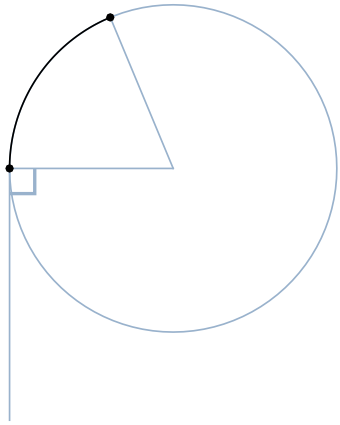


Fig. 15.8 – A flare connects a segment to an arc to which it is tangent

Line segments and these five types of curve description will form the basis for a simple textual description of piece-wise curves.

15.3 Describing piece-wise curves

A *pwc-list* is a list of pw-points based on the descriptions of the last section. This section shows the basic shapes that a pwc-list can describe. In the next section, these pwc-lists are converted into MDL `const` arrays of a custom type designed for using pw-curves for vector displacements.

Translating a pwc-list to an MDL array is done by a separate program that creates an efficient representation of the pwc-list for use during rendering. A graphical application could be designed in a similar manner, where an interactive drawing program provides the definition of a piece-wise curve, which is translated by the application into an efficient MDL representation.[13]

13. The pwc-list translation program will be available in a future version of the *MDL Handbook*.

15.3.1 Segments

The first example of a pwc-list only defines segments. The x and y values can be specified as fractions, which can be helpful in duplicating the measurements of *Palladio Londinensis*. In the rendering of the pw-curve, the blue square is the original form of the geometric object that is displaced by the pw-curve.

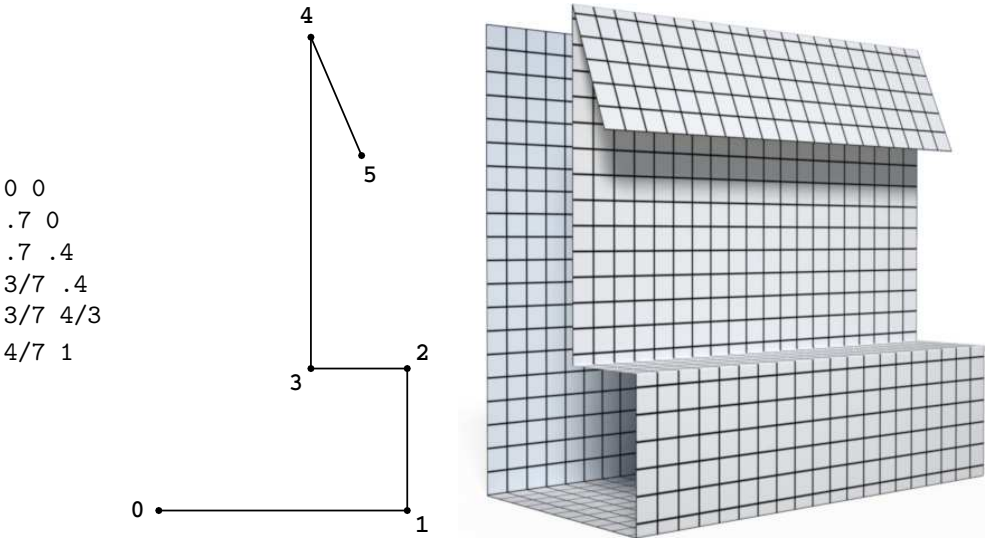


Fig. 15.9 – A series of segments

The `:relative` property specifies that the position of each pw-point is the relative x and y translation from the previous point in the pw-curve. This will be very useful in representing the shapes of Figure 15.6 (page 272), which are based on relative measurements throughout.

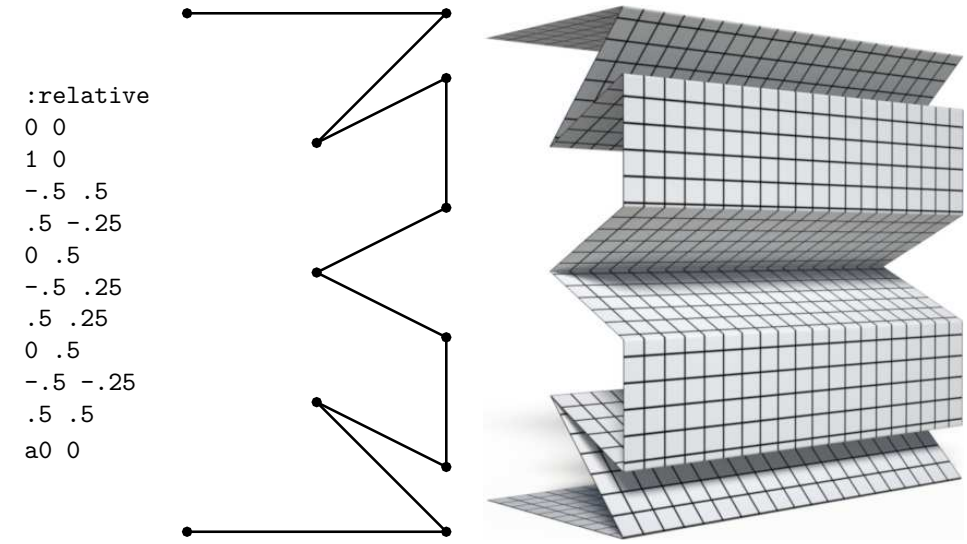


Fig. 15.10 – Relative offsets for points

15.3.2 Angle

The `:angle` property is a negative value for a clockwise arc between the pw-points. In this diagram, the *x* and *y* offsets are equal, forming a rounded corner.

```
:relative
0 0
.25 0
0 .55
.125 .125 :angle -90
.375 0
.125 -.125 :angle -90
.125 0
0 1
.25 .25 :angle -90
0 .25
a0 0
```

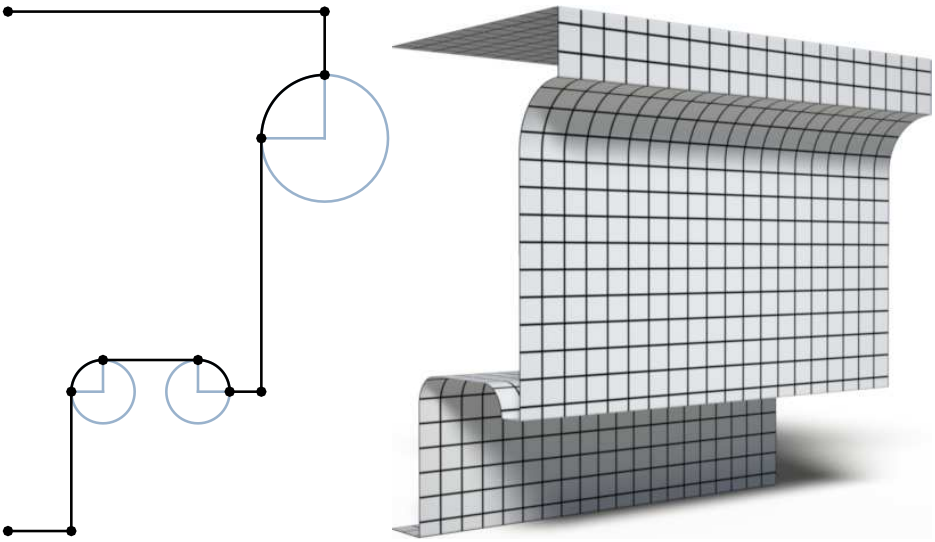


Fig. 15.11 – Arcs defined by their angle measurement

15.3.3 Sagitta

An arc produced by the `:sag` property in pw-curve diagrams includes a line segment showing the sagitta distance.

```
:relative
0 0
.5 0
0 .25
0 1 :sag .3
0 .25
-.5 0
```

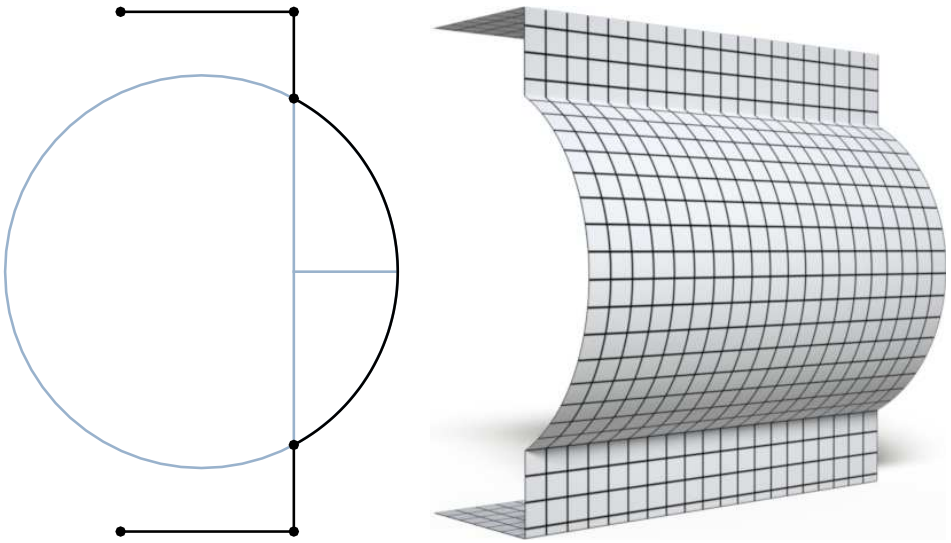


Fig. 15.12 – Defining an arc by its sagitta

The sagitta distance can simplify the expression of an arc's relationship to other portions of the pw-curve. In Figure 15.12 (page 275), a distance of 0.3 extends the arc a slight amount past the upper and lower segments.

```
:relative
0 0
.5 0
0 .25
-.25 0
0 1 :sag .3
.25 0
0 .25
-.5 0
```

