A Gentle Introduction to Precomputed Radiance Transfer

Marcos Paulo Berteli Slomp[†]

Manuel M. Oliveira[†]

Diego Inácio Patrício[†]

Abstract: Realistic image synthesis is one of the most relevant subjects in computer graphics, and many algorithms have been developed to try to reproduce the visual complexity perceived in the real world. However, rendering realistic images in real time remains a challenge even with the support of modern graphics hardware. Precomputed Radiance Transfer (PRT) is a new graphics technique capable of synthesizing highly realistic images in real time. This is achieved by restricting the solution of the rendering equation to a subset of the light transport paths that handle only energy exchange among diffuse surfaces. Due to the quality of its results, PRT has attracted the attention of many computer graphics researchers and practitioners. However, understanding and implementing PRT requires familiarity with concepts such as projection into basis functions, empirical function integration and light transport theory. This tutorial provides a gentle introduction to PRT and its required background, enabling the readers to understand and implement the technique.

Keywords: precomputed radiance transfer, global illumination, spherical harmonic lighting, real-time rendering, Monte Carlo integration.

† {slomp | oliveira | dipatricio}@inf.ufrgs.br

1 Introduction

Global illumination algorithms are at the heart of photorealistic image synthesis, but the high cost usually associated with them has limited their use in real-time applications. Because of that, a common practice has been to use algorithms that only implement part of the light transport defined by the rendering equation [12]. More specifically, a solution for the transport among diffuse surfaces in static scenes can be precomputed and then used for real-time rendering. This is the case of radiosity [9] methods and, more recently, precomputed radiance transfer (PRT) [24]. This new class of algorithms is capable of producing high-quality renderings, can take advantage of recent developments in programmable graphics hardware, and has already been incorporated into Microsoft's DirectX SDK [7]. However, its use by the research community and practitioners is still timid, probably due to the technique's elaborate formulation, which has traditionally involved the use of Monte Carlo methods [11] for approximating the solution of some integrals of empirical functions, and the use of spherical harmonics as a basis for the space of both lighting and transfer functions [6].

This tutorial presents a step-by-step introduction to precomputed radiance transfer, providing a detailed treatment of all aspects necessary for understanding and efficiently implementing the technique. In particular, it focus on practical aspects not covered in sufficient detail in previous publications. The presentation is illustrated with sample code extracted from a real application, which can be used as a reference by researchers, game developers and graphics programmers in general. The goal is to provide the readers with a solid intuition and understanding of the underlying algorithms, enabling them to incorporate the technique in their future applications.

The tutorial is organized as follows: Section 2 presents a review of the rendering equation. Section 3 introduces the notion of precomputed radiance transfer and derives the PRT equation from the rendering equation. Sections 4 and 5 provide the background on spherical harmonics and Monte Carlo integration, respectively, necessary for understanding and implementing PRT. Section 6 describes a PRT implementation, providing code fragments for all the core functions. Finally, Section 7 discusses possible extensions to the plain PRT rendering, including the incorporation of specular highlights and the use of tone mapping.

2 Illumination Models and the Rendering Equation

Real world scenes tend to be visually rich (Figure 1), resulting from a complex interaction among several factors, such as object geometry, material properties, and the many paths followed by the light on its way from its source to the object surfaces. In computer graphics, illumination models are abstract representations that attempt to express the interplay between light and object surfaces, and can be classified as *local* and *global*. *Local* *illumination models* compute the shading at a point on a surface considering only its local properties. As a result, such a simplification does not handle blocking light (*i.e.*, occlusions), which is responsible for producing shadows (an important element for achieving realism), and also ignores indirect lighting (*i.e.*, light reflected from other objects that ends up arriving at the given point). *Global illumination models*, in contrast, take into account the entire scene when computing the illumination at a surface point. While they can produce more sophisticated renderings, this comes at higher computational costs, which has traditionally prevented their use in real-time applications.

Figure 1: A real world scene exhibiting a variety of geometric shapes, different materials and light paths. Its appearance results from the interplay of all such elements.

