9 Illumination Models and Lights

In the past two chapters we have glossed over exactly how shaders respond to light, as well as how light sources themselves operate, which of course are important aspects of the overall appearance of materials. These are the issues that will be addressed by this chapter. In doing so, we will build up a variety of library routines that implement different material appearances.

9.1 Built-in Local Illumination Models

In earlier chapters, all of our shaders have ended with the same three lines:

```
Ci = Ct * (Ka*ambient() + Kd*diffuse(Nf)) +
     specularcolor * Ks*specular(Nf,-normalize(I),roughness);
Oi = Os;  Ci *= Oi;
```

Three functions are used here that have not been previously described:

`color diffuse(vector N)`

Calculates light widely and uniformly scattered as it bounces from a light source off of the surface. Diffuse reflectivity is generally approximated by Lambert's law:

$$\sum_{i=1}^{\text{nlights}} Cl_i \max(0, N \cdot L_i)$$

where for each of the i light sources, L_i is the unit vector pointing toward the light, Cl_i is the light color, and N is the unit normal of the surface. The max function ensures that lights with $N \cdot L_i < 0$ (i.e., those *behind* the surface) do not contribute to the calculation.

`color specular(vector N, V; float roughness)`

Computes so-called *specular* lighting, which refers to the way that glossier surfaces have noticeable bright spots or highlights resulting from the narrower (in angle) scattering of light off the surface. A typical formula for such scattering might be the Blinn-Phong model:

$$\sum_{i=1}^{\text{nlights}} Cl_i \max(0, N \cdot H)^{1/\text{roughness}}$$

where H is the vector halfway between the viewing direction and the direction of the light source (i.e., `normalize(normalize(-I)+normalize(L))`). The equation above is for the Blinn-Phong reflection model, which is what is dictated by the *RenderMan Interface Specification*. *PRMan* actually uses a slightly different, proprietary formula for `specular()`. *BMRT* also uses a slightly nonstandard formulation of `specular()` in order to more closely match *PRMan*. So beware—though the spec dictates Blinn-Phong, individual implementations can and do substitute other reflection models for `specular()`.

`color ambient()`

Returns the contribution of so-called *ambient* light, which comes from no specific location but rather represents the low level of scattered light in a scene after bouncing from object to object.[1]

[1] In most renderers, ambient light is typically approximated by a low-level, constant, nondirectional light contribution set by the user in a rather ad hoc manner. When renderers try to accurately calculate this interreflected light in a principled manner, it is known as *global illumination*, or, depending on the exact method used, as *radiosity*, *path tracing*, *Monte Carlo integration*, and others.

Listing 9.1 MaterialPlastic computes a local illumination model approximating the appearance of ordinary plastic.

```
/* Compute the color of the surface using a simple plastic-like BRDF.
 * Typical values are Ka=1, Kd=0.8, Ks=0.5, roughness=0.1.
 */
color MaterialPlastic (normal Nf;  color basecolor;
                       float Ka, Kd, Ks, roughness;)
{
    extern vector I;
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf))
           + Ks*specular(Nf,-normalize(I),roughness);
}
```

Therefore, those three lines we had at the end of our shaders calculate a weighted sum of ambient, diffuse, and specular lighting components. Typically, the diffuse and ambient light is filtered by the base color of the object, but the specular contribution is not (or is filtered by a separate `specularcolor`).

We usually assign `Oi`, the opacity of the surface, to simply be the default surface opacity `Os`. Finally, we scale the output color by the output opacity, because RenderMan requires shaders to compute premultiplied opacity values.

When `specularcolor` is 1 (i.e., white), these calculations yield an appearance closely resembling plastic. Let us then formalize it with the Shading Language function `MaterialPlastic` (in Listing 9.1). With this function in our library, we could replace the usual ending lines of our shader with:

```
Ci = MaterialPlastic (Nf, V, Cs, Ka, Kd, Ks, roughness);
Oi = Os;  Ci *= Oi;
```

For the remainder of this chapter, functions that compute completed material appearances will be named with the prefix `Material`, followed by a description of the material family. Arguments passed will generally include a base color, surface normal, viewing direction, and a variety of weights and other knobs that select individual appearances from the family of materials.

The implementation of the `Material` functions will typically be to compute a weighted sum of several *primitive local illumination functions*. Before long, it will be necessary to move beyond `ambient()`, `diffuse()`, and `specular()` to other, more sophisticated local illumination functions. When we start writing our own local illumination functions, we will use the convention of naming them with the prefix `LocIllum` and will typically name them after their inventors (e.g., `LocIllumCook-Torrance`).

But first, let us see what effects we can get with just the built-in `specular()` and `diffuse()` functions.

Listing 9.2 `MaterialMatte` computes the color of the surface using a simple Lambertian BRDF.

```
color MaterialMatte (normal Nf;  color basecolor;  float Ka, Kd;)
{
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf));
}
```

9.1.1 Matte Surfaces

The typical combination of the built-in `diffuse()` and `specular()` functions, formalized in `MaterialPlastic()`, is great at making materials that look like manufactured plastic (such as toys). But many objects that you model will not be made from materials that feature a prominent specular highlight. You could, of course, simply call `MaterialPlastic()` passing `Ks` = 0. However, that seems wasteful to call the potentially expensive `specular()` function only to multiply it by zero. Our solution is to create a separate, simpler `MaterialMatte` that only calculates the ambient and Lambertian (via `diffuse()`) contributions without a specular highlight, as shown in Listing 9.2.

9.1.2 Rough Metallic Surfaces

Another class of surfaces not well modeled by the `MaterialPlastic` function is that of metals. Deferring until the next section metals that are polished to the point that they have visible reflections of surrounding objects, we will concentrate for now on roughened metallic surfaces without coherent reflections.

Cook and Torrance (1981,1982) realized that an important difference between plastics and metals is the effect of the base color of the material on the specular component. Many materials, including paint and colored plastic, are composed of a transparent substrate with embedded pigment particles (see Figure 9.1). The outer, clear surface boundary both reflects light specularly (without affecting its color) and transmits light into the media that is permeated by pigment deposits. Some of the transmitted light is scattered back diffusely after being filtered by the pigment color. This white highlight contributes greatly to the perception of the material as plastic.

Homogeneous materials, including metals, lack a transparent outer layer that would specularly reflect light without attenuating its color. Therefore, in these materials, all reflected light (including specular) is scaled by the material's base color. This largely contributes to the metallic appearance. We implement this look in `MaterialRoughMetal` (Listing 9.3).

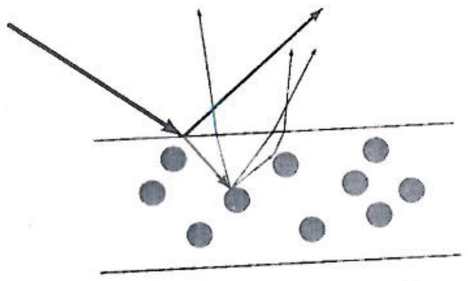


Figure 9.1 Cross section of a plastic-like surface.

Listing 9.3 `MaterialRoughMetal` calculates the color of the surface using a simple metal-like BRDF.

```
/* Compute the color of the surface using a simple metal-like BRDF.
 * To give a metallic appearance, both diffuse and specular
 * components are scaled by the color of the metal.  It is
 * recommended that Kd < 0.1, Ks > 0.5, and roughness > 0.15
 * to give a believable metallic appearance.
 */
color MaterialRoughMetal (normal Nf;  color basecolor;
                          float Ka, Kd, Ks, roughness;)
{
    extern vector I;
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf) +
                        Ks*specular(Nf,-normalize(I),roughness));
}
```

9.1.3 Backlighting

So far, the materials we have simulated have all been assumed to be of substantial thickness. In other words, lights on the same side of the surface as the viewer reflect off the surface and are seen by the camera, but lights "behind" the surface (from the point of view of the camera) do not scatter light around the corner so that it contributes to the camera's view.

It is not by accident that such backlighting is excluded from contributing. Note that the `diffuse()` function takes the surface normal as an argument. The details of its working will be revealed in Section 9.3; let us simply note here that this normal parameter is used to exclude the contribution of light sources that do not lie on the same side of the surface as the viewer.

But thinner materials--paper, lampshades, blades of grass, thin sheets of plastic—do have appearances that are affected by lights behind the object. These

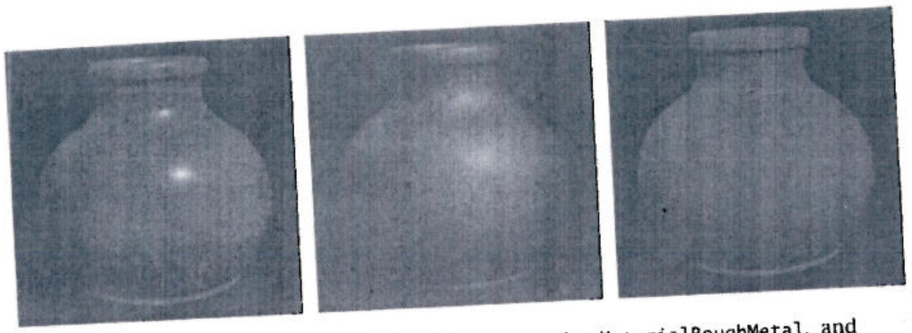


Figure 9.2 The finishes (left to right) `MaterialPlastic`, `MaterialRoughMetal`, and `MaterialMatte` applied to a vase. See also color plate 9.2.

objects are translucent, so lights shine *through* the object, albeit usually at a lower intensity than the reflections of the lights in front of the object.[2] Therefore, since

```
diffuse(Nf)
```

sums the Lambertian scattering of lights on the viewer's side of the surface, then the lights from the *back side* should be described by

```
diffuse(-Nf)
```

In fact, this works exactly as we might hope. Thus, making a material translucent is as easy as adding an additional contribution of `diffuse()` oriented in the backwards direction (and presumably with a different, and smaller, weight denoted by `Kt`). This is exemplified by the `MaterialThinPlastic` function of Listing 9.4.