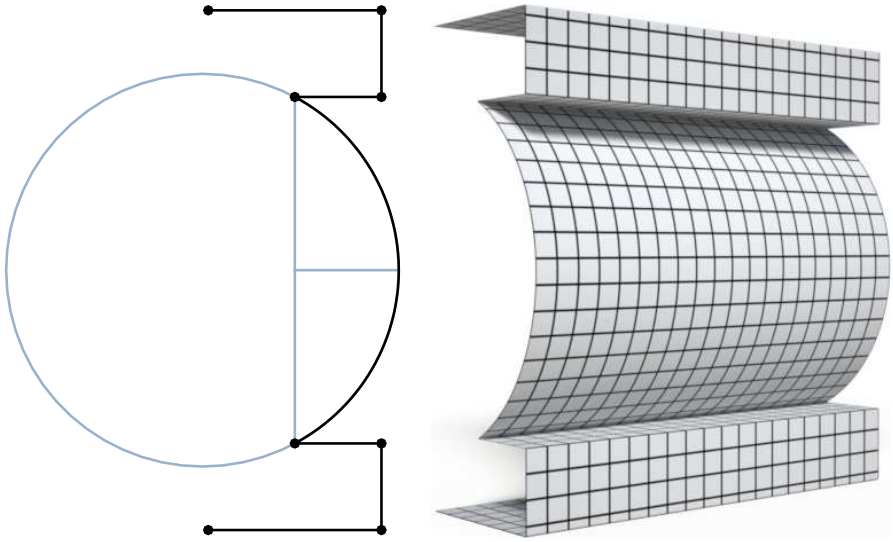


Fig. 15.13 – Combing segments and curves

15.3.4 Cyma reversa and cyma recta

The `:cyma_reversa` and `:cyma_recta` properties have no additional arguments; the position of the two pw-points uniquely describes the resulting curve. The prefix "`a`" in the last pw-point specifies that the number should be treated as an absolute x or y value. This will also be useful when defining the complex curves of *Palladio Londinensis*.

```
:relative
0 0
.25 0
0 1 :cyma_reversa
a0 0
```

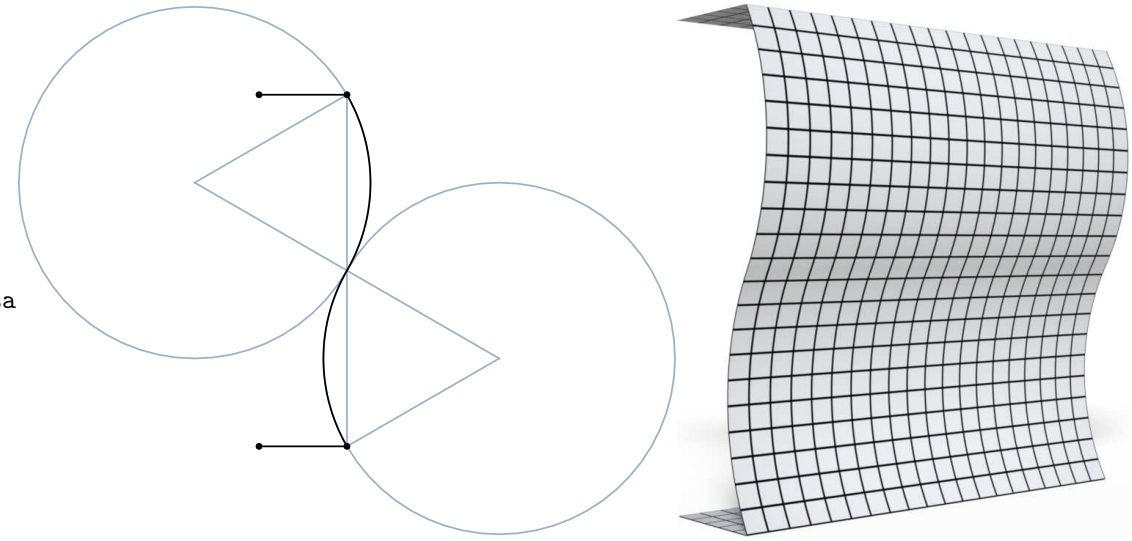


Fig. 15.14 – Cyma reversa

To distinguish between the two types of cyma, note that the first arc of the cyma reversa—in the order of the pw-points—is always concave whereas the first arc of the cyma recta is convex.

```
:relative
0 0
.25 0
0 1 :cyma_recta
a0 0
```

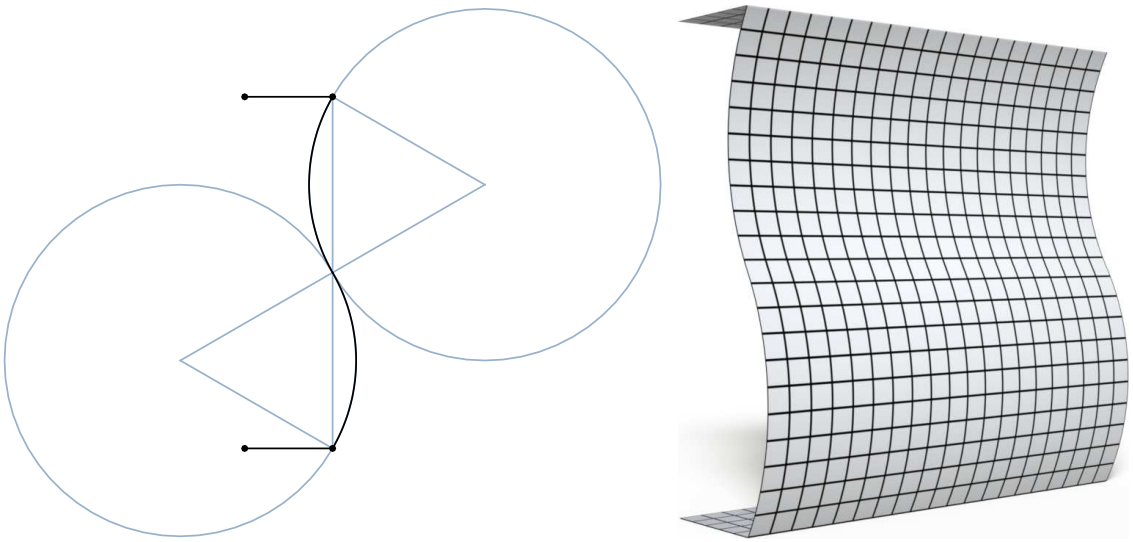


Fig. 15.15 – Cyma recta

15.3.5 Flare

Like the cyma reversa and recta, the `:flare` property has no arguments. The constraint that the arc is tangent to the preceding segment at the first point defines the center of the circle.

```
:relative
0 0
.1 0
0 .55
.2 .3 :flare
0 .3
.3 .1 :flare
0 .2
a0 0
```

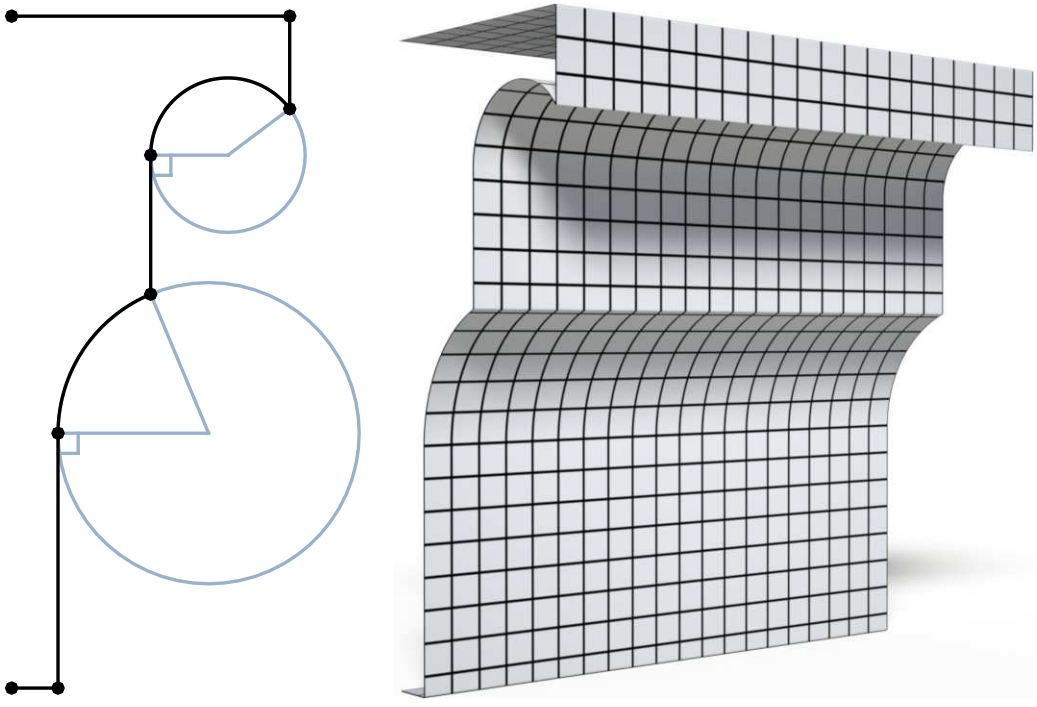


Fig. 15.16 – Flare

15.4 Implementing curves from *Palladio Londinensis*

With the types of curves developed in the last section, a pwc-list can define curves in the *Palladio Londinensis*. For example, Figure 15.17 is an encoding of the first diagram in Plate XXIX.

```
:relative
0 0
1.5 0
0 3
.5 0
0 4
.5 0
1.5 2 :cyma_reversa
.5 0
0 1.5
a1.5 0
0 7.5 :sag 2.5
.5 0
2 2 :sag -.5
0 .5
1.5 2 :sag .25
.5 0
0 .5
3.5 0
.5 -.5 :angle -90
1 0
0 3
.5 .5 :angle -90
.5 0
0 .5
3 3 :cyma_recta
0 1
a0 0
```

