

Real-Time BRDF-based Lighting using Cube-Maps

NVIDIA Corporation

Chris Wynn

Please send me comments/questions/suggestions

<mailto:chris.wynn@nvidia.com>

Introduction and Overview

The goal of this paper is to present a practical overview and an implementation guide for real-time rendering using Bi-directional Reflectance Distribution Functions (BRDFs). This document assumes that the reader has a basic understanding of BRDFs and that they are familiar with real-time rendering of polygonal models.

A BRDF can be thought of as a weighting function that describes how light is reflected when it makes contact with a surface. In general, it is a function of wavelength as well as two directions – an incoming light direction \mathbf{w}_i , and an outgoing viewer direction \mathbf{w}_o . Since these two directions are unit vectors, it's convenient to write these vectors as $\mathbf{w}_i = (\theta_i, \phi_i)$ and $\mathbf{w}_o = (\theta_o, \phi_o)$. In this notation (known as spherical coordinates), the angle θ represents the angular rotation of a vector from being aligned with the up vector and the angle ϕ represents the angular rotation of a vector in the plane perpendicular to the up vector. Figure 1 shows intuitively how these two angles together define a direction on a hemisphere with radius = 1.

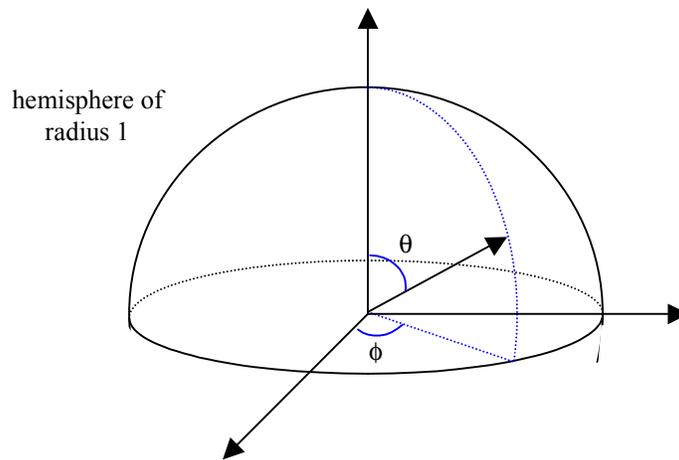


Figure 1. Relationship between a direction vector and associated angles θ and ϕ .

Suppose we have surface point along with a local coordinate system defined by the normal, tangent, and binormal at the surface point. Given a light vector, \mathbf{w}_i , and a view

vector, \mathbf{w}_o , (both defined relative to the local coordinate system) the general BRDF lighting equation for a single point light source is

$$L_o = BRDF(\theta_i, \phi_i, \theta_o, \phi_o) L_i \cos \theta_i$$

where L_i represents the intensity of the light arriving at the surface and L_o represents the intensity of the surface point as seen from direction \mathbf{w}_o . For multiple light sources the lighting equation is

$$L_o = \sum_{j=1}^n BRDF(\theta_i^j, \phi_i^j, \theta_o, \phi_o) L_i^j \cos \theta_i^j .$$

where L_i^j is the intensity of the j^{th} light source and $w_i^j = (\theta_i^j, \phi_i^j)$ is the direction to the j^{th} light source.

Since a BRDF is a 4D function, ideally it would be nice to be able to use a hardware accelerated 4D lookup table (i.e. a 4D texture) to store the BRDF and then at run-time do a per-pixel texture lookup to compute the lighting equation above. Unfortunately, the current generation of graphics hardware does not provide support for 4D textures. As a consequence, we have to try to come up with a clever way of computing the same expression using the features of currently available graphics accelerators.

One way to compute this lighting expression is by using two cube-maps (or two 2D textures) to approximate the 4D function. The basic idea is to in a preprocess, take the 4D BRDF and decompose or “separate” it into the product of two 2D functions (i.e. $BRDF(\theta_i, \phi_i, \theta_o, \phi_o) = G(\theta_i, \phi_i) \cdot H(\theta_o, \phi_o)$). Then at runtime, rather than using a single 4D texture lookup per-pixel, a pair of cube-map lookups (or 2D texture lookups) can be modulated together to compute the BRDF weighting function. Using this approach, the BRDF lighting equation for a single point light becomes

$$L_o = G(\theta_i, \phi_i) \cdot H(\theta_o, \phi_o) L_i \cos \theta_i$$

where the functions $G(\theta_i, \phi_i)$ and $H(\theta_o, \phi_o)$ correspond to the results obtained from a texture lookup.

So loosely speaking, the rendering approach consists of two phases – a preprocess stage which is responsible for separating the BRDF into two 2D functions represented as a pair of texture maps (or cube-maps) and a runtime stage which is responsible for reconstructing the BRDF and computation of the BRDF lighting equation.

The Separation Preprocess

There are a couple of ways to perform the separation preprocess, but this paper will only discuss one such approach called Normalized Decomposition¹. This approach has the advantage that it is relatively simple and for the most part fast. Also, it can be written in such a way that its memory requirements are not overly taxing for today's pc's.

At a high level, the basic idea of this decomposition strategy is to take the 4D function and project it down into a two-dimensional space. In this two-dimensional space, some simple math operations are then performed which produce two 2D functions. This all sounds a bit cryptic at the moment, but it will become clearer as the details are revealed.

To understand how this process works, let's first suppose we were going to create a simple 4D texture to represent the BRDF. Since the BRDF is a function of 4 parameters, namely $(\theta_i, \phi_i, \theta_o, \phi_o)$ it makes sense to pick a sampling resolution and sample each input parameter uniformly over the domain of possible input values. So if we wanted to create a 16x16x16x16 texture, we would sample each parameter uniformly 16 times over its domain. This leads to the question of what is the domain of each input parameter? The input parameters represent direction vectors on a unit hemisphere relative to a plane and a normal (see figure 1). From this figure and our knowledge of spherical coordinates we can see that by allowing θ to vary over $[0, \text{PI}/2]$ and ϕ to vary over $[0, 2\text{PI})$, we can define any incoming or outgoing direction vector in spherical coordinates. This means that our θ inputs should be sampled 16 times over the domain $[0, \text{PI}/2]$ and our ϕ inputs should be sampled 16 times over the domain $[0, 2\text{PI})$. This naturally leads to the following pseudocode for computing a 4D texture.

```
double deltat = (0.5 * M_PI) / (16-1);
double deltap = (2.0 * M_PI) / 16;
double theta_i, phi_i;
double theta_o, phi_o;

for ( int h = 0; h < 16; h++ )
  for ( int i = 0; i < 16; i++ )
    for ( int j = 0; j < 16; j++ )
      for ( int k = 0; k < 16; k++ )
        {
          theta_o = h * deltat;
          phi_o   = i * deltap;
          theta_i = j * deltat;
          phi_i   = k * deltap;
          /* Compute or lookup the brdf value. */
          val = f( theta_i, phi_i, theta_o, phi_o )
          /* Store it in a 4D array. */
          BRDF[h][i][j][k] = val;
        }
```

Code Listing 1. Sampling the 4D function.

¹ The Normalized Decomposition as well as another approach are described in detail in the references [3] and [4].