9.2 Reflections

We have covered materials that are linear combinations of Lambertian diffuse and specular components. However, many surfaces are polished to a sufficient degree that you can see coherent reflections of the surrounding environment. This section will discuss two ways of simulating this phenomenon and show several applications.

People often assume that mirror-like reflections require ray tracing. But not all renderers support ray tracing (and, in fact, those renderers are typically much faster than ray tracers). In addition, there are situations where even ray tracing does not help. For example, if you are compositing a CG object into a live-action shot, you

[2] Note the difference between *translucency*, the diffuse transmission of very scattered light through a thickness of material, and *transparency*, which means you can see a coherent image through the object. Ordinary paper is translucent, whereas glass is transparent.

Listing 9.4 `MaterialThinPlastic` implements a simple, thin, plastic-like BRDF.

```
/* Compute the color of the surface using a simple, thin, plastic-like
 * BRDF. We call it _thin_ because it includes a transmission component
 * to allow light from the _back_ of the surface to affect the appearance.
 * Typical values are Ka=1, Kd=0.8, Kt=0.2, Ks=0.5, roughness=0.1.
 */
color MaterialThinPlastic (normal Nf;  vector V;  color basecolor;
                           float Ka, Kd, Kt, Ks, roughness;)
{
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf) + Kt*diffuse(-Nf))
           + Ks*specular(Nf,V,roughness);
}
```

may want the object to reflect its environment. This is not possible even with ray tracing because the environment does not exist in the CG world. Of course, you could laboriously model all the objects in the live-action scene, but this seems like too much work for a few reflections.

Luckily, RenderMan Shading Language provides support for faking these effects with texture maps, even for renderers that do not support any ray tracing. In this case, we can take a multipass approach, first rendering the scene from the points of view of the reflectors, then using these first passes as special texture maps when rendering the final view from the main camera.

9.2.1 Environment Maps

Environment maps take images of six axis-aligned directions from a particular point (like the six faces of a cube) and allow you to look up texture on those maps, indexed by a direction vector, thus simulating reflection. An example of an "unwrapped" environment map is shown in Figure 9.3.

Accessing an environment map from inside your shader is straightforward with the built-in `environment` function:

type `environment (string filename, vector R, ...)`

The `environment` function is quite analogous to the `texture()` call in several ways:

- The return type can be explicitly cast to either `float` or `color`. If you do not explicitly cast the results, the compiler will try to infer the return type, which could lead to ambiguous situations.
- A `float` in brackets immediately following the filename indicates a starting channel (default is to start with channel 0).
- For environment maps, the texture coordinates consist of a direction vector. As with `texture()`, derivatives of this vector will be used for automatic filtering of

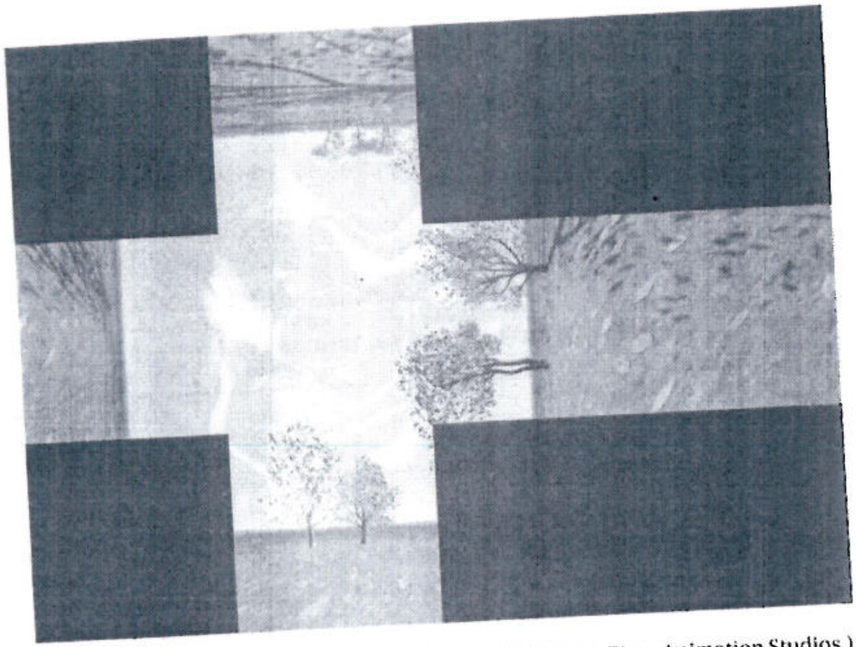


Figure 9.3 *Geri's Game*—An example environment map. (© Pixar Animation Studios.) See also color plate 9.3.

the environment map lookup. Optionally, four vectors may be given to bound the angle range, and in that case no derivatives will be taken.

- The `environment` function can take the optional arguments `"blur"`, `"width"`, and `"filter"`, which perform the same functions as for `texture()`.

Environment maps typically sample the mirror direction, as computed by the Shading Language built-in function `reflect()`. For example,

```
normal Nf = normalize (faceforward (N, I));
vector R = normalize (reflect (I, N));
color Crefl = color environment (envmapname, R);
```

Note that the `environment()` is indexed by direction only, not position. Thus, not only is the environment map created from the point of view of a single location but all lookups are also made from that point. Alternatively, you can think of the environment map as being a reflection of a cube of infinite size. Either way, two points with identical mirror directions will look up the same direction in the environment map. This is most noticeable for flat surfaces, which tend to have all of their points index the same spot on the environment map. This is an obvious and objectionable artifact, especially for surfaces like floors, whose reflections are *very* sensitive to position.

We can partially overcome this difficulty with the following strategy:

1. We assume that the environment map exists on the interior of a sphere with a *finite* and known radius as well as a known center. Example: if the environment map is of a room interior, we choose a sphere radius representative of the room size. (Even if we have assembled an environment map from six rectangular faces, because it is indexed by direction only, for simplicity we can just as easily think of it as a spherical map.)
2. Instead of indexing the environment by direction only, we define a ray using the position and mirror direction of the point, then calculate the intersection of this ray with our aforementioned environment sphere.
3. The intersection with the environment sphere is then used as the environment lookup direction.

Thus, if a simple `environment()` lookup is like ray tracing against a sphere of infinite radius, then the scheme above is simply ray tracing against a sphere of a radius appropriate to the actual scene in the environment map.

As a subproblem, we must be able to intersect an environment sphere with a ray. A general treatment of ray/object intersections can be found in (Glassner, 1989), but the ray/sphere case is particularly simple. If a ray if described by end point E and unit direction vector I (expressed in the coordinate system of a sphere centered at its local origin and with radius r), then any point along the ray can be described as $E + It$ (for free parameter t). This ray intersects the sphere anyplace that $|E + It| = r$. Because the length of a vector is the square root of its dot product with itself, then

$$(E + It) \cdot (E + It) = r^2$$

Expanding the x, y, and z components for the dot product calculation yields

$$\begin{aligned}
(E_x + I_x t)^2 + (E_y + I_y t)^2 + (E_z + I_z t)^2 - r^2 &= 0 \\
E_x^2 + 2E_x I_x t + I_x^2 t^2 + E_y^2 + 2E_y I_y t + I_y^2 t^2 & \\
+ E_z^2 + 2E_z I_z t + I_z^2 t^2 - r^2 &= 0 \\
(I \cdot I)t^2 + 2(E \cdot I)t + E \cdot E - r^2 &= 0
\end{aligned}$$

for which the value(s) of t can be solved using the quadratic equation. This solution is performed by the `raysphere` function, which in turn is used by our enhanced `Environment` routine (see Listing 9.5). Note that `Environment` also returns an `alpha` value from the environment map, allowing us to composite multiple environment maps together.

9.2.2 Creating Cube Face Environment Maps

The creation of cube face environment maps is straightforward. First, six reflection images are created by rendering the scene from the point of view of the reflective object in each of six orthogonal directions in `"world"` space, given in Table 9.1.

Listing 9.5 `Environment` function replaces a simple `environment` call, giving more accurate reflections by tracing against an environment sphere of finite radius.

```
/* raysphere - calculate the intersection of ray (E,I) with a sphere
 * centered at the origin and with radius r.  We return the number of
 * intersections found (0, 1, or 2), and place the distances to the
 * intersections in t0, t1 (always with t0 <= t1).  Ignore any hits
 * closer than eps.
 */
float
raysphere (point E; vector I;    /* Origin and unit direction of the ray */
           float r;              /* radius of sphere */
           float eps;            /* epsilon - ignore closer hits */
           output float t0, t1;  /* distances to intersection */
    )
{
    /* Set up a quadratic equation -- note that a==1 if I is normalized */
    float b = 2 * ((vector E) . I);
    float c = ((vector E) . (vector E)) - r*r;
    float discrim = b*b - 4*c;
    float solutions;
    if (discrim > 0) {            /* Two solutions */
        discrim = sqrt(discrim);
        t0 = (-discrim - b) / 2;
        if (t0 > eps) {
            t1 = (discrim - b) / 2;
            solutions = 2;
        } else {
            t0 = (discrim - b) / 2;
            solutions = (t0 > eps) ? 1 : 0;
        }
    } else if (discrim == 0) {  /* One solution on the edge! */
        t0 = -b/2;
        solutions = (t0 > eps) ? 1 : 0;
    } else {                    /* Imaginary solution -> no intersection */
        solutions = 0;
    }
    return solutions;
}


/* Environment() - A replacement for ordinary environment() lookups, this
 * function ray traces against an environment sphere of known, finite
 * radius.  Inputs are:
 *     envname - filename of environment map
 *     envspace - name of space environment map was made in
 *     envrad - approximate supposed radius of environment sphere
 *     P, R - position and direction of traced ray
 *     blur - amount of additional blur to add to environment map
 * Outputs are:
 *     return value - the color of incoming environment light
 *     alpha - opacity of environment map lookup in the direction R.
```

▶

```
 * Warning -  the environment call itself takes derivatives, causing
 * trouble if called inside a loop or varying conditional!  Be cautious.
 */
color Environment ( string envname, envspace;  uniform float envrad;
                    point P;  vector R;  float blur; output float alpha;)
{
    /* Transform to the space of the environment map */
    point Psp = transform (envspace, P);
    vector Rsp = normalize (vtransform (envspace, R));
    uniform float r2 = envrad * envrad;
    /* Clamp the position to be *inside* the environment sphere */
    if ((vector Psp).(vector Psp) > r2)
        Psp = point (envrad * normalize (vector Psp));
    float t0, t1;
    if (raysphere (Psp, Rsp, envrad, 1.0e-4, t0, t1) > 0)
        Rsp = vector (Psp + t0 * Rsp);
    alpha = float environment (envname[3], Rsp, "blur", blur, "fill", 1);
    return color environment (envname, Rsp, "blur", blur);
}
```

Table 9.1: The six view directions in an environment map.