For the specific case of static scenes containing only diffuse surfaces, real-time renderings can be achieved using pre-computed solutions for the light transport. This tutorial describes one such technique called precomputed radiance transfer. The reader should notice, however, that with the advent of modern programmable graphics hardware capable of performing several computations in parallel, the gap separating global illumination algorithms and real-time performance gets thinner every day. This opens up exciting new possibilities and the prospect for rendering more complex light phenomena in real time on commodity hardware.

2.1 The Rendering Equation

All rendering algorithms can be seen as solutions to particular formulations of a general expression known as the rendering equation [12] (Equation 1). Its interpretation can be stated as: *the light leaving any point x in a given direction $\vec{\omega}_o$ is computed as the amount of light that x emits in the direction $\vec{\omega}_o$, plus the reflected/scattered light from x in the outgoing direction $\vec{\omega}_o$ after it has reached x from all incoming directions.*

$$L(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + \int_\Omega L(x',\vec{\omega}_i)\,\rho(x,\vec{\omega}_i,\vec{\omega}_o)\,V(x,x')\,G(x,x')\,d\vec{\omega}_i \qquad (1)$$

where

Ω represents the domain of all possible directions
$L(x',\vec{\omega}_i)$ is the amount of light arriving at x from another point x' along $\vec{\omega}_i$
$\rho(x,\vec{\omega}_i,\vec{\omega}_o)$ is a function that tells how much of the incoming light arriving at x along the direction ω_i is reflected along the outgoing direction $\vec{\omega}_o$
$V(x,x')$ is a binary visibility function involving x and x'
$G(x,x')$ is the geometric relationship between x and x'

In Equation (1) one can identify the factors responsible for the visual complexity observed in real scenes: local geometric aspects are implicitly defined by x; material properties are expressed as ρ; the incoming light is represented by L. Occlusions are obtained as the result of the visibility function V, so it is related to light transport. Since the rendering equation is defined recursively (L appears in both sides of the equation), it is useful to think about it as an infinite series, expressed by the Neumann expansion:

$$L(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + L_0(x,\vec{\omega}_o) + L_1(x,\vec{\omega}_o) + L_2(x,\vec{\omega}_o) + \ldots \qquad (2)$$

The first term of the expansion (L_e) represents the light emitted by x in the direction $\vec{\omega}_o$. The second term (L_0) represents the light reflected/scattered at x in the direction $\vec{\omega}_o$ after arriving directly from light sources or other emitting surfaces along all possible incoming directions. The following term (L_1) represents the light reflected/scattered at x in the direction $\vec{\omega}_o$ after it has bounced once before reaching x. Likewise, L_i, $i \geq 2$, represents the light reflected/scattered at x in the direction $\vec{\omega}_o$ after it has bounced (or been transmitted through) i times on scene before reaching x.

All terms besides L_e and L_0 represent indirect lighting (bounces). The first level of indirect lighting can be computed using the results obtained with directing lighting. Similarly, a second bounce can be performed with the results of the first one, and so on. This behavior is expressed by Equation (3), which describes the indirect light contribution exiting a point x along the direction $\vec{\omega}_o$ after n bounces. Notice the emittance term has been omitted since it has already been accounted for in L_0. Also, the visibility function must be inverted, as one must only consider, for indirect lighting purposes, directions that were blocked when direct lighting was computed.

$$L_n(x,\vec{\omega}_o) = \int_\Omega L_{n-1}(x',\vec{\omega}_i)\,\rho(x,\vec{\omega}_i,\vec{\omega}_o)\,(1 - V(x,x'))\,G(x,x')\,d\vec{\omega}_i \qquad (3)$$

Despite its elegance and compactness, directly solving the rendering equation is not practical. Its integration domain consists of infinite directions and its recursive nature also leads to an infinite number of levels to solve for. Precomputed radiance transfer is an attempt to surpass the challenge of real-time global illumination by splitting the problem into two parts: one that is solved in a precomputed step and another that is solved in real time. The PRT equation can be derived directly from the rendering equation though a series of simplifications and approximations.

