01 THE IMPLEMENTATION OF A HAIR SCATTERING MODEL

Matt Pharr

16 October 2016

Although the third edition of *Physically Based Rendering* includes an implementation of a shape that allows for efficient intersections of rays with a "flat ribbon" primitive that can be used for modeling fine hair and fur, the system doesn't include any BSDFs or BSSRDFs that model light scattering from hair. (This state of affairs was due to both time and space limits.)

Therefore, here we will describe the implementation of the hair scattering model described in the paper *A Practical and Controllable Hair and Fur Model for Production Path Tracing*, by Matt Jen-Yuan Chiang, Benedikt Bitterli, Chuck Tappan, and Brent Burley. The paper itself is available here: *benedikt-bitterli.me/pchfm/*. See the "Further Reading" section at the end of this document for more information about previous work in scattering from hair (some of which is also incorporated in our implementation here.) Figure 1.1 shows a model of curly hair rendered using this BSDF.

This implementation includes both a new `BxDF` and a new `Material` for hair. Both of these are defined in the files `materials/hair.h` and `materials/hair.cpp`, which are now included in the "master" branch of `pbrt`. A few unit tests are in the file `tests/hair.cpp`. Note that this is a different organization than the rest of `pbrt`, where `BxDF`s are generally defined in `core/reflection.{h,cpp}`. For this extension, we wanted to localize the additions to all-new files as much as possible in order to minimize changes to preexisting parts of the system.

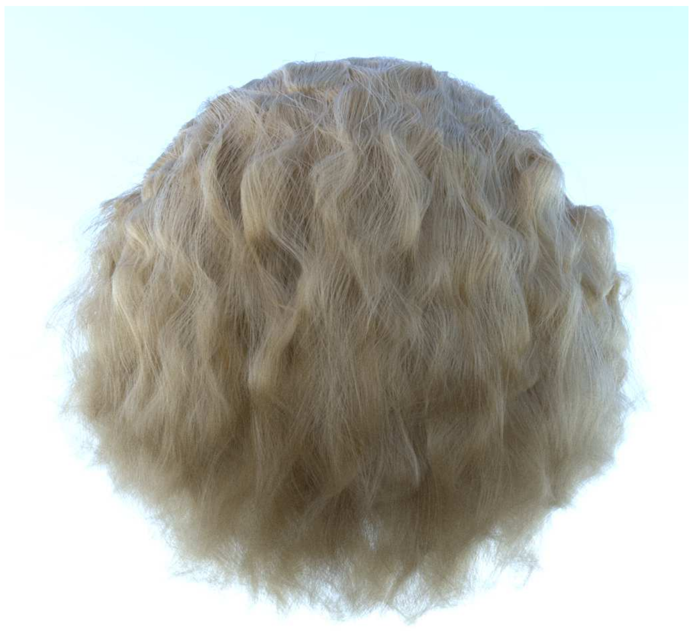

Figure 1.1: Curly hair model represented by nearly 3.3 million `Curve` shapes, rendered using the scattering model described in this document. Path tracing was used to accurately model the effect of multiple scattering; Figure 1.2 shows the difference global illumination makes in hair. For this image, 1024 samples per pixel were used. *(Hair geometry courtesy Cem Yuksel.)*

1.1 GEOMETRY

Before discussing radiometry and light scattering from hair, we'll start by defining some ways of measuring incident and outgoing directions from intersection points on hair.

We will assume that the hair BSDF is only used with the `Curve` shape that was defined in Section 3.7. in the third edition of *Physically Based Rendering*.[1] A `Curve` can represent the shape defined by circle swept along the path of a Bézier curve, giving a generalized cylinder, and provides a reasonably efficient intersection test for this primitive. For the geometric discussion to follow, we'll assume that the `Curve` variant corresponding to a flat ribbon that is always facing the incident ray is being used. However, in the BSDF model, we'll interpret intersection points as if they were on the surface of the swept cylinder. If there is no interpenetration

1 All page and section references in the remainder of this document will refer to the third edition.

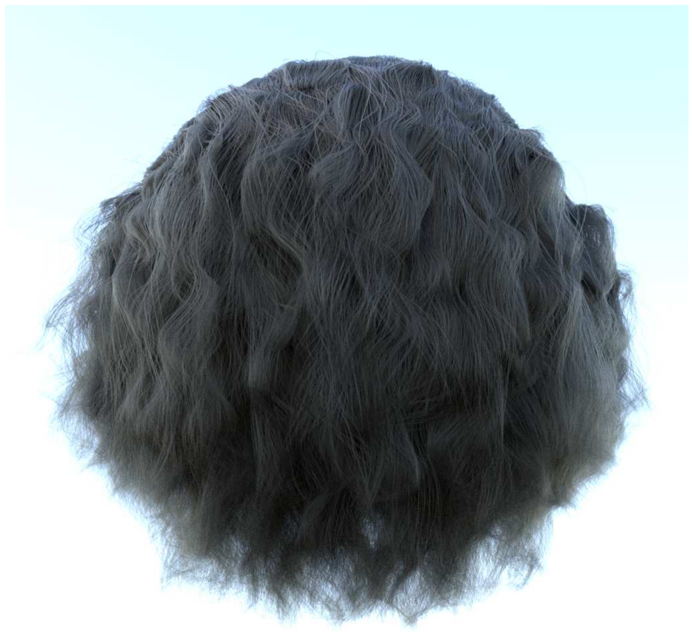

Figure 1.2: The Importance of Multiple Scattering. When the hair model in Figure 1.1 is rendered with direct lighting only, the apparent difference is substantial. Especially for light-colored hair, multiple scattering makes a significant contribution to hair's appearance. *(Hair geometry courtesy Cem Yuksel.)*

between hairs and if the hair's width is not much larger than a pixel's width, there's no harm in switching between these interpretations.

Throughout our implementation of this scattering model, we will regularly find it useful to separately consider scattering in the longitudinal plane, effectively using a side view of the curve, and scattering the azimuthal plane, considering it head-on at a particular point along it. To understand these parameterizations, first recall that `Curve`s are parameterized such that the u direction is along the length of the curve and v spans its width. At a given u, all of the possible surface normals of the curve are given by the surface normals of the circular cross-section at that point. All of these normals lie in a plane; we will call this the *normal plane* (Figure 1.3).

We'll find it useful to represent directions at a ray–curve intersection point with respect to coordinates (θ, ϕ) that are defined with respect to the normal plane at the u position where the ray intersected the curve. The angle θ is the *longitudinal angle*, which is the offset of the ray with respect to the normal plane (Figure 1.4(a)); θ ranges from $-\pi/2$ to $\pi/2$, where $\pi/2$ corresponds to a direction aligned with $\partial \mathrm{p}/\partial u$ and $-\pi/2$ corresponds to $-\partial \mathrm{p}/\partial u$. As explained at the start

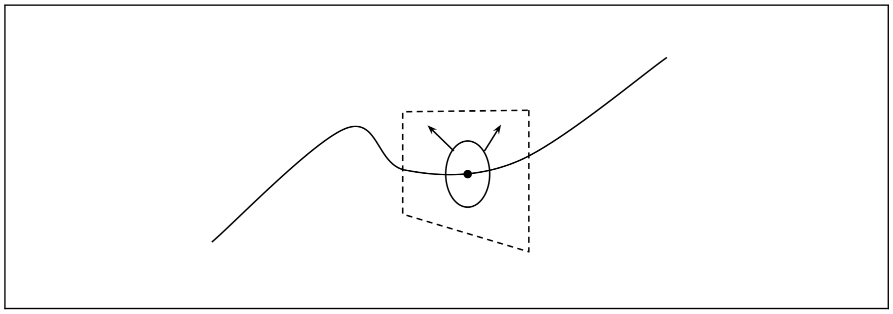

Figure 1.3: At any parametric point u along a `Curve` shape, the cross-section of the curve is defined by a circle. All of the circle's surface normals at u (arrows) lie in a plane (dashed lines), dubbed the "normal plane".

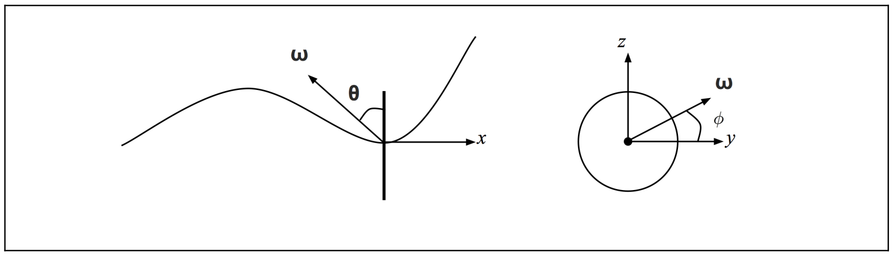

Figure 1.4: (a) Given a direction ω at a point on a curve, the angle θ is defined by the angle between ω and the normal plane at the point (thick line). The curve's tangent vector at the point is aligned with the x axis in the BSDF coordinate system. (b) For a direction ω, the angle ϕ is found by projecting the direction into the normal plane and computing its angle with the y axis, which corresponds to the curve's $\partial \mathrm{p}/\partial v$ in the BSDF coordinate system.

of Chapter 8, in `pbrt`'s regular BSDF coordinate system, $\partial \mathrm{p}/\partial u$ is aligned with the $+x$ axis, so given a direction in the BSDF coordinate system, we have $\sin\theta = \omega_x$, since the normal plane is perpendicular to $\partial \mathrm{p}/\partial u$.

In the BSDF coordinate system, the normal plane is spanned by the y and z coordinate axes. (y corresponds to $\partial \mathrm{p}/\partial v$ for curves, which is always perpendicular to the cure's $\partial \mathrm{p}/\partial u$, and z is aligned with the ribbon normal.) The *azimuthal angle* ϕ is found by projecting a direction ω into the normal plane and computing its angle with the y axis. It thus ranges from 0 to 2π. (See Figure 1.4(b).)

One more measurement with respect to the curve will be useful in the following. Consider incident rays with some direction ω: at any given parametric u value, all such rays that intersect the curve can only possibly intersect one half of the circle swept along the curve (Figure 1.5). We will parameterize the circle's diameter with the variable h, where $h = \pm 1$ corresponds to the ray grazing the edge of the circle, and $h = 0$ corresponds to hitting it edge-on. Because `pbrt` parameterizes curves with v across the curve and $v \in [0, 1]$, we can compute $h = -1 + 2v$.

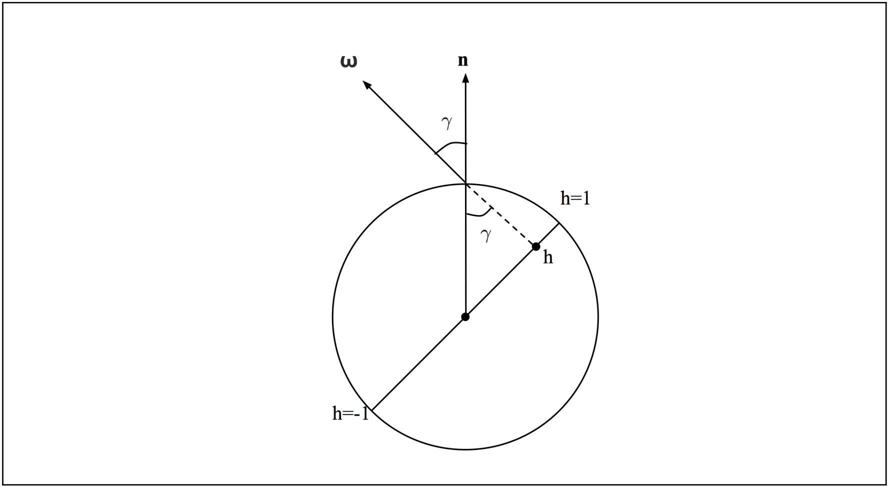

Figure 1.5: Given an incident direction ω of a ray that intersected a `Curve` projected to the normal plane, we can parameterize the curve's width with $h \in [-1, 1]$. Given the h for a ray that has intersected the curve, trigonometry shows how to compute the angle γ between ω and the surface normal on the curve's surface at the intersection point. The two angles γ are equal, and because the circle's radius is 1, $\sin\gamma = h$.

Given the h for a ray intersection, we can compute the angle between the surface normal (which is by definition in the normal plane) and the direction ω, which we'll denote by γ. (Note: this is unrelated to the γ_n notation used for floating-point error bounds in Section 3.9). See Figure 1.5, which shows that $\sin\gamma = h$.

1.2 UTILITY ROUTINES

Before going forward to the hair scattering model implementation, we'll introduce a few utility functions that will be repeatedly useful in the following. (In retrospect, these would have been nice to have included in `pbrt-v3`; they will likely be part of the core system in a future version.)

