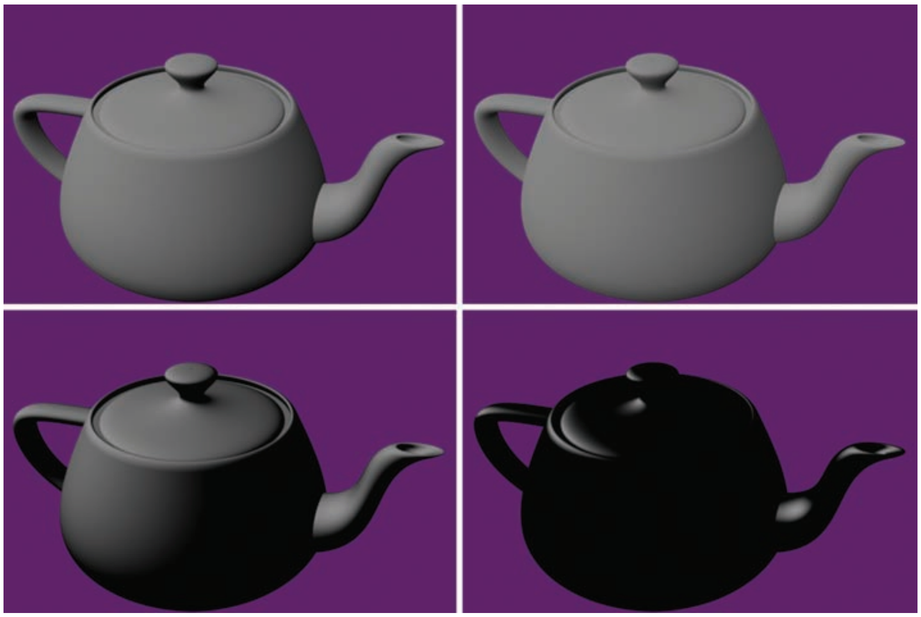

Figure 11.3 Wrapped diffuse shading.

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;
}
```

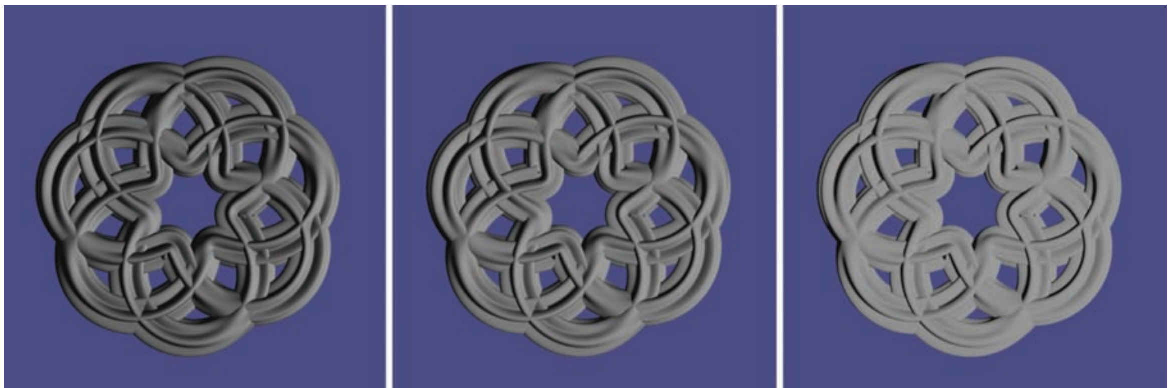

Figure 11.4 Biased diffuse shading.

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.

