# Physically Based Lighting at Pixar

by Christophe Hery and Ryusuke Villemin
Pixar Animation Studios

Figure 1: Mike looks both physical and illuminated! *Monsters University*, ©Disney/Pixar 2013.

## 1 Introduction

For *Monsters University* and more recently for *The Blue Umbrella*, the lighting pipeline at Pixar was completely rewritten and switched to a physically based and ray-traced system. Although ray-tracing was already used in some cases—in the movie *Cars* for example—we were still using point lights, shadow maps, ad hoc shading models, etc. Over the years, the original pipeline has gotten more and more complex, with thousands of lights and shader tweaking in every shot. The idea was to let the computer handle all this complexity, freeing the lighter to concentrate on the artistic side.

Moving to physically based shaders, the main focus is now to solve the rendering equation[1]:

$$L(x,\omega_o) = \int_\Omega f(x,\omega_i,\omega_o) L(x,\omega_i)\cos(\theta) d\omega$$

In this equation there are two components that need to be updated, so that they cooperate in creating physically based lighting: the light, $L$, representing the energy emitted in the scene, and the BRDF, $f$, describing how the light will react in the scene. It is crucial that both work in tandem; physically

[1] For simplicity, in this paper we will focus on reflection only, ignoring all transmission effects like refraction, sub-surface scattering, and volumetric phenomena.

correct lights interacting with non-normalized BRDFs, or correct BRDFs interacting with incorrect lights, won't make much sense. In fact you will probably end up with more disadvantages, without seeing the full potential benefits.

We typically use Monte Carlo ray-tracing to solve this equation in the general case, but a naive implementation would lead to unbearable render times. To avoid this, different strategies are used—depending on what light path we consider—in order to optimize the convergence time. The equation is divided into sub-components, each of which will be resolved by a specific *coshader*[2] we call an *integrator*.

$$
\begin{aligned}
L(x,\omega_o) &= \int_\Omega f(x,\omega_i,\omega_o)\left(L_{direct}(x,\omega_i) + L_{indirect}(x,\omega_i)\right)\cos(\theta)d\omega \\
L(x,\omega_o) &= \int_\Omega f(x,\omega_i,\omega_o)L_{direct}(x,\omega_i)\cos(\theta)d\omega \\
&+ \int_\Omega \left(f_{diffuse}(x,\omega_i,\omega_o) + f_{specular}(x,\omega_i,\omega_o)\right)L_{indirect}(x,\omega_i)\cos(\theta)d\omega \\
L(x,\omega_o) &= \int_\Omega f(x,\omega_i,\omega_o)L_{direct}(x,\omega_i)\cos(\theta)d\omega \\
&+ \int_\Omega f_{diffuse}(x,\omega_i,\omega_o)L_{indirect}(x,\omega_i)\cos(\theta)d\omega \\
&+ \int_\Omega f_{specular}(x,\omega_i,\omega_o)L_{indirect}(x,\omega_i)\cos(\theta)d\omega
\end{aligned}
$$

That is:

$$L(x,\omega_o) = L_{direct} + L_{indirectDiffuse} + L_{indirectSpecular}$$

| | |
|---|---|
| $L(x,\omega_o)$ | radiance from $x$ in the $\omega_o$ direction $[Wm^{-2}sr^{-1}]$ |
| $L(x,\omega_i)$ | radiance to $x$ in the $\omega_i$ direction $[Wm^{-2}sr^{-1}]$ |
| $f_s(x,\omega_i,\omega_o)$ | surface BRDF at $x$ from direction $\omega_i$ to direction $\omega_o$ $[sr^{-1}]$ |
| $\theta$ | angle between the surface normal $N$ and $\omega_i$ $[r]$ |

Table 1: Notations

$L_{direct}$ is resolved by the `directLighting` integrator (light path = E{D,S}L, using [Heckbert 1990] path notation[3]). $L_{indirectDiffuse}$ is resolved by the `indirectDiffuse` integrator (light path = ED{D,S}*L), $L_{indirectSpecular}$ by the `reflection` integrator (light path = ES{D,S}*L). To take advantage of RenderMan's capabilities, the `indirectDiffuse` integrator is then divided into sub parts. The caustic integrator is going to solve specular paths ending with a diffuse bounce (EDS*L). For multiple diffuse bounces, we can either use the photon integrator or recursively use the indirect integrator (EDD*L). The photon and caustic integrators make use of photon mapping capability [The RenderMan Team 2013b]; the `indirectDiffuse` integrator uses the radiosity cache (described in [Christensen et al. 2012]) and some irradiance caching technology.

[2]A coshader is a specialized object construct in Pixar's RenderMan: for more information, refer to [The RenderMan Team 2009].

[3]Eye(E), specular(S), diffuse(D), light(L).

$$
\begin{aligned}
L_{indirectDiffuse}(x,\omega_o) &= \int_{\Omega} f_{diffuse}(x,\omega_i,\omega_o) L_{indirect}(x,\omega_i)\cos(\theta)d\omega \\
&= \int_{\Omega} f_{diffuse}(x,\omega_i,\omega_o) \\
&\quad \Bigg( \int_{\Omega} f_{specular}(x,\omega_i,\omega_o) L_{specular}(x,\omega_i)\cos(\theta)d\omega \\
&\quad + \int_{\Omega} f_{diffuse}(x,\omega_i,\omega_o) L_{diffuse}(x,\omega_i)\cos(\theta)d\omega \Bigg) \cos(\theta)d\omega
\end{aligned}
$$

The same recursive algorithm applies for the reflection integrator.

Breaking down the equation into parts that are solved by coshader integrators lets us create a very versatile and expandable system, in which we can change parts without rebuilding anything. For example, we recently added volumetric integrators; when plugged into our system, everything worked with minimal changes, since they solve non-overlapping parts of the same problem. You'll notice that there are difficult light paths that we choose to ignore for now; in the future we plan to solve those using bidirectional path-tracing and vertex merging techniques. We end up with four main integrators[4]:

- `directLighting` integrator
- `indirectDiffuse` integrator
- `reflection` integrator
- `photonCaustic` integrator

In these course notes, we are going to focus on the implementation of the `directLighting` integrator, which—even in the case of global illumination—represents the major part of the lighting. It is also the part we can optimize the most; contrary to indirect integrators that usually don't[5] have a-priori knowledge of the scene (before shooting rays into it), the direct lighting integrator knows about the light sources.

The direct lighting computation can be decomposed into three main parts:

- Light coshader
- BRDF coshader
- Integrator coshader

The integrator coshader is the main shader that will compute the final lighting result using Multiple Importance Sampling (see [Veach and Guibas 1995]). The light and the BRDF shaders are responsible for providing all the samples and weights with respect to their sampling strategies. There is a nice symmetry between the two: they both have a `sample` function for generating samples using their own strategy, and respective `emissionAndPDF` and `valueAndPDF` functions to generate values corresponding to the other strategy.

We provide many light and BRDF shaders in our library. To name a few, we have: `lambertianDiffuse`, `orenNayarDiffuse`, `kajiyaHairDiffuse`, `beckmannIsotropicSpecular`, `beckmannAnisotropicSpecular`, `gtrIsotropicSpecular`[6], `ggxAnisotropicSpecular` and `marschnerHairSpecular`. For lights, we have `dome`[7], `portal`, `rect`, `disk`, `sphere` and `distant` types. As coshaders, they are all abstracted in our system and interchangeable in the scene descriptions.

[4] Alongside the specialized volumes, transmission and sub-surface scattering integrators.