Face view	Axis toward top	Axis toward right
px (positive x)	$+y$	$-z$
nx (negative x)	$+y$	$+z$
py	$-z$	$+x$
ny	$+z$	$+x$
pz	$+y$	$+x$
nz	$+y$	$-x$

Next, these six views (which should be rendered using a square 90° field of view to completely cover all directions) are combined into a single environment map. This can be done from the RIB file:

`MakeCubeFaceEnvironment` *px nx py ny pz nz envfile*

Here *px*, *nx*, and so on are the names of the files containing the individual face images, and *envfile* is the name of the file where you would like the final environment map to be placed.

Alternatively, most renderers will have a separate program that will assemble an environment map from the six individual views. In the case of *PRMan*, this can be done with the `txmake` program:

`txmake -envcube` *px nx py ny pz nz envfile*

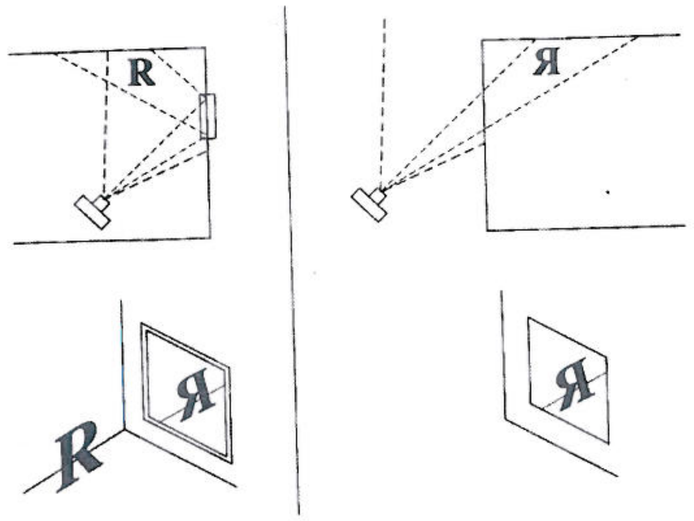


Figure 9.4 Creating a mirrored scene for generating a reflection map. On the top left, a camera views a scene that includes a mirror. Below is the image it produces. On the top right, the camera instead views the *mirror scene*. Notice that the image it produces (below) contains the required reflection.

9.2.3 Flat Surface Reflection Maps

For the special case of flat objects (such as floors or flat mirrors), there is an even easier and more efficient method for producing reflections, which also solves the problem of environment maps being inaccurate for flat objects.

For the example of a flat mirror, we can observe that the image in the reflection would be identical to the image that you would get if you put another copy of the room on the other side of the mirror, *reflected* about the plane of the mirror. This geometric principle is illustrated in Figure 9.4.

Once we create this reflection map, we can turn it into a texture and index it from our shader. Because the pixels in the reflection map correspond exactly to the reflected image in the same pixels of the main image, we access the texture map by the texture's pixel coordinates, not the s, t coordinates of the mirror. We can do this by projecting `P` into `"NDC"` space. This is done in the `ReflMap` function in Listing 9.6.

9.2.4 General Reflections and Shiny Surfaces

We would prefer to write our shaders so that we may use either reflection or environment maps. Therefore, we can combine both into a single routine, `SampleEnvironment()`, given in Listing 9.7.

Listing 9.6 `ReflMap` function performs simple reflection mapping.

```
color ReflMap ( string reflname; point P; float blur;
                output float alpha; )
{
    /* Transform to the space of the environment map */
    point Pndc = transform ("NDC", P);
    float x = xcomp(Pndc), y = ycomp(Pndc);
    alpha = float texture (reflname[3], x, y, "blur", blur, "fill", 1);
    return color texture (reflname, x, y, "blur", blur);
}
```

Listing 9.7 `SampleEnvironment` function makes calls to either or both of `Environment` or `ReflMap` as needed.

```
#define ENVPARAMS      envname, envspace, envrad

#define DECLARE_ENVPARAMS                                    \
        string envname, envspace;  uniform float envrad

#define DECLARE_DEFAULTED_ENVPARAMS                          \
        string envname = "", envspace = "world";             \
        uniform float envrad = 100


color
SampleEnvironment (point P;  vector R;  float Kr, blur;
                   DECLARE_ENVPARAMS;)
{
    color C = 0;
    float alpha;
    if (envname != "") {
        if (envspace == "NDC")
            C = ReflMap (envname, P, blur, alpha);
        else C = Environment (envname, envspace, envrad, P, R, blur,
                              alpha);
    }
    return Kr*C;
}
```

A few comments on the source code in Listing 9.7. To allow easy specification of the many environment-related parameters, we define macros `ENVPARAMS`, `DECLARE_ENVPARAMS`, and `DECLARE_DEFAULTED_ENVPARAMS`, which are macros containing the parameter names, declarations, and declarations with default values, respectively. These macros allow us to succinctly include them in any shader, as we have done in shader `shinymetal` (Listing 9.8).

Listing 9.8 `MaterialShinyMetal` and the `shinymetal` shader.

```
/* Compute the color of the surface using a simple metal-like BRDF.  To
 * give a metallic appearance, both diffuse and specular components are
 * scaled by the color of the metal.  It is recommended that Kd < 0.1,
 * Ks > 0.5, and roughness > 0.15 to give a believable metallic appearance.
 */
color MaterialShinyMetal (normal Nf;  color basecolor;
                          float Ka, Kd, Ks, roughness, Kr, blur;
                          DECLARE_ENVPARAMS;)

{
    extern point P;
    extern vector I;
    vector IN = normalize(I), V = -IN;
    vector R = reflect (IN, Nf);
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf) +
                        Ks*specular(Nf,V,roughness) +
                        SampleEnvironment (P, R, Kr, blur, ENVPARAMS));
}


surface
shinymetal ( float Ka = 1, Kd = 0.1, Ks = 1, roughness = 0.2;
             float Kr = 0.8, blur = 0;  DECLARE_DEFAULTED_ENVPARAMS; )

{
    normal Nf = faceforward (normalize(N), I);
    Ci = MaterialShinyMetal (Nf, Cs, Ka, Kd, Ks, roughness, Kr, blur,
                             ENVPARAMS);
    Oi = Os;  Ci *= Oi;
}
```

With this routine in our toolbox, we can make a straightforward shader that can be used for shiny metals like mirrors, chrome, or copper (see Listing 9.8). If you are using the shader for a flat object with a prepared flat reflection map, you need only pass "NDC" as the `envspace` parameter and the `SampleEnvironment` function will correctly access your reflection map.

9.2.5 Fresnel and Shiny Plastic

All materials are more reflective at glancing angles than face-on (in fact, materials approach 100% reflectivity at grazing angles). For polished metals, the face-on reflectivity is so high that this difference in reflectivity with angle is not very noticeable. You can convince yourself of this by examining an ordinary mirror—it appears nearly equally shiny when you view your reflection head-on versus when you look at the mirror at an angle. So we tend to ignore the angular effect on reflectivity, assuming for simplicity that polished metals are equally reflective in all directions (as we have in the previous section).

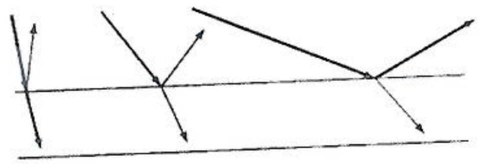


Figure 9.5 The ratio of reflected to transmitted light at a material boundary changes with the angle of the incident light ray.

Dielectrics such as plastic, ceramics, and glass are much less reflective than metals overall, and especially so when viewed face-on. Therefore, their higher reflectivity at grazing angles is a much more significant visual detail (see Figure 9.5). The formulas that relate angle to reflectivity of such materials are known as the *Fresnel equations*.

A function that calculates the Fresnel terms is included in Shading Language:

```
void fresnel (vector I; normal N; float eta;
              output float Kr, Kt; output vector R, T);
```

According to Snell's law and the Fresnel equations, `fresnel` computes the reflection and transmission direction vectors `R` and `T`, respectively, as well as the scaling factors for reflected and transmitted light, `Kr` and `Kt`. The `I` parameter is the normalized incident ray, `N` is the normalized surface normal, and `eta` is the ratio of refractive index of the medium containing `I` to that on the opposite side of the surface.

We can use `fresnel()` to attenuate the relative contributions of environmental and diffuse reflectances. This is demonstrated in the `MaterialShinyPlastic` and `shinyplastic`, both shown in Listing 9.9. Figure 9.6 compares the appearance of `MaterialShinyMetal` and `MaterialShinyPlastic`.

The index of refraction of air is very close to 1.0, water is 1.33, ordinary window glass is approximately 1.5, and most plastics are close to this value. In computer graphics, we typically assume that 1.5 is a reasonable refractive index for a very wide range of plastics and glasses. Therefore, a reasonable value for the `eta` parameter passed to `fresnel()` is 1/1.5. Nearly any optics textbook will list the indices of refraction for a variety of real materials, should you wish to be more physically accurate.