First, `Sqr()` just computes the square of the given value. Although this function provides trivial functionality, it makes it possible to transcribe equations to code more succinctly than if we did not have this helper.

$\langle$*General Utility Functions*$\rangle \equiv$

```
inline Float Sqr(Float v) { return v * v; }
```

In the following, we will also need to compute relatively large integer powers of floating-point values. Because the integer powers will be compile-time constants, it's possible to use C++ templates to generate much more efficient code to compute these powers than is generally possible using standard library routines.

Consider for example the task of computing the value v^{20}. This computation is equivalently expressed as

$$(((v^2)^2)^2)^2(v^2)^2.$$

Counting up multiplications, we can see that given v, just five multiplies are necessary to compute this value (assuming that the value $(v^2)^2$ is computed once and reused). More generally, raising a value to an integer power n can be done with $O(\log n)$ multiplies.

An obvious (and indeed correct) approach to compute v^{20} would be to call `std::pow(v, 20)`. Doing so is likely to be much less efficient than five multiplies; in general, `std::pow()` is implemented by computing a logarithm, multiplying by the exponent, and then exponentiating. Evaluating these transcendental functions generally requires tens or hundreds of machine instructions, taking much longer than a handful of multiplies. Some `std::pow()` implementations check for small integer exponents and handle them specially, and some compilers detect calls like `std::pow(v, 2)` and directly turn them into multiplies, but neither of these can be depended on.

C++ template functions offer a way to turn exponentiation like this into an efficient series of multiplies. Consider this use of templates:

```
template <int n> Float Pow(Float v) { return v * Pow<n-1>(v); }
template <> Float Pow<0>(Float v) { return 1; }
```

A call like `Pow<5>(v)` will be turned into `v*v*v*v*v`, which can be directly compiled into a series of multiplies. There's a catch, however: recall from Section 3.9 of the third edition that the IEEE floating-point standard prohibits the compiler from reassociating floating-point expressions. Thus, a call `Pow<20>(v)` will be compiled to nineteen multiply operations—likely more efficient than `std::pow()`, but not yet the logarithmic number that's possible assuming we don't mind reassociation.

The following template functions give us a logarithmic number of multiplication operations. The main `Pow()` function splits the exponent in half before recursively calling itself; template specializations handle the base cases.

$\langle$*General Utility Functions*$\rangle \equiv$

```
template <int n>
static Float Pow(Float v) {
    static_assert(n > 0, "Power can't be negative");
    Float n2 = Pow<n / 2>(v);
    return n2 * n2 * Pow<n & 1>(v);
}
template <> Float Pow<1>(Float v) { return v; }
template <> Float Pow<0>(Float v) { return 1; }
```

In benchmarks on a 2016-era laptop, this implementation was 4.6x faster than calling `std::pow` for its uses later in this code.

Though our implementation of `Pow()` is straightforward, it's always a good idea to have a unit test. `pbrt` uses the Google Test framework for unit tests; see the `pbrt-v3` User's Guide for more information. Here, we test integer powers up to 29. The value 2 for `v` is chosen carefully; recall from the discussion on p. 214 of the third edition that with IEEE floating-point arithmetic, multiplication by a factor of two gives an exact result as long as there's no underflow or overflow. Thus, we can reasonably expect exact equality with the integer power of two reference values computed for the test.

⟨*Hair Tests*⟩ ≡

```
TEST(Hair, Pow) {
    EXPECT_EQ(Pow<0>(2.f), 1 << 0);
    EXPECT_EQ(Pow<1>(2.f), 1 << 1);
    EXPECT_EQ(Pow<2>(2.f), 1 << 2);
    ⟨Test remainder of pow template powers to 29⟩
}
```

In the following, we'll need to compute the arcsine of various values. These values may be slightly outside the legal range $[-1, 1]$ due to floating-point round-off error; the `SafeASin()` utility function handles clamping to this range, which makes calling code a bit cleaner. A runtime assertion makes sure that the value provided isn't too far out of the valid range.

⟨*General Utility Functions*⟩ ≡

```
inline Float SafeASin(Float x) {
    CHECK(x >= -1.0001 && x <= 1.0001);
    return std::asin(Clamp(x, -1, 1));
}
```

Similarly, we need to compute the square root of values that may be slightly negative due to round-off error; again, the clamp to the valid range is nice to have in a single place.

⟨*General Utility Functions*⟩ ≡

```
inline Float SafeSqrt(Float x) {
    CHECK_GE(x, -1e-4);
    return std::sqrt(std::max(Float(0), x));
}
```

1.3 SCATTERING FROM HAIR

Geometric setting and utility functions in hand, we will now turn to discuss the general scattering behaviors that give hair its distinctive appearance and some of the assumptions that we'll make in the following.

Hair and fur have three main components:

- Cuticle: the outer layer, which forms the boundary with air. The cuticle's surface is a nested series of scales at a slight angle to the hair surface.
- Cortex: the next layer inside the cuticle. The cortex generally accounts for around 90% of hair's volume but less for fur. It is typically colored with pigments that mostly absorb light.
- Medulla: the center core at the middle of the cortex. It is larger and more significant in thicker hair and fur. The medulla is also pigmented. Scattering

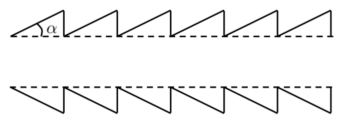

Figure 1.6: The surface of hair is formed by scales that deviate a by a small angle α from the ideal cylinder. (α is generally around $2-4^{\circ}$; the angle shown here is larger for illustrative purposes.)

from the medulla is much more significant than scattering from the medium in the cortex.

For the following model, we'll make a few assumptions. (Approaches for relaxing some of them are discussed in the exercises at the end of this document.) First, we assume that the cuticle can be modeled as a rough dielectric cylinder with scales that are all angled at the same angle α (effectively giving a nested series of cones.) (Figure 1.6.) We also treat the hair interior as a homogeneous medium that only absorbs light—scattering inside the hair is not modeled directly.

We will also make the assumption that scattering can be modeled accurately by a BSDF—we model light as entering and exiting the hair at the same place. (A BSSRDF could certainly be used instead; it's unclear how important subsurface light transport is in practice.) Note that this assumption does require that the hair's diameter be fairly small with respect to how quickly illumination changes over the surface; this assumption is generally fine in practice.

Incident light arriving at a hair may be scattered one more more times before leaving the hair; Figure 1.7 shows a few of the possible cases. We use p to denote the number of path segments it follows inside the hair before being scattered back out to air. We will sometimes refer to terms with a shorthand that describes the corresponding scattering events at the boundary: $p=0$ corresponds to R, for reflection, $p=1$ is TT, for two transmissions $p=2$ is TRT, $p=3$ is TRRT, and so forth.

In the following, we will find it useful to consider these scattering modes separately and so will write the hair BSDF as a sum over terms p:

$$f(\omega_o, \omega_i) = \sum_{p=0}^{\infty} f_p(\omega_o, \omega_i). \tag{1.1}$$

To make the scattering model implementation and importance sampling easier, many hair scattering models factor f into terms where one depends only on the angles θ and another on ϕ, the difference between ϕ_o and ϕ_i. This *semi-separable* model is given by:

$$f_p(\omega_o, \omega_i) = \frac{M_p(\theta_o, \theta_i)\, A_p(\omega_o)\, N_p(\phi)}{|\cos\theta_i|}, \tag{1.2}$$

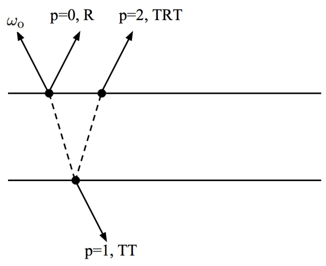

Figure 1.7: Incident light arriving at a hair can be scattered in a variety of ways. $p = 0$ corresponds to light reflected from the surface of the cuticle. Light may also be transmitted through the hair and leave the other side: $p = 1$. It may be transmitted into the hair and reflected back into it again before being transmitted back out: $p = 2$, and so forth.

where we have a *longitudinal scattering function* M_p, an *attenuation function* A_p, and an *azimuthal scattering function* N_p.[2] The division by $|\cos\theta_i|$ cancels out the corresponding factor in the reflection equation.

In the following implementation, we will evaluate the first few terms of the sum in Equation (1.1) and then represent all higher-order terms with a single one. The `pMax` constant controls how many are evaluated before the switch-over.

$\langle$*HairBSDF Constants*$\rangle \equiv$

```
static const int pMax = 3;
```

The model implemented in the `HairBSDF` is parameterized by six values:

- `h`: the $[-1, 1]$ offset along the curve width where the ray intersected the oriented ribbon (h was defined in Section 1.1).
- `eta`: the index of refraction of the interior of the hair. (Typically, 1.55).
- `sigma_a`: the absorption coefficient of the hair interior, where distance is measured with respect to the hair cylinder's diameter.
- `beta_m`: the longitudinal roughness of the hair, mapped to the range $[0, 1]$.
- `beta_n`: the azimuthal roughness, also mapped to $[0, 1]$.
- `alpha`: the angle that the small scales on the surface of hair are offset from the base cylinder, expressed in degrees. (Typically, 2).

2 Other authors generally include A_p in the N_p term, though we find it more clear to keep them separate for the following exposition. Here we also use f for the BSDF, which most hair scattering papers denote by S.

$\langle$*HairBSDF Method Definitions*$\rangle \equiv$

```
HairBSDF::HairBSDF(Float h, Float eta, const Spectrum &sigma_a, Float beta_m,
           Float beta_n, Float alpha)
    : BxDF(BxDFType(BSDF_GLOSSY | BSDF_REFLECTION | BSDF_TRANSMISSION)),
      h(h), gamma0(SafeASin(h)), eta(eta), sigma_a(sigma_a), beta_m(beta_m),
      beta_n(beta_n), alpha(alpha) {
    ⟨Compute longitudinal variance from β_m⟩
    ⟨Compute azimuthal logistic scale factor from β_n⟩
    ⟨Compute α terms for hair scales⟩
}
```

$\langle$*HairBSDF Private Data*$\rangle \equiv$

```
const Float h, gamma0, eta;
const Spectrum sigma_a;
const Float beta_m, beta_n, alpha;
```

We'll proceed to the method that evaluates the BSDF, leaving implementation of the code fragments in the constructor for later, closer to where the values they compute are used.

$\langle$*HairBSDF Method Definitions*$\rangle \equiv$

```
Spectrum HairBSDF::f(const Vector3f &wo, const Vector3f &wi) const {
    ⟨Compute hair coordinate system terms related to wo⟩
    ⟨Compute hair coordinate system terms related to wi⟩
    ⟨Compute cos θ_t for refracted ray⟩
    ⟨Compute γ_t for refracted ray⟩
    ⟨Compute the transmittance T of a single path through the cylinder⟩
    ⟨Evaluate hair BSDF⟩
}
```

There are a few quantities related to the directions ω_o and ω_i that are needed for evaluating the hair scattering model—specifically, the sine and cosine of the angle θ that each direction makes with the plane perpendicular to the curve, and the angle ϕ in the azimuthal coordinate system.

As explained in Section 1.1, $\sin\theta_\mathrm{o}$ is given by the x component of ω_o in the BSDF coordinate system. Given $\sin\theta_\mathrm{o}$, because $\theta_\mathrm{o} \in [-\pi/2, \pi/2]$, we know that $\cos\theta_\mathrm{o}$ must be positive, and so we can compute $\cos\theta_\mathrm{o}$ using the identity $\sin^2\theta + \cos^2\theta = 1$. The angle ϕ_o in the perpendicular plane can be computed with `std::atan`.

$\langle$*Compute hair coordinate system terms related to* `wo`$\rangle \equiv$

```
Float sinThetaO = wo.x;
Float cosThetaO = SafeSqrt(1 - Sqr(sinThetaO));
Float phiO = std::atan2(wo.z, wo.y);
```

Equivalent code, not included here, computes these values for `wi`.

1.3.1 LONGITUDINAL SCATTERING

Onward to M_p, the function that defines the component of scattering related to the angles θ—longitudinal scattering. Longitudinal scattering is responsible for the specular lobe along the length of hair and the longitudinal roughness β_m controls the size of this highlight. Figure 1.8 shows a hair model rendered with three different longitudinal scattering roughnesses.

The model implemented here was developed from d'Eon et al. (2011). The mathematical details of the derivation are complex, so we won't include them here. The goals (which they achieved) were to derive a scattering function that is normalized (ensuring both energy conservation and no energy loss) and can be sampled directly. Although the model isn't derived based on a physical model of how hair scatters light, it matches measured data well and has parametric control of roughness v.