[5] See [The RenderMan Team 2013b] for exceptions.

[6] This is the generalized Trowbridge-Reitz model introduced by [Burley 2012].

[7] Similar to [Pharr and Humphreys 2004].

Figure 2: Note the various luminaires in this shot (neon signs, car headlights and various street lights). *The Blue Umbrella*, ©Disney/Pixar 2013.

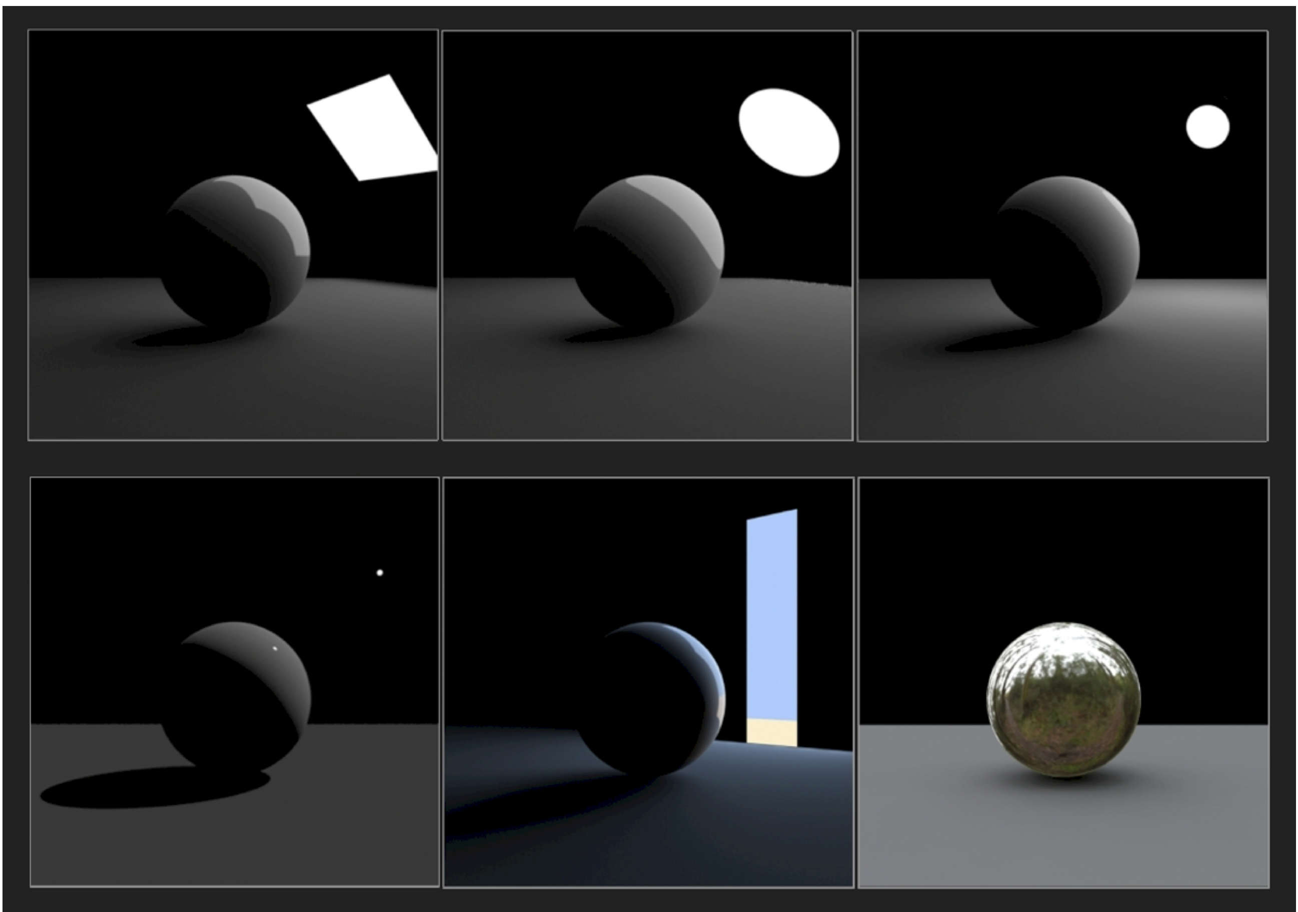


Figure 3: Different area light types.

The pieces of data (samples) are passed between these coshaders using structs of arrays:

- `LightSampleStruct` contains the results of the light sampling:
  - sample position `P` on the light
  - normalized vector `Ln` from the shading point to `P`
  - sample color `Cl`
  - sample pdf `pdf`
- `BSDFSampleStruct` contains the results of the BRDF sampling:
  - weighted sample value (value/pdf) `weight`
  - sample pdf `pdf`
  - sample direction `dir`
- `LightEmissionStruct` contains the light-side values of the BRDF sampling:
  - sample position `P` on the light
  - sample color `Cl`
  - sample pdf `pdf`
- `BSDFValueStruct` contains the BRDF-side values of the light sampling:
  - sample value `value`
  - sample pdf `pdf`

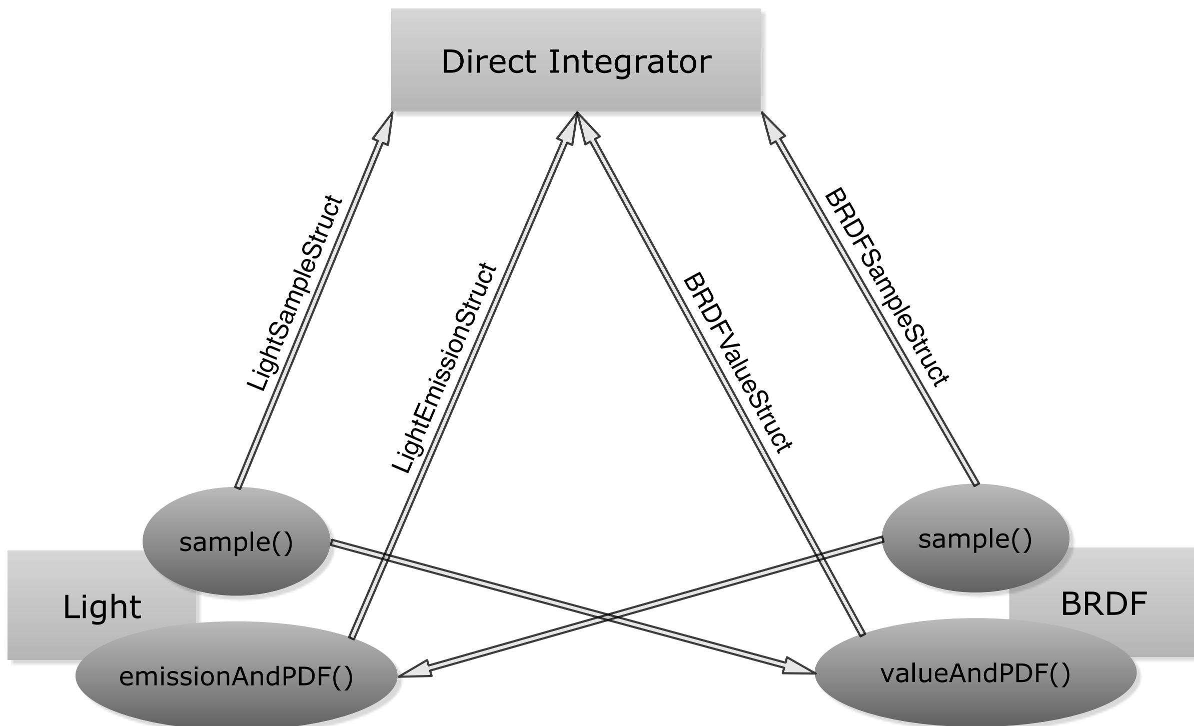



Figure 4: How the various structures communicate data between the three main coshaders.