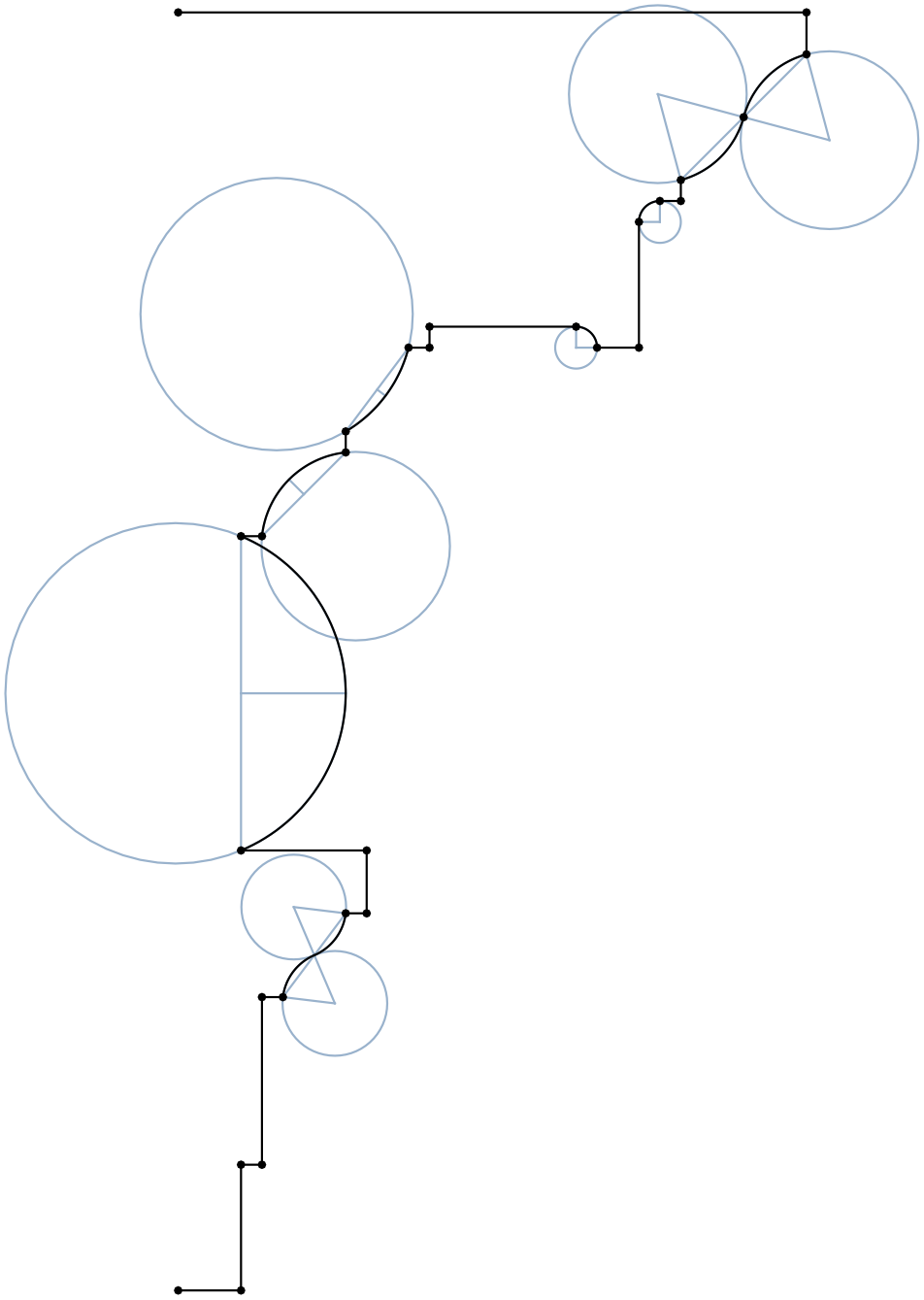


Fig. 15.17 – The first ornament from Figure 15.3 (page 271), Plate XXIX in Palladio Londinensis

Figure 15.18 (page 279) encodes the second diagram from Plate XXIX.

```
:relative
0 0
1.5 0
0 2.75
1/3 0
0 3.75
a2.5 1 :sag .4
a2.75 0
a4 2-1/3 :cyma_reversa
a4.25 0
0 1.5
a1.5 0
0 8.5 :sag 2.75
.5 0
1.5 2 :cyma_reversa
.5 0
0 .5
2 2 :sag .5
.5 0
0 .5
4 0
.5 -.5 :angle -90
.5 0
0 4
.5 0
1 1.5 :cyma_reversa
.5 0
0 .5
3 3 :cyma_recta
0 1
a0 0
```

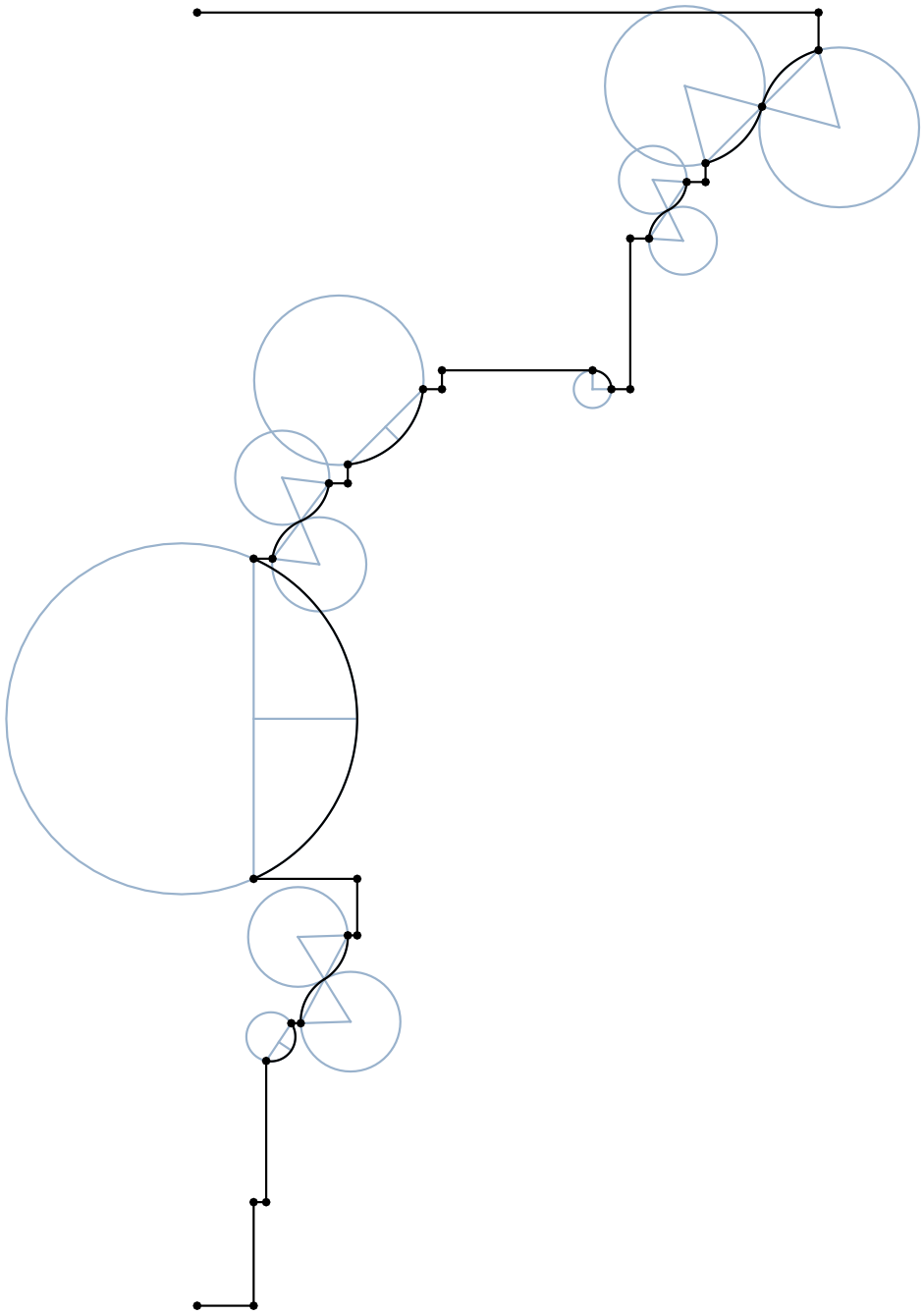


Fig. 15.18 – The second ornament in Plate XXIX

As another example of encoding a diagram in *Palladio Londinensis*, Figure 15.19 is an arch support, called an *impost* in Plate XIIII:

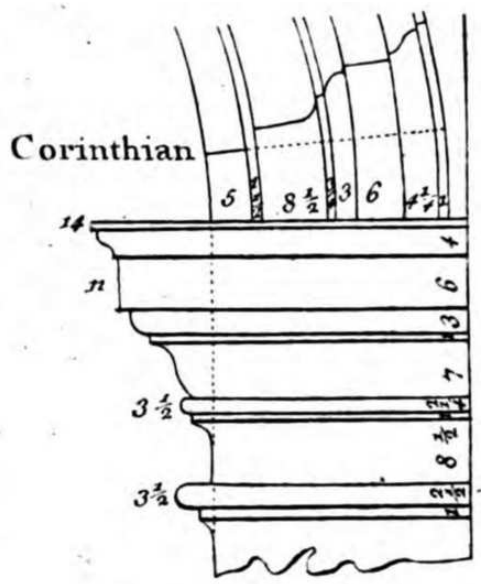


Fig. 15.19 – Diagram of Corinthian impost and arch in Palladio Londinensis*, from Plate XIIII*