Their model is:

$$M_p(\theta_\mathrm{o}, \theta_\mathrm{i}) = \frac{1}{2v \sinh(1/v)} \, \mathrm{e}^{-\frac{\sin\theta_\mathrm{i} \sin\theta_\mathrm{o}}{v}} \, I_0\left(\frac{\cos\theta_\mathrm{o} \cos\theta_\mathrm{i}}{v}\right), \tag{1.3}$$

where I_0 is the modified Bessel function of the first kind and v is the roughness variance. (Note that this is a different usage of v than earlier in this document when it was used for the parametric coordinate along the width of a curve.) Figure 1.9 shows plots of M_p.

It turns out that that this model isn't numerically stable for low roughness variance values, so d'Eon (2013) derived a different approach for that case that operates on the log of I_0 before taking an exponent at the end. The `v <= .1` test in the implementation below selects between the two formulations.

⟨Hair Local Functions⟩ ≡

```
static Float Mp(Float cosThetaI, Float cosThetaO, Float sinThetaI,
                Float sinThetaO, Float v) {
    Float a = cosThetaI * cosThetaO / v;
    Float b = sinThetaI * sinThetaO / v;
    Float mp = (v <= .1) ?
        (std::exp(LogI0(a) - b - 1/v + 0.6931f + std::log(1 / (2*v)))) :
        (std::exp(-b) * I0(a)) / (std::sinh(1 / v) * 2 * v);
    return mp;
}
```

`I0()` and `LogI0()` compute the values of the modified Bessel function of the first kind its logarithm, respectively. We won't include their implementations here, which are based on numerical approximations to those transcendental functions.

⟨Hair Local Declarations⟩ ≡

```
inline Float I0(Float x), LogI0(Float x);
```

One challenge with this model is choosing a roughness v to achieve a desired look. Here we have implemented a perceptually uniform mapping from roughness $\beta_m \in [0, 1]$ to v where a roughness of 0 is nearly perfectly smooth and 1 is

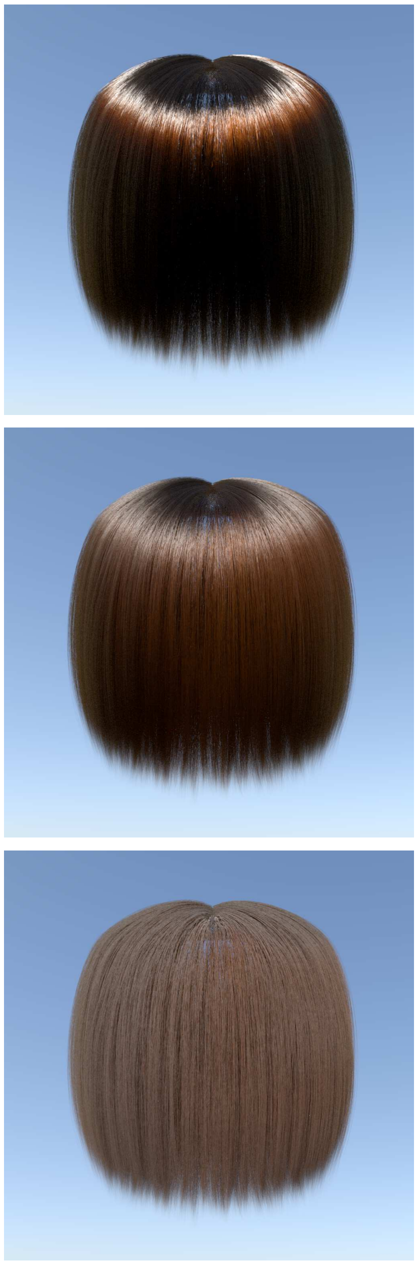

Figure 1.8: The Effect of Varying the Longitudinal Roughness β_m. Hair model illuminated by a skylight environment map rendered with varying longitudinal roughness. (a) With a very low roughness, $\beta_m = 0.1$, the hair appears too shiny—almost metallic. (b) With $\beta_m = 0.25$, the highlight is similar to typical human hair. (c) At high roughness, $\beta_m = 0.6$, the hair is unrealistically flat and diffuse. *(Hair geometry courtesy Cem Yuksel.)*

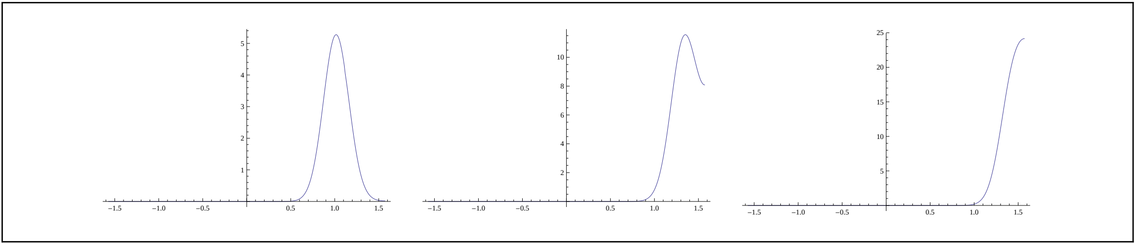

Figure 1.9: Plots of the Longitudinal Scattering Function. The shape of M_p as a function of θ_i for fixed θ_o (-1 radian, -1.3 radians, and -1.4 radians, from left to right). In all cases a roughness variance of $v = 0.02$ was used. Note that for grazing angles, the peak is slightly shifted from the perfect specular reflection direction (at 1, 1.3, and 1.4, respectively.) Note also that the y axis scales are all different, reflecting the functions being normalized. *(After d'Eon et al. (2011), Figure 4.)*

extremely rough. Different roughness values are used for different values of p. For $p = 1$, roughness is reduced by an empirical factor that models the focusing of light due to refraction through the circular boundary of the hair. It is then increased for $p = 2$ and subsequent terms, which models the effect of light spreading out after multiple reflections at the rough cylinder boundary in interior of the hair. (See the "Further Reading" section for more on this variation.)

$\langle$*Compute longitudinal variance from* $\beta_m\rangle \equiv$

```
v[0] = Sqr(0.726f * beta_m + 0.812f * Sqr(beta_m) +
           3.7f * Pow<20>(beta_m));
v[1] = .25 * v[0];
v[2] = 4 * v[0];
for (int p = 3; p <= pMax; ++p)
    v[p] = v[2];
```

$\langle$*HairBSDF Private Data*$\rangle \equiv$

```
Float v[pMax + 1];
```

1.3.2 ABSORPTION IN FIBERS

The A_p term describes how much of the incident light is affected by each of the scattering modes p. It incorporates two effects: Fresnel reflection and transmission at the hair–air boundary and absorption of light that passes through the hair (for $p > 0$). This absorption is what gives hair and fur its color. Figure 1.10 has rendered images of hair with varying absorption coefficients, showing the effect that absorption has. The A_p function that we will implement here models all reflection and transmission at the hair boundary as perfectly specular—a very different assumption that M_p (and N_p to come), which model glossy reflection and transmission. This assumption simplifies the implementation and gives reasonable results in practice (presumably in that the specular paths are in a sense averages over all of the possibly glossy paths.)

We'll start by finding the transmittance of a single transmitted segment through the hair. To do so, we need to find the distance the ray travels until it exits the

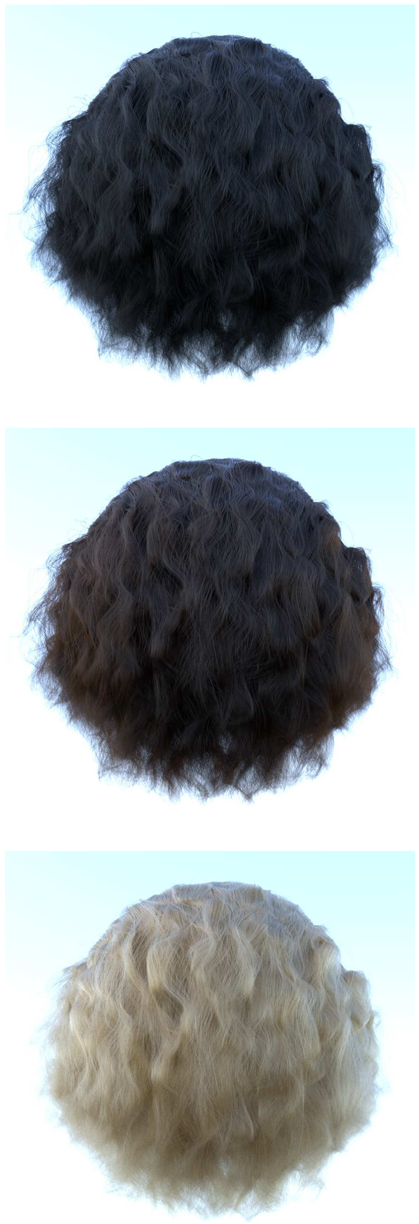

Figure 1.10: Hair Rendered with Various Absorption Coefficients. In all cases, $\beta_m = 0.125$ and $\beta_n = 0.3$. (a) $\sigma_\mathrm{a} = (3.35, 5.58, 10.96)$ (RGB coefficients): in black hair, almost all transmitted light is absorbed. The white specular highlight from the $p = 0$ term is the main visual feature. (b) $\sigma_\mathrm{a} = (0.84, 1.39, 2.74)$, giving brown hair, where the $p > 1$ terms all introduce color to the hair. (c) With a very low absorption coefficient of $\sigma_\mathrm{a} = (0.06, 0.10, 0.20)$, we have blonde hair. *(Hair geometry courtesy Cem Yuksel.)*

cylinder; the easiest way to do this is to compute the distances in the longitudinal and azimuthal projections separately.

To compute these distances, we need the transmitted angles of the ray ω_o, in the longitudinal and azimuthal planes, which we'll denote by θ_t and γ_t, respectively. Application of Snell's law using the hair's index of refraction η allows us to compute $\sin\theta_\mathrm{t}$ and $\cos\theta_\mathrm{t}$.

⟨*Compute* $\cos\theta_\mathrm{t}$ *for refracted ray*⟩ ≡

```
Float sinThetaT = sinThetaO / eta;
Float cosThetaT = SafeSqrt(1 - Sqr(sinThetaT));
```

For γ_t, although we could compute the transmitted direction ω_t from ω_o and then project ω_t into the normal plane, it's possible to compute γ_t directly using a *modified index of refraction* that accounts for the effect of the longitudinal angle on the refracted direction in the normal plane. The modified index of refraction is given by

$$\eta' = \frac{\sqrt{\eta^2 - \sin^2\theta_\mathrm{o}}}{\cos\theta_\mathrm{o}}.$$

Given η', we can compute the refracted direction γ_t directly in the normal plane.[3] Since $h = \sin\gamma_\mathrm{o}$, we can apply Snell's law (p. 546) to compute γ_t.

⟨*Compute* γ_t *for refracted ray*⟩ ≡

```
Float etap = std::sqrt(eta * eta - Sqr(sinThetaO)) / cosThetaO;
Float sinGammaT = h / etap;
Float cosGammaT = SafeSqrt(1 - Sqr(sinGammaT));
Float gammaT = SafeASin(sinGammaT);
```

If we consider the azimuthal projection of the transmitted ray in the normal plane, we can see that the segment makes the same angle γ_t with the circle normal at both of its endpoints (Figure 1.11). If we denote the total length of the segment by l_a, then basic trigonometry tells us that $l_a/2 = \cos\gamma_\mathrm{t}$, assuming a unit radius circle.

Now considering the longitudinal projection, we can see that the distance that a transmitted ray travels before exiting is scaled by a factor of $1/\cos\theta_\mathrm{t}$ as it passes through the cylinder (Figure 1.12). Putting these together, the total segment length in terms of the hair diameter is

$$l = \frac{2\cos\gamma_\mathrm{t}}{\cos\theta_\mathrm{t}}.$$

Recall that for the `HairBSDF` we defined σ_a to be measured with respect to the hair diameter (so that adjusting the hair geometry's width doesn't completely change its color). Therefore, we do not consider the hair cylinder diameter when

3 This is due to the *Bravais properties* of cylindrical scattering. See Appendix B of Marschner et al. (2003) for a nice derivation and further explanation.