This document will first detail two examples and their implementations: the `sphere area light` in section 2 and the `beckmannIsotropicSpecular` BRDF in section 3. For anyone interested in our `marschnerHairSpecular` approach, we invite you to check [Hery and Ramamoorthi 2012]. Later, in section 4, we will show exactly how the sampling data is pulled together inside our direct integrator.

## 2 Sphere Area Light

Figure 5: A nice (sphere) moon scene. *Monsters University*, ©Disney/Pixar 2013.

The sphere area light is our simplest illumination object, and we provide algorithms below for sampling and evaluating this particular luminaire. Originally these functions were written in RenderMan's Shading Language ([The RenderMan Team 1987-2013]); for ultimate performance, we later ported some of them to C++ (see [Villemin and Hery 2012]) and leveraged Intel's new `ispc` compiler—freely available here [Pharr and Mark 2012].

## 2.1 Sphere Area Light Sampling

First (lines 7 to 9), we compute the ideal number of samples, with a heuristic based on the solid angle observed from the shading point P. Also, from the importance of the ray (`rayWeight` in the code), we reduce this number of samples in recursion. Then for sampling, we roughly follow [Shirley et al. 1996]—a straightforward approach, where we distribute samples within the solid angle. At lines 14 and 16 of the algorithm, we rely on two random numbers[8]: $\xi_1$ and $\xi_2$. If P is inside the light (see lines 3 and 26), no samples are valid.

**Algorithm 1**. (Generating samples from a sphere area light)

```
1  void sample (out LightSamplingStruct ls)
   // Check whether we are inside or not
2  vector lightCenterDir = lightCenterPos − P;
3  float d2 = lightCenterDir · lightCenterDir;
4  if (d2 − radius²) ≥ 1e-4 then
5      float d = √d2;
       // Build an orthonormal basis towards the center of the sphere
6      vector ONBU, ONBV, ONBW;
7      CreateBasisFromW (lightCenterDir / d, ONBU, ONBV, ONBW);
       // Determine desired number of samples, between min and max and handle recursion
8      float solidAngle = 1 − √(d2 / (radius² + d2)); // Actually, the solid angle is 2π times this quantity
9      int numSamples = ceil(rayWeight ∗ solidAngle ∗ (maxSamples − minSamples));
10     ls → numValid = numSamples;
11     float costhetamax = √(1 − radius² / d2);
12     float pdf = 1/(2π ∗ (1 − costhetamax));
13     for int i = 0; i < numSamples; i += 1 do
14         float costheta = 1 + ξ1[i] ∗ (costhetamax − 1);
15         float sin2theta = 1 − costheta²;
16         vector lightDir = SphericalDir(√sin2theta, costheta, 2π ∗ ξ2[i]);
17         lightDir = TransformFromBasis(lightDir, ONBU, ONBV, ONBW);
18         float Δ = √(radius² − sin2theta ∗ d2);
19         ls → P[i] = P + (costheta ∗ d − Δ) ∗ lightDir;
20         ls → Ln[i] = lightDir;
21         ls → pdf[i] = pdf;
22         ls → Cl[i] = lightColor;
23     end
24 end
25 else
26     ls → numValid = 0;
27 end
```

[8]These numbers can, for instance, be generated via [Kensler 2013].

## 2.2 Sphere Area Light Evaluation

Here is the evaluation method of the light, given an array of BRDF samples as input. We use a simple quadratic formula, lines 15-19, to check on the potential hits to the light. If a given BRDF sample direction does not see the light (through this intersection routine), we mark the record with a PDF value of 0 (line 30).

**Algorithm 2**. (Evaluating BRDF samples on a sphere area light)

```
1  void emissionAndPDF (in BSDFSamplingStruct bs; out LightEmissionStruct le)
2  if bs → numValid = 0 then
3      return;
4  end
   // Check whether we are inside or not
5  vector lightCenterDir = P − lightCenterPos;
6  float d2 = lightCenterDir · lightCenterDir;
7  if (d2 − radius^2) ≥ 1e-4 then
8      float costhetamax = sqrt(1 − radius^2/d2);
9      float pdf = 1/(2π ∗ (1 − costhetamax));
       // intersect light
10     for int i = 0; i < numValid; i += 1 do
11         boolean isValid = false;
12         vector dir = bs → dir[i]; // direction towards the light
13         float b = 2 ∗ dir · lightCenterDir;
14         float c = lightCenterDir · lightCenterDir − radius^2;
15         float Δ = b^2 − 4 ∗ c;
16         if Δ > 0 then
17             float t = (−b − sqrt(Δ))/2;
18             if t < 1e-5 then
19                 t = (−b + sqrt(Δ))/2;
20             end
21             if t ≥ 1e-5 and t ≤ 1e20 then
                   // we have a hit
22                 isValid = true;
23                 le → P[i] = P + t ∗ dir;
24                 le → Cl[i] = lightColor;
25                 le → pdf[i] = pdf;
26             end
27         end
28         if isValid = false then
29             le → Cl[i] = color(0);
30             le → pdf[i] = 0;
31         end
32     end
33 end
```

### 2.3 Sphere Area Light Diffuse Convolution

This function is used in the integrator for the control variates method (refer to 4.5). We follow [Snyder 1996]. Note that we check on the surface type to determine whether we can provide a valid convolution or not.

**Algorithm 3**. (Diffuse convolution of a sphere area light)

```
1  boolean diffConvolution (out color diffConv)
   // Check whether we can provide a diffuse convolution to the integrator
2  if enableControlVariates = false or isHair = true then
3      return false;
4  end
5  diffConv = color(0);
6  if lightColor != color(0) then
       // Check whether we are inside or not
7      vector lightCenterDir = lightCenterPos − P;
8      float d2 = lightCenterDir · lightCenterDir;
9      float cosTheta = (lightCenterDir·N) / √d2;
10     if (d2 − radius^2) ≥ 1e-4 then
11         float sinAlpha = radius / √d2;
12         float cosAlpha = √(1 − sinAlpha^2);
13         float alpha = asin(sinAlpha);
14         float theta = acos(cosTheta);
           // we use the faster cubic formula
15         if theta < (π/2 − alpha) then
16             diffConv = cosTheta ∗ sinAlpha^2;
17         end
18         else if theta < π/2 then
19             float g0 = sinAlpha^3;  // d
20             float g1 = 1/π ∗ (alpha − cosAlpha ∗ sinAlpha);
21             float gp0 = −cosAlpha ∗ sinAlpha^2 ∗ alpha;  // c
22             float gp1 = −sinAlpha^2 ∗ alpha/2;
23             float a = gp1 + gp0 − 2 ∗ (g1 − g0);
24             float b = 3 ∗ (g1 − g0) − gp1 − 2 ∗ gp0;
25             float y = (theta − (π/2 − alpha))/alpha;
26             diffConv = g0 + y ∗ (gp0 + y ∗ (b + y ∗ a));  // ay^3 + by^2 + cy + d
27         end
28         else if theta < (π/2 + alpha) then
29             float g0 = 1/π ∗ (alpha − cosAlpha ∗ sinAlpha);  // d
30             float gp0 = −(sinAlpha^2 ∗ alpha)/2;  // c
31             float a = gp0 + 2 ∗ g0;
32             float b = −3 ∗ g0 − 2 ∗ gp0;
33             float y = (theta − π/2)/alpha;
34             diffConv = g0 + y ∗ (gp0 + y ∗ (b + y ∗ a));  // ay^3 + by^2 + cy + d
35         end
           // else leave diffConv at 0
36     end
37     diffConv *= lightColor;
38 end
39 return true;
```

## 3 Beckmann BRDF

This particular specular model was derived with a view towards simplicity and efficiency. It is largely inspired by [Walter et al. 2007], and also has much in common with the grand-daddy of all models: [Cook and Torrance 1982].

