shader uses these values to eventually compute an outgoing `Ci`. Such a surface/light interaction scheme makes for very efficient computations and is in fact the only reasonable way to use programmable lights and shaders together.

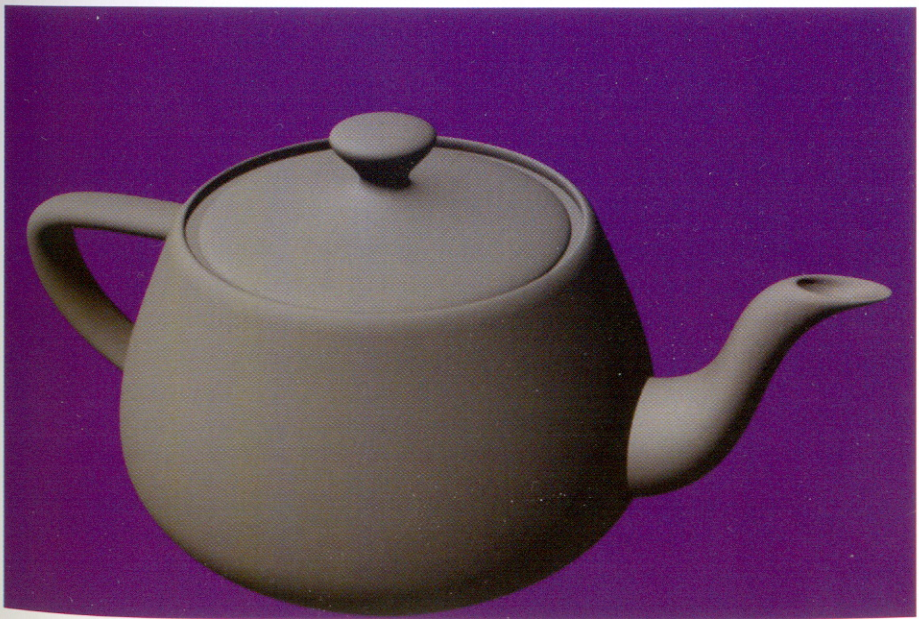
Examples of Using illuminance()

We will now illustrate a variety of uses for `illuminance()`, including the specification of simplified BRDFs. The list of shader variables we will use consists of the usual suspects: `P`, `N`, `I`, `L`, and such. Think of these examples as elements that you can mix and match to come up with your own customized illumination models. In the following subsections, we'll use the terms "lighting model," "illumination model," and "shading model" interchangeably in an informal manner.

diffuse()

As a first example, the `diffuse()` function can be expressed using `illuminance()` as follows (see Figure 11.2):

```
normal Nf = faceforward(normalize(N),I);
color Cdiff=0;
illuminance(P,Nf,PI/2)
{
 vector Ln = normalize(L);
 Cdiff += Cl*Ln.Nf;
}
```



Since the cone angle is PI/2, the illumination cone's influence is twice PI/2, or PI, which is 180 degrees. The color contribution from a single light is proportional to Ln.Nf, the dot product between that light's direction and the surface normal. If the surface normal is parallel to the light, the corresponding point receives full illumination from the light. Conversely, if the normal is perpendicular, the point underneath it receives zero illumination. In between these extreme cases, the 0.0 to 1.0 dot product creates the classic Lambertian ("diffuse") lighting. This represents a perfectly diffuse material, where there is no dependence of Cdiff on the viewing angle. In other words, the Lambertian model amounts to constant BRDF.

From this you can see that the illumination from a straight-on orientation to a sphere will be limited to the front hemisphere, since the back hemisphere is beyond the reach of a front light due to the PI/2 cone angle. But what if we want the light to wrap around and extend beyond the hemisphere? We would be able to simulate the look of an area light this way. In some cases doing so will better integrate a CG element with a live-action background.