In addition to the piece-wise elements used to encode the previous two diagrams, the Corinthian impost also uses the `:flare` element.

```
:relative
0 0
15 0
0 7.5
1 1.5 :flare
0 1
1 0
0 2.5 :angle 180
a15 0
0 6
2 2.5 :flare
0 1
.25 0
0 2.25 :angle 180
-.25 0
7 8.5 :cyma_recta
0 1
3 3 :sag .7
2 0
0 6
1 0
4 4 :cyma_reversa
1 0
0 1
a0 0
```

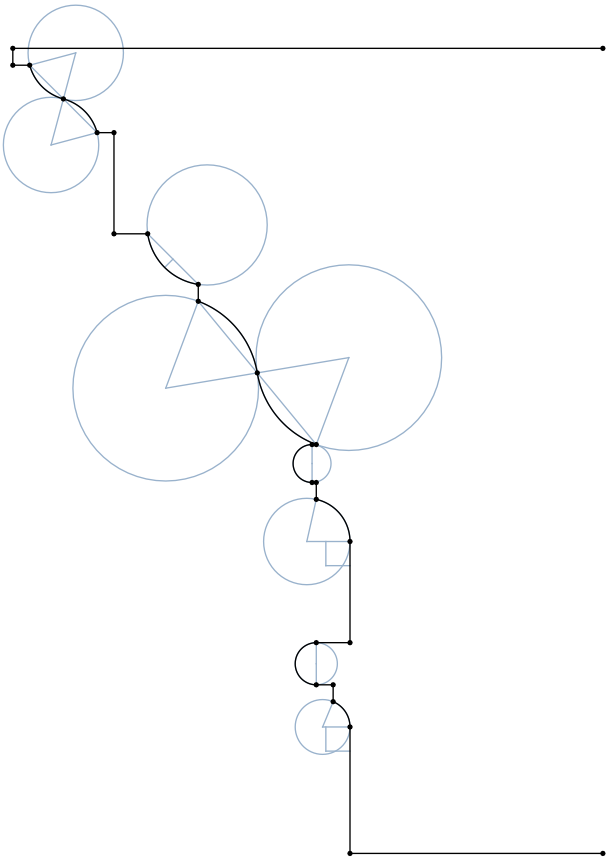


Fig. 15.20 – Corinthian impost defined by a pwc-list

When encoding diagrams in the *Palladio Londinensis*, some parts of the original drawing may be implemented in more than one way—a flare could also be used to define a 90° angle. Additional piece-wise curve types—for example, splines—would be required for encoding other *Palladio Londinensis* diagrams.

15.5 Defining a piece-wise curve in MDL

How should a piece-wise curve be represented in MDL? The five curve types —angle, sagitta, cyma recta, cyma reversa and flare—are simply convenient ways of describing arcs of a circle. The piece-wise curve in MDL can be therefore be represented by a series of arcs or segments, with the cyma recta and cyma reversa defined by two arcs. A separate translation program converts the human-readable pwc-list into an MDL array. An array is an efficient representation for calculating the appropriate point along the curve for use in defining the displacement vector.

The six types of pw-points are identified by a custom `enum` name `pw_type`:

Listing 15.1

```
enum pw_type {
    segment,
    angle,
    sagitta,
    cyma_recta,
    cyma_reversa,
    flare
};
```

Types of pw-points in the pwc-list

The `pw_type` enum is the first field of the MDL definition of a pw-point, the custom struct `pw_point`:

Listing 15.2

```
struct pw_point {
    pw_type type;
    float t = 0.0;
    float2 pt = float2(0.0);
    float2 center = float2(0.0);
    float radius = 0.0;
    float start_angle = 0.0;
    float end_angle = 0.0;
};
```

Type from pw_type enum

Parameterized distance along the pw-curve (0.0 → 1.0)

Position of point

Circular arc parameters

The pwc-list translation program uses the `pw_point struct` and the `pw_type enum` to define a `const` array of elements of type `pw_curve::pw_point`, that is, the struct `pw_point` defined in the `pw_curve` module.

The field `t` is the parametric position of the pw-point along the pw-curve, with a value from 0.0 to 1.0. The mapping from texture space to a point on the pw-curve will use this value to determine the piece in which the desired point will be found.

The translation from the pwc-list also calculates the length of the pw-curve. This value can be passed as an argument to the `material_surface` calculation to maintain the same texture scaling in u and v.

As a convenience in debugging, the pwc-list element is added as a comment to each pw-point.

Listing 15.3 – Constructed MDL array for Figure 1.7 (page 274)

```
const float segments_length = 2.00056;
```
Length of pw-curve for texture scaling

```
const pw_curve::pw_point[6] segments (
  pw_curve::pw_point( // 0 0
    pw_curve::segment, 0.0, float2(0.0, 0.0)),
  pw_curve::pw_point( // .7 0
    pw_curve::segment, 0.26243, float2(0.525, 0.0)),
  pw_curve::pw_point( // .7 .4
    pw_curve::segment, 0.41238, float2(0.525, 0.3)),
  pw_curve::pw_point( // 3/7 .4
    pw_curve::segment, 0.51414, float2(0.32142857, 0.3)),
  pw_curve::pw_point( // 3/7 4/3
    pw_curve::segment, 0.86404, float2(0.32142857, 1.0)),
  pw_curve::pw_point( // 4/7 1
    pw_curve::segment, 1.0, float2(0.42857143, 0.75)));
```
Constructor for a `pw_curve::pw_point` includes the pwc-list element as a comment

Properties of the entire pwc-list, like `:relative`, are placed as a comment to the declaration of the array. For the definition of the four circular arc types, additional fields to the `pw_curve::pw_point` struct define the circle's center, its radius, and the beginning and ending angle measurements.

Listing 15.4 – Constructed MDL array for Figure 1.9 (page 275)

```
const float right_angle_arc_length = 2.23678;

const pw_curve::pw_point[11] right_angle_arc ( // :relative
  pw_curve::pw_point( // 0 0
    pw_curve::segment, 0.0, float2(0.0, 0.0)),
  pw_curve::pw_point( // .25 0
    pw_curve::segment, 0.05452, float2(0.12195122, 0.0)),
  pw_curve::pw_point( // 0 .55
    pw_curve::segment, 0.17447, float2(0.12195122, 0.26829268)),
```
Property of the pwc-list

```
pw_curve::pw_point( // .125 .125 :angle -90
  pw_curve::angle, 0.21729, float2(0.18292683, 0.32926829),
  float2(0.18292683, 0.26829268), 0.06098, 3.14159, 1.5708),
```

Constructor for a `pw_curve::pw_point` that creates a 90° arc

```
pw_curve::pw_point( // .375 0
  pw_curve::segment, 0.29907, float2(0.36585366, 0.32926829)),
pw_curve::pw_point( // .125 -.125 :angle -90
  pw_curve::angle, 0.34189, float2(0.42682927, 0.26829268),
  float2(0.36585366, 0.26829268), 0.06098, 1.5708, 0.0),
pw_curve::pw_point( // .125 0
  pw_curve::segment, 0.36915, float2(0.48780488, 0.26829268)),
pw_curve::pw_point( // 0 1
  pw_curve::segment, 0.58723, float2(0.48780488, 0.75609756)),
pw_curve::pw_point( // .25 .25 :angle -90
  pw_curve::angle, 0.67287, float2(0.6097561, 0.87804878),
  float2(0.6097561, 0.75609756), 0.12195, 3.14159, 1.5708),
pw_curve::pw_point( // 0 .25
  pw_curve::segment, 0.7274, float2(0.6097561, 1.0)),
pw_curve::pw_point( // a0 0
  pw_curve::segment, 1.0, float2(0.0, 1.0)));
```

The function `point_along_pwc` searches for the piece that contains the `curve_position`. If `curve_position` is less than the parametric value `t` of a point, then the desired point is in that piece.

Listing 15.5

```
float2 point_along_pwc(
    float curve_position,
```

Position on the piece-wise curve: 0.0 → 1.0

```
    pw_curve::pw_point[<N>] pwc)
```

Size-deferred array of N pw_curve:pw_point instances

```
{
    float2 result = pwc[N-1].pt;
```

All but the last pw-point will be examined, so the last point is the default

```
    for (int i = 1; i < N; i++) {
```

Start at 1 so that pwc[i-1] and pwc[i] define the current piece

```
        pw_curve::pw_point current = pwc[i];
```

Current point in the pwc-list

```
        if (curve_position < current.t) {
```

If the position on the curve is less than the pw-point's parameter value, the desired point is in this piece

```
            pw_curve::pw_point last = pwc[i-1];
```

The previous point that will define the piece along with the current point

```
            debug::assert(
               (current.t - last.t) != 0, "duplicated t value");
            float piece_position =
                (curve_position - last.t) / (current.t - last.t);
```

Calculate position in the current piece, but check that the denominator is not zero.

```
          if (current.type == pw_curve::segment) {
            result = math::lerp(
              last.pt, current.pt, piece_position);
```

If the piece is a line segment, use linear interpolation

```
          } else {
            float angle =
               math::lerp(
                  current.start_angle, current.end_angle,
                  piece_position);
            result = current.center + current.radius *
              float2(math::cos(angle), math::sin(angle));
```

If the piece is an arc, use linear interpolation for the angle, then calculate the point

```
          }
          break;
```

The piece containing the point has been found; exit from the loop

```
      }
    }
    return result;
}
```

Once the point on the pw-curve has been found, the function `displace_to_point` uses the surface normal and the specified tangent vector in texture space to construct the displacement vector.