We decided to employ the trusted Beckmann distribution, with a roughness term $\alpha$ (between 0 and 1). For an arbitrary microfacet normal $\mathbf{m}$, using $\theta_m$ as the angle between this $\mathbf{m}$ direction and the macroscopic surface normal $\mathbf{n}$, the distribution is expressed as:

$$D_b(\mathbf{m}) = \frac{e^{-\tan^2(\theta_m)/\alpha^2}}{\pi\,\alpha^2\,\cos^4(\theta_m)} = \frac{e^{\frac{(\mathbf{n}\cdot\mathbf{m})^2-1}{\alpha^2(\mathbf{n}\cdot\mathbf{m})^2}}}{\pi\,\alpha^2\,(\mathbf{n}\cdot\mathbf{m})^4}$$

As we saw in [Hoffman 2013], microfacet BRDFs always evaluate normal distributions at $\mathbf{h}$, the halfway vector from the light direction $\mathbf{l}$ and the view direction $\mathbf{v}$. [Walter 2005] derives a sampling strategy for the dominant term in $D_b$, i.e. the exponential. In this scheme, each sample is picked with a probability:

$$pdf = \frac{D_b(\mathbf{h})\,(\mathbf{n}\cdot\mathbf{h})}{4\,(\mathbf{v}\cdot\mathbf{h})}$$

Suppose we have our full BRDF model (which has many terms in addition to $D_b$), and use "value" to denote its evaluation for a given set of $\mathbf{l}$, $\mathbf{v}$ and $\mathbf{n}$ (and consequently $\mathbf{h}$). By definition of radiance and sampling, we have:

$$color = \frac{value\,(\mathbf{n}\cdot\mathbf{l})}{pdf}$$

One desirable property of a BRDF model in a production environment is that it preserves energy as much as possible. A common verification method is the "white furnace" test[9], which should produce a uniformly white result. Mathematically, this translates to:

$$\frac{value\,(\mathbf{n}\cdot\mathbf{l})}{pdf} = 1$$

In other words:

$$f_\mu(\mathbf{l},\mathbf{v}) = value = \frac{pdf}{\mathbf{n}\cdot\mathbf{l}} = \frac{\frac{D_b(\mathbf{h})\,(\mathbf{n}\cdot\mathbf{h})}{4\,(\mathbf{v}\cdot\mathbf{h})}}{\mathbf{n}\cdot\mathbf{l}} = \frac{D_b(\mathbf{h})\,(\mathbf{n}\cdot\mathbf{h})}{4\,(\mathbf{n}\cdot\mathbf{l})\,(\mathbf{v}\cdot\mathbf{h})}$$

Accounting for Fresnel, $F$, this gives us our BRDF:

$$f_\mu(\mathbf{l},\mathbf{v}) = \frac{F(\mathbf{l},\mathbf{h})\,D_b(\mathbf{h})\,(\mathbf{n}\cdot\mathbf{h})}{4\,(\mathbf{n}\cdot\mathbf{l})\,(\mathbf{v}\cdot\mathbf{h})}$$

In practice, we pull out the Fresnel term from this definition and incorporate the regular $\mathbf{n}\cdot\mathbf{l}$ cosine instead. This cancels out, leaving:

$$f^*_\mu(\mathbf{l},\mathbf{v}) = \frac{D_b(\mathbf{h})\,(\mathbf{n}\cdot\mathbf{h})}{4\,(\mathbf{v}\cdot\mathbf{h})} = pdf$$

This has the advantage of being a very simple (and fast) expression and it also guarantees that we pass the furnace test, at least as far as the Beckmann distribution goes. With small roughness values, say $\alpha \le 0.1$, $D_b$ (and thus $f_\mu$) will indeed abide by the furnace requirement. For larger values, we still lose some energy at grazing angles. This is the reason for pulling the Fresnel term out: artists can use it to compensate for the loss. In summary, our BRDF (with built-in cosine) is:

$$f^*_\mu(\mathbf{l},\mathbf{v}) = \frac{e^{\frac{(\mathbf{n}\cdot\mathbf{h})^2-1}{\alpha^2(\mathbf{n}\cdot\mathbf{h})^2}}}{4\,\pi\,\alpha^2\,(\mathbf{n}\cdot\mathbf{h})^3(\mathbf{v}\cdot\mathbf{h})}$$

A perfect Distribution-based BRDF! (See [Ashikhmin and Premoze 2007].)

[9] A pure white object is uniformly lit without shadows under a pure-white dome.

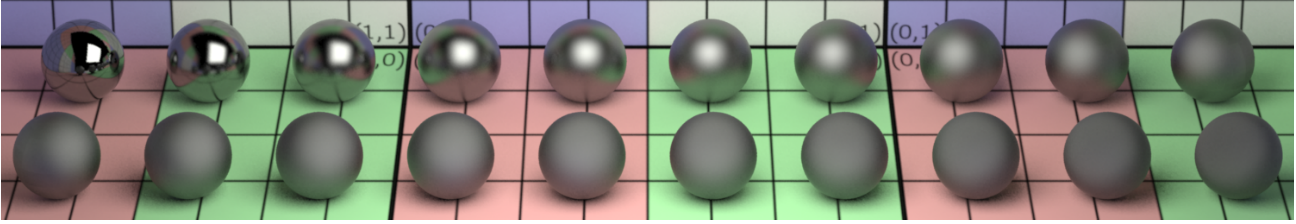


Figure 6: Beckmann specular: increasing roughness values from 0.001 to 1.0

### 3.1 Beckmann BRDF sampling

Now that we have our target distribution, we show how BRDF samples are generated using that distribution. Note that we do not enforce energy conservation between the various lobes: this is left as an "exercise" for the shading artist, who has to guarantee (or not) that the sum of the `K` terms is $\leq 1$. In the algorithms below, it is assumed that Fresnel is baked externally into `specColor` and that the facing-ratio test, $\mathrm{V} \cdot \mathrm{Ng}$, has been done prior to the call. Lines 8 and 14 are directly from [Walter 2005].