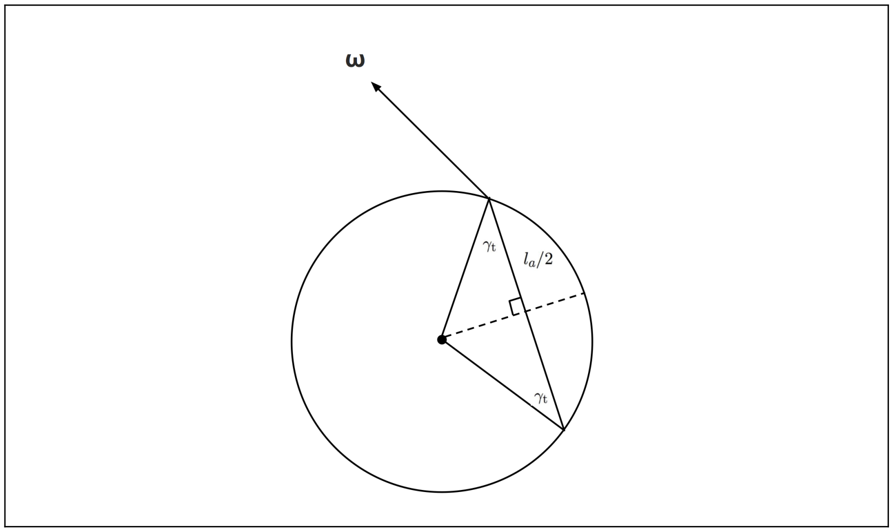

Figure 1.11: Computing the Transmitted Segment's Distance. For a transmitted ray with angle γ_t with respect to the circle's surface normal, half of the total distance l_a is given by $\cos\gamma$, assuming a unit radius. Because γ_t is the same at both halves of the segment, $l_a = 2\cos\gamma_t$.

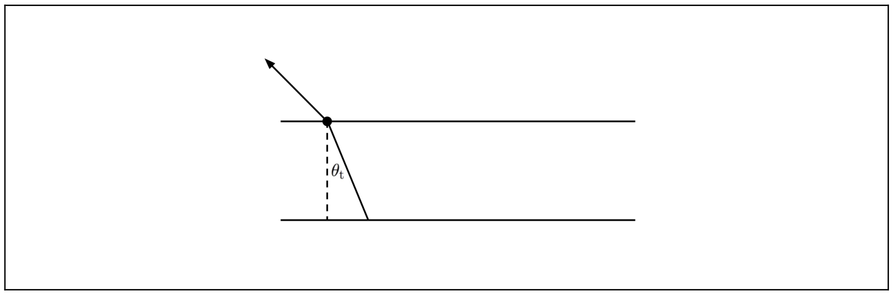

Figure 1.12: The Effect of θ_t on the Transmitted Segment's Length. The length of the transmitted segment through the cylinder is increased by a factor of $1/\cos\theta_t$ versus a direct vertical path.

we apply Beer's law, and transmittance is given by

$$T = \mathrm{e}^{-\sigma_a l}. \tag{1.4}$$

$\langle$*Compute the transmittance* `T` *of a single path through the cylinder*$\rangle \equiv$

```
Spectrum T = Exp(-sigma_a * (2 * cosGammaT / cosThetaT));
```

Given a single segment's transmittance, we can now describe the function that evaluates the full A_p function. `Ap()` returns an array with the values of A_p up to $p_{\max}$ and a final value that accounts for the sums of attenuations for all of the higher-order scattering terms.

⟨*Hair Local Functions*⟩ ≡

```
static std::array<Spectrum, pMax + 1> Ap(Float cosThetaO, Float eta,
                                         Float h, const Spectrum &T) {
    std::array<Spectrum, pMax + 1> ap;
    ⟨Compute p = 0 attenuation at initial cylinder intersection⟩
    ⟨Compute p = 1 attenuation term⟩
    ⟨Compute attenuation terms up to p = pMax⟩
    ⟨Compute attenuation term accounting for remaining orders of scattering⟩
    return ap;
}
```

For the A_0 term, corresponding to light that reflects at the cuticle, the Fresnel reflectance at the air–hair boundary gives the fraction of light that is reflected. We can find the cosine of the angle between the surface normal and the direction vector with angles θ_o and γ_o in the hair coordinate system by $\cos\theta_\mathrm{o}\cos\gamma_\mathrm{o}$.

⟨*Compute $p=0$ attenuation at initial cylinder intersection*⟩ ≡

```
Float cosGammaO = SafeSqrt(1 - h * h);
Float cosTheta = cosThetaO * cosGammaO;
Float f = FrDielectric(cosTheta, 1.f, eta);
ap[0] = f;
```

For the TT term, $p=1$, we have two $1-f$ terms, accounting for transmission into and out of the cuticle boundary, and a single T term for one transmission path through the hair. For all of the $p>0$ terms, which include transmission, we can neglect the scaling of radiance based on the different indices of refraction of the exterior and interior media (recall the discussion of this effect on p. 527): because the viewer and light source are both assumed to be outside the hair, all of those factors cancel out.

⟨*Compute $p=1$ attenuation term*⟩ ≡

```
ap[1] = Sqr(1 - f) * T;
```

The $p=2$ term has one more reflection event, reflecting light back into the hair, and then a second transmission term. Since we assume perfect specular reflection at the cuticle boundary, both segments inside the hair make the same angle γ_t with the circle's normal (Figure 1.13). From this, we can see that both segments must have the same length (and so forth for subsequent segments.) In general, for $p>0$,

$$A_p = (1-f)^2 T^p f^{p-1}.$$

⟨*Compute attenuation terms up to $p=$* `pMax`⟩ ≡

```
for (int p = 2; p < pMax; ++p)
    ap[p] = ap[p - 1] * T * f;
```

After `pMax`, a final term accounts for all further orders of scattering. We'd like to compute the sum of the infinite series of remaining terms, which fortunately can be found in closed form, since both $T<1$ and $f<1$:

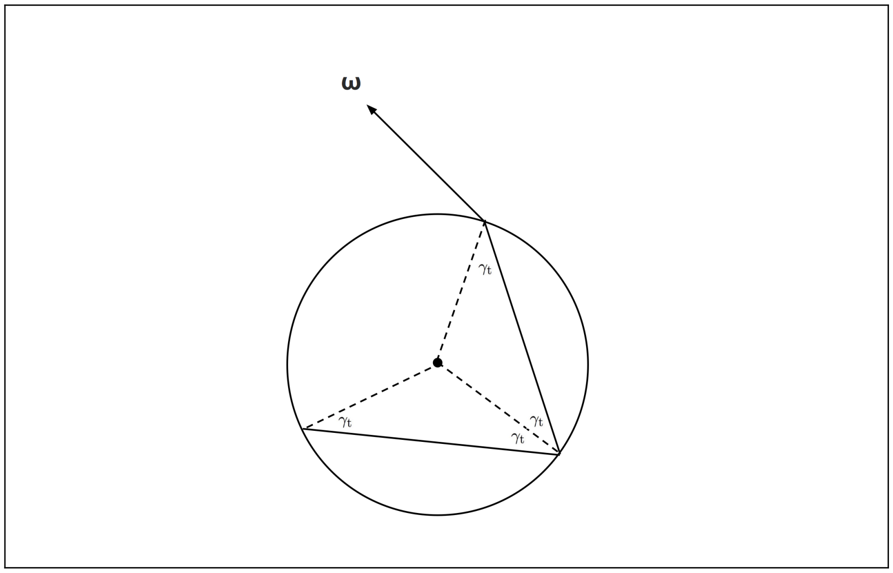

Figure 1.13: When a transmitted ray undergoes specular reflection at the interior of the hair cylinder, it makes the same angle γ_t with the circle's surface normal as the original transmitted ray did. From this, it follows that the lengths of all ray segments for a path inside the cylinder must be equal.

$$\sum_{p=p_{\max}}^{\infty} (1-f)^2 T^p f^{p-1} = \frac{(1-f)^2 T^{p_{\max}} f^{p_{\max}-1}}{1-Tf}.$$

⟨*Compute attenuation term accounting for remaining orders of scattering*⟩ ≡

```
ap[pMax] = ap[pMax - 1] * f * T / (Spectrum(1.f) - T * f);
```

1.3.3 AZIMUTHAL SCATTERING

Finally, we will model the component of scattering dependent on the angle ϕ. We will do this work entirely in the normal plane. The azimuthal scattering model is based on first computing a new azimuthal direction assuming perfect specular reflection and transmission and then defining a distribution of directions around this central direction, where increasing roughness gives a wider distribution. Therefore, we will first consider how an incident ray is deflected by specular reflection and transmission in the normal plane; Figure 1.14 illustrates the cases for the first two values of p.

Following the reasoning from Figure 1.14, we can derive the function Φ, which gives the net change in azimuthal direction:

$$\Phi(p,h) = 2p\gamma_t - 2\gamma_o + p\pi.$$

(Recall that γ_o and γ_t are derived from h.) Figure 1.15 shows a plot of this function for $p = 1$.

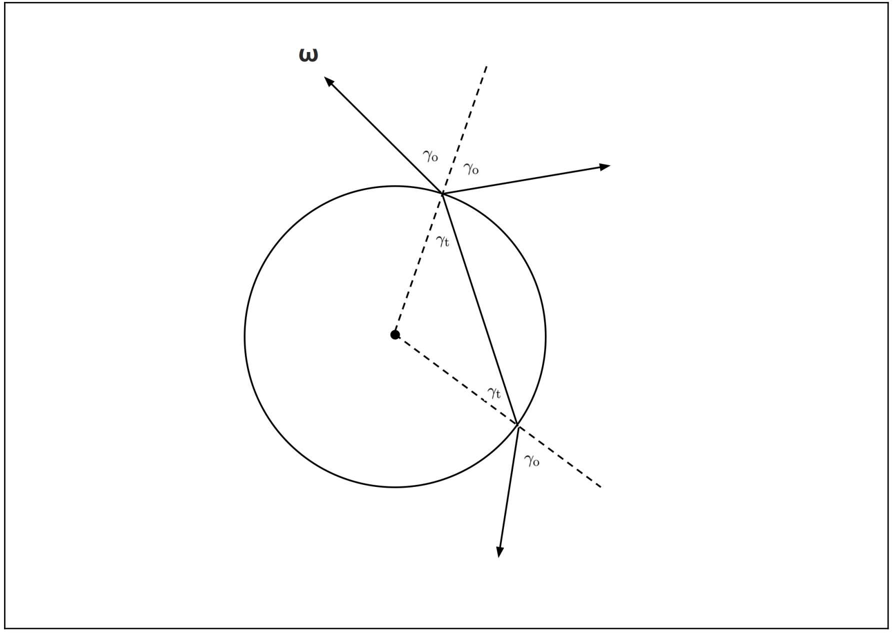

Figure 1.14: For specular reflection, with $p = 0$, the incident and reflected directions make the same angle γ_o, with the surface normal. The net change in angle is thus $-2\gamma_o$. For $p = 1$, the ray is deflected from γ_o to γ_t when it enters the cylinder and then correspondingly on the way out. We can also see that when the ray is transmitted again out of the circle, it again makes an angle γ_o with the surface normal there. Adding up the angles, the net deflection is $2\gamma_t - 2\gamma_o + \pi$.

⟨*Hair Local Functions*⟩ ≡
```
inline Float Phi(int p, Float gammaO, Float gammaT) {
    return 2 * p * gammaT - 2 * gammaO + p * Pi;
}
```

Now that we know how to compute new angles in the normal plane after specular transmission and reflection, we need a way to represent surface roughness, so that a range of directions centered around the specular direction can contribute to scattering. The *logistic distribution* provides a good option: it is a generally useful one for rendering, since it has a similar shape to the Gaussian (which of course comes up often in rendering), while also being normalized and integrable in closed-form (unlike the Gaussian).

The logistic distribution takes a scale factor s, which controls its width:

$$l(x, s) = \frac{\mathrm{e}^{-x/s}}{s(1 + \mathrm{e}^{-x/s})^2}.$$

The implementation is straightforward, though it is worth taking the absolute value of x to avoid numerical instability for when the ratio x/s is relatively

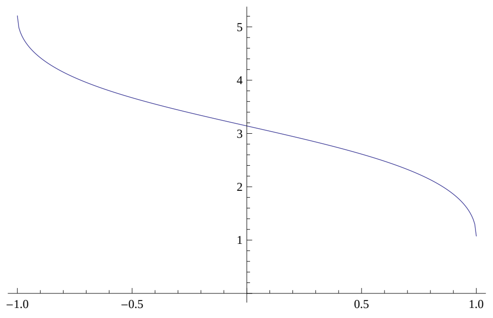

Figure 1.15: Plot of $\Phi(p, h)$ For $p = 1$. As h varies from -1 to 1, we can see that the range of orientations ϕ for the specularly transmitted ray varies rapidly. By examining the range of ϕ values, we can see that the possible transmitted directions cover roughly $2/3$ of all possible directions on the circle.

large. (The function is symmetric around the origin, so this is mathematically equivalent.)