Wrap diffuse()

We can modify the Lambertian illumination loop by including a user-specified wrap parameter, which will define how much the light is allowed to reach beyond the hemisphere (see Figure 11.3).

```
surface wdiffuse(float Ka=0.1, Kd=0.5, wrap=1., gam=1)
{
  normal Nf = normalize(faceforward(N,I));
  color Cwd=0;
  float wrp=1-0.5*wrap;
  float wa = acos(wrp*2-1); // wrap angle

  illuminance(P, Nf, wa)
    {
      vector Ln = normalize(L);
      float diffuze;
      float dotp = 0.5*(1+Ln.Nf);
      if(dotp<=wrp)
       {
         diffuze=pow((wrp-dotp)/(wrp),gam);
       }
      else
         diffuze = pow((dotp-wrp)/(1-wrp),gam);
      Cwd += Cl * diffuze;
    }
  Ci = Cs*Os*(Ka*ambient() + Kd*Cwd);
```

Figure 11.3 Wrapped diffuse shading.

To understand how the `wrap` parameter works, we can directly express the wrap angle `wa` in terms of `wrap` by eliminating the `wrp` parameter:

```
wa = acos(1-wrap); // acos((1-0.5*wrap)*2-1)
```

When `wrap` is 1.0 (the default value), we get the usual behavior because the wrap angle `wa` evaluates to 90 degrees (`acos(0)`). Larger values of `wrap`, such as 1.5, increase `wa` to beyond 90 degrees, past the usual hemispherical boundary. At the extreme case of `wrap` being 2.0, the wrap angle becomes 180 degrees (`acos(-1)`), which means that the frontal illumination can reach all the way around to the back of the object. Likewise, `wrap` can also be less than 1.0 to allow us to restrict the illumination to just a portion of the front hemisphere. Non-unity values of the `gam` parameter can be used to alter the wrapping distribution via the `pow()` function by biasing the wrapping of the illumination. This produces the effect of expanding the darker range of illumination values while compressing the brighter range, or vice versa.

Biased diffuse()

We can also make another simple but useful modification to the original Lambert model. As we saw above, in this model `Ln.Nn` provides the diffuse component. The dot product is equivalent to the cosine of the angle between `Ln` and `Nn` (this is always true when the two vectors that form the dot product are normalized). Going from 0 to 90 degrees—that is, head-on to sideways illumination—the profile of the

diffuse illumination is the classic cosine curve. An easy modification to make is to subject the dot product to a `bias()` function to nonlinearly alter the illumination profile. The code looks like this:

```
float biasFunc(float t;float a;)
{
   return pow(t,-(log(a)/log(2)));
}
color bdiff (float bias;)
{
  color C = 0;
  extern point P;
  extern normal N;
  illuminance (P, N, PI/2)
    {
      extern vector L;
      vector Ln = normalize(L);
      normal Nn = normalize(N);
      float cos_theta_i = Ln.Nn;
      C += biasFunc(cos_theta_i,bias);
    }
  return C;
}
```