In Code Listing 1, you may notice that the `delta_t` value was computed by subtracting one from the sampling resolution before doing the division. The reason this is done is because we would like the samples in θ to cover the domain $[0, \text{PI}/2]$ and to include a sample at each of the endpoints of the domain. For ϕ , it is not necessary to do the same thing. The reason this is the case is that while we desire the samples in ϕ to cover the domain $[0, 2\text{PI}]$, the samples taken at 0 and at 2PI will produce the same value because 0 and 2PI correspond to the same amount of rotation in spherical coordinates. This is a very subtle point, but something worth mentioning as a coding precaution.

Also, you may have noticed that the value returned from the lookup function is written in bold. Recall that a BRDF is really a vector-valued function and can be thought of as a 3-component R, G, B vector. Everything regarding BRDFs that is discussed in this tutorial, really applies to each of the R, G, and B color components individually, except where noted. Thus the lookup function in the code listing actually returns a trio of values representing the BRDF.

Now that we know what the basic pseudocode looks like for creating a 4D texture, let's talk about how this gets mapped into a 2D space. Notice how in the innermost loop of the pseudocode the result of the BRDF lookup is being stored into a four-dimensional array. Instead of storing the values in a four-dimensional array, what we can do is store the data in a matrix. The way that we do this might seem tricky at first but it is really not too bad. Suppose for simplicity that we chose to sample the 4D BRDF 2 times in each dimension. This means there would be a total of $16 = 2 \times 2 \times 2 \times 2$ samples from the 4D space that must be mapped down into our matrix representation. The way the mapping is done is by allowing each row to represent a fixed outgoing direction \mathbf{w}_o and allowing each column of the matrix to represent a fixed incoming direction \mathbf{w}_i . Each position in the matrix then corresponds to specific incoming and outgoing directions and the BRDF value associated with each position is the value of the BRDF at the associated incoming and outgoing directions. To understand this better consider the mapping of BRDF samples to the matrix as shown in figure 2.

$$\begin{array}{l}
 \theta_{o0}, \phi_{o0} \\
 \theta_{o0}, \phi_{o1} \\
 \theta_{o1}, \phi_{o0} \\
 \theta_{o1}, \phi_{o1}
 \end{array}
 \begin{array}{c}
 \begin{array}{cccc}
 \theta_{o0}, \phi_{o0} & \theta_{o0}, \phi_{o0} & \theta_{o0}, \phi_{o0} & \theta_{o0}, \phi_{o0} \\
 \left[\begin{array}{cccc}
 F(\theta_{i0}, \phi_{i0}, \theta_{o0}, \phi_{o0}) & F(\theta_{i0}, \phi_{i1}, \theta_{o0}, \phi_{o0}) & F(\theta_{i1}, \phi_{i0}, \theta_{o0}, \phi_{o0}) & F(\theta_{i1}, \phi_{i1}, \theta_{o0}, \phi_{o0}) \\
 F(\theta_{i0}, \phi_{i0}, \theta_{o0}, \phi_{o1}) & F(\theta_{i0}, \phi_{i1}, \theta_{o0}, \phi_{o1}) & F(\theta_{i1}, \phi_{i0}, \theta_{o0}, \phi_{o1}) & F(\theta_{i1}, \phi_{i1}, \theta_{o0}, \phi_{o1}) \\
 F(\theta_{i0}, \phi_{i0}, \theta_{o1}, \phi_{o0}) & F(\theta_{i0}, \phi_{i1}, \theta_{o1}, \phi_{o0}) & F(\theta_{i1}, \phi_{i0}, \theta_{o1}, \phi_{o0}) & F(\theta_{i1}, \phi_{i1}, \theta_{o1}, \phi_{o0}) \\
 F(\theta_{i0}, \phi_{i0}, \theta_{o1}, \phi_{o1}) & F(\theta_{i0}, \phi_{i1}, \theta_{o1}, \phi_{o1}) & F(\theta_{i1}, \phi_{i0}, \theta_{o1}, \phi_{o1}) & F(\theta_{i1}, \phi_{i1}, \theta_{o1}, \phi_{o1})
 \end{array} \right]
 \end{array}
 \end{array}$$

Figure 2. Mapping of BRDF samples to the 2D matrix representation (sampling rate of 2x in both θ and ϕ).

In the figure, for each row of the matrix, θ_o and ϕ_o remain fixed. Within a row, however, the incoming direction is allowed to vary. The way this is done is by first fixing a θ_i and then allowing ϕ_i to vary over its domain of input values, then fixing θ_i with its next sample value and then again allowing ϕ_i to vary over its domain of input values. This is a somewhat trivial example, but the basic idea is that each row is essentially a two-dimensional array unrolled into a one-dimensional array. Likewise, each column can be considered to be a two-dimensional array unrolled into a one-dimensional array. The Pseudocode for sampling the 4D BRDF N times in each dimension and mapping the function into a matrix is given in code listing 2.

```

double deltat = (0.5 * M_PI) / (N-1);
double deltap = (2.0 * M_PI) / N;
double theta_i, phi_i;
double theta_o, phi_o;

for ( int h = 0; h < N; h++ )
  for ( int i = 0; i < N; i++ )
    for ( int j = 0; j < N; j++ )
      for ( int k = 0; k < N; k++ )
        {
          theta_o = h * deltat;
          phi_o   = i * deltap;
          theta_i = j * deltat;
          phi_i   = k * deltap;
          /* Compute or lookup the brdf value. */
          val = f( theta_i, phi_i, theta_o, phi_o );
          /* Store it in a N^2 x N^2 matrix. */
          BRDFMatrix[h*N+i][j*N+k] = val;
        }

```

Code Listing 2. Mapping the 4D BRDF into a 2D matrix. Each input domain is sampled N times.

Now that we know how to map the 4D BRDF into a 2D matrix, we can now talk about the separation process – the process of separating the 4D function represented in the matrix into two 2D functions. The goal of the process we are about to describe is to produce two functions $G(\theta_i, \phi_i)$ and $H(\theta_o, \phi_o)$ such that

$$BRDF(\theta_i, \phi_i, \theta_o, \phi_o) \cong G(\theta_i, \phi_i) \cdot H(\theta_o, \phi_o).$$

The steps for performing a separation using the Normalized-Decomposition approach are as follows:

1. For each row, compute the norm² of all BRDF values in that row.
2. For each column, divide each column value by its corresponding row-norm and compute the mean average of these “normalized” values.

² A *norm* is a type of vector average. Some common norms include:

$$\text{One-norm: } \|\mathbf{x}\|_1 = |x_1| + |x_2| + \dots + |x_n|$$

$$\text{Two-norm: } \|\mathbf{x}\|_2 = \left(|x_1|^2 + |x_2|^2 + \dots + |x_n|^2 \right)^{\frac{1}{2}}$$

$$\text{Max-norm: } \|\mathbf{x}\|_\infty = \text{MAX}(|x_1|, |x_2|, \dots, |x_n|)$$

At first this may sound and seem complicated, but it is actually fairly simple. The matrix and the values computed in each step are shown in figures 3 and 4.

From figure 3 we can observe that the resulting $n \times 1$ “norm vector” encodes for each row a norm value. More specifically, this means that for each sampled outgoing direction (θ_o, ϕ_o) there is a corresponding norm value in the norm vector. As a result, this norm vector can serve as the function $H(\theta_o, \phi_o)$. Similarly, from figure 4 we observe that the resulting $1 \times n$ “average vector” encodes for each column an average value. Specifically, this means that for each sampled incoming direction (θ_i, ϕ_i) there is a corresponding average value in the average vector. As a result, this average vector can serve as the function $G(\theta_i, \phi_i)$.