Of course, for real materials the refractive index is wavelength dependent. We tend to ignore this effect, since we have such an ad hoc approach to spectral sampling and color spaces anyway. Although metals also have wavelength-dependent indices of refraction, Fresnel effects, and angle-dependent spectral reflectivities, they are much less visually noticeable in most metals, so we usually just pretend

Listing 9.9 shinyplastic shader is for highly polished plastic and uses fresnel() so that reflections are stronger at grazing angles.

```
color
MaterialShinyPlastic (normal Nf;  color basecolor;
                      float Ka, Kd, Ks, roughness, Kr, blur, ior;
                      DECLARE_ENVPARAMS;)

{
    extern vector I;      extern point P;
    vector IN = normalize(I), V = -IN;
    float fkr, fkt;  vector R, T;
    fresnel (IN, Nf, 1/ior, fkr, R, T);
    fkt = 1-fkr;
    return  fkt * basecolor * (Ka*ambient() + Kd*diffuse(Nf))
                  + (Ks*fkr) * specular(Nf,V,roughness)
                  + SampleEnvironment (P, R, fkr*Kr, blur, ENVPARAMS));
}


surface
shinyplastic ( float Ka = 1, Kd = 0.5, Ks = .5, roughness = 0.1;
               float Kr = 1, blur = 0, ior = 1.5;
               DECLARE_DEFAULTED_ENVPARAMS; )

{
    normal Nf = faceforward (normalize(N), I);
    Ci = MaterialShinyPlastic (Nf, Cs, Ka, Kd, Ks, roughness, Kr,
                               blur, ior,  ENVPARAMS);

    Oi = Os;  Ci *= Oi;
}
```

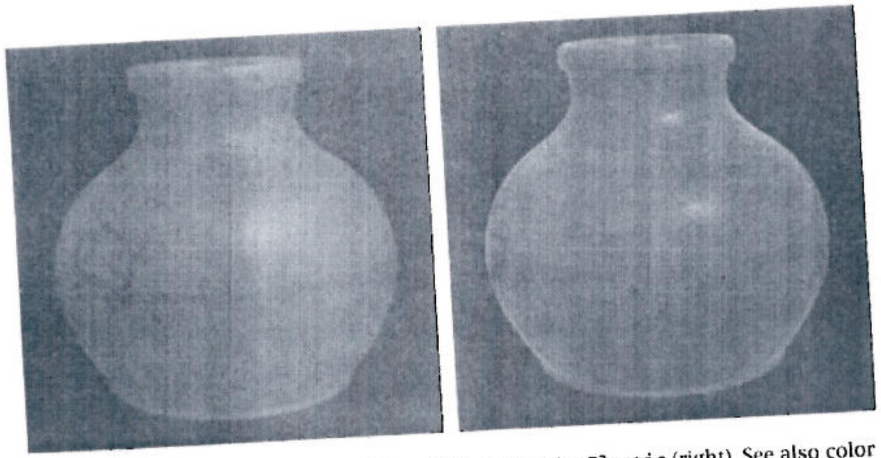


Figure 9.6 MaterialShinyMetal (left) and MaterialShinyPlastic (right). See also color plate 9.6.

that shiny metals reflect uniformly. You are, of course, perfectly free to go to the extra effort of computing the wavelength and angle dependency of metals!

Important note: in `MaterialShinyPlastic` we set `fkt=1-fkr`, overriding the value returned by `fresnel()`. The reasons for this are extremely subtle and well beyond the scope of this book. Suffice it to say that the value that `fresnel()` is supposed to return for `Kt` assumes a more rigorous simulation of light propagation than either *PRMan* or *BMRT* provides. In light of these inaccuracies, the intuitive notion that `Kt` and `Kr` ought to sum to 1 is about as good an approximation as you might hope for. We will therefore continue to use this oversimplification.

9.3 Illuminance Loops, or How `diffuse()` and `specular()` Work

We have seen classes of materials that can be described by various relative weightings and uses of the built-in `diffuse()` and `specular()` functions. In point of fact, these are just examples of possible illumination models, and it is quite useful to write alternative ones. In order to do that, however, the shader needs access to the lights: how many are there, where are they, and how bright are they?

The key to such functionality is a special syntactic structure in Shading Language called an `illuminance` loop:

```
illuminance ( point position ) {
    statements;
}

illuminance ( point position; vector axis; float angle ) {
    statements;
}
```

The `illuminance` statement loops over all light sources visible from a particular *position*. In the first form, all lights are considered, and in the second form, only those lights whose directions are within *angle* of *axis* (typically, *angle*=$\pi/2$ and *axis*=`N`, which indicates that all light sources in the visible hemisphere from `P` should be considered). For each light source, the *statements* are executed, during which two additional variables are defined: `L` is the vector that points to the light source, and `Cl` is the color representing the incoming energy from that light source.

Perhaps the most straightforward example of the use of `illuminance` loops is the implementation of the `diffuse()` function:

```
color diffuse (normal Nn)
{
    extern point P;
    color C = 0;
    illuminance (P, Nn, PI/2) {
        C += Cl * (Nn . normalize(L));
    }
    return C;
}
```

Briefly, this function is looping over all light sources. Each light's contribution is computed using a Lambertian shading model; that is, the light reflected diffusely is the light arriving from the source multiplied by the dot product of the normal with the direction of the light. The contributions of all lights are summed and that sum is returned as the result of `diffuse()`.

9.4 Identifying Lights with Special Properties

In reality, surfaces simply scatter light. The scattering function, or BRDF (which stands for bidirectional reflection distribution function) may be quite complex. Dividing the scattering function into diffuse versus specular, or considering the BRDF to be a weighted sum of the two, is simply a convenient oversimplification. Many other simplifications and abstractions are possible.

In the physical world, light scattering is a property of the surface material, not a property of the light itself or of its source. Nonetheless, in computer graphics it is often convenient and desirable to place light sources whose purpose is solely to provide a highlight, or alternatively to provide a soft fill light where specular highlights would be undesirable. Therefore, we would like to construct our `diffuse()` function so that it can ignore light sources that have been tagged as being "nondiffuse"; that is, the source itself should only contribute specular highlights. We can do this by exploiting the *message passing* mechanism of Shading Language, wherein the surface shader may peek at the parameters of the light shader.

`float lightsource ( string paramname; output` *type* `result )`

> The `lightsource()` function, which may only be called from within an `illuminance` loop, searches for a parameter of the light source named `paramname`. If such a parameter is found and if its type matches that of the variable `result`, then its value will be stored in the `result` and the `lightsource()` function will return 1.0. If no such parameter is found, the variable `result` will be unchanged and the return value of `lightsource` will be zero.

We can use this mechanism as follows. Let us assume rather arbitrarily that any light that we wish to contribute only to specular highlights (and that therefore should be ignored by the `diffuse()` function) will contain in its parameter list an `output float` parameter named `__nondiffuse`. Similarly, we can use an `output float` parameter named `__nonspecular` to indicate that a particular light should not contribute to specular highlights. Then implementations of `diffuse()` and `specular()`, as shown in Listing 9.10, would respond properly to these con-

Listing 9.10 The implementation of the built-in `diffuse()` and `specular()` functions, including controls to ignore nondiffuse and nonspecular lights.

```
color diffuse (normal Nn)
{
    extern point P;
    color C = 0;
    illuminance (P, Nn, PI/2) {
        float nondiff = 0;
        lightsource ("__nondiffuse", nondiff);
        C += Cl * (1-nondiff) * (Nn . normalize(L));
    }
    return C;
}

color specular (normal Nn; vector V; float roughness)
{
    extern point P;
    color C = 0;
    illuminance (P, Nn, PI/2) {
        float nonspec = 0;
        lightsource ("__nonspecular", nonspec);
        vector H = normalize (normalize(L) + V);
        C += Cl * (1-nonspec) * pow (max (0, Nn.H), 1/roughness);
    }
    return C;
}
```

trols. In both *PRMan* and *BMRT*, the implementations of `diffuse()` and `specular()` respond to `__nondiffuse` and `__nonspecular` in this manner (although as we explained earlier, the implementation of `specular()` is not quite what is in Listing 9.10).

This message passing mechanism may be used more generally to pass all sorts of "extra" information from the lights to the surfaces. For example, a light may include an `output` parameter giving its *ultraviolet* illumination, and special surface shaders may respond to this parameter by exhibiting fluorescence.

There is an additional means of controlling `illuminance` loops with an optional *light category* specifier:

```
illuminance ( string category; point position )
    statements;

illuminance ( string category; point position;
              vector axis; float angle )
    statements;
```

Ordinary `illuminance` loops will execute their body for every nonambient light source. The named category extension to the `illuminance` syntax causes the *statements* to be executed only for a subset of light sources.

Light shaders can specify the categories to which they belong by declaring a string parameter named `__category` (this name has two underscores), whose value is a comma-separated list of categories into which the light shader falls. When the `illuminance` statement contains a string parameter *category*, the loop will only consider lights for which the *category* is among those listed in its comma-separated `__category` list. If the `illuminance` *category* begins with a - character, then only lights *not* containing that category will be considered. For example,

```
float uvcontrib = 0;
illuminance ("uvlight", P, Nf, PI/2) {
    float uv = 0;
    lightsource ("uv", uv);
    uvcontrib += uv;
}
Ci += uvcontrib * uvglowcolor;
```

will look specifically for lights containing the string `"uvlight"` in their `__category` list and will execute those lights, summing their `"uv"` output parameter. An example light shader that computes ultraviolet intensity might be

```
light
uvpointlight (float intensity = 1, uvintensity = 0.5;
              color lightcolor = 1;
              point from = point "shader" (0,0,0);
              output varying float uv = 0;
              string __category = "uvlight";)
{
    illuminate (from) {
        Cl = intensity * lightcolor / (L . L);
        uv = uvintensity / (L . L);
    }
}
```

Don't worry too much that you haven't seen light shaders yet: before the end of this chapter, this example will be crystal clear.