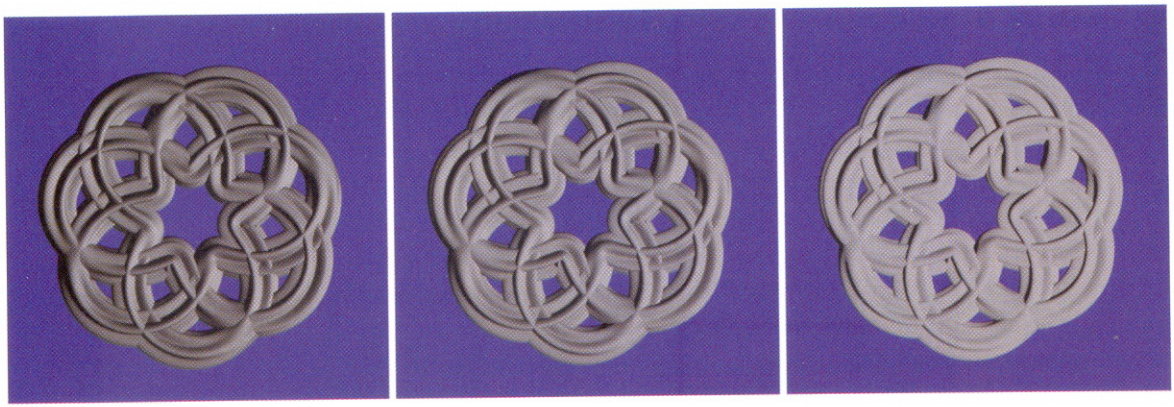


Figure 11.4 Biased diffuse shading.

The results of applying `bias` values of 0.3, 0.5, and 0.8 are shown in Figure 11.4. As you can see, the `bias()` function can be used to exaggerate diffuse illumination or to downplay it.

```
    }
    return C;
}
```

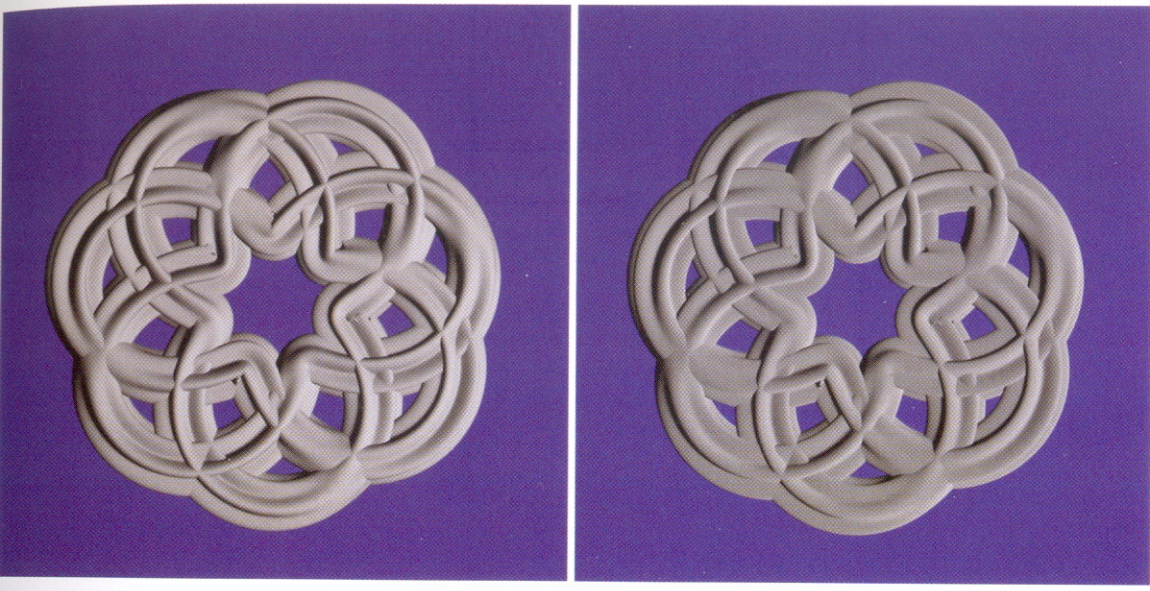


Figure 11.7 Oren-Nayar shading model.

Next, we look at three variations of the Fresnel function—a simple version, Schlick's approximation, and the built-in RSL function, which is the one commonly used in production. The two alternatives to the built-in call are presented just for comparison and study.

Fresnel—Simple Approximation

The dependence on the viewing angle of a material's reflectance/transmittance proportion at an air/solid boundary is called the *Fresnel effect* or *Fresnel phenomenon.* While it is not a full-blown lighting model, it is worth including here along with BRDFs and shading models because it offers one more way to vary shading across a surface.