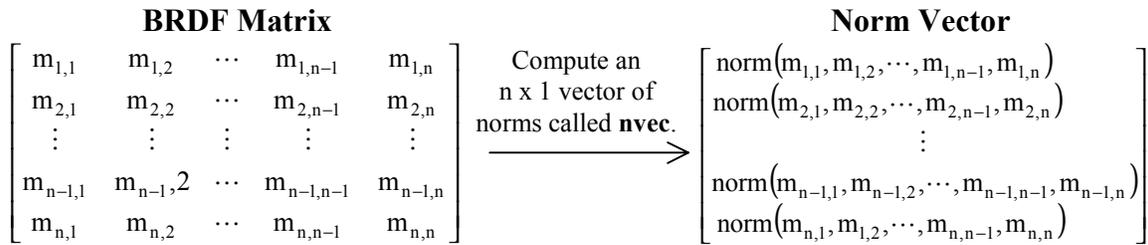


Figure 3. Step 1 - A norm is computed for each row resulting in an $n \times 1$ vector of norms called *nvec*. (This is performed for each color channel independently.)

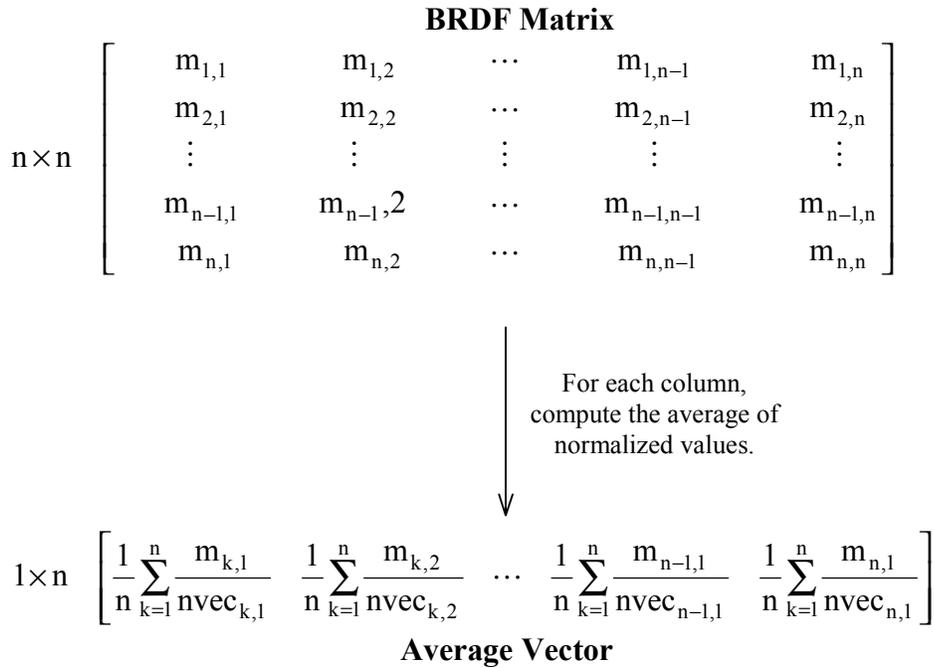
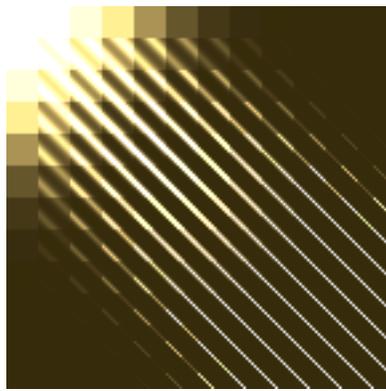


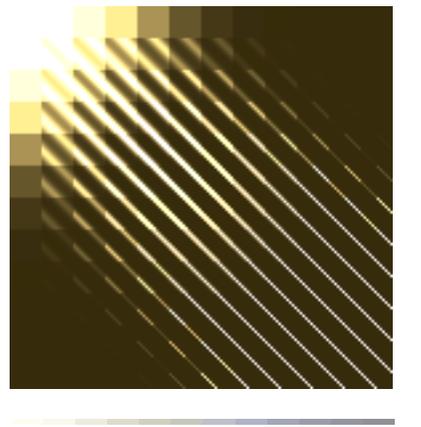
Figure 4. Step 2 – The average of normalized column values is computed. This results in a $1 \times n$ vector of averages. (This computation is performed for each color channel independently.)

Using these two vectors for $G(\theta_i, \phi_i)$ and $H(\theta_o, \phi_o)$, we can do a fairly good job of approximating the original BRDF.

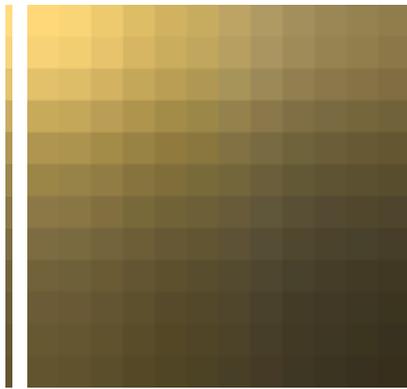
To illustrate this separation process and the quality of the reconstruction that we may achieve, consider the sequence of images shown in figure 5. Figure a) shows an image of a 144x144 image of a BRDF matrix. This image corresponds to a 12x12x12x12 sampling of the BRDF over its input domain. This image is derived directly from the measured or analytical BRDF data and can be thought of as a true representation of the BRDF. The separation process produces two vectors $G(\theta_i, \phi_i)$ and $H(\theta_o, \phi_o)$ as shown in b). Finally, figure c) shows how the approximate BRDF is reconstructed for any incoming and outgoing direction combination.



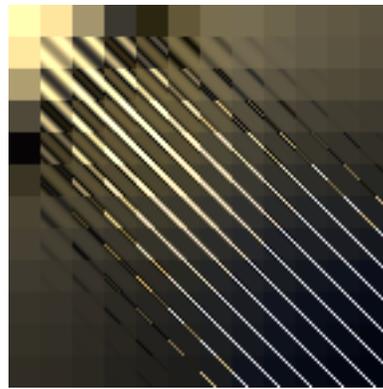
a) Image of the sampled “true” BRDF matrix.



b) The separation of the BRDF matrix into functions G and H.



c) The “approximate” BRDF matrix reconstructed using $G * H$.



d) The “difference” BRDF matrix which represents the error associated with the approximation.

Figure 5. The BRDF separation and reconstruction process.

Figures a) and c), demonstrate that the approach of separating a BRDF into two functions and then reconstructing can produce results that significantly differ from the true BRDF. Figure d) is a representation of the absolute error introduced in the separation and reconstruction process. Later in this tutorial, we'll discuss a way of reducing the error by parameterizing the BRDF differently.

Limiting the Dynamic Range

Before we discuss how to create texture maps for use in real-time rendering using BRDFs, we need to first address an issue pertaining to the large dynamic range of BRDFs. Recall that BRDFs can be unbounded. That is, rather than being functions which take on range values limited to $[0,1]$, an individual BRDF may take on values much larger than 1 and in many cases may take on values in the thousands for certain incoming and outgoing directions. Typically, the percentage of function values that are extremely large is small; nonetheless, something must be done in order to keep these extreme values from disproportionately impacting the quality of the separation process. Additionally, the BRDF values will eventually need to be mapped to the $[0,1]$ required by current graphics hardware. If the dynamic range of the BRDF is too large, the mapping to the $[0,1]$ range will yield poor results – specifically images will appear too dark.