9.5 Custom Material Descriptions

We have seen that the implementation of `diffuse()` is that of a Lambertian shading model that approximates a rough surface that scatters light equally in all directions. Similarly, `specular()` implements a Blinn-Phong scattering function (according to the RI spec). As we've seen, different weighted combinations of these two functions can yield materials that look like a variety of plastics and metals. Now that we understand how they operate, we may use the `illuminance` construct ourselves to create custom primitive local illumination models. Past ACM SIGGRAPH proceedings are a treasure trove of ideas for more complex and realistic local illumination models. In this section we will examine three local illumination models (two physically based and one ad hoc) and construct shader functions that implement them.

9.5.1 Rough Surfaces

Lambert's law models a perfectly smooth surface that reflects light equally in all directions. This is very much an oversimplification of the behavior of materials. In the proceedings of SIGGRAPH '94, Michael Oren and Shree K. Nayar described a surface scattering model for rough surfaces. Their model (and others, including Beckman, Blinn, and Cook/Torrance) considers rough surfaces to have microscopic grooves and hills. This is modeled mathematically as a collection of microfacets having a statistical distribution of relative directions. Their results indicated that many real-world rough materials (like clay) could be more accurately modeled using the following equation:

$$L_r(\theta_r, \theta_i, \phi_r - \phi_i, \sigma) = \frac{\rho}{\pi} E_0 \cos\theta_i (A + B \max[0, \cos(\phi_r - \phi_i)] \sin\alpha \tan\beta),$$

where

$$A = 1.0 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = \max[\theta_i, \theta_r]$$

$$\beta = \min[\theta_i, \theta_r],$$

and the terms mean

- ρ is the reflectivity of the surface (`Kd*Cs`).
- E_0 is the energy arriving at the surface from the light (`Cl`).
- θ_i is the angle between the surface normal and the direction of the light source.
- θ_r is the angle between the surface normal and the vector in the direction the light is reflected (i.e., toward the viewer).
- $\phi_r - \phi_i$ is the angle (about the normal) between the incoming and reflected light directions.
- σ is the standard deviation of the angle distribution of the microfacets (in radians). Larger values represent more rough surfaces; smaller values represent smoother surfaces. If $\sigma = 0$, the surface is perfectly smooth, and this function reduces to a simple Lambertian reflectance model. We'll call this parameter "roughness."

Listing 9.11 LocIllumOrenNayar implements a BRDF for diffuse, but rough, surfaces.

```
/*
 * Oren and Nayar's generalization of Lambert's reflection model.
 * The roughness parameter gives the standard deviation of angle
 * orientations of the presumed surface grooves.  When roughness=0,
 * the model is identical to Lambertian reflection.
 */
color
LocIllumOrenNayar (normal N;  vector V;  float roughness;)
{
    /* Surface roughness coefficients for Oren/Nayar's formula */
    float sigma2 = roughness * roughness;
    float A = 1 - 0.5 * sigma2 / (sigma2 + 0.33);
    float B = 0.45 * sigma2 / (sigma2 + 0.09);
    /* Useful precomputed quantities */
    float  theta_r = acos (V . N);          /* Angle between V and N */
    vector V_perp_N = normalize(V-N*(V.N)); /* Part of V perpendicular to N
 */

    /* Accumulate incoming radiance from lights in C */
    color  C = 0;
    extern point P;
    illuminance (P, N, PI/2) {
        /* Must declare extern L & Cl because we're in a function */
        extern vector L;  extern color Cl;
        float nondiff = 0;
        lightsource ("__nondiffuse", nondiff);
        if (nondiff < 1) {
            vector LN = normalize(L);
            float cos_theta_i = LN . N;
            float cos_phi_diff = V_perp_N . normalize(LN - N*cos_theta_i);
            float theta_i = acos (cos_theta_i);
            float alpha = max (theta_i, theta_r);
            float beta = min (theta_i, theta_r);
            C += (1-nondiff) * Cl * cos_theta_i *
                 (A + B * max(0,cos_phi_diff) * sin(alpha) * tan(beta));
        }
    }
    return C;
}
```

These equations are easily translated into an illuminance loop, as shown in Listing 9.11.

Figure 9.7 shows a teapot with the Oren/Nayar reflectance model. The left teapot uses a roughness coefficient of 0.5, while the right uses a roughness coefficient of 0, which has reflectance identical to Lambertian shading. Notice that as roughness

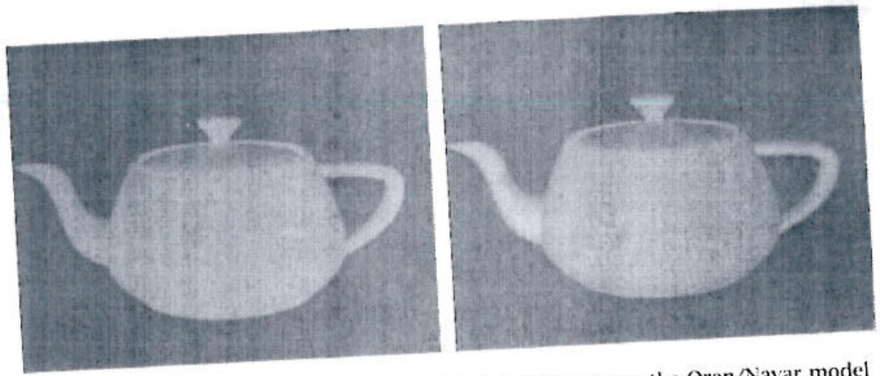


Figure 9.7 When the light source is directly behind the viewer, the Oren/Nayar model (left) acts much more as a retroreflector, compared to the Lambertian model (right). See also color plate 9.7.

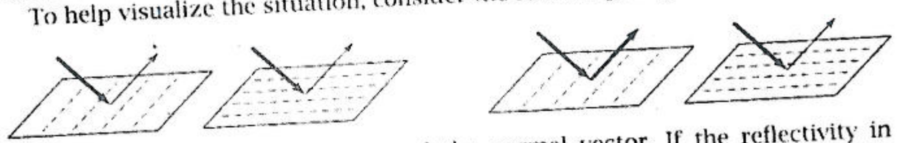
increases, a strong backscattering effect is present. The object begins to act as a retroreflector, so that light that comes from the same direction as the viewer bounces back at a rate nearly independent of surface orientation.

It is this type of backscattering that accounts for the appearance of the full moon. Everyone has noticed that a full moon (when the viewer and sun are nearly at the same angle to the moon) looks like a flat disk, rather than like a ball. The moon is not Lambertian—it is more closely modeled as a rough surface, and this reflection model is a good approximation to the behavior of lunar dust.

9.5.2 Anisotropic Metal

The `MaterialRoughMetal` function described earlier does an adequate job of simulating the appearance of a metal object that is rough enough that coherent reflections (from ray tracing or environment maps) are not necessary. However, it does make the simplifying assumption that the metal scatters light only according to the angular relationship between the surface normal, the eye, and the light source. It specifically does not depend on the orientation of the surface as it spins around the normal vector.

To help visualize the situation, consider the following diagram:



Imagine rotating the material around the normal vector. If the reflectivity in a particular direction is independent of the surface orientation, then the material is said to be *isotropic*. On the other hand, if the material reflects preferentially depending on surface orientation, then it is *anisotropic*.

Anisotropic materials are not uncommon. Various manufacturing processes can produce materials with microscopic grooves that are all aligned to a particular direction (picture the surface being covered with tiny half-cylinders oriented in parallel or otherwise coherently). This gives rise to anisotropic BRDFs. A number of papers have been written about anisotropic reflection models, including Kajiya (1985) and Poulin and Fournier (1990).

Greg Ward Larson described an anisotropic reflection model in his SIGGRAPH '92 paper, "Measuring and Modeling Anisotropic Reflection" (Ward, 1992). In this paper, anisotropic specular reflection was given as

$$\frac{1}{\sqrt{\cos\theta_i \cos\theta_r}} \frac{1}{4\pi\alpha_x\alpha_y} \exp\left[-2\frac{\left(\frac{\hat{h}\cdot\hat{x}}{\alpha_x}\right)^2 + \left(\frac{\hat{h}\cdot\hat{y}}{\alpha_y}\right)^2}{1+\hat{h}\cdot\hat{n}}\right],$$

where

- θ_i is the angle between the surface normal and the direction of the light source.
- θ_r is the angle between the surface normal and the vector in the direction the light is reflected (i.e., toward the viewer).
- $\hat{x}$ and $\hat{y}$ are the two perpendicular tangent directions on the surface.
- α_x and α_y are the standard deviations of the slope in the $\hat{x}$ and $\hat{y}$ directions, respectively. We will call these `xroughness` and `yroughness`.
- $\hat{n}$ is the unit surface normal (`normalize(N)`).
- $\hat{h}$ is the half-angle between the incident and reflection rays (i.e., `H = normalize (normalize(-I) + normalize(L))`).

Listing 9.12 lists the function `LocIllumWardAnisotropic`, which implements the anisotropic specular component of Larson's model.[3] This function can be used instead of an ordinary `specular()` call. It differs from `specular()` in that it takes *two* roughness values: one for the direction aligned with surface tangent `xdir`, and the other for the perpendicular direction. Figure 9.8 shows this model applied to a teapot.

9.5.3 Glossy Specular Highlights

The previous subsections listed Shading Language implementations of two local illumination models that are physically based simulations of the way light reflects

[3] When comparing the original equation to our Shading Language implementation, you may wonder where the factor of $1/\pi$ went and why there appears to be an extra factor of $L \cdot N$. This is not an error! Greg Ward Larson's paper describes the BRDF, which is only part of the kernel of the light integral, whereas shaders describe the result of that integral. This is something that must be kept in mind when coding traditional BRDFs in `illuminance` loops.