Listing 15.6

```
float3 displace_to_point(
  float2 pw_point, float uv_position, float3 tangent)
{
  float3 result =
```

The x component scales the surface normal vector

```
    pw_point.x * state::normal();
  result +=
    (pw_point.y - uv_position) * tangent;
```

The offset of the y component from the current uv position scales the tangent vector

```
  return result;
}
```

Until function `displace_to_point`, it has not been necessary to specify *which* tangent will be used to construct the displacement vector. A natural choice in modeling the vertical structures of *Palladio Londinensis* would be to use the tangent vector in the v direction, given the canonical texture space of a square polygon. However, a pw-curve can just as easily be used to displace a surface in the u direction. To parameterize this choice, the `displacement_direction enum` provides a clearer argument value than, say, a Boolean variable in which v displacements are expressed as "not in the u direction."

Listing 15.7

```
enum displacement_direction {
  along_u,
  along_v
};
```

Function `displace_with_pwc` determines a *uv* value from the pw-curve. This value, together with the corresponding tangent vector, is used to calculate the resulting displacement vector.

Listing 15.8

```
float3 displace_with_pwc(
  uniform pw_curve::pw_point[<N>] pwc,
  displacement_direction displace_dir)
{
  int uv_index = displace_dir == along_u ? 0 : 1;
```

Index for the u or v coordinate

```
  float uv_pos = state::texture_coordinate(0)[uv_index];
```

Position in uv texture space

```
  float3 tangent =
    displace_dir == along_u
    ? state::texture_tangent_u(0)
    : state::texture_tangent_v(0);
```

Select the tangent vector to be scaled by the x component of the piece-wise curve

```
  float2 pw_point =
    point_along_pwc(uv_pos, pwc);
```

Point on the piece-wise curve for normal vector modification

```
  return
    displace_to_point(pw_point, uv_pos, tangent);
```

Return the modified normal based on the curve point

```
}
```

The function `displace_with_pwc` provides the transformation from the MDL array (created from the pwc-list) to the displacement vector for use in the materials developed in the next section.

15.6 Using piece-wise curves in materials

Section 1.4 (page 277) created piece-wise curves based on measurements from the *Palladio Londinensis*. After translating the pwc-list into an MDL array, Section 1.5 (page 281) defined a series of functions to derive vector displacements from those arrays. This section uses those arrays to create vector displacements in a material.

Material `grid_ornament_1` use the material `vector_displacement`, developed in "A template material for vector displacement" (page 243). The first parameter is the point's displacement vector, defined by function `displace_with_pwc`:

Listing 15.9

```
material grid_ornament_1(
  float grid_count=50) =
let {
  float normalized_grid_count =
    grid_count *
    ornament_1::ornament_1_length;
```

Use the calculated length of the ornament_1 curve to normalize the texture space

```
} in
geometry::vector_displacement(
```

```
    displace_with_pwc(
      ornament_1::ornament_1, along_v),
  geometry::grid(
    float2(grid_count, normalized_grid_count)));
```

Create displacement vector from array ornament_1

The length of the pw-curve scales the texture in v using the length calculated during the construction of the MDL array of pw-points.

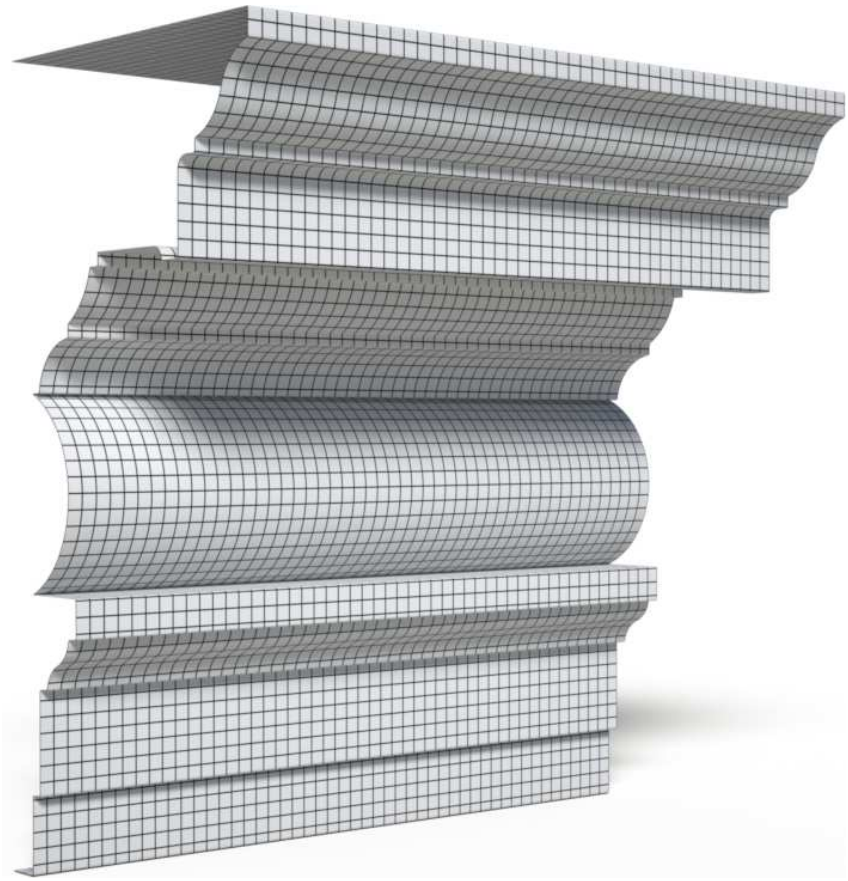


Fig. 15.21 – The first ornament rendered with the material `pwc_grid`

Figure 15.21 demonstrates that the texture scaling remains consistent in the u and v directions, enabling other texture-based materials to be used for the surface property. The `material_surface` property of Listing 15.10 is almost the simplest material that uses a texture for the diffuse reflection value; only the texture scaling has been parameterized.

Listing 15.10

```
material travertine(
  float texture_u_scale = 1.0,
  float texture_v_scale = 1.0) =
material (
  surface: material_surface (
```

```
    scattering: df::diffuse_reflection_bsdf (
        tint: tex::lookup_color(
          texture_2d("travertine.png"),
          float2(texture_u_scale,
                texture_v_scale)))));
```

Texture map used for diffuse color

Material `stone_ornament_2` uses the material `travertine` in combination with the array for the second ornament in Plate XXIX of *Palladio Londinensis*.

Listing 15.11

```
material stone_ornament_2() =
  geometry::vector_displacement(
    displace_with_pwc(
      ornament_2::ornament_2, along_v),
    travertine(1.0, ornament_2::ornament_2_length));
```

Fig. 15.22 – The second ornament rendered with a stone texture map in material `travertine`

15.7 Radial displacment

In the previous sections, pwc-lists described a three-dimensional shape from a two-dimensional curve translated in single direction. For more complex shapes—column capitals, for example—the pwc-list can be rotated around a central axis.

Function `y_rotation_3` creates a 3x3 matrix that defines rotation around the y axis.

Listing 15.12

```
float3x3 y_rotation_3(float angle) {
  return float3x3(
    math::cos(angle), 0.0, math::sin(angle),
    0.0, 1.0, 0.0,
    -math::sin(angle), 0.0, math::cos(angle));
}
```

Function `displace_to_angular_point` maps the parameter `t`, which will be a uv coordinate value $0.0 \rightarrow 1.0$, to $0^\circ \rightarrow 360^\circ$.

Listing 15.13

```
float3 displace_to_angular_point(
  float2 pw_point, float t, float radius = 0)
{
  float angle = t * math::PI * 2;
```

Map $0.0 \rightarrow 1.0$ to $0^\circ \rightarrow 360^\circ$

```
  float3 p = float3(
    pw_point[0] + radius,
    pw_point[1],
    0.0);
```

Construct a three-dimensional point from the pw-point

```
  p *= y_rotation_3(angle);
```

Multiple the point by the rotation matrix

```
  p -= state::position();
```

Compensate for the point's original position

```
  return p;
}
```

Both u and v coordinates are used in `pwc_angular_displace`: one determines the amount of rotation, the other determines the point on the piece-wise curve to be rotated.

Listing 15.14

```
float3 pwc_angular_displace(
  uniform pw_curve::pw_point[<N>] pwc,
  displacement_direction displace_dir)
{
  int position_index =
    displace_dir == along_u ? 0 : 1;
```

uv coordinate index for position along pwc

```
  int rotation_index =
    displace_dir == along_u ? 1 : 0;
```

uv coordinate index for amount of rotation

```
  float position =
    state::texture_coordinate(0)[position_index];
```

Position in uv texture space

```
    float rotation =
      state::texture_coordinate(0)[rotation_index];
```
Rotation factor ($0 \rightarrow 1$)

```
    float2 pw_point =
      point_along_pwc(position, pwc);
```
Point on the piece-wise curve for normal vector modification

```
    float3 result =
      displace_to_angular_point(
        pw_point, rotation);
```
Result is rotation of pwc point based on the value of the other uv coordinate

```
  return result;
}
```

Material `radial_pwc` parameterizes the surface material to be used with the displaced object geometry.

Listing 15.15

```
material radial_pwc(
  uniform pw_curve::pw_point[<N>] pwc,
  material surface_material,
  displacement_direction displace_dir=along_v) =
geometry::vector_displacement(
  pwc_angular_displace(pwc, displace_dir),
  surface_material);
```

Material `corinthian_capital_grid` uses the piece-wise curve constructed for the Corinthian impost, with a grid size of 50 in both u and v directions.

Listing 15.16

```
material corinthian_capital_grid()
= radial_pwc(
  corinthian::corinthian,
  geometry::grid(float2(50.0)));
```

Figure 15.23 (page 290) uses material `corinthian_capital_grid`. Like the previous renderings, the initial geometry is a simple square, the radial form constructed solely by the displacement vectors calculated from the piece-wise curve.