The simplest and most practical way of limiting the dynamic range is to clamp the values of the BRDF to some maximum value. This may be a value that was experimentally chosen to yield visually pleasing results or alternatively, the distribution of the BRDF may be considered to select a maximum value that ensures some percentage of the original BRDF range is maintained. Regardless of how this maximum allowable value is determined, it's necessary select one and clamp the BRDF values to ensure that no values larger than the maximum get stored in the BRDF matrix.

In this way, by artificially limiting the range of the BRDF, its possible to decrease the negative influence of extreme values and also ensure that the significant range of the BRDF maps to visible regions of the $[0,1]$ range.

Rescaling the Dynamic Range of the G and H Functions

Now that we know how to determine the functions G and H, the goal is to create textures that can be used at run-time to reconstruct the BRDF. Current graphics hardware requires that textures be limited to the range $[0,1]$. Unfortunately, G and H are not necessarily limited to that range. Before we can actually create textures, we need to discuss how to determine modified G and H functions that are limited to the range $[0,1]$. Fortunately, there are some clever properties that we take advantage to determine these modified functions.

Recall, that

$$\text{BRDF} = G * H$$

Denoting the maximum of G and H by M_G and M_H respectively, we have

$$\text{BRDF} = M_G * M_H * (G/M_G) * (H/M_H)$$

If we define $\hat{G} = (G/M_G)$ and $\hat{H} = (H/M_H)$

$$\text{BRDF} = M_G * M_H * \hat{G} * \hat{H}$$

Letting $\delta = M_G * M_H$, we have

$$\text{BRDF} = \delta * \hat{G} * \hat{H}$$

Notice that the functions \hat{G} and \hat{H} are now both limited to the range $[0,1]$ and may be used as textures used to approximate the true BRDF. The only problem that remains is that the hardware must be able to compute on a per-fragment (per-pixel) basis, $\text{BRDF} = \delta * \hat{G} * \hat{H}$ where δ is some arbitrary floating point scaling value with arbitrary range. Unfortunately, most hardware does not allow for scaling by such an arbitrary value during texture combining. In general however, current hardware does allow for a scaling by an integer value of 1, 2, or 4. In order to reformulate this equation so that this computation can be performed on a per-fragment basis, we use a clever trick.

What we can do is select the minimum integer value D in $\{1,2,4\}$ such that $\delta \leq D$ (but if $\delta > 4$ we just select $D = 4$). Then,

$$\begin{aligned} \text{BRDF} &= D * (\delta / D) * \hat{G} * \hat{H} \\ \text{BRDF} &= D * \text{sqrt}(\delta / D) * \text{sqrt}(\delta / D) * \hat{G} * \hat{H} \\ \text{BRDF} &= D * \text{sqrt}(\delta / D) * \hat{G} * \text{sqrt}(\delta / D) * \hat{H} \end{aligned}$$

If we scale the functions of \hat{G} and \hat{H} , we have $\hat{g} = \text{sqrt}(\delta / D) * \hat{G}$ and $\hat{h} = \text{sqrt}(\delta / D) * \hat{H}$. Consequently,

$$\text{BRDF} = D * \hat{g} * \hat{h}$$

Since there is the possibility that $\delta > D$ (i.e. when $\delta > 4$), the functions \hat{g} and \hat{h} must be clamped to have values no larger than 1 so that they may be used as textures with valid range $[0,1]$. The effective result of this clamping is that the dynamic range of the BRDF is further clamped. In general, this does not severely impact quality.

The above discussion glosses over a very important issue that should be clarified. Recall that a BRDF is really a vector-function. That is, it takes on potentially different values for each of its red, green, and blue color channels. Thus the functions G and H , and the derived functions (\hat{G} , \hat{H} and \hat{g} , \hat{h}) all have values that are defined per-color channel. Current hardware, however, only allows for a single per-fragment integer scaling of 1, 2, or 4 for all three color channels on a single pass. Specifically, it's not possible to scale each of the red, green, and blue color channels by a per-channel integer scale value on a single pass. To circumvent this problem, one D value must be selected that is suitable for

all of the color channels. Thus the D value that should be used during the rescaling process is the maximum D value over all of the color channels.

Pseudocode for rescaling the dynamic range of the H and G functions and determining the D scale value is presented in code listing 3. Notice that the δ values and rescaled G and H functions are computed on a per-channel basis, but that a single D scale value is derived and used for the per-channel computations of the \hat{g} and \hat{h} functions.

```
// Compute delta for each color channel.
double delta[3];
delta[0] = Mg[0] * Mh[0];
delta[1] = Mg[1] * Mh[1];
delta[2] = Mg[2] * Mh[2];

// Compute "D" scale value.
D = 1;
for ( int cidx = 0; cidx < 3; cidx++ )
{
    if ( delta[cidx] < 1 )          d = 1;
    else if ( delta[cidx] < 2 ) d = 2;
    else                          d = 4;
    if ( d > D ) D = d;
}

// Scale all function values by 1/max to normalize.
// Each function h and g is assumed to have N values.
for( int cidx = 0; cidx < 3; cidx++ )
{
    double invmaxG = 1.0 / maxG[cidx];
    double invmaxH = 1.0 / maxH[cidx];
    for ( i = 0; i < N; i++ )
    {
        g[cidx][i] = invmaxG * g[cidx][i];
        h[cidx][i] = invmaxH * h[cidx][i];
    }
}

// Scale all values by sqrt( delta[cidx] / D )
for( int cidx = 0; cidx < 3; cidx++ )
{
    double term = sqrt( delta[cidx] / (float)D );
    for ( i = 0; i < N; i++ )
    {
        g[cidx][i] = term * g[cidx][i];
        h[cidx][i] = term * h[cidx][i];
        if ( g[cidx][i] > 1 ) g[cidx][i] = 1.0;
        if ( h[cidx][i] > 1 ) h[cidx][i] = 1.0;
    }
}

// Now the functions h and g are limited to [0,1] range
// for each of the red, green, and blue channels.
// A valid "D" scale value has been computed as well.
```

Code Listing 3. Rescaling the dynamic range of the H and G functions and computing the D scale value.

Resampling the Functions to Create Textures

Now that we know how to rescale the textures such that they have the proper range, we can finally create the textures that will actually be used during the run-time rendering phase. The goal here is to encode each function into a texture in a manner such that at run-time, the correct function value can be retrieved on a per-fragment basis.

Given a set of spherical coordinates at each vertex of a triangle, the goal is to on a per-fragment basis, retrieve the texel (i.e. function value) that corresponds to function value associated with the linearly interpolated spherical coordinates of the fragment.

A simplified 2D version of this goal is depicted in figure 6.