3 Precomputed Radiance Transfer

PRT is based on two main assumptions: (i) all objects in the scene are non-emitters, and (ii) the light sources are infinitely distant. The second assumption makes the incoming light direction independent of the position of point x. As a result, the term L_0 (direct lighting) in Equation (2) is illustrated in Figure 2 and rewritten as:

$$L_0(x,\vec{\omega}_o) = \int_\Omega L_\epsilon(\vec{\omega}_i)\,\rho(x,\vec{\omega}_i,\vec{\omega}_o)\,V(x,x')\,G(x,x')\,d\vec{\omega}_i \qquad (4)$$

Another common assumption in PRT is to treat all surfaces in the scene as Lambertian reflectors. In this case, ρ becomes just the surface reflectance divided by π [3], and can be taken out of the integral, substantially simplifying the equation. In this tutorial, however, we will continue to use the more general form of Equation (4), postponing such a simplification until the implementation section. The idea is to give the reader a better appreciation for the potential of the technique. The functions ρ, $\boldsymbol{V}$ and $\boldsymbol{G}$ can be grouped into a single function T, known as *transfer function*.

$$L_0(x,\vec{\omega}_o) = \int_\Omega L_\epsilon(\vec{\omega}_i)\,T(x,\vec{\omega}_o,\vec{\omega}_i,x')\,d\vec{\omega}_i \qquad (5)$$

Basically, the lighting function L_ϵ describes the arriving light intensity at point x, while the transfer function expresses how x responds to the incoming illumination. In PRT, these functions are independent from each other and, therefore, can be estimated separately and combined later to produce the final result.

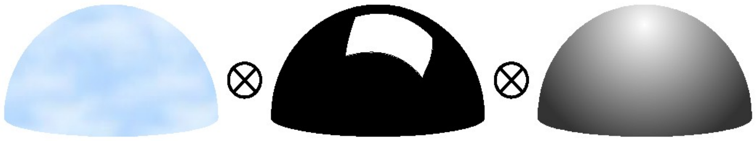

Figure 2: To determine the illumination of a point over a surface, it is necessary to integrate, along with its material properties, the lighting function (leftmost), the visibility function (center) and the geometric factor (rightmost).

Even with such simplifications, Equation (5) is not yet practical for real-time implementation. First, one needs a way to estimate the functions L_ϵ and T, none of which has an analytical solution. And second, we need to efficiently evaluate Equation (5) during runtime. In PRT, the lighting and transfer functions are estimated during a preprocessing step and their values are used to approximate the integral in real time. These operations will be discussed in more detail later. Next, we describe the use of basis functions to project and reconstruct arbitrary functions. These operations are essential for understanding how PRT can evaluate an estimate of Equation (5) in real time. Figure 3 illustrates how lighting and transfer functions can be combined to produce the final shading.

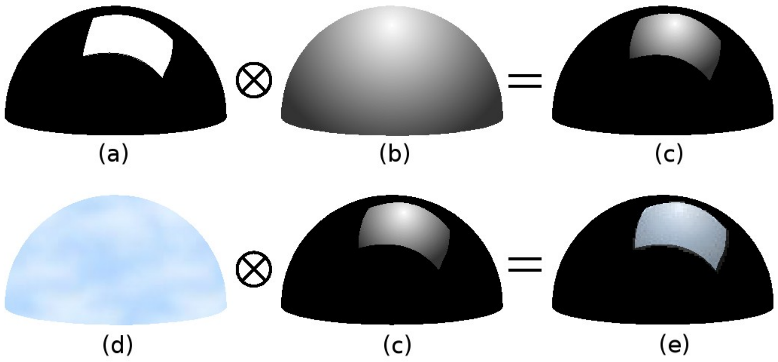

Figure 3: The visibility term (a) is combined with the geometric factor (b), which gives a transfer factor (c). This transfer factor (c) will be then combined with the incoming radiance (d) to determine the final illumination (e).