Fig. 15.23 – Using function `pwc_angular_displace` *to construct a solid of revolution*

15.8 Three-dimensional texture mapping

Figure 15.23 does not compensate for the different scaling in the u and v directions; unlike the ornaments, there is no obvious way to extend a two-dimensional texture map to such a geometric transformation. However, *three-dimensional texture mapping* can create a consistent surface appearance as surface points map to a pattern defined in three dimensions.

For example, material `marbloid` uses the Perlin noise function from the "Noise" (page 179) chapter to create a mapping from a surface point's three-dimensional position to a value in three-dimensional noise space.

Listing 15.17

```
material marbloid (
  float scale = 1.0,
  int level_count = 5,
  functions::mapping_mode space = functions::object) =
let {
  double3 scaled_point =
    scale * functions::position(space);
  double noise_value =
    perlin_noise::summed_perlin_noise(
    scaled_point, level_count,
    level_scale: 0.8, turbulence: true);
```

Create a noise value based on position of current point

```
  double z = 2 * scaled_point.z;
  double veins = 0.03 +
    0.3 * (1 + math::sin(z + 8.0 * noise_value));
```

Repeat the noise pattern for vein-like structure


```
    color tint(
      float(math::pow(veins, 1/1.8)),
      float(math::pow(veins, 1/1.4)),
      float(math::pow(veins, 1/1.1) * 0.9));
} in
  material (
    surface: material_surface (
      scattering: df::diffuse_reflection_bsdf(
        tint: tint)));
```

Modify the noise value differently for the RGB values to create a non-grayscale value

The `marbloid` material provides the surface property for the generic `radial_pwc` material.

Listing 15.18

```
material corinthian_capital_stone()
= radial_pwc(
  corinthian::corinthian,
  marbloid(scale: 12));
```

Figure 15.24 uses material `corinthian_capital_stone`. The structure of the pattern, though based on the random results of the noise function, feels consistent throughout, a result of the mapping from the noise function to spatial position rather than to the distorted *uv* texture space of the surface.

Fig. 15.24 – Using three-dimensional texture mapping material `marbloid`

15.9 Displacement with two piece-wise curves

Rather than rotating a piece-wise curve around an axis, a second piece-wise curve can define the form that the first curve should follow.

Listing 15.19

```
float3 displace_to_pwc_point(
  float2 pw_point, float t,
  uniform pw_curve::pw_point[<N>] pwc,
  float radius_offset = 0.0)
{
  float angle = t * math::PI * 2;
  float3 p = float3(pw_point[0] + radius_offset,
                pw_point[1],
                0.0);
  float2 square_pt = point_along_pwc(t, pwc);
  float scale = math::length(square_pt);
  p.x *= scale;
  p *= y_rotation_3(angle);
  p -= state::position();
  return p;
}
```

Material `double_pwc` uses two piece-wise curves, the arguments `pwc_profile` and `pwc_sweep`.

Listing 15.20

```
material double_pwc(
  uniform pw_curve::pw_point[<N>] pwc_profile,
  uniform pw_curve::pw_point[<M>] pwc_sweep,
  material surface_material,
  float radius_offset = 0.0) =
let {
  float u = state::texture_coordinate(0)[0];
  float v = state::texture_coordinate(0)[1];
  float2 profile_point = point_along_pwc(
    v, pwc_profile);
  float3 displace_point = displace_to_pwc_point(
    profile_point, u, pwc_sweep, radius_offset);
} in
geometry::vector_displacement(
  displace_point,
  surface_material);
```

The *Palladio Londinensis* includes diagrams for the construction of *column fluting*, vertical groves that provide more visual detail to columns.

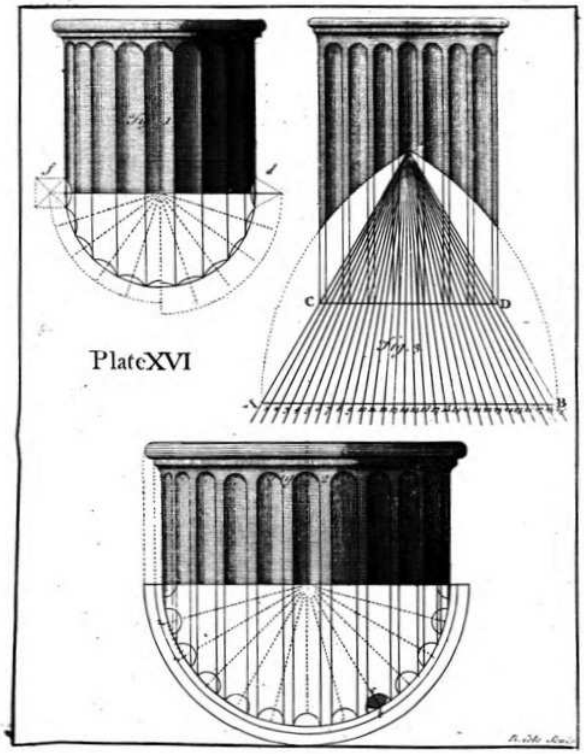


Fig. 15.25 – Construction methods for column fluting from Palladio Londinensis

The traditional Doric fluting geometry can be defined as a piece-wise curve of twenty arcs. By constructing a square using the distance between flute edges as one side, the flute arc can be constructed from the circle with a center in the center of the constructed square.

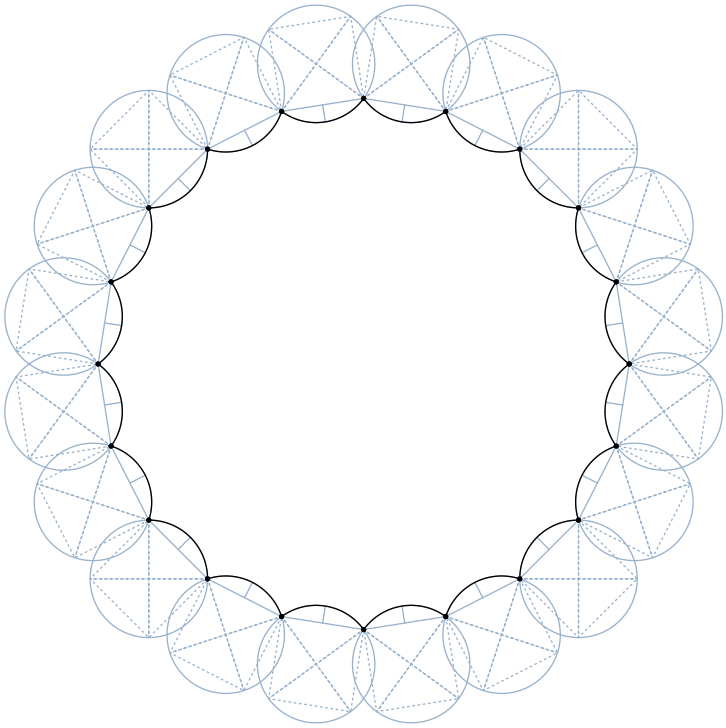


Fig. 15.26 – Traditional number and proportions of Doric column flutes implemented with the `:sag` *property*

The material `doric_matte_stone` uses the MDL array constructed from the Doric piece-wise curve to define the positioning of the Corinthian capital of the previous section.

Listing 15.21

```
material doric_matte_stone()
= double_pwc(
   corinthian::corinthian,
   pwc_examples::doric_flutes,
   diffuse_reflection::diffuse_reflection(
      tint: pink_stone),
   radius_offset: 0.25);
```

The color of the diffuse_reflection is defined by a `const color` variable, `pink_stone`:

```
const color pink_stone = color(.7, .66, .63);
```

Figure 15.27 uses material `doric_matte_stone`.

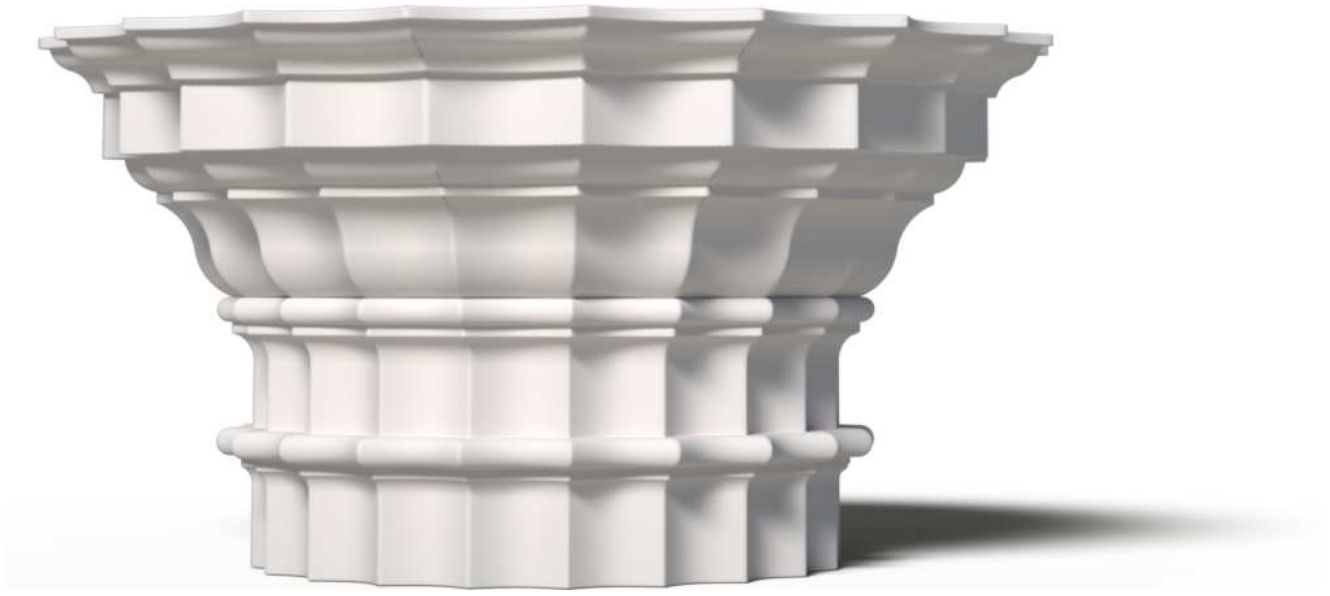


Figure 15.27

In contrast to doric fluting, traditional iconic fluting consists of twenty-four semicircular arcs, separated by a distance that is one-third the width of the flutes.