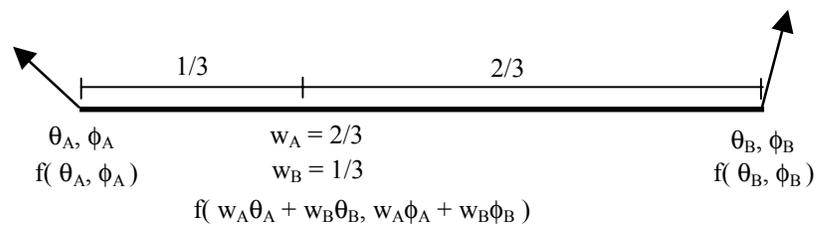


Figure 6. Linear interpolation of spherical coordinates

Depending on what type of texturing hardware is available, any one of 3 types of textures can be used at run-time: 2D textures, dual-paraboloid maps, or cube-maps. Using cube-maps for spherical interpolation is preferred to both dual-paraboloid maps and 2D textures because they have better interpolation properties and produce relatively few interpolation artifacts. Consequently we will describe how to encode both the G and H functions into cube-maps that may be used at run-time for BRDF-based lighting. 2D textures on the other hand are most susceptible to incorrect spherical interpolation. Since 2D textures are available on almost every graphics accelerator currently available, we'll also describe a fallback method that can be used to encode the two functions when cube-mapping is not available.

Resampling to Create Cube-Maps

Encoding the directional G or H function into a cube-map is fairly straightforward. The idea behind encoding the function into a cube-map is to think of each texel location within the cube-map as representing a direction, and the value stored in the cube-map as representing the function value for that direction. Since each function H and G is defined only over the hemisphere of incoming directions, only one half of the texels in the cube map need store any valid data. For simplicity, consistency, and debugging purposes it makes sense to encode the function on the upper half (+z) of the cube-map only. (To minimize memory requirements, it is possible to encode both hemisphere functions into a single cube-map. For the sake clarity, this optimization is not described in this tutorial.)

An overview of the algorithm follows:

For each face of the cube map

For each texel of the face

0. If texel is on bottom half of hemisphere write (0,0,0) RGB value into texel and continue.
1. Compute the vector from the origin through the texel center
2. Compute the spherical coordinate (θ, ϕ) representation of the vector.
3. Look up the red, green, and blue function values in the G or H function.
4. Write the RGB values into the texel.

Step 3 of the algorithm requires some clarification. Recall that the functions H and G are discrete functions that have per-channel values defined only for hemispherical directions for which an incoming or outgoing sample direction was used in sampling the BRDF. (Remember that each row or column of the BRDF matrix corresponded to a single incoming or outgoing direction given in its (θ, ϕ) representation. As a result, each element of the H or G “separation vectors” corresponds to the function value associated with a particular direction in its spherical coordinate representation – a specific (θ, ϕ) pair. However, when encoding this function into the cube-map, it’s likely that the spherical coordinates for the texel, (θ_t, ϕ_t) , do not correspond exactly to any of the (θ, ϕ) pairs that were used during BRDF sampling. Instead, the texel’s θ_t value falls in between two of the sampled domain positions (say somewhere in the range of $[\theta_i, \theta_{i+1}]$). Likewise, the texel’s ϕ_t value falls in between two of the sampled domain positions (say somewhere in the range of $[\phi_j, \phi_{j+1}]$). This situation is shown in figure 7.

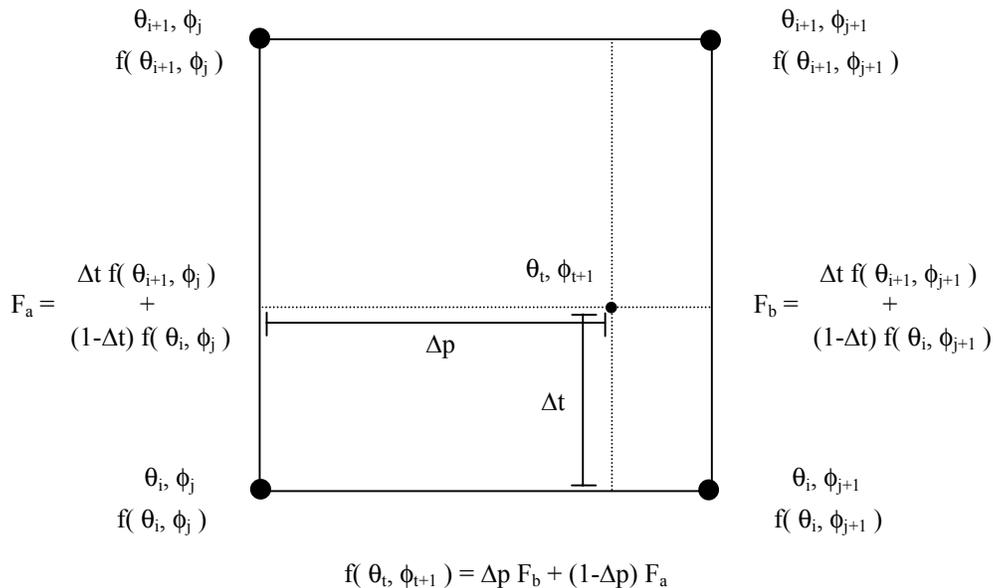


Figure 7. Bilinear interpolation to determine the function value(s) for a texel.

In order to determine the function value for the texel, either a nearest-neighbor approach or a bilinear interpolation approach could be used. Using a nearest-neighbor approach generates poor results during the run-time reconstruction stage so bilinear interpolation of the surrounding points should be used. When doing the bilinear interpolation, it's important to remember that ϕ corresponds to the angular rotation of a vector projected into a reference plane (see figure 1). This means that there is a "wrap-around" that occurs at $\phi_0 = 0$. So it's necessary to determine when ϕ_t is in the range $[\phi_{n-1}, 2\pi]$ (equivalently $[\phi_{n-1}, \phi_0]$) and to interpolate appropriately.

Run-Time Rendering

Now that we've seen how to perform the pre-processing necessary for generating a separated BRDF, we can now consider the run-time portion of the algorithm. Given a scaling value (1, 2, or 4) and the two cube-maps representing the rescaled functions H and G described above, the goal is to evaluate the BRDF lighting equation on a per-pixel basis. Recall, that the BRDF-lighting equation is

$$L_o = BRDF(\theta_i, \phi_i, \theta_o, \phi_o) L_i \cos \theta_i$$

Since we are using the two functions computed during the pre-process we have

$$L_o = G(\theta_i, \phi_i) \cdot H(\theta_o, \phi_o) L_i \cos \theta_i$$

$$L_o \cong G(\theta_i, \phi_i) \cdot H(\theta_o, \phi_o) L_i \cos \theta_i$$

which is equivalent to

$$L_o = D \cdot g(\theta_i, \phi_i) \cdot h(\theta_o, \phi_o) L_i \cos \theta_i$$

after the re-scaling operations described above. Since the g and h functions have been encoded into cube-maps, we can use the two cube-map textures and multi-texturing to compute the product of the g and h functions. Also, the scaling by D can be performed using a scale operation. Finally, the $L_i \cos \theta_i$ term can be computed using a single, colored hardware light to compute $L_i (\mathbf{N} \cdot \mathbf{L}) = L_i \cos \theta_i$. This can all be done on a single pass and leads to the following high-level algorithm for rendering an object with BRDF-based lighting:

1. For each vertex, compute the texture coordinates used to index the G texture.
2. For each vertex, compute the texture coordinates used to index the H texture.
3. Enable a hardware light to compute $L_i \cos \theta_i$ (diffuse Lambertian lighting only).
4. Bind and enable the G texture to texture unit 0.
5. Bind and enable the H texture to texture unit 1.

6. Setup the texture combining to compute $D * t_0 * t_1 * \text{incoming fragment color}$.
7. Render the object using the texture coordinates computed in steps 1 and 2.