3.2 Projection and Reconstruction of Functions

Given a function space, a basis functions β is an infinite set of functions used to project and reconstruct arbitrary functions. The *projection* of a function f into any function β_i gives a measure c_i of how similar these two functions are, and is accomplished by integrating the product of f and β_i over the entire domain of f:

$$c_i = \int f(x)\, \beta_i(x)\, dx \qquad (6)$$

The original function f can be recovered as a linear combination of the basis functions β_i, each modulated by its corresponding coefficient c_i. This process is known as *reconstruction* and its accuracy depends on how many terms are added. The more basis functions are used, the better the reconstruction. In fact, when using an infinite number of basis functions, which requires an infinite number of coefficients, one is guaranteed to recover the original function:

$$f(x) = \lim_{n\to\infty} \sum_{i=1}^{n} c_i \, \beta_i(x)$$

By using just a finite number n of terms will produce an approximation $\tilde{f}$ of the original function f.

$$f(x) \approx \tilde{f}(x) = \sum_{k=1}^{n} c_k \, \beta_k(x)$$

For PRT purposes, the use of orthonormal basis functions, *i.e.*, functions exhibiting the following property, is highly desirable:

$$\int \beta_i(x) \, \beta_j(x) dx = \begin{cases} 0 & when \;\; i \neq j \\ 1 & when \;\; i = j \end{cases}$$

Orthonormality guarantees that the integral of the product of two basis functions will be either zero, if the functions are different, or one, in case they are the same. Thus, let c_k and d_k be the coefficients associated with the projections of any two functions $f(x)$ and $g(x)$, respectively, into the basis function β_k, for all k. The integral of the product $f(x)g(x)$ can then be obtained as the sum of the products $c_k d_k$. More formally:

$$\begin{aligned} c_k &= \int f(x) \, \beta_k(x) \, dx \\ d_k &= \int g(x) \, \beta_k(x) \, dx \\ \int f(x) \, g(x) \, dx &\approx \int \sum_{k=1}^{n} c_k \, \beta_k(x) \sum_{k=1}^{n} d_k \, \beta_k(x) \, dx = \\ &\sum_{k=1}^{n} c_k \, d_k \int \beta_k(x) \, \beta_k(x) \, dx = \\ &\sum_{k=1}^{n} c_k \, d_k \qquad (7) \end{aligned}$$

This provides a powerful mechanism for factoring the evaluation of Equation (5) in two steps: (i) the projection of both L_ϵ and T into an orthonormal basis, which can be done during preprocessing, and (ii) the evaluation of the integral as a dot product (Equation 7), which can be efficiently done in real time. This will be explored in detail next.

3.3 Precomputed Radiance Transfer Equation Derivation

Let both lighting and transfer functions from Equation (5) be projected into the same set basis functions β_k. In this case, the original functions can be approximated as:

$$L_\epsilon(\vec{\omega}_i) \approx \sum_{k=1}^{n} l_k \, \beta_k(\vec{\omega}_i) \quad \text{where} \quad l_k = \int L_\epsilon(\vec{\omega}_i) \, \beta_k(\vec{\omega}_i) \, d\vec{\omega}_i \qquad (8)$$

and

$$T(x,\vec{\omega}_o,\vec{\omega}_i,x') \approx \sum_{k=1}^{n} t_k\, \beta_k(\vec{\omega}_i) \quad \text{where} \quad t_k = \int T(x,\vec{\omega}_o,\vec{\omega}_i,x')\, \beta_k(\vec{\omega}_i)\, d\vec{\omega}_i \qquad (9)$$

Substituting the value of the approximate lighting function (Equation 8) into Equation (5):

$$L_0(x,\vec{\omega}_o) = \int_\Omega \left[\sum_{k=1}^{n} l_k\, \beta_k(\vec{\omega}_i)\, T(x,\vec{\omega}_o,\vec{\omega}_i,x') \right] d\vec{\omega}_i$$

Using the fact that integration is a linear operation – the integral of sums equals the sum of integrals – the equation above can be rewritten as:

$$L_0(x,\vec{\omega}_o) = \sum_{k=1}^{n} l_k \int_\Omega \beta_k(\vec{\omega}_i)\, T(x,\vec{\omega}_o,\vec{\omega}_i,x')\, d\vec{\omega}_i$$