$\langle$*Hair Local Functions*$\rangle \equiv$

```
inline Float Logistic(Float x, Float s) {
    x = std::abs(x);
    return std::exp(-x / s) / (s * Sqr(1 + std::exp(-x / s)));
}
```

Because the logistic distribution is normalized, it is its own PDF. Its integral is reasonably straightforward:

$$\int l(x, s)\,\mathrm{d}x = \frac{1}{1 + \mathrm{e}^{-x/s}}, \tag{1.5}$$

and the function that implements its CDF follows directly.

$\langle$*Hair Local Functions*$\rangle \equiv$

```
inline Float LogisticCDF(Float x, Float s) {
    return 1 / (1 + std::exp(-x / s));
}
```

In the following, we'll find it useful to define a normalized logistic function over a range $[a, b]$; we'll call this the *trimmed logistic*, l_t. (In practice, we'll always use the range $[-\pi, \pi]$, but will derive the next few functions for arbitrary ranges for flexibility.)

$$l_\mathrm{t}(x, s, [a, b]) = \frac{l(x, s)}{\int_a^b l(x', s)\,\mathrm{d}x'}.$$

The implementation follows directly using Equation (1.5). Figure 1.16 shows plots of the trimmed logistic distribution for a few values of s.

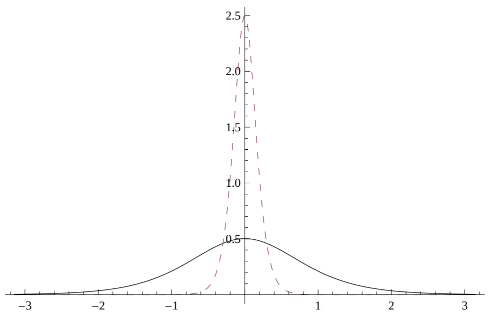

Figure 1.16: Plots of The Trimmed Logistic Function Over $\pm\pi$. The curve for $s = 0.5$ (solid line) is broad and flat, while at $s = 0.1$ (dashed line), the curve is peaked. Because the function is normalized, the peak at 0 generally doesn't have the value 1, unlike the Gaussian.

$\langle$*Hair Local Functions*$\rangle \equiv$

```
inline Float TrimmedLogistic(Float x, Float s, Float a, Float b) {
    return Logistic(x, s) / (LogisticCDF(b, s) - LogisticCDF(a, s));
}
```

Now we have the pieces to be able to implement the azimuthal scattering distribution. The `Np()` function computes the N_p term, computing the angular difference between ϕ and $\Phi(p, h)$ and evaluating the azimuthal distribution with that angle.

$\langle$*Hair Local Functions*$\rangle \equiv$

```
inline Float Np(Float phi, int p, Float s, Float gamma0,
                Float gammaT) {
    Float dphi = phi - Phi(p, gamma0, gammaT);
    ⟨Remap dphi to [−π, π]⟩
    return TrimmedLogistic(dphi, s, -Pi, Pi);
}
```

The difference between ϕ and $\Phi(p, h)$ may be outside the range we've defined the logistic over, $[-\pi, \pi]$, so we rotate around the circle as needed to get the value to the right range. Because `dphi` never gets too far out of range for the small p used here, we just use the simple approach of adding or subtracting 2π as needed.

$\langle$*Remap* `dphi` *to* $[-\pi, \pi]\rangle \equiv$

```
while (dphi > Pi) dphi -= 2 * Pi;
while (dphi < -Pi) dphi += 2 * Pi;
```

As with the longitudinal roughness, it's helpful to have a roughly perceptually linear mapping from azimuthal roughness $\beta_n \in [0, 1]$ to the logistic scale factor s.

$\langle$*Compute azimuthal logistic scale factor from* $\beta_n\rangle \equiv$

```
s = SqrtPiOver8 * (0.265f * beta_n + 1.194f * Sqr(beta_n) +
                   5.372f * Pow<22>(beta_n));
```

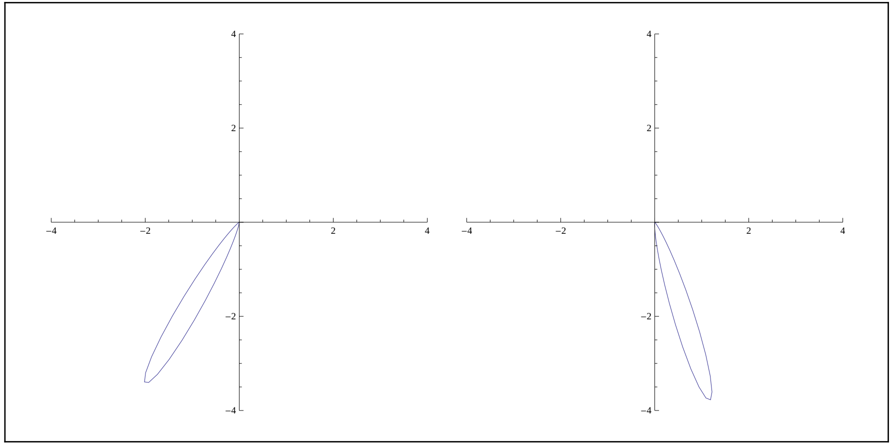

Figure 1.17: Polar plots of N_p for $p = 1$ with a low roughness, $\beta_n = 0.1$, for $h = -0.5$ (left) and $h = 0.3$ (right). We can see that N_p varies rapidly over the width of the hair.

The mapping uses the constant $\sqrt{\pi/8}$.

$\langle$*HairBSDF Constants*$\rangle \equiv$

```
static const Float SqrtPiOver8 = 0.626657069f;
```

$\langle$*HairBSDF Private Data*$\rangle \equiv$

```
Float s;
```

Figure 1.17 shows polar plots of azimuthal scattering for the TT term, $p = 1$, with a fairly low roughness. The scattering distributions for the two different points on the curve's width are quite different. Because we expect the hair width to be roughly pixel-sized, many rays per pixel are needed to resolve this variation well.

Figure 1.18 shows renderings of the hair model from Figure 1.1 with a fixed longitudinal roughness and varying azimuthal roughness. We can see that higher azimuthal roughness causes the hair to be lighter in color; this is because more light is able to exit the hair after multiple scattering when the distribution is broader.

1.3.4 SCATTERING MODEL IMPLEMENTATION

We now have almost all of the pieces we need to be able to evaluate the model. The last detail is to account for the effect of scales on the hair surface (recall Figure 1.6). Suitable adjustments to θ_i work well to model this characteristic of hair.

For the R term, adding the angle 2α to θ_i can model the effect of evaluating the hair scattering model with respect to the surface normal of a scale. We can then go ahead and evaluate M_0 with this modification to θ_i. For TT, we have to account for two transmission events through scales. Rotating by α in the opposite direction approximately compensates. (Because the refraction angle is

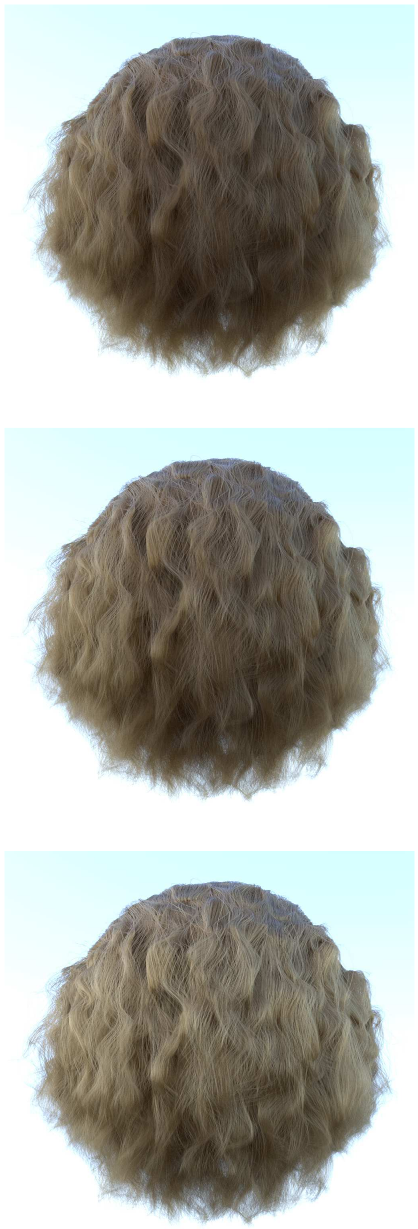

Figure 1.18: Hair Rendered With Varying Azimuthal Roughness. Top $\beta_n = 0.3$, middle: $\beta_n = 0.6$, and bottom: $\beta_n = 0.9$. In all cases, $\beta_m = 0.3$. As the longitudinal roughness increases, the hair lightens, as more multiply-scattered light can exit the hair volume. *(Hair geometry courtesy Cem Yuksel.)*

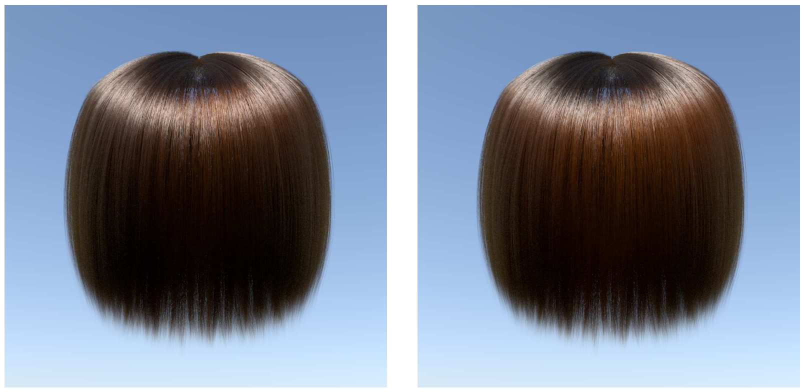

Figure 1.19: The Effect of Scales on Hair. (a) Hair rendered without modeling scales on the hair surface, $\alpha = 0$. (b) With $\alpha = 2^\circ$, the specular highlights from the $p = 0$ term (white) and $p = 2$ (hair colored) are distinct, and we can now see a secondary hair-colored highlight below the white highlight. *(Hair geometry courtesy Cem Yuksel.)*

non-linear with respect to changes in normal orientation, there is some error in this approximation, though the error is low for the typical case of small values of α.) TRT has a reflection term inside the hair; a rotation by -4α compensates for the overall effect.

The effects of these shifts are that the primary reflection lobe R is offset to be above the perfect specular direction and the secondary TRT lobe is shifted below it. Together, these lead to two distinct specular highlights of different colors, since R isn't affected by the hair's color, while TRT picks up the hair color due to absorption. This effect can be seen in human hair. Figure 1.19 shows the visual result of accounting for the tilted scales.

To support computing these offsets, in the `HairBSDF` constructor we precompute $\sin 2^k\alpha$ and $\cos 2^k\alpha$ for $k = 0, 1, 2$. These values can be computed particularly efficiently using trigonometric double angle identities: $\cos 2\theta = \cos^2\theta - \sin^2\theta$ and $\sin 2\theta = 2\cos\theta\sin\theta$.

$\langle$*Compute α terms for hair scales*$\rangle \equiv$

```
sin2kAlpha[0] = std::sin(alpha);
cos2kAlpha[0] = SafeSqrt(1 - Sqr(sin2kAlpha[0]));
for (int i = 1; i < 3; ++i) {
    sin2kAlpha[i] = 2 * cos2kAlpha[i - 1] * sin2kAlpha[i - 1];
    cos2kAlpha[i] = Sqr(cos2kAlpha[i - 1]) - Sqr(sin2kAlpha[i - 1]);
}
```

$\langle$*HairBSDF Private Data*$\rangle \equiv$

```
Float sin2kAlpha[3], cos2kAlpha[3];
```

Evaluating the model is now mostly just a matter of calling functions that have already been defined and summing the individual terms f_p.

⟨*Evaluate hair BSDF*⟩ ≡

```
Float phi = phiI - phiO;
std::array<Spectrum, pMax + 1> ap = Ap(cosThetaO, eta, h, T);
Spectrum fsum(0.);
for (int p = 0; p < pMax; ++p) {
    ⟨Compute sin θi and cos θi terms accounting for scales⟩
    fsum += Mp(cosThetaIp, cosThetaO, sinThetaIp, sinThetaO, v[p]) * ap[p] *
            Np(phi, p, s, gammaO, gammaT);
}
⟨Compute contribution of remaining terms after pMax⟩
if (AbsCosTheta(wi) > 0) fsum /= AbsCosTheta(wi);
return fsum;
```

As discussed earlier, θ_i is rotated to model the effect of hair scales. Fortunately we only need the sine and cosine of the angle θ_i to evaluate M_p. We can therefore use the trigonometric identities

$$\sin\theta \pm \alpha = \sin\theta\cos\alpha \pm \cos\theta\sin\alpha$$
$$\cos\theta \pm \alpha = \cos\theta\cos\alpha \mp \sin\theta\sin\alpha$$

to efficiently compute the rotated angle, without needing to evaluate any additional trigonometric functions.