In accordance with the Fresnel effect, materials such as skin and water are more reflective when viewed edge-on (view direction and surface normal are almost perpendicular) instead of head-on (the directions are nearly parallel). This can be very simply approximated via the term `0.5*(1+Nn.Vn)`. The resulting shading looks brighter at the edges compared to the interior. To control the extent of this effect, the value is raised to a user-supplied exponent (often this is 2). The Fresnel effect is also wavelength dependent, but we ignore this dependency for our purposes.

Such differential edge enhancement can be used to create rim lighting, transparent edges, electron microscope-like images, two-tone paint (by using the value to blend between two colors), and so on. Figure 11.8 shows the Fresnel effect using our simple approximation.

```
color frnl(float INDEX;)
{
  color C = 0;
  extern point P;
  extern normal N;
  extern vector I;

  vector Vn = -normalize(I);
  normal Nn = normalize(N);

  float Kr = 0.5*(1+Nn.Vn);
  Kr = pow(Kr,INDEX);

  C = color(Kr,Kr,Kr);
  return C;
}
```

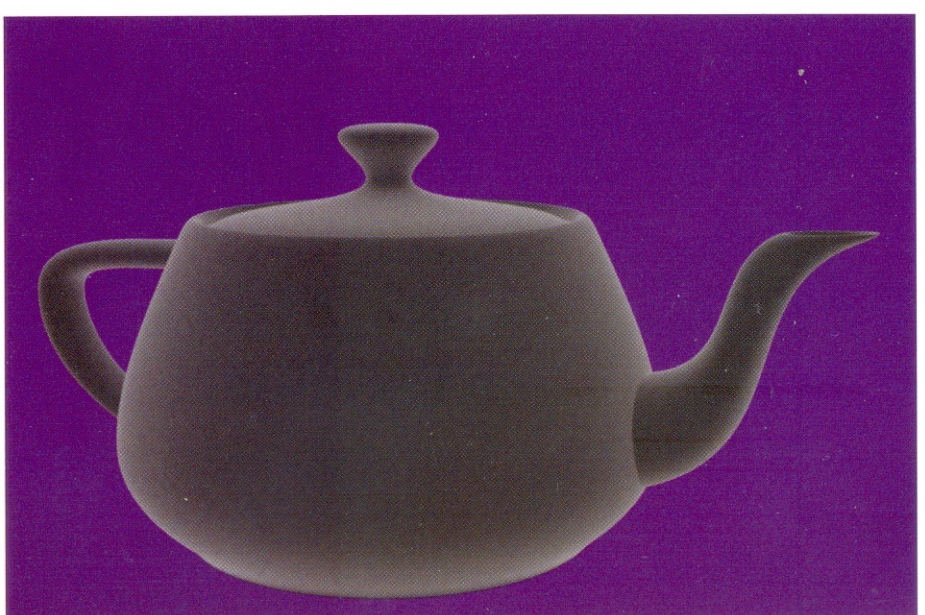


Figure 11.8 Simple approximation of the Fresnel effect.

```
        Cdiff += Cl;
    }

    Oi = Os;
    Ci = Oi * (mix(rootcolor, tipcolor, v) * (Ka*ambient() + Kd*Cdiff) +
        (Ks * Cspec * specularcolor));
}
```