For each term k of the sum, the associated lighting coefficient will be multiplied by the entire integral. But this integral represents a projection operation. So, each term of the sum will produce a new coefficient that is the result of projecting the transfer function into the associated basis function, resulting in the following equation:

$$L_0(x,\vec{\omega}_o) = \sum_{k=1}^{n} l_k\, t_k \qquad (10)$$

This equation is identical to Equation (7), and it can be seen as a dot product of the lighting and the transfer coefficients. One should note that Equation (10) only handles direct lighting (*i.e.*, the light leaving x after having arrived there directly from the light source). Notice, however, that it handles occlusions with respect to the light source. To be able to perform extra bounces, a similar derivation can be done with higher order terms of the Neumann Expansion, which leads to the final PRT equation, shown below:

$$L(x,\vec{\omega}_o) = \sum_{k=0}^{n} l_k\, (t_k^0 + t_k^1 + t_k^2 + \ldots) \qquad (11)$$

It can be seen that one set of transfer coefficients should be obtained for each outgoing direction, which is explicit in the transfer function T. This is the general case, which can be simplified to a single set of transfer coefficients (*i.e.*, independent of the outgoing direction) by assuming that all surfaces are ideal diffuse reflectors (Lambertian surfaces). Such a simplification will be discussed later in Section 6.

At this point, one is still left with two important questions: which basis function should be used and how to solve the integrals required to perform the projection? These questions will be addressed in the next sections.

4 Spherical Harmonics

Spherical harmonics (SH) are orthonormal functions defined over the unit sphere, where the 2D domain can be seen as the set of all possible directions. In their general formulation, SH functions are defined over complex numbers, but for PRT purposes, we are interested in approximating real functions. Therefore, this tutorial will focus on real spherical harmonics only.

4.1 The Legendre Polynomials

Legendre Polynomials are the core of the spherical harmonic functions. The Associated Legendre polynomials return real numbers and are recursively defined as follows:

$$\{ l \in \mathbb{N} \}$$
$$\{ m \in \mathbb{N} \mid m \leq l \}$$
$$\{ x \in \mathbb{R} \mid -1 \leq x \leq 1 \}$$
$$P_l^m(x) = \begin{cases} x(2m+1)P_m^m & when \;\; l=m+1 \\ (-1)^m (2m-1)!!\,(1-x^2)^{m/2} & when \;\; l=m \\ \dfrac{x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m}{(l-m)} & otherwise \end{cases}$$

They are characterized by two parameters, l, which is usually called *band*, and m, that varies according to l, and they are defined over real numbers in the interval between $[-l, l]$. The unusual operator $!!$ is called *double factorial*, and is defined as follows:

$$n!! = \begin{cases} 1 & when \;\; n \leq 1 \\ n(n-2)!! & when \;\; n > 1 \end{cases}$$

4.2 Spherical Harmonic Functions

Spherical harmonic functions are parameterized using θ and ϕ, as shown below, with scaling factors K_l^m for normalizing the functions.

$$\{ l \in \mathbb{N} \}$$
$$\{ m \in \mathbb{Z} \mid -l \leq m \leq l \}$$
$$y_l^m(\theta, \phi) = \begin{cases} \sqrt{2}\, K_l^m \cos(m\phi)\; P_l^m(\cos(\theta)) & when \;\; m>0 \\ \sqrt{2}\, K_l^m \sin(-m\phi)\; P_l^{-m}(\cos(\theta)) & when \;\; m<0 \\ K_l^0\; P_l^0(\cos(\theta)) & when \;\; m=0 \end{cases}$$

where

$$K_l^m=\sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}$$

A spherical harmonics approximation of a function using l bands requires l^2 coefficients. An alternative representation using a single index can be obtained using the following relationship:

$$\{\ y_l^m(\theta,\phi)=y_i(\theta,\phi)\ \mid\ i=l(l+1)+m\ \}$$

That is all one needs to know about spherical harmonics to use them in PRT. The next section discusses Monte Carlo integration, a method for numerically evaluating the integral of any function, be it analytical or empirical.