Here we only include the case for $p = 0$, where θ_i is rotated by 2α. The remaining cases follow the same structure. (For $p = 1$, the rotation is by $-\alpha$ and for $p = 2$, -4α.)

⟨*Compute* $\sin\theta_\mathrm{i}$ *and* $\cos\theta_\mathrm{i}$ *terms accounting for scales*⟩ ≡

```
Float sinThetaIp, cosThetaIp;
if (p == 0) {
    sinThetaIp = sinThetaI * cos2kAlpha[1] + cosThetaI * sin2kAlpha[1];
    cosThetaIp = cosThetaI * cos2kAlpha[1] - sinThetaI * sin2kAlpha[1];
}
⟨Handle remainder of p values for hair scale tilt⟩
⟨Handle out-of-range cos θi from scale adjustment⟩
```

When ω_i is nearly parallel with the hair, the scale adjustment may give a slightly negative value for $\cos\theta_\mathrm{i}$—effectively, in this case, it represents a θ_i that is slightly greater than $\pi/2$, the maximum expected value of θ in the hair coordinate system. This angle is equivalent to $\pi - \theta_\mathrm{i}$, and $\cos(\pi - \theta_\mathrm{i}) = |\cos\theta_\mathrm{i}|$, so we can easily handle that here.

⟨*Handle out-of-range* $\cos\theta_\mathrm{i}$ *from scale adjustment*⟩ ≡

```
cosThetaIp = std::abs(cosThetaIp);
```

A final term accounts for all higher-order scattering inside the hair. We just use a uniform distribution $N(\phi) = 1/(2\pi)$ for the azimuthal distribution; this is a reasonable choice, as the varied direction offsets from $\Phi(p, h)$ for $p \geq p_\mathrm{max}$

generally have wide variation and the final A_p term generally represents less than 15% of the overall scattering, so little error is introduced in the final result.

⟨*Compute contribution of remaining terms after* `pMax`⟩ ≡

```
fsum += Mp(cosThetaI, cosThetaO, sinThetaI, sinThetaO, v[pMax]) *
        ap[pMax] / (2.f * Pi);
```

1.3.5 THE “WHITE FURNACE” TEST

We have, we hope, implemented a hair scattering model wherein if hair doesn't absorb any of the light passing through it (i.e., $\sigma_a = 0$), then all of the incident light should be reflected. If such a hair is illuminated with uniform incident radiance, the reflected radiance should be exactly the same as the incident radiance. The *white furnace test* checks this, making sure that reflected radiance is one given unit incident radiance. Our implementation tests a variety of azimuthal and longitudinal roughnesses.

⟨*Hair Tests*⟩ ≡

```
TEST(Hair, WhiteFurnace) {
    RNG rng;
    Vector3f wo = UniformSampleSphere({rng.UniformFloat(),
                                       rng.UniformFloat()});
    for (Float beta_m = .1; beta_m < 1; beta_m += .2) {
        for (Float beta_n = .1; beta_n < 1; beta_n += .2) {
            ⟨Estimate reflected uniform incident radiance from hair⟩
        }
    }
}
```

For each roughness, we compute a Monte Carlo estimate of the spherical–directional reflectance,

$$\int_{\mathcal{S}^2} f(\omega_o, \omega_i)\, |\cos\theta_i|\, d\omega_i.$$

Each sample is evaluated by first sampling a random offset along the hair h and then computing the fraction of reflected radiance for a random incident direction.

⟨*Estimate reflected uniform incident radiance from hair*⟩ ≡

```
Spectrum sum = 0.f;
int count = 300000;
for (int i = 0; i < count; ++i) {
    Float h = -1 + 2. * rng.UniformFloat();
    Spectrum sigma_a = 0.f;
    HairBSDF hair(h, 1.55, sigma_a, beta_m, beta_n, 0.f);
    Vector3f wi = UniformSampleSphere({rng.UniformFloat(),
                                       rng.UniformFloat()});
    sum += hair.f(wo, wi) * AbsCosTheta(wi);
}
Float avg = sum.y() / (count * UniformSpherePdf());
EXPECT_TRUE(avg >= .95 && avg <= 1.05);
```

The `WhiteFurnaceSampled` test, not included here, uses the `Sample_f()` method (which will be defined shortly) to sample incident directions rather than a uniform distribution, dividing BSDF values by the PDF to compute the estimate of reflectance. It uses a tighter tolerance ($1 \pm .01$) than the first white furnace test, since importance sampling gives much faster convergence than uniform sampling over the sphere. (Note, however, that it's useful to have both variants of this test: if there was a bug in the importance sampling code and the second white furnace test failed, we wouldn't know whether the bug was in the sampling code or how the hair BSDF computed reflection. This way, if one white furnace test fails or both fail, we can have a better idea of whether the underlying problem is in the evaluation of the model or in the code that samples it.)

1.4 IMPORTANCE SAMPLING

Being able to generate sampled directions and compute the PDF for sampling a given direction according to a distribution that is similar to the overall BSDF is critical for efficient rendering, especially at low roughnesses, where the hair BSDF varies rapidly as a function of direction. In the approach implemented here, samples are generated with a two step process: first we choose a p term to sample according to a probability based on each term's A_p function value, which gives its contribution to the overall scattering function. Then, we find a direction by sampling the corresponding M_p and N_p terms. Fortunately, both the M_p and N_p terms of the hair BSDF can be sampled perfectly, leaving us with a sampling scheme that exactly matches the PDF of the full BSDF.

1.4.1 COMPUTING ADDITIONAL SAMPLE VALUES

In the following, we'll need a total of four random samples to sample the direction `wi`. This presents a challenge in that only two sample values are passed into `pbrt`'s `BxDF::Sample_f()` interface. One option would be to modify the interfaces to provide more sample values, though these would be unused by all of the other `BxDF` implementations; rendering efficiency would suffer from the time to generate these unused samples, and rendering quality would also likely suffer, as many `Sampler`s generate better samples in the lower dimensions than in higher dimensions; these extra samples would usually be wasted.

Therefore, here we'll implement an approach that lets us extract two separate samples from each provided sample. The `DemuxFloat()` function, to be defined shortly, decomposes a sample $\xi \in [0, 1)$ into a pair of samples, while also making some effort to preserve stratification in the returned sample value.

To understand the operation of `DemuxFloat()`, first recall the discussion of the Morton curve in Section 4.3.3. The Morton curve is a space-filling 1D curve that maps real numbers in $[0, 1]$ to n-dimensional numbers $[0, 1]^n$. If we use a 1D sample value ξ as an offset into a Morton curve, we can use each of the dimensions' values as independent samples.

An important advantage of using a Morton curve for this task is that it preserves stratification in the sample values it returns. To see why this is so, consider a collection of four stratified 1D sample values (i.e., one in $[0, 1/4)$, one in $[1/4, 1/2)$, and so forth.) The first sample value will be mapped mapped to a point in $[0, 1/2) \times [0, 1/2)$ by the Morton curve, since the first quarter of the 2D Morton curve traces out that region of space in two dimensions. Similarly, the next stratified 1D sample will be in the range $[1/2, 1) \times [0, 1/2)$, and so on.

To help with this, the `Compact1By1()` function takes a 32-bit integer, deletes all of the bits with odd indices, and compacts the remaining bits. The comment lines in the implementation illustrate the effect of each operation: after the surviving bits are initially masked off, a series of masked shifts compacts them until they are in contiguous positions.[4]

⟨*General Utility Functions*⟩ ≡
```
static uint32_t Compact1By1(uint32_t x) {
    // x = -f-e -d-c -b-a -9-8 -7-6 -5-4 -3-2 -1-0
    x &= 0x55555555;
    // x = --fe --dc --ba --98 --76 --54 --32 --10
    x = (x ^ (x >> 1)) & 0x33333333;
    // x = ---- fedc ---- ba98 ---- 7654 ---- 3210
    x = (x ^ (x >> 2)) &  0x0f0f0f0f;
    // x = ---- ---- fedc ba98 ---- ---- 7654 3210
    x = (x ^ (x >> 4)) & 0x00ff00ff;
    // x = ---- ---- ---- ---- fedc ba98 7654 3210
    x = (x ^ (x >> 8)) & 0x0000ffff;
    return x;
}
```

Here, we'll treat the sample ξ as a fixed-point value `v` computed by multiplying by 2^{32}. The 2D Morton curve effectively takes alternating bits from this value, giving two values between 0 and 2^{16}. In turn, these are mapped back to `Float`s.

⟨*General Utility Functions*⟩ ≡
```
static Point2f DemuxFloat(Float f) {
    uint64_t v = f * (1ull << 32);
    uint32_t bits[2] = {Compact1By1(v), Compact1By1(v >> 1)};
    return {bits[0] / Float(1 << 16), bits[1] / Float(1 << 16)};
}
```

1.4.2 A DISTRIBUTION FOR SAMPLING P

Next, we'll define the `ComputeApPdf()` method, which returns a discrete PDF with probabilities for sampling each term A_p according to its contribution relative to all of the A_p terms, given θ_o.

4 This code is thanks to Fabian Giesen, *fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/*.

⟨*HairBSDF Method Definitions*⟩ ≡

```
std::array<Float, pMax + 1> HairBSDF::ComputeApPdf(Float cosThetaO) const {
    ⟨Compute array of A_p values for cosThetaO⟩
    ⟨Compute A_p PDF from individual A_p terms⟩
    return apPdf;
}
```

The method starts by computing the values of A_p for `cosThetaO`. We're able to reuse some previously-defined fragments to make this task easier.

⟨*Compute array of* A_p *values for* `cosThetaO`⟩ ≡

```
Float sinThetaO = SafeSqrt(1 - cosThetaO * cosThetaO);
⟨Compute cos θt for refracted ray⟩
⟨Compute γt for refracted ray⟩
⟨Compute the transmittance T of a single path through the cylinder⟩
std::array<Spectrum, pMax + 1> ap = Ap(cosThetaO, eta, h, T);
```

Next, the spectral A_p values are converted to scalars using their luminance and these values are normalized to make a proper PDF.

⟨*Compute* A_p *PDF from individual* A_p *terms*⟩ ≡

```
std::array<Float, pMax + 1> apPdf;
Float sumY = std::accumulate(ap.begin(), ap.end(), Float(0),
    [](Float s, const Spectrum &ap) { return s + ap.y(); });
for (int i = 0; i <= pMax; ++i)
    apPdf[i] = ap[i].y() / sumY;
```

1.4.3 SAMPLING INCIDENT DIRECTIONS

With these preliminaries out of the way, we can now implement the `Sample_f()` method.

⟨*HairBSDF Method Definitions*⟩ ≡

```
Spectrum HairBSDF::Sample_f(const Vector3f &wo, Vector3f *wi,
        const Point2f &u2, Float *pdf, BxDFType *sampledType) const {
    ⟨Compute hair coordinate system terms related to wo⟩
    ⟨Derive four random samples from u2⟩
    ⟨Determine which term p to sample for hair scattering⟩
    ⟨Sample M_p to compute θi⟩
    ⟨Sample N_p to compute Δφ⟩
    ⟨Compute wi from sampled hair scattering angles⟩
    ⟨Compute PDF for sampled hair scattering direction wi⟩
    return f(wo, *wi);
}
```

⟨*Derive four random samples from* `u2`⟩ ≡

```
Point2f u[2] = { DemuxFloat(u2[0]), DemuxFloat(u2[1]) };
```

Given the PDF over A_p terms, we just loop over PDF values until we find the first value of `p` where the sum of preceding PDF values is greater than the sample value. Because we only need to generate one sample from the PDF's distribution, the work to compute an explicit CDF array (for example, by using `Distribution1D`) isn't worthwhile.

⟨*Determine which term p to sample for hair scattering*⟩ ≡

```
std::array<Float, pMax + 1> apPdf = ComputeApPdf(cosThetaO);
int p;
for (p = 0; p < pMax; ++p) {
    if (u[0][0] < apPdf[p]) break;
    u[0][0] -= apPdf[p];
}
```

Now that we've chosen a term, we can sample the corresponding M_p term given θ_o to find θ_i. The derivation of this sampling method is fairly involved, so we'll neither include the derivation nor the implementation here. This fragment, ⟨*Sample M_p to compute θ_i*⟩, consumes both of the sample values `u[1][0]` and `u[1][1]` and initializes variables `sinThetaI` and `cosThetaI` according to the sampled direction. After sampling a direction θ_i, this fragment then applies the inverse of the rotation that will later be used to account for hair scales when the BSDF is evaluated.