In OpenGL, setup of the texture combining hardware in step 6 can be achieved through either the `NV_texture_env_combine4`, `EXT_texture_combine`, or `NV_register_combiners` extension. The fundamental goal of this step is to set the appropriate hardware state so that fragment combining computes a result that is the product of the incoming fragment, the result of texture unit 0, the result of texture unit 1, and the scalar D .

Possibly the most difficult aspect of the above algorithm is steps 1 and 2 – computing the appropriate texture coordinates. Since G is a function of the incoming (light) direction, the texture coordinates required per-vertex is a direction vector indicating the direction from the vertex to the light source. Similarly, since H is a function of the outgoing (view) direction, the texture coordinates required per-vertex is a direction vector indicating the direction from the vertex to the viewer.

While this seems simple enough to compute, the tricky part arises in that these two direction vectors must be specified in a local tangent space (also known as TBN-space). This requires that a local tangent, binormal, and normal be defined for every vertex of the model. (For a good explanation of tangent space and how tangent space vectors can be computed for an arbitrary polygonal model see [1]). Given a tangent, T , binormal B , and normal, N , defining a local coordinate system, the texture coordinates specified to index G must be the incoming direction defined relative to that coordinate system and the texture coordinates specified when indexing the H texture must be the outgoing direction defined relative to the local TBN coordinate frame. Code listing 4 demonstrates how to compute texture coordinates for both the H and G textures.

```

// Computes 2 sets of texture coordinates for every vertex of the
// object.  These texture coordinates are used to index the 2 cube
// maps which store the separated BRDF.  The basic idea is as
// follows:  given a light position and viewer position in world
// space, compute a light vector, vl, and a direction vector, ve.
// The first set of texture coordinates are given by the "incoming"
// direction which is the light vector relative to the local tangent
// space.  The second set of texture coordinates are given by the
// "outgoing" direction which is the view vector relative to the
// local tangent space.
// "eye" is the world-space coordinates of the eye position.
// "plight" is the world-space coordinates of the light position.
// "vertexdata" is an array of world-space vertices.
// "tangendata" is an array of tangent vectors in world-space.
// "normaldata" is an array of normal vectors in world-space.
// "binormdata" is an array of binormal vectors in world-space.

void ComputeCubeMapTexCoords( Vec &eye, Vec &plight )
{
    Vectorf vl, ve;
    Vectorf coordo, coordi;

    // For each vertex.
    for ( unsigned int i = 0; i < vertcount; i++ )
    {
        // Compute world-space incoming and outgoing vectors.
        vl = plight - vertexdata[i];
        ve = eye - vertexdata[i];

        // Transform incoming vector to local tangent space.
        coordG.x = dot( vl, tangendata[i] );
        coordG.y = dot( vl, normaldata[i] );
        coordG.z = dot( vl, binormdata[i] );

        // Transform outgoing vector to local tangent space.
        coordH.x = dot( ve, tangendata[i] );
        coordH.y = dot( ve, normaldata[i] );
        coordH.z = dot( ve, binormdata[i] );

        // Store the results in a global array.
        texcoordsG[i] = coordG;
        texcoordsH[i] = coordH;
    }
}

```

Code Listing 4. Computing texture coordinates for indexing the G and H textures.

Improving the Results through Reparametrization

The BRDF separation and reconstruction technique described above works well for some BRDFs and does not work well for others. In particular, some BRDFs do not separate well using the separation previously described above and tend to do well using an alternate *parametrization*. By parameterizing or defining such BRDFs in terms of an alternative set of variables, it may be possible to achieve vastly improved results.

Consider again figure 5. If you make some comparisons between parts a) and c) it won't take too long to figure out that the reconstructed approximated BRDF shown in c) has significant differences from the actual BRDF shown in a). Specifically, you'll probably notice that many of the diagonal features present in the original BRDF were lost during the separation and reconstruction process. The main idea behind reparameterizing the BRDF is to try to re-structure the sampling domains so that there are fewer diagonal features in the true BRDF matrix.

If you recall, in the previously described approach, the BRDF matrix was constructed in a way that had each row corresponding to a fixed outgoing direction and each column corresponding to a fixed incoming direction. Since the BRDF matrix was constructed by sampling the BRDF over the incoming and outgoing directions, this parameterization is generally referred to as the Outgoing-Incoming (OI) parameterization. While many other parameterizations are conceivable, one parameterization that tends to work well is known as the Gram-Schmidt Halfangle-Difference (GSHD) vector parameterization. The idea behind the GSHD parameterization is to define two different direction vectors (that are related to the incoming and outgoing direction) and sample the BRDF over this domain instead of the incoming direction outgoing direction domain.

The derivation and the details of the approach are explained in [...] but the main idea is to sample the BRDF matrix over the domain of the Halfangle vector and a "Difference" vector. The Halfangle vector, \mathbf{h} , is the vector that is halfway between the incoming light vector and the outgoing view vector. This vector is easily computed by normalizing the vector sum of \mathbf{w}_i and \mathbf{w}_o . That is,

$$\mathbf{h} = \text{normalize}(\mathbf{w}_i + \mathbf{w}_o)$$

The Difference vector, \mathbf{d} , is unfortunately not quite so intuitive. The best way to think about \mathbf{d} is to think of it as the incoming direction \mathbf{w}_i relative to a new coordinate frame $\{\mathbf{h}, \mathbf{t}', \mathbf{s}'\}$ where \mathbf{h} corresponds to the "up" direction (i.e. pole of the hemisphere) and \mathbf{t}' and \mathbf{s}' are chosen to form an orthogonal frame. \mathbf{t}' and \mathbf{s}' are created using the Gram-Schmidt orthogonalization process. Given incoming and outgoing directions \mathbf{w}_i and \mathbf{w}_o , halfangle and difference vectors may be computed using Gram-Schmidt process as follows:

$$h = \frac{w_i + w_o}{|w_i + w_o|}$$

$$t'' = T - (T \cdot h)h$$

$$t' = \frac{t''}{|t''|}$$

$$s' = h \times t'$$

$$d = \begin{pmatrix} w_i \cdot t' \\ w_i \cdot s' \\ w_i \cdot h \end{pmatrix}$$

In order to use this alternative parameterization when separating and reconstructing a BRDF, some modifications must be made to the separation process (specifically the process of constructing the BRDF matrix). Modifications must also be made in the way that texture coordinates are computed during the run-time phase of the algorithm.

Sampling the BRDF using the GSHD Parameterization

Sampling the BRDF using the GSHD parameterization requires only some minor modifications in the way in which the original BRDF was sampled. Specifically, instead of sampling the 4D BRDF over the incoming and outgoing directions in spherical coordinates $(\theta_i, \phi_i, \theta_o, \phi_o)$, we instead need to sample the BRDF over the halfangle and difference vector directions in spherical coordinates $(\theta_h, \phi_h, \theta_d, \phi_d)$. The pseudocode for sampling over this alternative sampling domain is shown in code listing 5. Notice that this is just the same algorithm as that of code listing 2 but that only the sampling domain has changed.

```
double deltat = (0.5 * M_PI) / (N-1);
double deltap = (2.0 * M_PI) / N;
double theta_h, phi_h;
double theta_d, phi_d;

for ( int h = 0; h < N; h++ )
  for ( int i = 0; i < N; i++ )
    for ( int j = 0; j < N; j++ )
      for ( int k = 0; k < N; k++ )
        {
          theta_d = h * deltat;
          phi_d = i * deltap;
          theta_h = j * deltat;
          phi_h = k * deltap;
          /* Compute or lookup the brdf value. */
          val = f( ... );
          /* Store it in a N^2 x N^2 matrix. */
          BRDFMatrix[h*N+i][j*N+k] = val;
        }
```

Code Listing 5. Mapping the 4D BRDF into a 2D matrix using the GSHD parameterization. Each input domain is sampled N times.