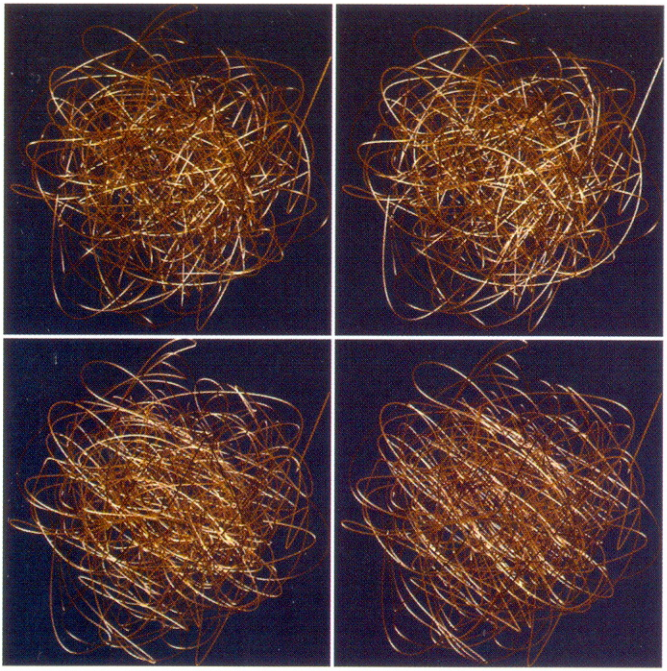


Figure 11.19 Anisotropic shading of a hairball.

Glossy Illumination

The classical Blinn/Phong highlight has a brightness profile across its circular cross section where maximum brightness occurs at the center, tapering off toward the circumference. This is to be expected given that we use a dot product such as `R.V` or `N.H` and control its spread via a `pow()` function. What if we want the highlight to be "flat," relatively uniform across the spot profile? Ceramic coatings, certain types of glass, liquids, and so on exhibit such a featureless highlight profile. Flat highlights are also employed in cartoon rendering (computer-generated or hand-drawn) and poster art.

A glossy look can most easily be achieved using `smoothstep()`, which returns values between 0 and 1 based on three inputs: two values that specify a range and a third selector value. By making the `smoothstep()` function's transition from 0 to 1

sharp (narrow) by specifying a narrow range and the usual Blinn specular value as the selector, we can create a flat profile across our highlight (we are essentially thresholding the highlight). Such an illumination model is presented by Gritz and Apodaca in their *Advanced RenderMan* book. The code in our `Glossy()` function is derived from their `LocIllumGlossy()` function, including the use of magic constants 0.72 and 0.18, which they arrived at empirically. Figure 11.20 shows a Spirograph surface shaded using the `Glossy()` function.

```
color Glossy ( normal N;  vector V; float roughness, sharpness; )
{
    color C = 0;
    float w = .18 * (1-sharpness);
    extern point P;
    illuminance (P, N, PI/2)
      {
        extern vector L;
        extern color Cl;
        vector H = normalize(normalize(L)+V);
        // create a 'flatter' highlight by thresholding the usual specular value
        C += Cl * smoothstep (.72-w, .72+w, pow(max(0,N.H), 1/roughness));
      }
      return C;
}
```

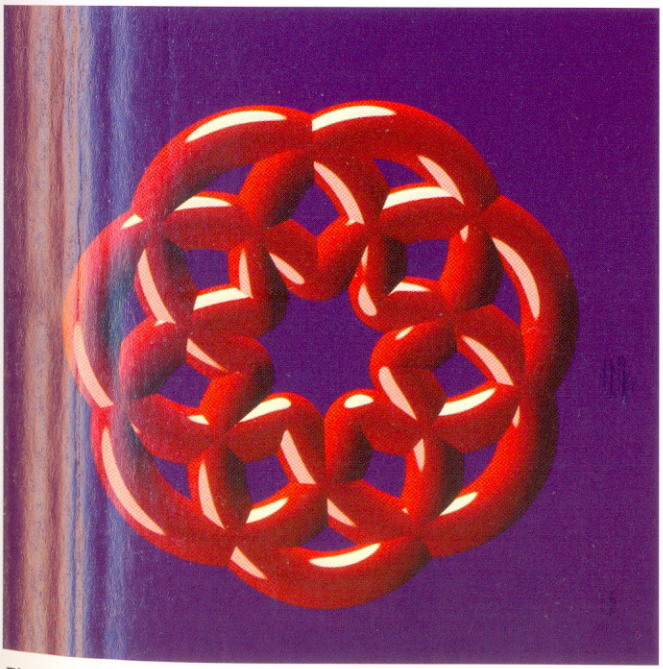


Figure 11.20 Glossy highlights using Ward's technique.