Next we'll sample the azimuthal distribution N_p. For terms up to p_max, we take a sample from the logistic distribution centered around the exit direction given by $\Phi(p, h)$. For the last term, we sample from a uniform distribution.

⟨*Sample N_p to compute $\Delta\phi$*⟩ ≡

⟨*Compute γ_t for refracted ray*⟩

```
Float dphi;
if (p < pMax)
    dphi = Phi(p, gammaO, gammaT) +
           SampleTrimmedLogistic(u[0][1], s, -Pi, Pi);
else
    dphi = 2 * Pi * u[0][1];
```

By inverting the CDF of the trimmed logistic, we can derive the recipe to sample from its distribution given a random variable $\xi \in [0, 1)$:

$$\begin{aligned}
\xi &= \int_a^x l_\mathrm{t}(x', s, [a, b])\,\mathrm{d}x' \\
&= \frac{1}{\int_a^b l(x, s)\,\mathrm{d}x} \int_a^x l(x', s)\,\mathrm{d}x' \\
&= \frac{1}{\int_a^b l(x, s)\,\mathrm{d}x} \left(\frac{1}{1 + \mathrm{e}^{-x/s}} - \frac{1}{1 + \mathrm{e}^{-a/s}} \right).
\end{aligned}$$

With a bit of algebra, we can solve for x:

$$x = -s \log \left(\frac{1}{\xi \int_a^b l(x, s)\,\mathrm{d}x + \frac{1}{1+\mathrm{e}^{-a/s}}} - 1 \right).$$

In practice, due to floating-point round-off, the implementation may compute an infinite value when $\xi \to 1$; a clamp at the end ensures that the returned value is in the range $[a, b]$.

⟨*Hair Local Functions*⟩ ≡

```
static Float SampleTrimmedLogistic(Float u, Float s, Float a,
                                   Float b) {
    Float k = LogisticCDF(b, s) - LogisticCDF(a, s);
    Float x = -s * std::log(1 / (u * k + LogisticCDF(a, s)) - 1);
    return Clamp(x, a, b);
}
```

Given θ_i and ϕ_i, we can compute the sampled direction `wi`. The math is similar to the `SphericalDirection()` function defined on p. 346, but with two important differences. First, because here θ is measured with respect to the plane perpendicular to the cylinder rather than the cylinder axis, we need to compute $\cos(\pi/2 - \theta) = \sin\theta$ for the coordinate with respect to the cylinder axis instead of $\cos\theta$. Second, because the hair shading coordinate system's (θ, ϕ) coordinates are oriented with respect to the $+x$ axis, the order of dimensions passed to the `Vector3f` constructor is adjusted correspondingly, since the direction returned from `Sample_f()` should be in the BSDF coordinate system.

⟨*Compute* `wi` *from sampled hair scattering angles*⟩ ≡

```
Float phiI = phiO + dphi;
*wi = Vector3f(sinThetaI, cosThetaI * std::cos(phiI),
               cosThetaI * std::sin(phiI));
```

Because we could sample directly from the M_p and N_p distributions, the overall PDF is

$$\sum_{p=0}^{p_\mathrm{max}} M_p(\theta_\mathrm{o}, \theta_\mathrm{i}) \tilde{A}_p(\omega_\mathrm{o}) N_p(\phi),$$

where $\tilde{A}_p$ are the normalized luminance-weighted PDF terms. Note that θ_i must be shifted to account for hair scales when evaluating the PDF; this is done in the same way (and with the same code fragment) as when the BSDF was evaluated.

⟨*Compute PDF for sampled hair scattering direction* `wi`⟩ ≡

```
*pdf = 0;
for (int p = 0; p < pMax; ++p) {
    ⟨Compute sin θi and cos θi terms accounting for scales⟩
    *pdf += Mp(cosThetaIp, cosThetaO, sinThetaIp, sinThetaO, v[p]) *
            apPdf[p] * Np(dphi, p, s, gammaO, gammaT);
}
*pdf += Mp(cosThetaI, cosThetaO, sinThetaI, sinThetaO, v[pMax]) *
        apPdf[pMax] * (1 / (2 * Pi));
```

The `HairBSDF::Pdf()` method performs the same computation was we just implemented for `Sample_f()`. Therefore, the implementation isn't included here.

1.4.4 TESTING SAMPLING ROUTINES

Because the sampling routine we have implemented exactly matches the PDF of the underlying BSDF, if we generate samples from the BSDF using `Sample_f()`, then the ratio between the BSDF value and the PDF computed for this direction should be one (as long as there is no absorption in the hair). The `SamplingWeights` test checks this for a variety of roughnesses and random sample values.

⟨*Hair Tests*⟩ ≡

```
TEST(Hair, SamplingWeights) {
    RNG rng;
    for (Float beta_m = .1; beta_m < 1; beta_m += .2)
        for (Float beta_n = .4; beta_n < 1; beta_n += .2) {
            int count = 10000;
            for (int i = 0; i < count; ++i) {
                ⟨Check HairBSDF::Sample_f() sample weight⟩
            }
        }
}
```

Performing the test is mostly a matter of setting up enough context to call `Sample_f()` and then verifying the ratio of the BSDF and the PDF.

⟨*Check* `HairBSDF::Sample_f()` *sample weight*⟩ ≡

```
Float h = -1 + 2 * rng.UniformFloat();
Spectrum sigma_a = 0;
HairBSDF hair(h, 1.55, sigma_a, beta_m, beta_n, 0.f);
Vector3f wo = UniformSampleSphere({rng.UniformFloat(), rng.UniformFloat()});
Vector3f wi;
Float pdf;
Point2f u = {rng.UniformFloat(), rng.UniformFloat()};
Spectrum f = hair.Sample_f(wo, &wi, u, &pdf, nullptr);
if (pdf > 0) {
    ⟨Verify that hair BSDF sample weight is close to 1 for wi⟩
}
```

Note that we accept a small amount of error, accepting values that are close to one but not exactly equal to it, in order to allow for floating-point round-off error.

⟨*Verify that hair BSDF sample weight is close to 1 for* `wi`⟩ ≡

```
EXPECT_GT(f.y() * AbsCosTheta(wi) / pdf, 0.999);
EXPECT_LT(f.y() * AbsCosTheta(wi) / pdf, 1.001);
```

Another useful test is based on computing reflected radiance from a varying incident radiance function with the scattering equation, Equation (5.9) in the third edition. Given a sufficient number of samples, we should get the same result both if we use the custom importance sampling scheme we have implemented and if we use a uniform distribution of directions over the unit sphere. This case is tested with the `SamplingConsistency` test.

⟨*Hair Tests*⟩ ≡

```
TEST(Hair, SamplingConsistency) {
    RNG rng;
    for (Float beta_m = .2; beta_m < 1; beta_m += .2)
        for (Float beta_n = .4; beta_n < 1; beta_n += .2) {
            ⟨Declare variables for hair sampling test⟩
            for (int i = 0; i < count; ++i) {
                ⟨Compute estimates of scattered radiance for hair sampling test⟩
            }
            ⟨Verify consistency of estimated hair reflected radiance values⟩
        }
}
```

The `Li` lambda function defines an incident radiance function with some (but not too much) variation as a function of direction. We keep this function fairly simple so that sampling the BSDF alone works well to compute reflected radiance and we can avoid the complexity of implementing multiple importance sampling in the test here.

⟨*Declare variables for hair sampling test*⟩ ≡

```
const int count = 64*1024;
Spectrum sigma_a = .25;
Vector3f wo = UniformSampleSphere({rng.UniformFloat(), rng.UniformFloat()});
auto Li = [](const Vector3f &w) -> Spectrum {
    return w.z * w.z;
};
Spectrum fImportance = 0, fUniform = 0;
```

For each sample in the Monte Carlo estimate, we choose a random point on the hair and use a pair of random numbers to sample an incident direction. We then use the regular Monte Carlo estimator to compute estimates using both sampling approaches.

⟨*Compute estimates of scattered radiance for hair sampling test*⟩ ≡

```
Float h = -1 + 2 * rng.UniformFloat();
HairBSDF hair(h, 1.55, sigma_a, beta_m, beta_n, 0.f);
Vector3f wi;
Float pdf;
Point2f u = {rng.UniformFloat(), rng.UniformFloat()};
Spectrum f = hair.Sample_f(wo, &wi, u, &pdf, nullptr);
if (pdf > 0) fImportance += f * Li(wi) * AbsCosTheta(wi) / (count * pdf);
wi = UniformSampleSphere(u);
fUniform += hair.f(wo, wi) * Li(wi) * AbsCosTheta(wi) /
            (count * UniformSpherePdf());
```

In the end, the two estimates should be very close. Here we treat 5% relative error as good enough to pass the test; this is a fairly low bar, but it allows the test to run quickly—if the total number of samples `count` was higher, we could expect closer agreement, but we prefer to have a test that runs in a second or so.

⟨*Verify consistency of estimated hair reflected radiance values*⟩ ≡
```
Float err = std::abs(fImportance.y() - fUniform.y()) / fUniform.y();
EXPECT_LT(err, 0.05);
```

1.5 HAIR ABSORPTION COEFFICIENTS

The color of hair is determined by how pigments in the cortex absorb light, which in turn is described by the normalized absorption coefficient where distance is measured in terms of the hair diameter. If a specific hair color is desired, there's a non-obvious relationship between the normalized absorption coefficient and the color of hair in a rendered image. Not only does changing the spectral values of the absorption coefficient have an unpredictable connection to the appearance of a single hair, but as we saw in Figure 1.2, multiple scattering between collections of many hairs has a significant effect each one's apparent color.[5] Therefore, here we provide implementations of two more intuitive ways to specify hair color.

The color of human hair is determined by the concentration of two pigments. The concentration of eumelanin is the primary factor that causes the difference between black, brown, and blonde hair. (Black hair has the most eumelanin and blonde hair has the least. White hair has none.) The second pigment, pheomelanin, causes hair to be orange or red. The `HairBSDF` class provides a convenience method that computes an absorption coefficient using the product of user-supplied pigment concentrations and absorption coefficients of the pigments computed by d'Eon et al. (2011), based on a model by Donner and Jensen (2006).

⟨*HairBSDF Method Definitions*⟩ ≡
```
Spectrum HairBSDF::SigmaAFromConcentration(Float ce, Float cp) {
    Float sigma_a[3];
    Float eumelaninSigmaA[3] = {0.419f, 0.697f, 1.37f};
    Float pheomelaninSigmaA[3] = {0.187f, 0.4f, 1.05f};
    for (int i = 0; i < 3; ++i)
        sigma_a[i] = (ce * eumelaninSigmaA[i] +
                      cp * pheomelaninSigmaA[i]);
    return Spectrum::FromRGB(sigma_a);
}
```

Eumelanin concentrations of roughly 8, 1.3, and 0.3 give reasonable representations of black, brown, and blonde hair, respectively.

It's also useful to specify the desired hair color directly. In order to make this possible, Chiang et al. (2016) created a cube of hair and rendered it with a variety of absorption coefficients and roughnesses, while it was illuminated with a uniform white dome. They then fit a function that mapped from the hair's azimuthal roughness and average color at the center of the front face

5 These issues are both related to those that make it difficult to directly set scattering and absorption coefficients for subsurface scattering, as discussed along with the `SubsurfaceFromDiffuse()` function defined on p. 938.

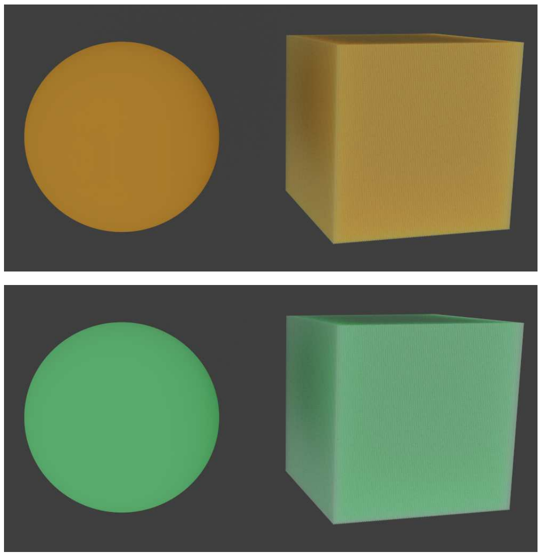

Figure 1.20: Two Spheres With Diffuse BSDFs Next To "Hair Cubes". The cubes are made of 100×100 densely packed `Curves`, with hair absorption coefficients computed using `SigmaAFromReflectance()`. Top, a RGB color of $(0.8, 0.4, 0.05)$ and a longitudinal roughness β_n of 0.3 was used; the computed σ_a value was $(0.003, 0.046, 0.488)$. Below, RGB was $(0.2, 0.8, 0.3)$ and $\beta_n = 0.8$, giving a σ_a value of $(0.141, 0.003, 0.079)$. In both cases, there is good agreement between the color of the sphere and the hair.

of the cube to an absorption coefficient. (Unlike the azimuthal roughness, the longitudinal roughness doesn't meaningfully affect the hair's color.) This function is implemented in the `SigmaAFromReflectance()` method; see Figure 1.20 for examples.