**Algorithm 4.** (Sampling of a Beckmann specular lobe)

1 void `sample` (in vector wi; in int lobeSamples; out BSDFSamplingStruct bs)

`// Note that we already rejected the cases where V · Ng < 0`

2 **if** specColor $=$ color(0) **or** lobeSamples $= 0$ **then**

3 return; `// no need to sample`

4 **end**

`// Need to rectify the probabilities from the fact that I am sampling all lobes at once`

5 float ratio $=$ bs $\rightarrow$ numSamples/lobeSamples;

6 int numCurrent $=$ bs $\rightarrow$ numValid; `// we will add to the list from there`

7 **for** int i $= 0$; i $<$ lobeSamples; i $+= 1$ **do**

`// Sample angle theta`

8 float tantheta2 $= -\ln \xi_1[i] * \text{roughness}^2$;

9 float costheta $= \frac{1}{\sqrt{1+\text{tantheta2}}}$;

`// Create a halfvector`

10 vector H $=$ `SphericalDir`($\sqrt{1-\text{costheta}^2}, \text{costheta}, 2\pi * \xi_2[i]$);

11 float VdotH $=$ wi $\cdot$ H;

`// Compute incident direction by reflecting about H`

12 vector wo $= -$wi $+ 2 *$ VdotH $*$ H;

13 **if** wo $\cdot$ N $> 0$ **then**

`//` $\xi_1[i]$ `represents the Beckmann exponential term`

14 bs $\rightarrow$ pdf[numCurrent] $= \frac{\xi_1[i]*\text{ratio}}{4\pi*\text{costheta}^3*\text{roughness}^2*\text{abs(VdotH)}}$;

`//` weight = $\frac{value*\text{LdotN}}{\text{pdf}}$`, which becomes by design:`

15 bs $\rightarrow$ weight[numCurrent] $= \frac{\text{specColor}}{\text{ratio}}$;

`// Corresponding sampling direction`

16 bs $\rightarrow$ dir[numCurrent] $=$ wo;

17 numCurrent $+= 1$;

18 **end**

19 **end**

20 bs $\rightarrow$ numValid $=$ numCurrent;

### 3.2 Beckmann BRDF evaluation

Our BRDF must also be able to evaluate the contribution of samples generated from the lighting distribution. Some light samples may resolve to zero contribution, because they face the wrong way (line 17). Also, remember that our returned value at line 13 is actually the product of the BRDF by the cosine.

**Algorithm 5**. (Evaluation of a beckmann specular lobe)

```
1  boolean valueAndPDF (in vector wi; in vector wos[]; out BSDFValueStruct bv)
2  boolean hasValidValues = false;
   // Note that we already rejected earlier the cases where V · Ng < 0
3  if specColor = color(0) then
4      return hasValidValues; // no valid values
5  end
6  for int i = 0; i < bv → numSamples; i += 1 do
7      vector wo = wos[i];
8      if wo · N > 0 then
9          hasValidValues = true;
           // Compute the microfacet distribution (Beckmann)
10         vector H = normalize(wo + wi);
11         float costheta = H · N;
12         float pdf = e^((costheta^2 − 1)/(roughness^2 ∗ costheta^2)) /(4π ∗ costheta^3 ∗ roughness^2 ∗ (wi · H));
13         bv → value[i] = specColor ∗ pdf;
14         bv → pdf[i] = pdf;
15     end
16     else
17         bv → value[i] = color(0);
18         bv → pdf[i] = 0;
19     end
20 end
21 return hasValidValues;
```

Line 12: $\text{pdf} = e^{\frac{\text{costheta}^2 - 1}{\text{roughness}^2 * \text{costheta}^2}} / (4\pi * \text{costheta}^3 * \text{roughness}^2 * (\text{wi} \cdot \text{H}))$

## 4 Direct lighting integrator

The `directLighting` integrator will gather all the samples from the BRDF coshaders and the light coshaders, and will compute the final result. For robustness, we combine these two sampling strategies using Multiple Importance Sampling (MIS). As described in the introduction, this integrator focuses on direct lighting only, so the computation is optimized for this sole task, completely ignoring all indirect effects. Since the BRDF's sampling is independent of the lights, it is done once outside of the light loop. Notice that we return two `BSDFSampleStruct`s here, one for "normal" area lights and one for infinite lights (see 4.3). Depending on the light type we will use `bs` or `bsbvh`. The `bsbvh` will only contain samples that are hitting at least one light. We still need the original sets of samples for infinite lights that are potentially hit by any direction. Then the results for all lights are accumulated into the final result. One important point here is that although we have area lights, by default they don't have any geometric representation in the scene and thus are not themselves visible. This is also why we have a separate acceleration structure for those lights.

**Algorithm 6**. (Direct Lighting Integration)

```
1  color integrate ()
   // Get BSDF samples -- sample according to BRDF strategy
2  IntegrateBrdf(bs, bsbvh, numSpecLobes, numSpecSamples);
   // Light loop
3  for int l = 0; l < lightCount; l += 1 do
4      shader li = lights[l];
       // Get Light samples -- sample according to Light strategy
5      if li → hasInfiniteBounds then
6          IntegrateLight(li, bs, numSpecLobes, numSpecSamples, finalDiff, finalSpec);
7      end
8      else
9          IntegrateLight(li, bsbvh, numSpecLobes, numSpecSamples, finalDiff, finalSpec);
10     end
11     resultDiff += finalDiff;
12     resultSpec += finalSpec;
13     result += finalDiff + finalSpec;
14 end
15 return result;
```

### 4.1 BSDF Struct

The `BSDFStruct` is a structure that contains all of the BRDF lobes for a given material[10]. We combine them in this struct for code clarity, and also for performance reasons. For diffuse components, in order to optimize computation, we "flatten" all the lobes into one, by using a merged albedo instead of multiple different albedos (if the lobes are standard Lambertian diffuse). Even if we have more complicated (potentially view-dependent) diffuse, we still have a gain by sampling all the lobes at once using a single strategy—uniform or cosine-weighted, for example. For specular lobes, since they each will use a different strategy (either because it is a set of distinct BRDF models or if they have different roughnesses), we need to sample them separately, but we can still store all the samples in one common struct. Once we have all the samples and their corresponding values and PDFs, we don't really need to know where and how they were computed: the values are enough to perform MIS. `valueAndPDF_Spec` is shown here, `valueAndPDF_Diff` is equivalent:

**Algorithm 7.** (BSDFStruct valueAndPDF_Spec)

```
1  void valueAndPDF_Spec (out BSDFValueStruct bv)
2  for int i = 0; i < numSpecularBRDFs; i += 1 do
3      boolean hasValidValues = specularBRDFs[i] → valueAndPDF(wi, wouts, onebv);
4      if hasValidValues = false then
5          int numDirections = arraylength(wouts);
6          for int k = 0; k < numDirections; k += 1 do
7              onebv → value[k] = color(0);
8              onebv → pdf[k] = 0;
9          end
10     end
11     push(bv, onebv);
       // one struct of arrays per lobe for MIS on specular
12 end
```

**Algorithm 8.** (BSDFStruct sample)

```
1  void sample ()
   // Sample each specular BRDF
2  for int i = 0; i < numSpecularBRDFs; i += 1 do
3      int thisNumSamples = specularSamples[i];
4      if thisNumSamples > 0 and facingRatio ≥ 0.0 then
5          specularBRDFs[i] → sample(wi, thisNumSamples, bs);
6      end
7  end
```

[10]If this was not obvious, let's state here that our materials can contain arbitrary numbers of lobes, each of arbitrary types.

### 4.2 Light integrator

This is the integration code for a single light. We assume that the BRDF sampling has already been done outside and has been passed here as an input parameter. First we compute the samples according to the light strategy. This is done using the `sample` function in the light coshader (as in section 2.1). Then, in order to use MIS, we will compute the corresponding values and PDFs for BRDF sampling. This is done through the `emissionAndPDF` function, that takes the `valueAndPDF` function for each of the BRDF coshaders. These will give us the first pair of values. Next we compute the corresponding light values for the BRDF samples that were passed as inputs. Since the BRDF sampling was already done outside, we only need to call the `emissionAndPDF` on the light coshader side. This, combined with the input samples of the BRDF, will give us the second pair of values for MIS.