Listing 9.12 LocIllumWardAnisotropic: Greg Ward Larson's anisotropic specular illumination model.

```
/*
 * Greg Ward Larson's anisotropic specular local illumination model.
 * The derivation and formulae can be found in:  Ward, Gregory J.
 * "Measuring and Modeling Anisotropic Reflection," ACM Computer
 * Graphics 26(2) (Proceedings of Siggraph '92), pp. 265-272, July, 1992.
 * Notice that compared to the paper, the implementation below appears
 * to be missing a factor of 1/pi, and to have an extra L.N term.
 * This is not an error!  It is because the paper's formula is for the
 * BRDF, which is only part of the kernel of the light integral, whereas
 * shaders must compute the result of the integral.
 *
 * Inputs:
 *   N - unit surface normal
 *   V - unit viewing direction (from P toward the camera)
 *   xdir - a unit tangent of the surface defining the reference
 *          direction for the anisotropy.
 *   xroughness - the apparent roughness of the surface in xdir.
 *   yroughness - the roughness for the direction of the surface
 *          tangent perpendicular to xdir.
 */
color
LocIllumWardAnisotropic (normal N;  vector V;
                         vector xdir;  float xroughness, yroughness;)
{
    float sqr (float x) { return x*x; }

    float cos_theta_r = clamp (N.V, 0.0001, 1);
    vector X = xdir / xroughness;
    vector Y = (N ^ xdir) / yroughness;

    color C = 0;
    extern point P;
    illuminance (P, N, PI/2) {
        /* Must declare extern L & Cl because we're in a function */
        extern vector L;  extern color Cl;
        float nonspec = 0;
        lightsource ("__nonspecular", nonspec);
        if (nonspec < 1) {
            vector LN = normalize (L);
            float cos_theta_i = LN . N;
            if (cos_theta_i > 0.0) {
                vector H = normalize (V + LN);
                float rho = exp (-2 * (sqr(X.H) + sqr(Y.H)) / (1 + H.N))
                    / sqrt (cos_theta_i * cos_theta_r);
                C += Cl * ((1-nonspec) * cos_theta_i * rho);
            }
        }
    }
    return C / (4 * xroughness * yroughness);
}
```

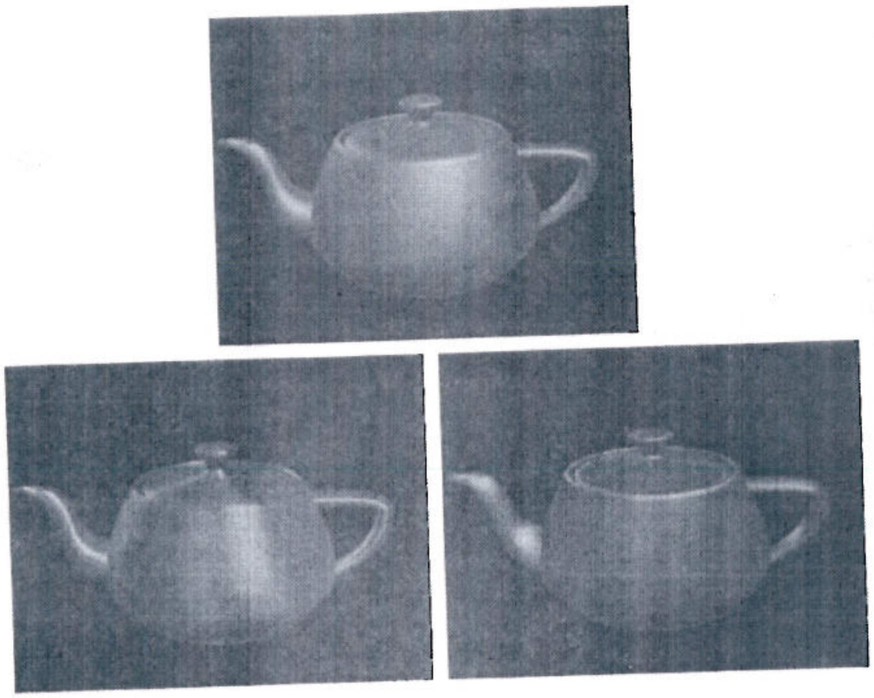


Figure 9.8 Examples of the Ward anisotropic reflection model. Isotropic reflection with `xroughness=yroughness=0.3` (top). Anisotropic reflection with `xroughness=0.15`, `yroughness=0.5` (bottom left). Anisotropic reflection with `xroughness=0.5`, `yroughness=0.15` (bottom right). In all cases, `xdir=normalize(dPdu)`. See also color plate 9.8.

off certain material types. Many times, however, we want an effect that achieves a particular look without resorting to simulation. The look itself may or may not match a real-world material, but the computations are completely ad hoc. This subsection presents a local illumination model that achieves a useful look but that is decidedly not based on the actual behavior of light. It works well for glossy materials, such as finished ceramics, glass, or wet materials.

We note that specular highlights are a lot like mirror reflections of the light source. The observation that they are big fuzzy circles on rough objects and small sharp circles on smooth objects is a combination of the blurry reflection model and the fact that the real light sources are not infinitesimal points as we often use in computer graphics, but extended area sources such as light bulbs, the sun, and so on.

For polished glossy surfaces, we would like to get a clear, distinct reflection of those bright area sources, even if there isn't a noticeable mirror reflection of the rest of the environment (and even if we aren't really using true area lights). We

Listing 9.13 LocIllumGlossy function: nonphysical replacement for specular() that makes a uniformly bright specular highlight.

```
/*
 * LocIllumGlossy - a possible replacement for specular(), having
 * more uniformly bright core and a sharper falloff.  It's a nice
 * specular function to use for something made of glass or liquid.
 * Inputs:
 *   roughness - related to the size of the highlight, larger is bigger
 *   sharpness - 1 is infinitely sharp, 0 is very dull
 */
color LocIllumGlossy ( normal N;  vector V;
                       float roughness, sharpness; )
{
    color C = 0;
    float w = .18 * (1-sharpness);
    extern point P;
    illuminance (P, N, PI/2) {
        /* Must declare extern L & Cl because we're in a function */
        extern vector L;  extern color Cl;
        float nonspec = 0;
        lightsource ("__nonspecular", nonspec);
        if (nonspec < 1) {
            vector H = normalize(normalize(L)+V);
            C += Cl * ((1-nonspec) *
                       smoothstep (.72-w, .72+w,
                                   pow(max(0,N.H), 1/roughness)));
        }
    }
    return C;
}
```

propose that we could achieve this look by thresholding the specular highlight. In other words, we modify a Blinn-Phong specular term (c.f. Listing 9.10) from

```
Cl * pow (max (0, Nn.H), 1/roughness);
```

to the thresholded version:

```
Cl * smoothstep (e0, e1, pow (max (0, Nn.H), 1/roughness));
```

With an appropriately chosen e0 and e1, the specular highlight will appear to be smaller, have a sharp transition, and be fully bright inside the transition region. Listing 9.13 is a full implementation of this proposal (with some magic constants chosen empirically by the authors). Finally, Figure 9.9 compares this glossy specular highlight to a standard plastic-like specular() function. In that example, we used roughness=0.1, sharpness=0.5.

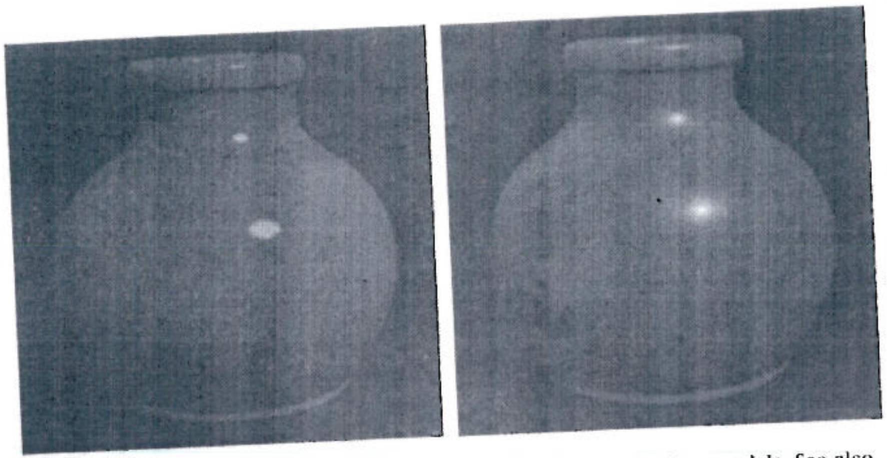


Figure 9.9 Comparing the glossy versus plastic specular illumination models. See also color plate 9.9.

9.6 Light Sources

Previous sections discuss how surface shaders respond to the light energy that arrives at the surface, reflecting in different ways to give the appearance of different types of materials. Now it is time to move to light shaders, which allow the shader author similarly detailed control over the operation of the light sources themselves.

9.6.1 Nongeometric Light Shaders

Light source shaders are syntactically similar to surface shaders. The primary difference is that the shader type is called `light` rather than `surface` and that a somewhat different set of built-in variables is available. The variables available inside light shaders are listed in Table 9.2. The goal of a light shader is primarily to determine the radiant energy `Cl` and light direction `L` of light impinging on `Ps` from this source. In addition, the light may compute additional quantities and store them in its `output` variables, which can be read and acted upon by `illuminance` loops using the `lightsource` statement.

The most basic type of light is an ambient source, which responds in a way that is independent of position. Such a shader is shown in Listing 9.14. The `ambientlight` shader simply sets `Cl` to a constant value, determined by its parameters. It also explicitly sets `L` to zero, indicating to the renderer that there is no directionality to the light.

For directional light, although we could explicitly set `L`, there are some syntactic structures for emitting light in light shaders that help us do the job efficiently. One such syntactic construct is the `solar` statement:

Table 9.2: Global variables available inside light shaders. Variables are read-only except where noted.

`point Ps`	Position of the point *on the surface* that requested data from the light shader.
`vector L`	The vector giving the direction of outgoing light from the source to the point being shaded, `Ps`. This variable can be set explicitly by the shader but is generally set implicitly by the `illuminate` or `solar` statements.
`color Cl`	The light color of the energy being emitted by the source. Setting this variable is the primary purpose of a light shader.