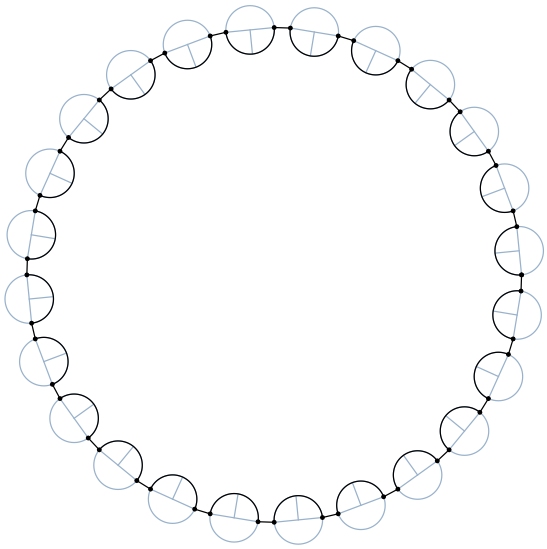


Fig. 15.28 – Ionic column flutes are separated by a straight line, one-third the width of the flute

Material `ionic_matte_stone` is the same in structure to `doric_matte_stone`, but uses the array defined by the piece-wise curve for the ionic fluting shape.

Listing 15.22

```
material ionic_matte_stone()
= double_pwc(
   corinthian::corinthian,
   pwc_examples::ionic_flutes,
   diffuse_reflection::diffuse_reflection(
      tint: pink_stone),
   radius_offset: 0.25);
```

Figure 15.29 uses material `ionic_matte_stone`.

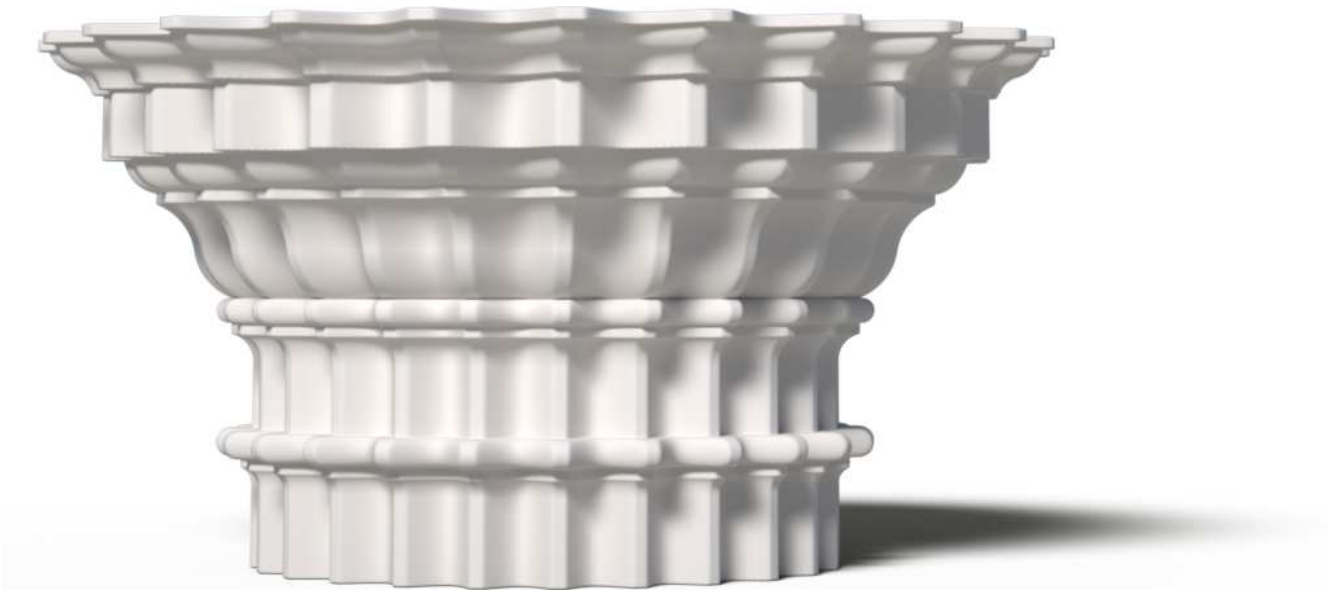


Figure 15.29

15.10 The procedural impulse

Providing a cross-platform description of surface appearance is an important design goal of MDL. The pwc-list extends the idea of portability to the data used as rendering parameters. A program written in a scripting language converted the pwc-lists—created in a text editor—to MDL arrays. Before rendering, the script calculates all the necessary data for point coordinates, angle descriptions and the parametric value for each pw-point. A generic data description like the pwc-list could also be created within a graphical application, providing an interactive display of the curve as it is developed, saved by the application in a generic format for use in the various rendering systems supported by the application.

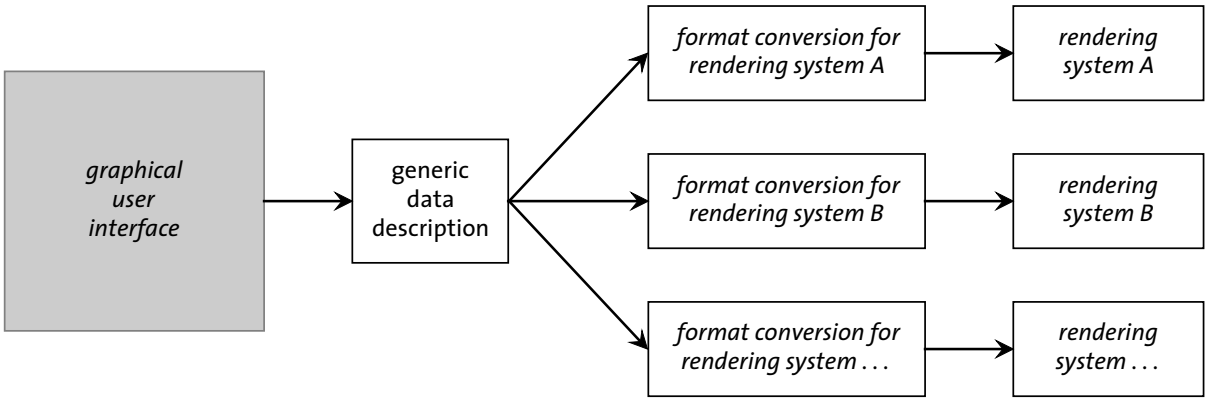


Fig. 15.30 – A generic data representation can be optimized for different rendering systems

But reusable templates in a generic format are hardly a digital invention. In Figure 15.31 (page 297), a portrait of an architect from the sixteenth century, the setting is surprisingly sparse. The architect's gaze directs us to the only tools of the trade that are visible, hanging on nails: a straightedge and two molding templates.

Fig. 15.31 – Bildnis eines Architekten *(Picture of an Architect), attributed to Ludger tom Ring der Älter, 1496–1547. Gemäldegalerie, Berlin*

Fig. 15.32 – Molding templates in Bildnis eines Architekten

Part 6 Reference

16 Terminology and syntax

16.1 Inputs

16.1.1 Parameters and arguments

Example parameters	*Corresponding arguments*
`color tint = color(1.0),` `float roughness = 0.0`	`tint: color(0.7),` `roughness: 0.1`

A *parameter* is a name that defines an input of a specified type to a struct or function. An *argument* is a datum of the type required by the parameter used in the struct or function. The ordered set of parameters of a struct or function is called its *signature*.

16.2 Instantiable types

A *data type* is described by a *constructor function*, or simply, a *constructor*. Calling the constructor function of a data type creates an *instance* of the data type. This process is also called *instancing* the data type.

16.2.1 Distribution function

Type declaration	*An instance of the type*

`bsdf diffuse_reflection_bsdf (`
`  color tint = color(1.0),`
`  float roughness = 0.0`
`);`
	`df::diffuse_reflection_bsdf (`
	`  tint: color(0.7),`
	`  roughness: 0.1)`

A *distribution function* describes the interaction of light with objects. MDL provides three distribution function types: the *bidirectional scattering distribution function* (or *BSDF*), the *emission distribution function* (or *EDF*) and the *volume distribution function* (or *VDF*).

Surface: BSDF	*Light: EDF*	*Volume: VDF*
`diffuse_reflection_bsdf`	`diffuse_edf`	`anisotropic_vdf`
`diffuse_transmission_bsdf`	`spot_edf`	
`specular_bsdf`	`measured_edf`	
`simple_glossy_bsdf`		
`backscattering_glossy_reflection_bsdf`		
`measured_bsdf`		

16.2.2 Material property struct

Type declaration

```
struct material_surface {
   bsdf scattering = bsdf();
   material_emission emission = material_emission();
};
```

An instance of the type

```
material_surface (
   scattering:
    df::diffuse_reflection_bsdf (
      tint: color(0.7),
      roughness: 0.1))
```

Each *fundamental appearance property* of MDL is defined by a *material property struct* type. These types contain *distribution function instances*.