```
        Cdiff += Cl;
    }

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

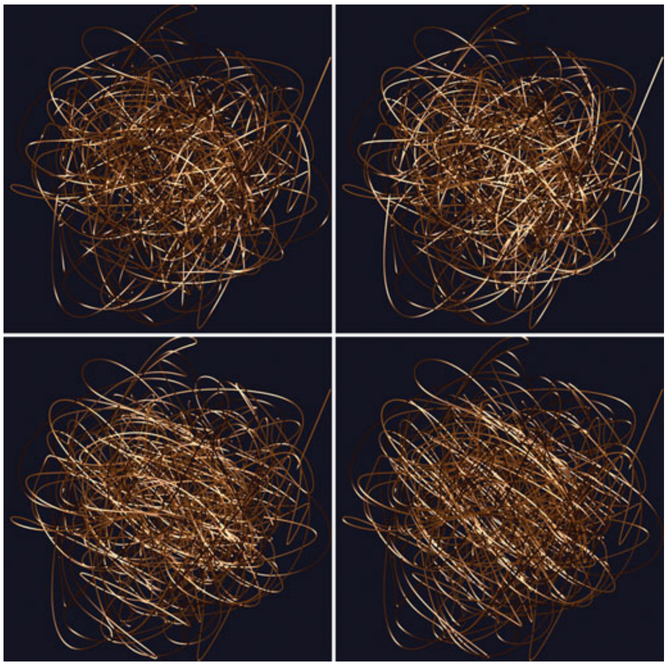
Figure 11.19 Anisotropic shading of a hairball.

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;
}
```

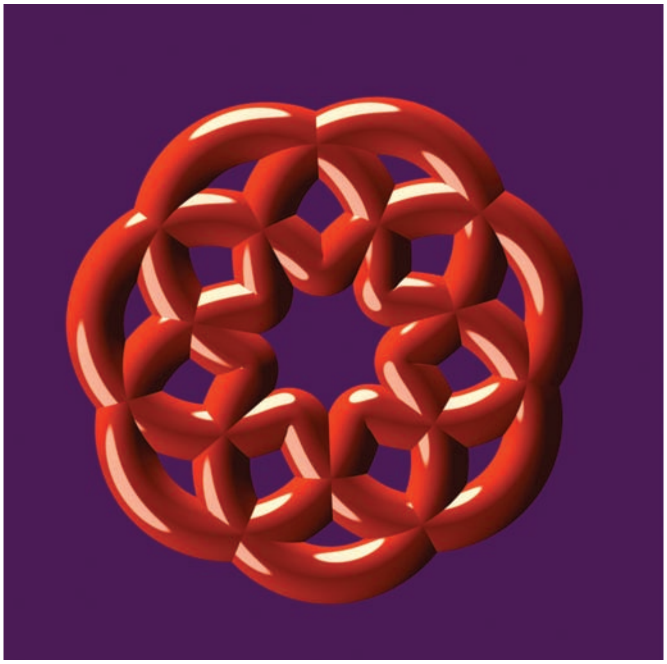

Figure 11.20 Glossy highlights using Ward's technique.

Fresnel—Schlick's Approximation

While the simple approximation to the Fresnel term provided in the previous subsection is adequate, a better approximation that matches well with real-world reflectance measurements was provided by Chris Schlick. Schlick uses a "rational" function approximation where a complex nonlinear curve is expressed using a ratio of polynomial functions that are relatively cheaper to evaluate.

Schlick's Fresnel approximation is that the Fresnel term for arbitrary values of `N` and `V` (both normalized) can be approximated by `Kr = eta + (1-eta)*pow(1-dotnv,5)`, where `dotnv` is `N.V` and `eta` is the value of `Kr` at normal incidence, that is, when `N` and `V` are parallel. By way of verification, you can see that when `N` and `V` are parallel, `dotnv` becomes 1, so `pow(1-dotnv,5)` is 0, therefore `Kr=eta`.

The resulting value `Kr` is further modified via `biasFunc()`, where inputs between 0.0 and 1.0 produce outputs that are also in the range 0.0 to 1.0. `biasFunc()` provides a nonlinear modification of its input and is used here to control the spread of the edge effect toward the interior. You can also use `biasFunc()` in other shaders to nonlinearly modulate color triplets or other parameters whose values lie in the range 0..1. Figure 11.9 shows an image rendered using Schlick's Fresnel approximation.

Note that the parameter we call `eta` needs to be between 0 and 1 to get meaningful results. Given a material's refractive index `n`, `eta` can be calculated as `sqr((n-1)/(n+1))`.

```
float biasFunc(float t;float a;)
{
   return pow(t,-(log(a)/log(2)));
}
color FresnelSchlickFunc(float bias;float eta;float Kfr)
{
  color C = 0;
  extern point P;
  extern normal N;
  extern vector I;

  normal Nn = normalize(N);
  vector Vn = -normalize(I);
  float dotnv = abs(Nn.Vn);
  float Kr = eta + (1-eta)*pow(1-dotnv,5);
  Kr = Kfr*biasFunc(Kr,bias);

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