Also, it's important to observe that in general, the function that performs the BRDF lookup (or computation) is parameterized by $(\theta_i, \phi_i, \theta_o, \phi_o)$, instead of $(\theta_h, \phi_h, \theta_d, \phi_d)$. This implies that in order to look up the appropriate BRDF value for a specific set of input parameters $(\theta_h, \phi_h, \theta_d, \phi_d)$, those parameters must be converted to the corresponding set of parameters in the incoming/outgoing parameterization (i.e. $(\theta_i, \phi_i, \theta_o, \phi_o)$). The relationship between these parameters is defined by the Gram-Schmidt process. Specifically, it can be shown that w_i and w_o can be derived from h and d from the following expressions:

$$w_i = (d.x)t' + (d.y)s' + (d.z)h$$

$$w_o = 2(h \bullet w_i)h - w_i$$

Thus the algorithm for sampling the BRDF using the GSHD parameterization should be modified slightly to do this $(\theta_h, \phi_h, \theta_d, \phi_d)$ to $(\theta_i, \phi_i, \theta_o, \phi_o)$ conversion prior to lookup. The appropriate modified pseudocode is shown in code listing 6.

```
double deltat = (0.5 * M_PI) / (N-1);
double deltap = (2.0 * M_PI) / N;
double theta_h, phi_h, theta_i, phi_i;
double theta_d, phi_d, theta_o, phi_o;

for ( int h = 0; h < N; h++ )
  for ( int i = 0; i < N; i++ )
    for ( int j = 0; j < N; j++ )
      for ( int k = 0; k < N; k++ )
        {
          theta_d = h * deltat;
          phi_d   = i * deltap;
          theta_h = j * deltat;
          phi_h   = k * deltap;

          /* Perform Gram-Schmidt conversion. */
          Vector tprime = t-h*dot(t,h);
          tprime.normalize();

          Vector bprime = cross(h,tprime);
          bprime.normalize();
          wi = tprime*d.x + bprime*d.y + h*d.z;

          wo = h*2.0*dot(h,wi) - wi;
          wo.normalize();

          ConvertVectorToSpherical( wi, &theta_i, &phi_i );
          ConvertVectorToSpherical( wo, &theta_o, &phi_o );

          /* Compute or lookup the brdf value. */
          val = f( theta_i, phi_i, theta_o, phi_o );
          /* Store it in a N^2 x N^2 matrix. */
          BRDFMatrix[h*N+i][j*N+k] = val;
        }
}
```

Code Listing 6. Mapping the 4D BRDF into a 2D matrix using the GSHD parameterization. (Parameter conversion shown)

The remainder of the separation algorithm remains unchanged. Using this alternate parameterization GSHD parameterization, a single row of the true BRDF matrix corresponds to BRDF samples for which the **d** direction (θ_d and ϕ_d) remains fixed and the **h** direction (θ_h and ϕ_h) varies. Similarly, each column of the true BRDF matrix corresponds to BRDF samples for which the **h** direction (θ_h and ϕ_h) remains fixed and the **d** direction (θ_d and ϕ_d) varies. Since the remainder of the separation algorithm remains unchanged, the result of the separation process is two functions $G(\theta_h, \phi_h)$ and $H(\theta_d, \phi_d)$ such that

$$BRDF(\theta_h, \phi_h, \theta_d, \phi_d) \cong G(\theta_h, \phi_h) \cdot H(\theta_d, \phi_d)$$

These functions again get represented as scale-adjusted cube-maps. In order to use these functions during the run-time rendering phase, the only thing that needs to change about the run-time algorithm is the way that texture coordinate generated. This is explained in the next section.

Before looking at how the run-time texture coordinate generation is modified as a result of this alternate parameterization, let's look at an illustration of the separation process using the GSHD parameterization. Figure 8 illustrates the separation process using the GSHD parameterization for the same BRDF as that shown in figure 5.

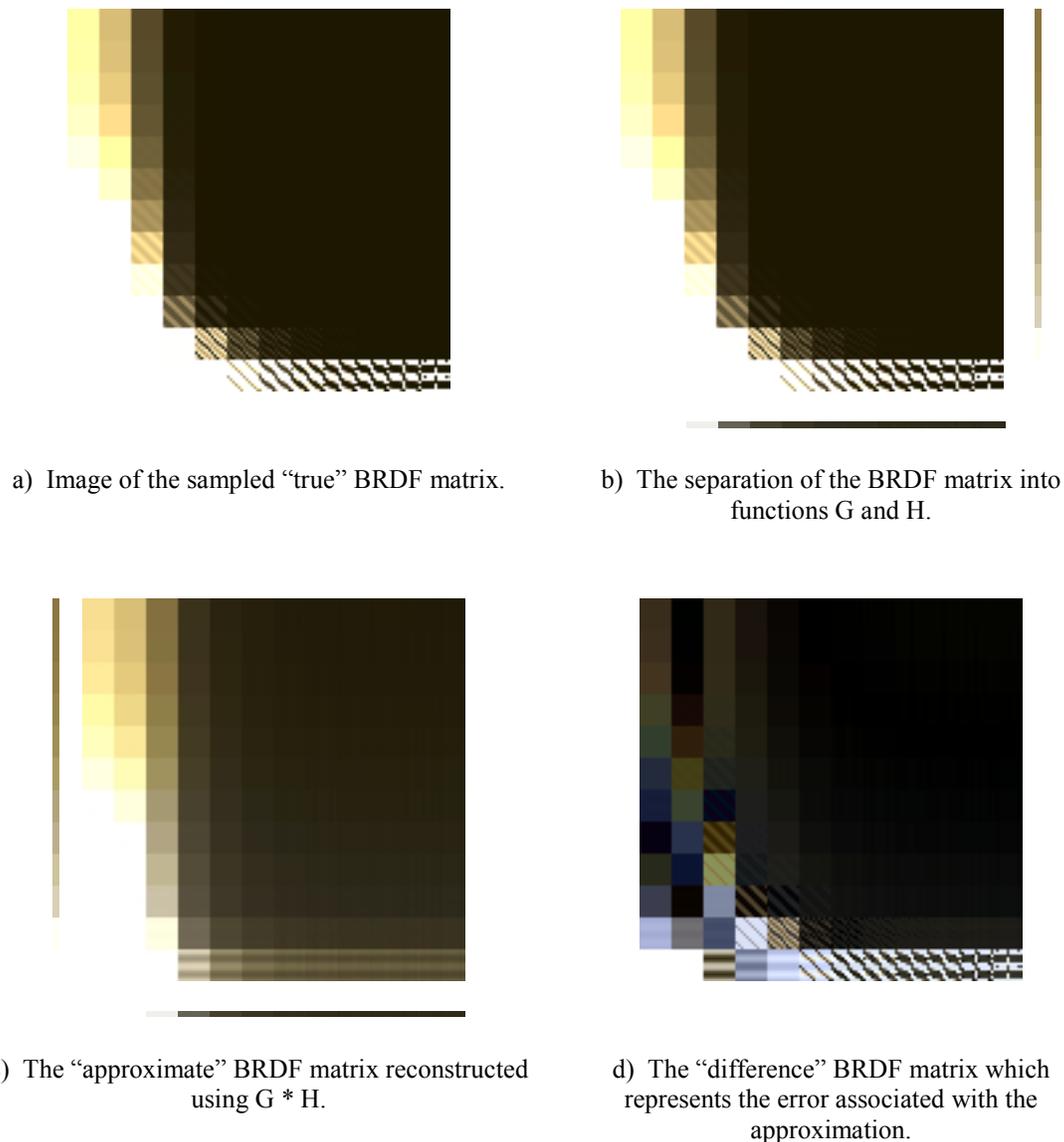


Figure 8. The BRDF separation and reconstruction process using GSHD.