$\langle$*HairBSDF Method Definitions*$\rangle \equiv$

```
Spectrum HairBSDF::SigmaAFromReflectance(const Spectrum &c, Float beta_n) {
    Spectrum sigma_a;
    for (int i = 0; i < Spectrum::nSamples; ++i)
        sigma_a[i] = Sqr(std::log(c[i]) /
                     (5.969f - 0.215f * beta_n + 2.532f * Sqr(beta_n) -
                      10.73f * Pow<3>(beta_n) + 5.574f * Pow<4>(beta_n) +
                      0.245f * Pow<5>(beta_n)));
    return sigma_a;
}
```

1.6 HAIR MATERIAL

Like most `Materials` in `pbrt`, the `HairMaterial` is pretty straightforward; it's mostly a matter of evaluating textures and creating a corresponding `HairBSDF` object. It allows the attenuation coefficient for the hair interior to be specified with one of three ways; the corresponding parameters are:

- "spectrum sigma_a": the absorption coefficient can be specified directly.
- "spectrum reflectance": a reflectance for `SigmaAFromReflectance()`.
- "float eumelanin" and/or "float pheomelanin": eumelanin and pheomelanin concentrations.

Only one of these approaches can be used; an error is issued if more than one is provided. If none is specified, an eumelanin concentration of 1.3 is used, giving a brownish color.

A few additional parameter are supported

- "float beta_m", "float beta_n": longitudinal and azimuthal roughnesses. Both default to 0.3.
- "float eta": index of refraction of the hair interior. (1.55 by default).
- "float alpha": hair scale angle in degrees (2 by default).

1.7 A NOTE ON RECIPROCITY

On p. 350 in the third edition of *Physically Based Rendering*, we noted that physically-based BRDFs are both reciprocal and energy conserving. (In particular, reciprocity means that swapping the two evaluation directions gives the same BRDF value: $f(\omega_\mathrm{o}, \omega_\mathrm{i}) = f(\omega_\mathrm{i}, \omega_\mathrm{o})$.) BTDFs are in general not reciprocal, however; this topic (and methods to address it) is discussed further on p. 960 when bidirectional light transport algorithms are introduced.

The model we have implemented is, unfortunately, not reciprocal. One immediate issue is that the rotation for hair scales is applied only to θ_i. However, there are more problems: first, all terms $p > 0$ that involve transmission are not reciprocal; the underlying issue is that the transmission terms use values based on ω_t, which itself only depends on ω_o. Thus, if ω_o and ω_i are interchanged, a completely different ω_t is computed, which in turn leads to different $\cos\theta_\mathrm{t}$ and γ_t values, which in turn give different values from the A_p and N_p functions.

Many earlier hair scattering models have worked around the lack of reciprocity due to $\cos\theta_\mathrm{t}$ by computing an angle $\theta_\mathrm{d} = (\theta_\mathrm{o} - \theta_\mathrm{i})/2$ and using that in place of θ_o when computing θ_t and related quantities. (Note that θ_d is symmetric, since it's measuring angles with respect to the normal plane.) We didn't use θ_d in our implementation for two reasons: first, when we're sampling the BSDF, only ω_o is known and thus θ_d can't be computed. In turn, θ_o must be used to generate the PDF of A_p terms. Later, when ω_i is known and the BSDF is evaluated, a different value of A_p would be computed than was used for sampling. This mismatch between function value and PDF can cause a variance spike.[6]

6 Previous approaches have addressed this issue by clamping the maximum ratio of function value and PDF, though doing so loses energy.

Second, even if θ_d is used, there is a second, subtle and more difficult, source of non-reciprocity. Recall that the position h along the curve width is computed based on finding the intersection of a ray with a ribbon oriented to face the ray. It is thus directly dependent on ω_o and independent of ω_i. In turn, because γ_o and γ_t depend on h, they depend on ω_o alone and thus reciprocity is lost. The models developed by Marschner et al. (2003) and d'Eon et al. (2011) didn't have this problem since both computed a BCSDF by integrating h across the curve width. As noted by Chiang et al. (2016), this integration is computationally expensive, especially with low azimuthal roughnesses where the function varies quickly and many evaluations are needed for deterministic quadrature methods (recall Figure 1.17).

Another option would be to integrate stochastically, sampling a random h. (In turn, any given evaluation of f wouldn't be reciprocal, but the expected value over a sum of many evaluations would be.) This approach doesn't fit well with `pbrt`'s current architecture, where no random sample values are made available to the `BSDF::f()` method. Changing this would require a fairly extensive set of code modifications and seems generally unappealing.

A final possibility would be to compute a h_i for ω_i, based on projecting the intersection point up to the surface of the cylinder and then finding h_i for a ray with direction ω_i that passes through the hit point. In turn, we could evaluate A_p and N_p twice for each ordering of directions and take the average. This is relatively straightforward (and h_i can be found working entirely in the azimuthal plane), but in turn may lead to noise spikes, as one of the evaluations may have a much larger value than the current implementations of the sampling and PDF methods expect.

Stuck for an elegant solution, we will leave this issue to an exercise and hope that future research on this topic addresses this issue. In practice, we haven't seen visual artifacts in rendered images from the lack of reciprocity.

1.8 FURTHER READING

Kajiya and Kay (1989) were the first to develop a hair scattering model for rendering. Their model combined a diffuse term with a Phong lobe for an empirical model of specular highlights.

Marschner et al. (2003) investigated the processes underlying scattering from hair and performed a variety of measurements of scattering from actual hair. They introduced the longitudinal/azimuthal decomposition and the use of the modified index of refraction to hair rendering. They then developed a scattering model where the longitudinal component was derived by first considering perfectly specular paths and then allowing roughness by centering a Gaussian around them, and their azimuthal model assumed perfectly specular reflections. They showed that this model agreed reasonably well with their measurements.

Zinke and Weber (2007) formalized different ways of modeling scattering from hair and clarified the assumptions underlying each of them. Starting with the *bidirectional fiber scattering distribution function* (BFSDF), which describes reflected differential radiance at a point on a hair as a fraction of incident differential power at another, they showed how assuming homogeneous scattering properties and a far away viewer and illumination made it possible to simplify the eight-dimensional BFSDF to a four-dimensional *bidirectional curve scattering distribution* (BCSDF).

d'Eon et al. (2011) made a number of improvements to Marschner et al.'s model. They showed that their M_p term wasn't actually energy conserving and derived a new one that was; this is the model from Equation (1.3) that our implementation uses. (See also d'Eon (2013) for a more numerically stable formulation of M_p for low roughness.) They also introduced a Gaussian to the azimuthal term, allowing for varying azimuthal roughness. A 1D quadrature method was used to integrate the model across the width of the hair h.

Being able to generate samples from a distribution that approximates the BSDF is important for efficient rendering. Hery and Ramamoorthi (2012) showed how to sample the first term of the Marschner et al. model, and d'Eon et al. (2013) showed how to sample all terms of their improved model. (See also Jakob (2012) for notes related to sampling their M_p term in a numerically stable way.)

d'Eon et al. (2014) performed extensive Monte Carlo simulations of scattering from dielectric cylinders with explicitly modeled scales and glossy scattering at the boundary based on a Beckmann microfacet distribution. They showed that separable models didn't model all of the observed effects and that in particular that the specular term modeled by M_p varies over the surface of the cylinder and also depends on ϕ. They developed a non-separable scattering model, where both α and β_m varied as a function of h, and showed that it fit their simulations very accurately.

All of the scattering models we have described so far have been BCSDFs—they represent the overall scattering across the entire width of the hair in a single model. Such "far field" models assume that both the viewer is far away and that incident illumination is uniform across the hair's width. In practice, both of these assumptions are invalid if one is using path tracing to model multiple scattering inside hair. Two recent models have considered scattering at a single point along the hair's width, making them more suitable for accurately modeling "near field" scattering.

Yan et al. (2015) generalized d'Eon et al.'s model to account for scattering in the medulla, modeling a scattering cylinder in the interior of fur. They validated their model with a variety of measurements of actual animal fur, and showed how previous hair scattering models didn't match measured fur scattering well. Their model didn't integrate across the hair width.

Chiang et al. (2016) showed a number of comparisons that eliminating the integral over width from d'Eon et al.'s model works well in practice, and that the sampling rates necessary for path tracing also worked well to integrate scattering over the curve width, giving a much more efficient implementation. They also developed the perceptually-uniform parameterization of β_m and β_n that we have implemented here as well as the inverse mapping from reflectance to σ_a used in our `HairBSDF::SigmaAFromReflectance()` method.

1.9 BIBLIOGRAPHY

Chiang, M. J.-Y., B. Bitterli, C. Tappan, and B. Burley. 2016. A practical and controllable hair and fur model for production path tracing. *Computer Graphics Forum (Proceedings of Eurographics 2016).*

d'Eon, E., G. Francois., M. Hill, J. Letteri, and J.-M. Aubry. 2011. An energy-conserving hair reflectance model. *Computer Graphics Forum 30*(4), 1181–87.

d'Eon, E., S. Marschner, and J. Hanika. 2013. Importance sampling for physically-based hair fiber models. In *SIGGRAPH Asia 2013 Technical Briefs*, 25:1–25:4.

d'Eon, E., S. Marschner, and J. Hanika. 2014. A fiber scattering model with non-separable lobes–supplemental report. In *SIGGRAPH 2014 Talks*.

d'Eon, E. 2013. Notes on *An energy-conserving hair reflectance model*. *https://publons.com/publon/2803/.*

Jakob, W. 2012. Numerically stable sampling of the von Mises Fisher distribution on S^2 (and other tricks). *https://www.mitsuba-renderer.org/~wenzel/files/vmf.pdf*

Marschner, S. R., H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan. 2003. Light scattering from human hair fibers. *ACM Transactions on Graphics 22*(3), 780–91.

Ogaki, S., Y. Tokuyoshi, and S. Schoellhammer. 2010. An empirical fur shader. In *SIGGRAPH Asia 2010 Sketches*.

Yan, L.-Q., C.-W. Tseng, H. W. Jensen, and R. Ramamoorthi. 2015. Physically-accurate fur reflectance: modeling, measurement, and rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2015) 34*(6), 185:1–13.

Zinke, A., and A. Weber. 2007. Light scattering from filaments. *IEEE Transactions on Visualization and Computer Graphics 13*(2), 342–356.

1.10 ACKNOWLEDGMENTS

Thanks to Benedikt Bitterli, Eugene d'Eon, and Lingqi Yan for cheerfully answering many questions about hair scattering models.

1.11 EXERCISES

1.1 Marschner et al. (2003) note that human hair actually has an elliptical cross section that causes glints in human hair due to caustics. Extend the implementation here to handle this case. One issue that you'll need to address is that the $\partial \mathrm{p}/\partial v$ returned by `Curve::Intersect()` is always perpendicular to the incident ray, which leads to different orientations of the azimuthal coordinate system. This isn't an issue for the model we have implemented here, since it operates only on the difference between angles ϕ in the hair coordinate system. For elliptical hairs, a consistent azimuthal coordinate system is necessary.

1.2 Ogaki et al. (2010) created an explicit geometric model of the cuticle surface and then shot a large number of rays at it for each of a set of discrete outgoing directions, modeling scattering at the boundary with a microfacet model and modeling absorption and scattering in the hair interior. They then created a tabular representation of the resulting scattering distribution and used it for rendering. Implement this approach and compare the result to the model here.

1.3 As discussed in "A Note on Reciprocity", the model implemented in this document doesn't obey reciprocity. Investigate this issue and derive an improved model that does.

1.4 Read Yan et al.'s paper on fur scattering (2015) and implement their model, which accounts from scattering in the medulla in fur. Render images that show the difference from accounting for this in comparison to the current implementation. You may want to also see Section 4.3 of Chiang et al. (2016), which discusses extensions for modeling the undercoat (which is shorter and curlier hair underneath the top level) and a more ad-hoc approach to account for the influence of scattering from the medulla.

1.5 Read the paper by d'Eon et al. (2014) on a non-separable hair scattering model and implement their approach in `pbrt`. Render images that show the difference between their approach and the current implementation.

1.12 REVISION HISTORY

October 16, 2016: original version posted.

November 5, 2016: very minor typos fixed.