The `computeMIS` function will then calculate the final values and weights for each of those samples. At this stage we have the results without any visibility taken into account. The final step is to compute the visibility term for all the resulting samples. Shadowing is done at the very end because it is usually the most expensive step, and doing it after everything enables to use the final weighting of each sample to optimize its computation using a cutoff (or Russian Roulette if we want to stay unbiased). This is also true if we have enabled resampling: the final number of samples will be reduced before doing the shadowing calculation.

**Algorithm 9**. (Light Integration)

```
1  color integrateLight (shader li; out BSDFSamplesStruct bs; out int numSpecLobes; out int
   numSpecSamples; out color lightDiff, lightSpec)
   // Light sampling
2  li → sample(ls);
   // Check generated samples
3  int numGeneratedLightSamples = ls → numValid;
4  int numActiveSpecSamples = bs → numValid;
   // BRDF evaluation
5  if numGeneratedLightSamples > 0 then
       // No MIS on diffuse:  do all diffuses together
6      bsdf → valueAndPDF_Diff(diffValues);
       // On the other hand, return an array for the specular lobes
7      bsdf → valueAndPDF_Spec(specValues);
8  end
   // Light evaluation
9  li → emissionAndPDF(bs, lightValues);
   // Importance sampling
10 computeMIS(ls, diffValues, specValues, bs, lightValues, Cdiff, Cspec, CspecBRDF);
   // Light shadows
11 if numGeneratedLightSamples > 0 then
12     computeLightShadows(Cdiff, Cspec, diffPerLight, diffPerLightNoShad, specPerLight, specPerLightNoShad);
13 end
   // BRDF shadows
14 if numActiveSpecSamples > 0 and facingRatio ≥ 0.0 then
15     computeBRDFShadows(CspecBRDF, specPerLight, specPerLightNoShad);
16 end
   // Per light integration output
17 lightDiff = mix(diffPerLightNoShad, diffPerLight, shadowDensity);
18 lightSpec = mix(specPerLightNoShad, specPerLight, shadowDensity);
```

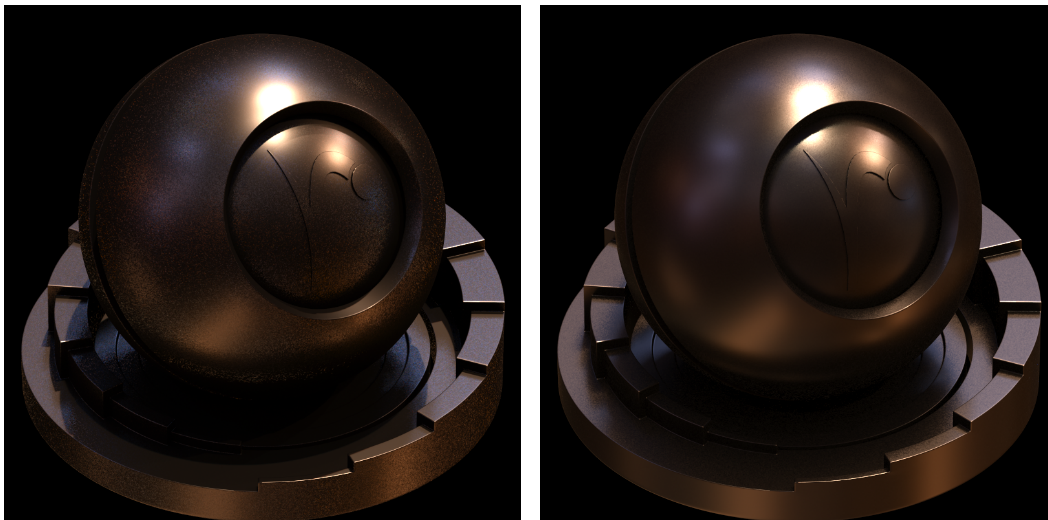
### 4.3 BRDF integrator

The BRDF's integration is done for all the specular lobes at once. The number of lobes is returned in `numSpecLobes`, and the number of samples per lobe is returned in an array `numLobeSamples`. Each BRDF can have a variable number of samples based on its properties (roughness, etc.). An optimization is performed at that stage: all the samples are tested against an acceleration structure containing all of the area lights. If a BRDF sample does not hit any light, it is discarded. This optimization is not required, but is recommended as the sampling can become very expensive if you have thousands of lights in the scene. We still keep the original set of samples, to use against infinite lights (`dome area light` in particular); the direct lighting integrator is responsible for passing the right sample set to the light depending on its type.

**Algorithm 10**. (BRDF Integration)

```
1 color integrateBrdf (out BSDFSamplesStruct bs; out BSDFSamplesStruct bsbvh; out int numSpecLobes;
  out int numSpecSamples)

  // Get BSDF samples -- sample according to BRDF strategy
2 bsdf → getNumSpecSamples(numLobeSamples, numSpecSamples);
3 numSpecLobes = arraylength(numLobeSamples);
4 if numSpecLobes > 0 then
5     bs = bsdf → sample();
      // Early reject BRDF samples (through bvh query) that do not intersect any lights
6     bsbvh = BVHReduce(P, bs);
7 end
```



(a) Light integration result (b) BRDF integration result

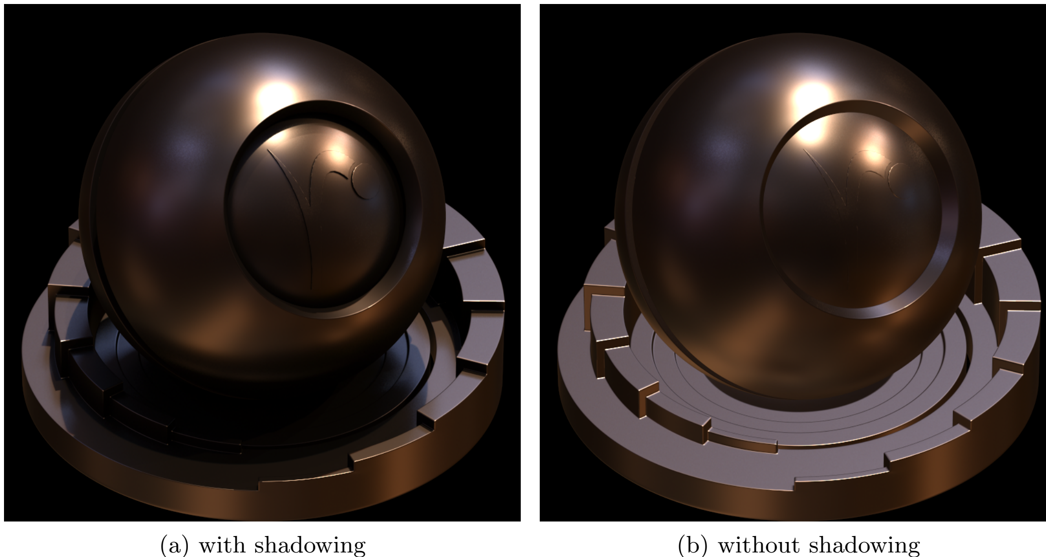
Figure 7: Two sampling strategies

## 4.4 MIS integrator

Once we have all the samples from the two strategies—Light and BRDF—we can perform MIS. (For diffuse lobes, we usually only use light sampling[11].) Our integrator also has the possibility to perform a resampling step[12], or resampled MIS (as in [Talbot et al. 2005]) in the case of MIS. This is sometimes useful to reduce the number of shadow rays to trace, especially if a lot of samples were necessary to solve a high frequency pattern in the emission of the light.