Material property structs

```
material_surface
material_emission
material_volume
material_geometry
```

16.2.3 Material struct

Type declaration

```
struct material {
   uniform bool thin_walled = false;
   material_surface surface = material_surface();
   material_surface backface = material_surface();
   uniform color ior = color(1.0);
   material_volume volume = material_volume();
   material_geometry geometry = material_geometry();
};
```

An instance of the type

```
material (
   surface: material_surface (
      scattering:
       df::diffuse_reflection_bsdf (
         tint: color(0.7),
         roughness: 0.1)))
```

The *material struct* is the fundamental data type of MDL. A *material struct instance* provides a complete specification for a rendering system to calculate surface appearance. A material struct instance contains instances of the surface, volume and geometry properties; the emission property is a field of the surface property. A material struct instance also defines the interpretation of the surface as a boundary or volume enclosure (the *thin_walled* field) and the index of refraction of the object (the *ior* field).

16.3 Modifying and combining distribution functions

16.3.1 Distribution function modifiers

Type declaration

```
bsdf tint (
   color tint,
   bsdf base);
```

An instance of the type

```
tint (
   tint: color(0.5),
   base: df::diffuse_reflection_bsdf (
      tint: color(0.7),
      roughness: 0.1));
```

A *BSDF modifier* creates a new BSDF instance from an existing BSDF instance and other parameters.

BSDF modifiers

```
tint
thin_film
directional_factor
measured_curve_factor
```

16.3.2 Mixing distribution functions

Type declaration

```
bsdf normalized_mix (
   bsdf_component[<N>] components);
```

An instance of the type

```
df::normalized_mix (
   components: df::bsdf_component[] (
      bsdf_component (
         weight: 0.5,
         base: df::diffuse_reflection_bsdf (
            tint: color(0.7),
            roughness: 0.1)),
      bsdf_component (
         weight: 0.5,
         base: df::specular_bsdf (
            tint: color(0.8,0.8,0.2))));
```

A *distribution function mixer* combines two or more *mixing components* to produce a new distribution function instance. Total values larger than one are adjusted either by *component normalization* or *component clamping*.

Mixing functions

```
bsdf normalized_mix
edf normalized_mix
vdf normalized_mix
bsdf clamped_mix
edf clamped_mix
vdf clamped_mix
```

16.3.3 Components for mixing distribution functions

Type declaration

```
struct bsdf_component (
   float weight = 0.0,
   bsdf component = bsdf())
```

An instance of the type

```
bsdf_component (
   weight: 0.5,
   base: df::diffuse_reflection_bsdf (
      tint: color(0.7),
      roughness: 0.1));
```

Each component specfifies a *mixing weight* used in calculating the result of the distribution function mixer.

Component types

```
bsdf_component
edf_component
vdf_component
```

16.3.4 Layering distribution functions

Type declaration

```
bsdf weighted_layer (
   float weight = 1.0,
   bsdf layer = bsdf(),
   bsdf base = bsdf(),
   float3 normal = state::normal());
```

An instance of the type

```
weighted_layer (
   weight: 0.1,
   layer: df::specular_bsdf (
      tint: color(0.8,0.8,0.2)),
   base: df::diffuse_reflection_bsdf (
      tint: color(0.7),
      roughness: 0.1))
```

A *layering function* combines two distribution function instances based on a weight to produce a new distribution function instance. The weighting value may be directionally dependent.

Layering functions

```
weighted_layer
fresnel_layer
custom_curve_layer
measured_curve_layer
```

16.4 Material definitions

16.4.1 Creating a material definition with parameters

Syntax

`material` *name* `(` *parameters* `) =`
material-struct-instance

Material creation

```
material diffuse (
   color tint = color(1.0)) =
material (
   surface: material_surface (
      scattering: df::diffuse_reflection_bsdf (
         tint: tint,
         roughness: 1.0)));
```

A *material definition* encapsulates a material instance, defining a name and providing *material parameters* in a form that that resembles a function definition. The values provided to parameters in the encapsulated material instance are called *interior arguments*. If no parameters are defined, then the material definition is *fully encapsulated*. If some, but not all, of the values of the interior arguments are supplied by material definition's parameters, the material definition is said to be *partially encapsulated*. If the value of all interior arguments are provided by the parameters of the material definition, the material definition is said to be *fully parameterized*. The example above, the material definition named `diffuse` is partially encapsulated: the material definition's `tint` parameter is used as an argument to the encapsulated material instance; the `roughness` parameter of the material instance is the constant value `1.0`.

16.4.2 Reuse of an existing material definition

Syntax

`material` *name* `(` *parameters* `) =`
material-definition-instance

Material creation

```
material blue_plaster() =
diffuse (
   tint: color(0.4,0.5,0.9),
   roughness: 0.8);
```


A material definition can provide the behavior for a new material definition through *material definition reuse*. The new definition can define a new signature for the reused material definition through *reparameterization*. In the example above, the new definition has been fully encapsulated; no parameters in the enclosed definition are available in the new signature.

16.4.3 Material definition reuse with duplicated parameters

Syntax

```
material name (*) =
 material-definition-instance
```

Material creation

```
material plaster(*) =
diffuse (
   tint: color(0.7,0.7,0.7),
   roughness: 1.0);
```

By defininining the reuse of a material definition with a *starred signature,* all the parameters of the enclosed material are available in the outer signature.

16.5 Material programming techniques

16.5.1 Material components as arguments

Syntax

```
material name ( parameters-including-material ) =
 material-definition-instance-using-material ;
```

Material creation

```
material add_clear_coat (
   material base,
   color ior = color(1.5)) =
material (
   volume: base.volume,
   geometry: base.geometry,
   surface: material_surface (
      emission: base.surface.emission,
      scattering: fresnel_layer (
         layer: specular_bsdf(),
         base: base.surface.scattering,
         ior: ior)));
```

A material definition can include one or more *material definition parameters* in its signature. The components of the material definition can be selected for use as interior arguments with *dot notation*.

16.5.2 Let-expressions

Syntax

```
material name ( parameters ) =
let {
    variable-declarations
} in
 material-instance ;
```

Material creation

```
material add_clear_coat (
   material base,
   color ior = color(1.5))
=
let {
   bsdf coat = specular_bsdf();
   bsdf coated_scattering = fresnel_layer (
      layer: coat,
      base: base.surface.scattering,
      ior: ior);
   material_surface coated_surface = material_surface (
      emission: base.surface.emission,
      scattering: coated_scattering);
} in
material (
   volume: base.volume,
   geometry: base.geometry,
   surface: coated_surface);
```

A *let-expression* defines a set of *local variables* that can be used as argument values for the material instance.

16.5.3 Conditional material expressions

Syntax

```
material name ( parameters ) =
 boolean-condition
   ? material-1
   : material-2 ;
```

Material creation

```
material level_of_detail (
   uniform bool high_resolution,
   material high_res_material,
   material low_res_material) =
high_resolution ? high_res_material : low_res_material;
```

A *conditional expression* selects between two values based on the Boolean value of the *conditional predicate*.

Epilog

The establishment of the fine arts and their division into various categories go back to a time that differed radically from ours and to people whose power over things and circumstances was minute in comparison to our own. However, the astounding growth that our resources have undergone in terms of their precision and adaptability will in the near future confront us with very radical changes indeed in the ancient industry of the beautiful. In all arts there is a physical component that cannot continue to be considered and treated in the same way as before; no longer can it escape the effects of modern knowledge and modern practice. Neither matter nor space nor time is what, up until twenty years ago, it always was. We must be prepared for such profound changes to alter the entire technological aspect of the arts, influencing invention itself as a result, and eventually, it may be, contriving to alter the very concept of art in the most magical fashion.

Paul Valéry, *Pièces sur l'art*, 1934. Quoted in Walter Benjamin, *The Work of Art in the Age of Mechanical Reproduction*, 1936, translated by J. A. Underwood.

Most people, if they pause to think about it, would probably accept that the business of material production has loomed so large in human history, absorbed such boundless resources of time and energy, provoked such internecine conflicts, engrossed so many human beings from cradle to grave and confronted so many of them as a matter of life or death, that it would be amazing if it were not to leave its mark on a good many other aspects of our existence. Other social institutions find themselves inexorably dragged into its orbit. It bends politics, law, culture and ideas out of true by demanding that rather than just flourish as themselves, they spend much of their time legitimating the prevailing social order. Think of contemporary capitalism, in which the commodity form has left is grubby thumbprints on everything from sport to sexuality, from how best to swing oneself a front-row seat in heaven to the ear-shattering tones in which U.S. television reporters hope to seize the viewer's attention for the sake of the advertisers. The most compelling confirmation of Marx's theory of history is late capitalist society. There is a sense in which his case is becoming truer as time passes. It is capitalism, not Marxism, which is economically reductionist. It is capitalism which believes in production for production's sake, in the narrower sense of the word "production."

Terry Eagleton, *Why Marx Was Right*, 2011.

All my life I have returned to this same question time and again: What is the need—that is, what is the truly objective constellation of forces working in us and the world—that justifies the creation of something like art? This question has certainly had a central place in my life, and has led me to distance myself from my initial involvement in the scientific field. Before I made this shift, provoked by this question, this search for answers, I began my studies in the natural sciences, and experienced certain things about the prevailing scientific paradigm, which made me realize that an answer would not be found here. In questioning the value of this kind of research, or alternatively, as a means of exploring the overall field of existing forces—including life forces, the forces of the mind, that is of the soul, psyche-spirit and their higher forms—I was compelled to consider, on purely experimental grounds, whether I should explore the sphere of art, that has manifested through time as a form of cultural activity.

Joseph Beuys, *What is Art?*, English translation, 2004; originally published in German as *Was ist Kunst? Werkstattgespräch mit Beuys*, 1986.

in memory, Lance Williams

424. Das Bild ist *da*; und ich bestreite seine *Richtigkeit* nicht. Aber *was* ist seine Anwendung? Denke an das Bild der Blindheit als einer Dunkelheit in der Seele oder im Kopf des Blinden.

424. The picture is there; and I do not dispute its *correctness*. But *what* is its application? Think of the picture of blindness as a darkness in the mind or in the head of a blind person.

Philosophische Untersuchungen, Ludwig Wittgenstein


NVIDIA