The sequence of images show: a) the true BRDF matrix, b) the separation into functions G and H, c) the reconstruction of the approximate BRDF matrix, and d) the error in the approximation. Notice how for this BRDF, the quality of the reconstruction is vastly improved using the GSHD parameterization. Specifically, the character of the reconstructed BRDF shown in figure 8c) is much more representative of the original true BRDF matrix of image 8a) than image 5c) is of image 5a). In general, the choice of parameterization is an important one. For some BRDFs the OI parameterization works better than GSHD; for others, GSHD is a better choice.

Run-Time Rendering using the GSHD Parameterization

The run-time algorithm for reconstructing the BRDF and computing the BRDF-based lighting using the GSHD parameterization is nearly identical to the approach using the OI parameterization. The only difference is that the texture coordinates should be generated differently in steps 1 and 2 of the run-time algorithm. Instead of computing the incoming and outgoing directions as texture coordinates for the G and H textures, the halfangle and difference vectors should be computed and used as texture coordinates. The modified algorithm for computing these texture coordinates using the Gram-Schmidt process is shown in code listing 7.

```

// Computes 2 sets of texture coordinates for every vertex of the
// object.  These texture coordinates are used to index the 2 cube
// maps which store the separated BRDF.  This version computes the
// texture coordinates for the Gram-Schmidt Half-Angle
// parametrization of a separable BRDF.  The basic idea is as
// follows:  given a light position and viewer position in world
// space, compute a light vector, vl, and a direction vector, ve.
// Compute the "half-way" vector between these two vectors as
// h = vl + ve.  This h vector transformed into the local-tangent
// frame gives one set of texture coordinates.  The second set of
// texture coordinates is a "difference" vector d.  This difference
// vector can be thought of as the direction of the light source
// (the light vector) relative to some coordinate system in which h
// is the up direction.  To find this coordinates, the Gram-Schmidt
// orthogonalization process is used to construct an orthonormal
// basis in which h is one of the basis vectors.  The second set of
// texture coordinates is vl relative to the new coordinate system.

void ComputeCubeMapTexCoordsGSHD( Vec &eye, Vec &plight )
{
    Vectorf vl, ve;
    Vectorf h, tprime, bprime;    Vectorf coordh, coordd;

    for ( unsigned int i = 0; i < vertcount; i++ )
    {
        ve = eye - vertexdata[i];    ve.normalize();
        vl = plight - vertexdata[i]; vl.normalize();

        h = vl + ve;
        h.normalize();

        // Derive a coordinate frame using the Gram-Schmidt process.
        tprime = tangendata[i] - (h*dot(tangendata[i],h));
        tprime.normalize();
        bprime = cross(h,tprime);
        bprime.normalize();

        // Transform h into local tangent space.
        coordh.x = dot(h,tangendata[i]);
        coordh.y = dot(h,normaldata[i]);
        coordh.z = dot(h,binormdata[i]);

        // Transform d into Gram-Schmidt derived coordinate frame.
        coordd.x = dot(vl,tprime);
        coordd.y = dot(vl,h);
        coordd.z = dot(vl,bprime);

        // Store the results in a global array.
        texcoordsG[i] = coordh;
        texcoordsH[i] = coordd;
    }
}

```

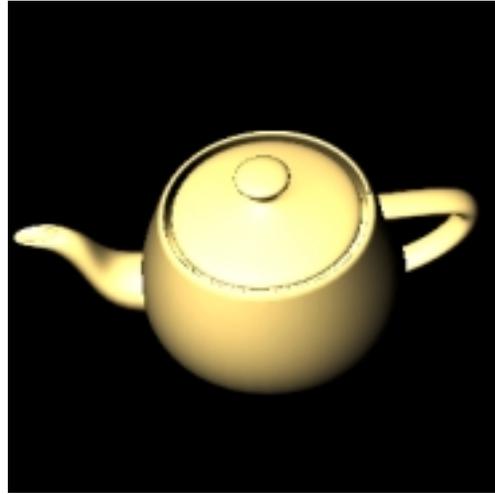
Code Listing 7. Computing texture coordinates for indexing the G and H textures using the GSHD parameterization.

Results

The following images show some examples of BRDF-based lighting for different BRDFs and separation parameters.



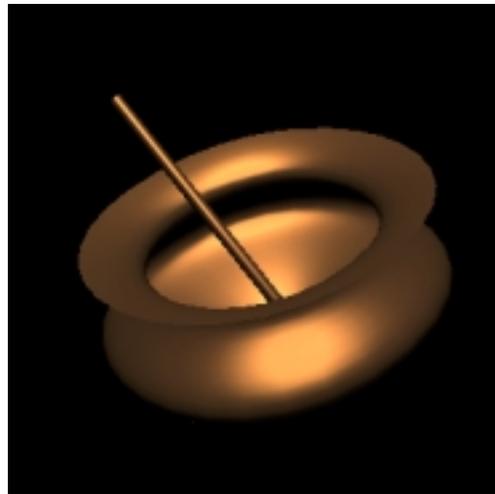
a) Anisotropic Gold BRDF using GSHD parameterization.



b) Anisotropic Gold BRDF using OI parameterization.



c) Anisotropic plastic BRDF using GSHD parameterization.



d) Bronze BRDF using GSHD parameterization.

Figure 9. Snapshots showing the results of the real-time BRDF-based lighting approach.

Summary/Conclusion

BRDF-based lighting is a physically-based lighting technique that is beginning to emerge in real-time computer graphics. This tutorial described a texture-based technique for performing real-time BRDF-based lighting that may be suitable for games and other applications that increased visual realism. While the run-time rendering phase is very simple to implement, the pre-processing algorithm is unfortunately rather complex. The good news is that once the pre-processing algorithm has been implemented, it can be used to process any BRDF. Additionally, the set of separation parameters can be used to control the quality and “look-and-feel” of a particular BRDF.

To alleviate the burden of implementing the pre-processing algorithm, NVIDIA has provided a sample BRDF separation implementation that may be used to perform the pre-processing algorithm. By using this sample implementation, it should be very simple to generate textures and incorporate BRDF-based lighting into existing applications and games. In addition, a sample application that demonstrates the run-time rendering phase is available as well. These example programs as well as additional information on BRDFs can be obtained from: <http://www.nvidia.com/Developer>

References

1. Sim Dietrich. “Hardware Bump Mapping.” In *Game Programming Gems*. Charles River Media, Inc. 2000.
2. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.
3. J. Kautz and M. McCool. *Interactive Rendering with Arbitrary BRDFs using Separable Approximations*. 10th Eurographics Rendering Workshop, 1999.
4. J. Kautz. Hardware Rendering with Bidirectional Reflectances. Technical Report CS-99-02, University of Waterloo, 1999.