**Algorithm 11.** (Compute MIS)

```
1  color computeMIS (LightSampleStruct ls; BSDFValueStruct diffValues; BSDFValueStruct specValues;
   BRDFStruct bs; LightEmissionStruct lightValues; out color Cdiff, Cspec, CspecBRDF)
2  if diffResampPercentage ≥ 1.0 then
3      integrateIS(diffValues, ls, Cdiff);
4  end
5  else
6      integrateRIS(diffResampPercentage, diffValues, ls, Cdiff);
7  end
8  if facingRatio ≥ 0.0 then
9      if specResampPercentage ≥ 1.0 then
10         integrateMIS(ls, specValues, numLobeSamples, Cspec, bs, lightValues, CspecBRDF);
11     end
12     else
13         integrateRMIS(specResampPercentage, ls, specValues, numLobeSamples, Cspec, bs, lightValues, CspecBRDF);
14     end
15 end
```



(a) with shadowing (b) without shadowing

Figure 8: MIS integration result

[11] We still might want to perform MIS even on diffuse lobes when a light source is very close to the shading point. In that case, the light sampling integral has a peak because of the division by the squared distance. Using BRDF sampling will get rid of this singularity.

[12] Because the resampling can be costly, we do this in an optimized DSO, which in RSL is a specialized function directly written in C/C++ to expand the language or for performance gains.

### 4.5 Shadow integrator

Light shadowing is done using ray-tracing of transmission (shadow) rays that are optimized to only compute visibility. This is not shown here, but specialized shadowing strategies can be implemented here. As long as the strategies do not overlap each other (i.e., not creating double shadows), we can have multiple calls to different shadowers. For example, after shadow tracing against standard objects in the scene, one can use a specialized Spherical Harmonics visibility scheme for heavy vegetation, and/or ray-march inside shadow maps for large hair objects (see [The RenderMan Team 2011]).

An additional step is done using control variates[13] if the light was able to provide an analytical solution of its non-shadowed lighting. This usually relies on a contour integral computation or a texture map pre-convolution. Common techniques, albeit somehow approximate, to obtain a diffuse convolution can make use of a Spherical Harmonics representation ([Ramamoorthi and Hanrahan 2001] and [Mehta et al. 2012]). Refer to 2.3 for an example on our Sphere area light.

The idea here is that we want to calculate:

$$I = \int fv$$

with $f$ being the product of the incoming radiance and the BRDF, and $v$, the visibility term. Without any changes, $I$ can be rewritten as:

$$I = \int fv + \alpha \left( \int f - \int f \right), \forall \alpha$$

If we have a way to calculate analytically $G = \int f$, this leads to:

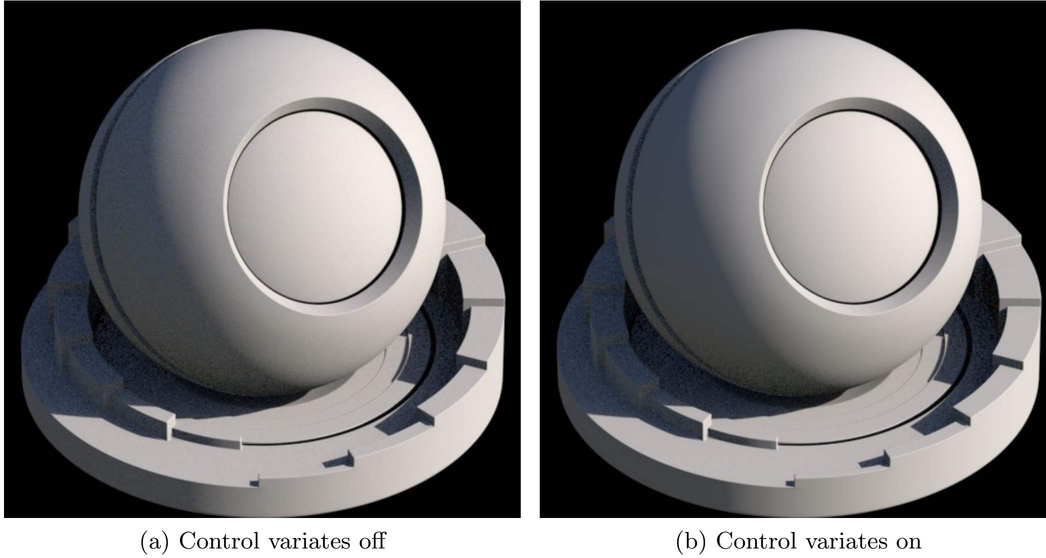
$$I = \int fv + \alpha \left( G - \int f \right)$$

For instance, we can take $\alpha = \bar{v}$, the average visibility. The advantage is that if $\bar{v}$ is 1 (i.e., v is 1 everywhere), then $I$ is exactly $G$. On the other hand, if $\bar{v}$ is exactly 0, then $I$ is the original sampling. If $\bar{v}$ is between 0 and 1, there is still variance reduction, as long as the correlation between $f$ and $fv$ is high enough. Here:

- G is our diffuse preconvolved solution, `diffConv`
- $\int fv$ is `diffPerLight`
- $\int f$ is `diffPerLightNoShad`

We direct the reader to [Clarberg and Akenine-Möller 2008], which describes a strategy for estimating an approximate and efficient $\bar{v}$ term.

[13] A classic Monte Carlo technique well documented in [Kalos and Whitlock 1986].



(a) Control variates off

(b) Control variates on

Figure 9: Control variates on diffuse. Observe the reduction of variance in the non-shadowed regions.

**Algorithm 12**. (Compute Light Shadows)

```
1  color computeLightShadows (color Cdiff[], Cspec[]; out color diffPerLight, diffPerLightNoShad,
   specPerLight, specPerLightNoShad)
2  color Lvis[];
3  avgVis = AreaShadowRays(ls → P, ls → pdf, Lvis);
4  for int i = 0; i < numGeneratedLightSamples; i += 1 do
5      diffPerLight += Cdiff[i] * (color(1) − Lvis[i]);
6      diffPerLightNoShad += Cdiff[i];
7  end
8  if facingRatio ≥ 0.0 then
9      for int i = 0; i < numGeneratedLightSamples; i += 1 do
10         specPerLight += Cspec[i] * (color(1) − Lvis[i]);
11         specPerLightNoShad += Cspec[i];
12     end
13 end
   // Check diffuse convolution
14 if li → diffuseConvolution(diffConv) = true then
       // Do control variates
15     diffConv *= bsdf → albedo();
16     diffPerLight += (color(1) − avgVis) * (diffConv − diffPerLightNoShad);
17     diffPerLightNoShad = diffConv;
18 end
```

BRDF shadowing is straightforward: there is no special trick here, we just trace all the shadow rays and add the results to the specular accumulation. The same method applies here if we want to use multiple shadowing strategies (SH, shadow maps, etc.), i.e. this is the place to run them.

**Algorithm 13.** (Compute BRDF Shadows)

```
1 color computeBRDFShadows (color CspecBRDF[]; out color specPerLight, specPerLightNoShad)
2 color LvisBRDF[];
3 color AreaShadowRays(lightValues → P, bs → pdf, LvisBRDF);
4 for int i = 0; i < numActiveSpecSamples; i += 1 do
5     specPerLight += CspecBRDF[i] * (color(1) − LvisBRDF[i]);
6     specPerLightNoShad += CspecBRDF[i];
7 end
```

# 5 Conclusion

RenderMan's versatility allows the shader writer to create his very own integrators, lights and BRDFs. One can thus develop a full physically based system—such as what we outlined in this document—without touching at the guts of the renderer. However, if you don't need this level of customization, or simply want the render engine to take care of all the implementation details and performance optimization, you can use the built-in integrators of RenderMan 17.0 [The RenderMan Team 2012], and the new geometric arealights API in RenderMan 18.0 [The RenderMan Team 2013a].