Listing 9.14 Ambient light source shader.

```
light
ambientlight (float intensity = 1;
              color lightcolor = 1;)
{
    Cl = intensity * lightcolor;    /* doesn't depend on position */
    L = 0;                          /* no light direction */
}
```

```
solar ( vector axis; float spreadangle ) {
    statements;
}
```

The effect of the `solar` statement is to send light to every `Ps` from the same direction, given by *axis*. The result is that rays from such a light are parallel, as if the source were infinitely far away. An example of such a source would be the sun.

Listing 9.15 is an example of the `solar` statement. The `solar` statement implicitly sets the `L` variable to its first argument; there is no need to set `L` yourself. Furthermore, the `solar` statement will compare this direction to the light gathering cone of the corresponding `illuminance` statement in the surface shader. If the light's `L` direction is outside the range of angles that the `illuminance` statement is gathering, the block of statements within the `solar` construct will not be executed.

The *spreadangle* parameter is usually set to zero, indicating that the source subtends an infinitesimal angle and that the rays are truly parallel. Values for *spreadangle* greater than zero indicate that a plethora of light rays arrive at each `Ps` from a range of directions, instead of a single ray from a particular direction. Such lights are known as *broad solar lights* and are analogous to very distant but very large area lights. (For example, the sun actually subtends a 1/2 degree angle when seen from Earth.) The exact mechanism for handling these cases may be implementation dependent, differing from renderer to renderer.

Listing 9.15 `distantlight`, a light shader for infinitely distant light sources with parallel rays.

```
light
distantlight ( float intensity = 1;
               color lightcolor = 1;
               point from = point "shader" (0,0,0);
               point to = point "shader" (0,0,1); )
{
    solar (to-from, 0) {
        Cl = intensity * lightcolor;
    }
}
```

Listing 9.16 `pointlight` radiates light in all directions from a particular point.

```
light
pointlight (float intensity = 1;
            color lightcolor = 1;
            point from = point "shader" (0,0,0);)
{
    illuminate (from) {
        Cl = intensity * lightcolor / (L . L);
    }
}
```

For lights that have a definite, finitely close position, there is another construct to use:

```
illuminate ( point from ) {
    statements;
}
```

This form of the `illuminate` statement indicates that light is emitted from position *from* and is radiated in all directions. As before, `illuminance` implicitly sets `L = Ps - from`. Listing 9.16 shows an example of a simple light shader that radiates light in all directions from a point `from` (which defaults to the origin of light shader space[4]). Dividing the intensity by `L.L` (which is the square of the length of `L`) results in what is known as $1/r^2$ falloff. In other words, the energy of light impinging on a surface falls off with the square of the distance between the surface and the light source.

[4] In a light shader, "`shader`" space is the coordinate system that was in effect at the point that the `LightSource` statement appeared in the RIB file.

Listing 9.17 `spotlight` radiates a cone of light in a particular direction.

```
light
spotlight ( float intensity = 1;
            color lightcolor = 1;
            point from = point "shader" (0,0,0);
            point to = point "shader" (0,0,1);
            float coneangle = radians(30);
            float conedeltaangle = radians(5);
            float beamdistribution = 2; )
{
    uniform vector A = normalize(to-from);
    uniform float cosoutside = cos (coneangle);
    uniform float cosinside  = cos (coneangle-conedeltaangle);
    illuminate (from, A, coneangle) {
        float cosangle = (L . A) / length(L);
        float atten = pow (cosangle, beamdistribution) / (L . L);
        atten *= smoothstep (cosoutside, cosinside, cosangle);
        Cl = atten * intensity * lightcolor;
    }
}
```

A second form of `illuminate` also specifies a particular cone of light emission, given by an axis and angle:

```
illuminate ( point from; vector axis; float angle ) {
    statements;
}
```

An example use of this construct can be found in the standard `spotlight` shader, shown in Listing 9.17. The `illuminate` construct will prevent light from being emitted in directions outside the cone. In addition, the shader computes $1/r^2$ distance falloff, applies a cosine falloff to directions away from the central axis, and smoothly fades the light out at the edges of the cone.

Both forms of `illuminate` will check the corresponding `illuminance` statement from the surface shader, which specified a cone axis and angle from which to gather light. If the *from* position is outside this angle range, the body of statements inside the `illuminate` construct will be skipped, thus saving computation for lights whose results would be ignored.

The lights presented here are very simple examples of light shaders, only meant to illustrate the `solar` and `illuminate` constructs. For high-quality image generation, many more controls would be necessary, including more flexible controls over the light's cross-sectional shape, directional and distance falloff, and so on. Such controls will be discussed in detail in Chapter 14.

Table 9.3: Global variables available inside area light shaders. Variables are read-only except where noted.

`point Ps`	Position of the point *on the surface* that requested data from the light shader.
`point P`	Position of the point *on the light*.
`normal N`	The surface normal of the light source geometry (at `P`).
`float u, v, s, t`	The 2D parametric coordinates of `P` (on the light source geometry).
`vector dPdu` `vector dPdv`	The partial derivatives (i.e., tangents) of the light source geometry at `P`.
`vector L`	The vector giving the direction of outgoing light from the source to the point being shaded, `Ps`. This variable can be set explicitly by the shader but is generally set implicitly by the `illuminate` or `solar` statements.
`color Cl`	The light color of the energy being emitted by the source. Setting this variable is the primary purpose of a light shader.

9.6.2 Area Light Sources

Area light sources are those that are associated with geometry (see Section 3.6). Table 9.3 lists the variables available inside area light sources. Many of the variables available to the area light shader describe a position *on the light* that has been selected by the renderer as the point at which the light shader is sampled. You need not worry about how this position is selected—the renderer will do it for you.

Positions, normals, and parameters on the light only make sense if the light is an area light source defined by a geometric primitive. If the light is an ordinary point source rather than an area light, these variables are undefined. Note that *PRMan* does not support area lights. Therefore, in that renderer the light geometry variables are undefined and should not be trusted to contain any meaningful data.

An example light shader that could be used for a simple area light source is given in Listing 9.18. Note the similarity to the `pointlight` shader—the main difference is that, rather than using a `from` parameter as the light position, we illuminate from the position `P` that the renderer chose for us by sampling the area light geometry. This shader illuminates only points on the *outside* of the light source geometry. It would also be fine to simply use `pointlight`, if you wanted the area light geometry to illuminate in all directions.

9.6.3 Shadows

Notice that light shaders are strictly "do-it-yourself" projects. If you want color, you have to specify it. If you want falloff, you need to code it (in the case of `distantlight` we have no distance-based falloff; for `spotlight` and `pointlight` we

Listing 9.18 `arealight` is a simple area light shader.

```
light
arealight (float intensity = 1;
           color lightcolor = 1;)
{
    illuminate (P, N, PI/2) {
        Cl = (intensity / (L.L)) * lightcolor;
    }
}
```

used $1/r^2$ falloff). Similarly, if you want the lights to be shadowed, that also needs to be in the shader.

A number of shadowing algorithms have been developed over the years, and their relative merits depend greatly on the overall rendering architectures. So as not to make undue demands on the renderer, the RenderMan standard provides for the "lowest common denominator": shadow maps. Shadow maps are simple, relatively cheap, very flexible, and can work with just about any rendering architecture.

The shadow map algorithm works in the following manner. Before rendering the main image, we will render separate images *from the vantage points of the lights*. Rather than render RGB color images, these light source views will record depth only (hence the name, *depth map*). An example depth map can be seen in Figure 9.10.

Once these depth maps have been created, we can render the main image from the point of view of the camera. In this pass, the light shader can determine if a particular surface point is in shadow by comparing its distance to the light against that stored in the shadow map. If it matches the depth in the shadow map, it is the closest surface to the light in that direction, so the object receives light. If the point in question is *farther* than indicated by the shadow map, it indicates that some other object was closer to the light when the shadow map was created. In such a case, the point in question is known to be in shadow. Figure 9.10 shows a simple scene with and without shadows, as well as the depth map that was used to produce the shadows.

Shading Language gives us a handy built-in function to access shadow maps:

```
float shadow ( string shadowmapname;  point Ptest; ... )
```

The `shadow()` function tests the point `Ptest` (in `"current"` space) against the shadow map file specified by `shadowmapname`. The return value is 0.0 if `Ptest` is unoccluded and 1.0 if `Ptest` is occluded (in shadow according to the map). The return value may also be between 0 and 1, indicating that the point is in partial shadow (this is very handy for soft shadows).

Like `texture()` and `environment()`, the `shadow()` call has several optional arguments that can be specified as token/value pairs:

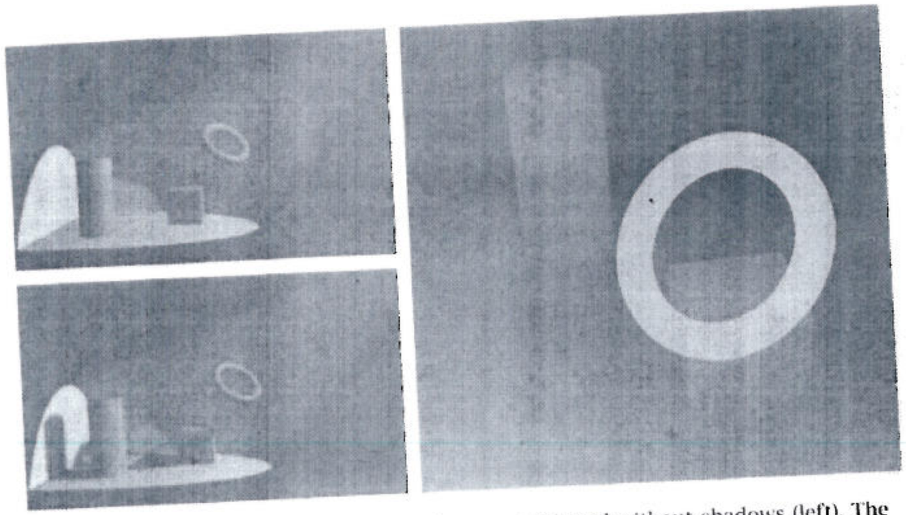


Figure 9.10 Shadow depth maps. A simple scene with and without shadows (left). The shadow map is just a depth image rendered from the point of view of the light source (right). (To visualize the map, we assign white to near depths, black to far depths.)