We want to acknowledge here the work of all the shading, lighting and rendering artists on Monsters University and The Blue Umbrella, and in particular want to thank their respective Directors of Photography (Jean-Claude Kalache on Monsters University and Brian Boyd for The Blue Umbrella). We also want to express our gratitude to the full RenderMan team and to the group of TDs that were part of the "GI" project.

Quote from Jean-Claude Kalache, the lighting Director of Photography on Monsters University:
"Physically based lights simplified our setups dramatically. Our master lighting productivity doubled and our shot lighting efficiency went up by 55%. We used multi-threading for interactive and batch renders. The overwhelming feedback we got from lighters was that the new lights were easy to use and provided more time for artistic exploration. We also used the physically based lights to create meaningful pre-vis lighting for the Sets Shading department. Every major set was pre-vised with these lights on average in 1-2 days. We used IBL[14] techniques for the Character department, providing half a dozen lighting setups. The lighting setups for both departments were normalized, which resulted in more robust shader behavior under any lighting conditions."

Quote from Chris Bernardi, the set shading lead on Monsters University:
"Physical shading required us to unify Specular and Reflection from the shading side: we had to rethink many long-held approaches, but the result was a more coherent representation of specular across our sets. There were far fewer adjustments that needed to be made in the context of lighting and our materials became far more portable as a result. A secondary result, was that since the lighting rigs became simpler to work with, it was easier for Lighting to deliver customized lighting setups for individual sets. This gave us the opportunity to fine tune our shaders in the context of some meaningful lighting, which also helped us deliver cleaner shading inventory in return. The combination of rendering with HDR[15] and the energy conservation of the specular model made it easier to hit a wider variety of materials and most of the tweaking was primarily with the roughness control."

[14] Image Based Lighting: this is the illumination mode triggered by our `dome` area light

[15] High Dynamic Range texture on the `dome` light

Figure 10: Monsters University: daylight scene ©Disney/Pixar 2013

Figure 11: The Blue Umbrella: nightime rainy scene ©Disney/Pixar 2013

# References


[Ashikhmin and Premoze 2007] Ashikhmin, M., and Premoze, S. 2007. Distribution-based BRDFs. Tech. rep., Program of Computer Graphics, Cornell University.

[Burley 2012] Burley, B., 2012. Physically-based shading at Disney. `http://selfshadow.com/s2012-shading/`.

[Christensen et al. 2012] Christensen, P. H., Harker, G., Shade, J., Schubert, B., and Batali, D., 2012. Multiresolution radiosity caching for efficient preview and final quality global illumination in movies. `http://graphics.pixar.com/library/RadiosityCaching/`.

[Clarberg and Akenine-Möller 2008] Clarberg, P., and Akenine-Möller, T. 2008. Exploiting Visibility Correlation in Direct Illumination. *Computer Graphics Forum (Proceedings of EGSR) 27*, 4.

[Cook and Torrance 1982] Cook, R. L., and Torrance, K. E. 1982. A reflectance model for computer graphics. *ACM Trans. Graph. 1*, 1 (Jan.), 7–24.

[Heckbert 1990] Heckbert, P. S. 1990. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph. 24*, 4 (Sept.), 145–154.

[Hery and Ramamoorthi 2012] Hery, C., and Ramamoorthi, R. 2012. Importance sampling of reflection from hair fibers. *Journal of Computer Graphics Techniques (JCGT) 1*, 1 (June), 1–17.

[Hoffman 2013] Hoffman, N., 2013. Background: Physics and math of shading. `http://selfshadow.com/s2013-shading/`.

[Kalos and Whitlock 1986] Kalos, M. H., and Whitlock, P. A. 1986. *Monte Carlo Methods*. John Wiley  Sons.

[Kensler 2013] Kensler, A. 2013. Correlated multi-jittered sampling. Tech. Rep. TM-13-01, Pixar Animation Studios.

[Mehta et al. 2012] Mehta, S. U., Ramamoorthi, R., Meyer, M., and Hery, C. 2012. Analytic tangent irradiance environment maps for anisotropic surfaces. *Comp. Graph. Forum 31*, 4 (June), 1501–1508.

[Pharr and Humphreys 2004] Pharr, M., and Humphreys, G., 2004. Infinite area light source with importance sampling. `http://www.pbrt.org/plugins/infinitesample.pdf`.

[Pharr and Mark 2012] Pharr, M., and Mark, W. R., 2012. ispc: A SPMD compiler for high-performance cpu programming. `http://llvm.org/pubs/2012-05-13-InPar-ispc.html`.

[Ramamoorthi and Hanrahan 2001] Ramamoorthi, R., and Hanrahan, P. 2001. An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '01, 497–500.

[Shirley et al. 1996] SHIRLEY, P., WANG, C., AND ZIMMERMAN, K. 1996. Monte carlo techniques for direct lighting calculations. *ACM Trans. Graph. 15*, 1 (Jan.), 1–36.

[Snyder 1996] SNYDER, J. M. 1996. Area light sources for real-time graphics. Tech. Rep. MSR-TR-96-11, Microsoft Research, Advanced Technology Division, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052.

[Talbot et al. 2005] TALBOT, J. F., CLINE, D., AND EGBERT, P. K. 2005. Importance resampling for global illumination. In *Proc. EGSR*.

[The RenderMan Team 1987-2013] THE RENDERMAN TEAM, 1987-2013. Shading language (RSL). `http://renderman.pixar.com/resources/current/rps/shadingLanguage.html`.

[The RenderMan Team 2009] THE RENDERMAN TEAM, 2009. Shader objects and co-shaders. `http://renderman.pixar.com/resources/current/rps/shaderObjects.html`.

[The RenderMan Team 2011] THE RENDERMAN TEAM, 2011. Area shadowing support. `http://renderman.pixar.com/resources/current/rps/areaShadow.html`.

[The RenderMan Team 2012] THE RENDERMAN TEAM, 2012. Physically plausible shading in RSL. `http://renderman.pixar.com/resources/current/rps/physicallyPlausibleShadingInRSL.html`.

[The RenderMan Team 2013a] THE RENDERMAN TEAM, 2013. Geometric area lights. `http://renderman.pixar.com/resources/current/rps/geometricAreaLights.html`.

[The RenderMan Team 2013b] THE RENDERMAN TEAM, 2013. New photon mapping features. `http://renderman.pixar.com/resources/current/rps/newPhotonMapping.html`.

[Veach and Guibas 1995] VEACH, E., AND GUIBAS, L. J. 1995. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '95, 419–428.

[Villemin and Hery 2012] VILLEMIN, R., AND HERY, C., 2012. Porting RSL to C++. `http://renderman.pixar.com/resources/current/rps/portingRSLtoC.html`.

[Walter et al. 2007] WALTER, B., MARSCHNER, S. R., LI, H., AND TORRANCE, K. E. 2007. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'07, 195–206.

[Walter 2005] WALTER, B. 2005. Notes on the Ward BRDF. Tech. Rep. PCG-05-06, Program of Computer Graphics, Cornell University.