- `"blur"` takes a `float` and controls the amount of blurring at the shadow edges, as if to simulate the penumbra resulting from an area light source. (See Figure 9.11.) A value of `"blur=0"` makes perfectly sharp shadows; larger values blur the edges. It is strongly advised to add some blur, as perfectly sharp shadows look unnatural and can also reveal the limited resolution of the shadow map.
- `"samples"` is a `float` specifying the number of samples used to test the shadow map. Shadow maps are antialiased by supersampling, so although having larger numbers of samples is more expensive, they can reduce the graininess in the blurry regions. We recommend a minimum of 16 samples, and for blurry shadows it may be quite reasonable to use 64 samples or more.
- `"bias"` is a `float` that *shifts the apparent depth of the objects from the light*. The shadow map is just an approximation, and often not a very good one. Because of numerical imprecisions in the rendering process and the limited resolution of the shadow map, it is possible for the shadow map lookups to incorrectly indicate that a surface is in partial shadow, even if the object is indeed the closest to the light. The solution we use is to add a "fudge factor" to the lookup to make sure that objects are pushed out of their own shadows. Selecting an appropriate bias value can be tricky. Figure 9.12 shows what can go wrong if you select a value that is either too small or too large.
- `"width"` is a `float` that multiplies the estimates of the rate of change of `Ptest` (used for antialiasing the shadow map lookup). This parameter functions analogously to the `"width"` parameter to `texture()` or `environment()`. Its use is

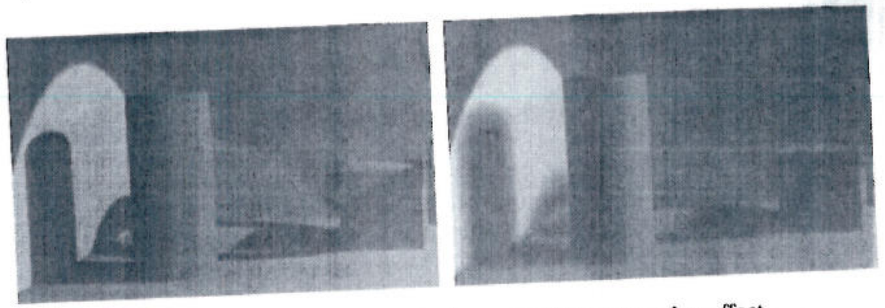


Figure 9.11 Adding blur to shadow map lookups can give a penumbra effect.

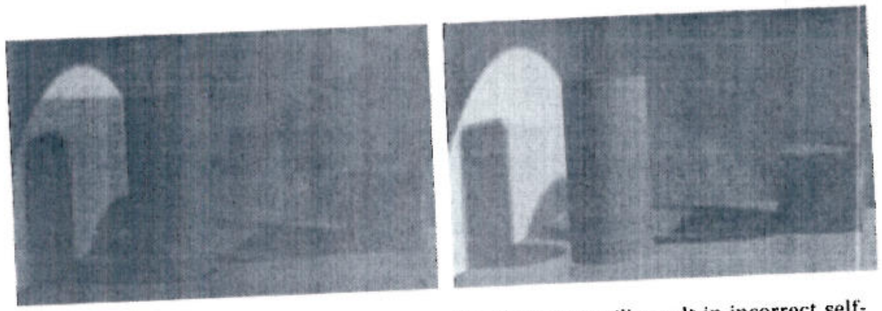


Figure 9.12 Selecting shadow bias. Too small a bias value will result in incorrect self-shadowing (left). Notice the darker, dirtier look compared to Figures 9.11 or 9.10. Too much bias can also introduce artifacts, such as the appearance of "floating objects" or the detached shadow at the bottom of the cylinder (right).

largely obsolete and we recommend using "`blur`" rather than "`width`" to make soft shadow edges.

The `Ptest` parameter determines the point at which to determine how much light is shadowed, but how does the renderer know the point of origin of the light? When the renderer creates a shadow map, it also stores in the shadow file the origin of the camera at the time that the shadow map was made—in other words, the emitting point. The `shadow()` function knows to look for this information in the shadow map file. Notice that since the shadow origin comes from the shadow map file rather than the light shader, it's permissible (and often useful, see Section 14.2.3) for the shadows to be cast from an entirely different position than the point from which the light shader illuminates. Listing 9.19 shows a modification of the `spotlight` shader that uses a shadow map. This light shader is still pretty simple, but the entirety of Chapter 14 will discuss more exotic features in light shaders.

Here are some tips to keep in mind when rendering shadow maps:

Listing 9.19 shadowspot is just like spotlight, but casts shadows using a shadow depth map.

```
light
shadowspot ( float  intensity = 1;
             color  lightcolor = 1;
             point  from = point "shader" (0,0,0);
             point  to = point "shader" (0,0,1);
             float  coneangle = radians(30);
             float  conedeltaangle = radians(5);
             float  beamdistribution = 2;
             string shadowname = "";
             float  samples = 16;
             float  blur = 0.01;
             float  bias = 0.01; )
{
    uniform vector A = normalize(to-from);
    uniform float cosoutside = cos (coneangle);
    uniform float cosinside  = cos (coneangle-conedeltaangle);

    illuminate (from, A, coneangle) {
        float cosangle = (L . A) / length(L);
        float atten = pow (cosangle, beamdistribution) / (L . L);
        atten *= smoothstep (cosoutside, cosinside, cosangle);
        if (shadowname != "") {
            atten *= 1 - shadow (shadowname, Ps, "samples",
                                 samples, "blur", blur, "bias",
                                 bias);
        }
        Cl = atten * intensity * lightcolor;
    }
}
```

- Select an appropriate shadow map resolution. It's not uncommon to use 2k × 2k or even higher-resolution shadow maps for film work.
- View the scene through the "shadow camera" before making the map. Make sure that the field of view is as small as possible, so as to maximize the effective resolution of the objects in the shadow map. Try to avoid your objects being small in the shadow map frame, surrounded by lots of empty unused pixels.
- Remember that depth maps must be one unjittered depth sample per pixel. In other words, the RIB file for the shadow map rendering ought to contain the following options:

```
PixelSamples 1 1
PixelFilter "box" 1 1
Hider "hidden" "jitter" [0]
Display "shadow.z" "zfile" "z"
ShadingRate 4
```

- In shadow maps, only depth is needed, not color. To save time rendering shadow maps, remove all `Surface` calls and increase the number given for `ShadingRate` (for example, as above). If you have surface shaders that displace significantly and those bumps need to self-shadow, you may be forced to run the surface shaders anyway (though you can still remove the lights). Beware!
- When rendering the shadow map, only include objects that will actually cast shadows on themselves or other objects. Objects that only receive, but do not cast, shadows (such as walls or floors) can be eliminated from the shadow map pass entirely. This saves rendering time when creating the shadow map and also eliminates the possibility that poorly chosen bias will cause these objects to incorrectly self-shadow (since they aren't in the maps anyway).
- Some renderers may create shadow map files directly. Others may create only "depth maps" (or "*z* files") that require an additional step to transform them into full-fledged shadow maps (much as an extra step is often required to turn ordinary image files into texture maps). For example, when using *PRMan*, *z* files must be converted into shadow maps as follows:

```
txmake -shadow shadow.z shadow.sm
```

  This command invokes the `txmake` program (*PRMan*'s texture conversion utility) to read the raw depth map file `shadow.z` and write the shadow map file `shadow.sm`.

It is also possible that some renderers (including *BMRT*, but not *PRMan*) support automatic ray-cast shadows that do not require shadow maps at all. In the case of *BMRT*, the following RIB attribute causes subsequently declared `LightSource` and `AreaLightSource` lights to automatically be shadowed:

```
Attribute "light" "shadows" ["on"]
```

There are also controls that let you specify which geometric objects cast shadows (consult the *BMRT* User's Manual for details). Chapter 17 also discusses extensions to Shading Language that allow for ray-cast shadow checks in light shaders.

Further Reading

Early simple local illumination models for computer graphics used Lambertian reflectance. Bui Tuong Phong (Phong, 1975) proposed using a specular illumination model $(L \cdot R)^n$ and also noted that the appearance of faceted objects could be improved by interpolating the vertex normals. Phong's reflection model is still commonly used in simple renderers, particularly those implemented in hardware. Blinn reformulated this model as $(N \cdot H)^n$, with H defined as the angle halfway between L and N. This gives superior results, but for some reason few renderers or graphics boards bother to use this improved version.

The fundamentals of environment mapping can be found in Greene (1986a, 1986b).

The anisotropic specular illumination model that we use came from Ward (1992). The reader is directed to that work for more information on the derivation, details, and use of Greg Ward Larson's model. Additional anisotropic local illumination models can be found in Kajiya (1985) and Poulin and Fournier (1990). Oren and Nayar's generalization of Lambert's law can be found in Oren and Nayar (1994). A similar reflection model for simulation of clouds and dusty and rough surfaces can be found in Blinn (1982). Treatments of iridescence can be found in Smits and Meyer (1989) and Gondek, Meyer, and Newman (1994).

An excellent overall discussion of surface physics, including refraction and the Fresnel equations (derived in all their gory detail), can be found in Hall (1989). This book contains equations and pseudocode for many of the more popular local illumination models. Unfortunately, it has come to our attention that this book is now out of print. Glassner's book (1995) also is an excellent reference on BRDFs.

Additional papers discussing local illumination models include Blinn and Newell (1976), Blinn (1977), Cook and Torrance (1981), Whitted and Cook (1985, 1988), Hall (1986), Nakamae, Kaneda, Okamoto, and Nishita (1990), He, Torrance, Sillion, and Greenberg (1991), Westin, Arvo, and Torrance (1992), Schlick (1993), Hanrahan and Krueger (1993), Lafortune, Foo, Torrance, and Greenberg (1997), and Goldman (1997).

Shadow maps are discussed in Williams (1978) and Reeves, Salesin, and Cook (1987).