5 Monte Carlo Integration

The expected value E of a function f with a random variable x associated to it is an average value that will tend to return most often if evaluated to a large number of samples. A random variable is a value that lies within a specific domain and has a probability p associated to it. Thus, E can be obtained by computing the following integral over the entire domain of the random variable:

$$E[f(x)]=\int f(x)\ p(x)\ dx \qquad (12)$$

Another way to get the expected value of a function is to take the mean of an infinite number of random samples within its domain:

$$E[f(x)]=\lim_{n\to\infty}\frac{1}{n}\sum_{i=1}^{n}f(x_i) \qquad (13)$$

Using a finite number of random samples will give only an approximation of the expected value:

$$E[f(x)]\approx\frac{1}{n}\sum_{i=1}^{n}f(x_i) \qquad (14)$$

Combining Equations (12) and (14), one gets a numerical solution for estimating the integral of an arbitrary function, which is known as *Monte Carlo integration* (Equation 15).

$$\int f(x)\,dx=\int\frac{f(x)\ p(x)}{p(x)}\,dx\approx\frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{p(x_i)} \qquad (15)$$

To solve an integral using the Monte Carlo, it is then necessary to take lots of samples of the function and each one must have an associated probability. Equation (15) can be rewritten in terms of a weight function ***w*** as:

$$\int f(x)\,dx \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i)\,w(x_i) \qquad where \quad w(x_i) = \frac{1}{p(x_i)} \qquad (16)$$

5.2 Using Monte Carlo to Solve the Projection into Spherical Harmonic basis

Replacing the general basis functions β used in Equation (6) by the spherical harmonic basis functions y and parameterizing the function f in terms of the space of directions, Equation (6) can be rewritten as:

$$c_l^m = \int f(\vec{\omega})\, y_l^m(\vec{\omega})\, d\vec{\omega} \qquad (17)$$

and it is useful to think of both spherical harmonic functions and coefficients in terms of a single index. In this case

$$c_k = \int f(\vec{\omega})\, y_k(\vec{\omega})\, d\vec{\omega} \qquad (18)$$

Applying Monte Carlo integration, the coefficients obtained with the projection operation can be approximated using n samples as shown in Equation (16). Note that the more samples are used, the better the approximation. It should be emphasized that each sample $f(\vec{\omega}_j)$ is associated with a direction $\vec{\omega}_j$. Thus, both the lighting and transfer functions need to be sampled along the set of directions $\vec{\omega}_j$, $1 \le j \le n$.

$$c_k \approx \frac{1}{n}\sum_{j=1}^{m} f(\vec{\omega}_j)\, y_k(\vec{\omega}_i)\, w(\vec{\omega}_j) \qquad (19)$$

Although one knows how to evaluate both the function f and the SH basis functions for an arbitrary direction, it is still necessary to associate a probability to the occurrence of this direction. Since these functions are defined over the space of all possible directions (geometrically, this space can be seen as the surface of a unit sphere), the probability of any sample (direction) occurring in such a surface will be one over the area of this unit sphere (*i.e.*, $1/4\pi$). So, the probability associated to each directions is:

$$p(\vec{\omega}_j) = \frac{1}{4\pi}$$

Substituting the weight function in Equation (19), one gets Equation (20), which describes the projection of an arbitrary function f defined over the space of all possible directions into the spherical harmonic basis.

$$c_k \approx \frac{4\pi}{n}\sum_{j=1}^{m} f(\vec{\omega}_j)\, y_k(\vec{\omega}_j) \qquad (20)$$

Since our Monte Carlo formulation assumes that each sampling direction is equally probable[1], the actual samples of the function f should be evenly distributed over the unit

1 An alternative to this approach is the use of importance sampling.

sphere. Moreover, the same set of sampling directions $\vec{\omega}_j$ should be used for both the lighting and the transfer functions. In order to guarantee a uniform sampling of the unit sphere, the following algorithm known as *stratified sampling* is often used:

(i) Evenly distribute the number n of samples over the unit square. For this, subdivide the unit square into $\sqrt{n} \times \sqrt{n}$ cells, and randomly select a sample inside each cell;

(ii) Map the coordinates of the samples in the unit square to coordinates on the unit sphere using Equation (21).

$$(x,\ y) \rightarrow (2\,acos(\sqrt{1-x}),\ 2\pi\, y) \rightarrow (\theta,\ \phi) \qquad (21)$$

The idea of stratified sampling is illustrated in Figure 4. On the left, one sees the distribution of the generated samples on the unit square. Each sample on the square has been mapped to one sample on the sphere on the right, according to Equation (21).

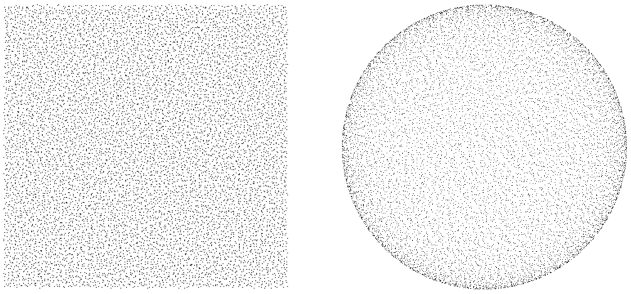

Figure 4: Uniform sampling of the sphere using stratified sampling. A total of 10000 random samples generated over the unit square (left). These samples are mapped to the surface of the sphere on the right using Equation (21).

6 PRT Implementation

At this point, the reader has all the background necessary to implement PRT, which is the subject of this section. As mentioned before, the PRT algorithm consists of two steps: (i) the projection of the lighting and transfer functions into some orthonormal basis functions, which happens as part of preprocessing, and (ii) the evaluation of Equation (11), which is done in real time.

For this tutorial, lighting functions will be represented as environment maps stored as spherical maps (light probes), but it is also possible to use cube maps. We also briefly discuss the use of analytical representations for lighting functions. For the case of transfer functions, two types will be considered: (i) *unshadowed transfer functions*, which ignore the visibility function ($V(x\,,x')$ in Equation 4), and (ii) *shadowed transfer functions,* which take the occlusion term into account. We also briefly discuss what it takes to add indirect lighting (also called *interreflected transfer function*) to the shadowed case. The transfer function will be evaluated for each vertex present in the scene.

The code fragments shown in this section were written in C/C++. There is also sample code for GPU execution written in Cg [15], with its corresponding CPU version.

6.2 Precomputation Step

This is the core of a PRT implementation, concentrating most of the coding effort. The following sections discuss each of the precomputation steps.

6.2.1 Sampling Directions

The first step is to generate the *m* sampling directions using the stratified sampling strategy described in Section 5.2 and illustrated in Figure 4. Before generating these directions, it is necessary to define the sample data structure:

```
struct Vector3
{
 float x;
 float y;
 float z;
};

struct Spherical
{
 float theta;
 float phi;
}

struct Sample
{
 Spherical spherical_coord;
 Vector3 cartesian_coord;
 float* sh_functions;
};
```

Each sample stores its spherical coordinates (θ, ϕ) and its Cartesian coordinates (x, y, z). The spherical coordinates will be used to evaluate the spherical harmonic functions, while the Cartesian ones will be used to evaluate the lighting function (compute dot products). The samples also store the result of the evaluation of the specified bands of the spherical harmonic functions (for the associated spherical coordinates). The details behind this extra storage will be explained later.

```
struct Sampler
{
 Sample* samples;
 int number_of_samples;
};
```

The sampler data structure (above) stores the samples, providing access and the ability to iterate over the samples (created using stratified sampling). The conversion between spherical coordinates obtained with Equation (21) to Cartesian coordinates is given by:

$$(\theta,\ \phi) \rightarrow (\sin\theta\cos\phi,\ \sin\theta\sin\phi,\ \cos\theta) \rightarrow (x, y, z)$$

The following code fragment performs the sampling and initialization of the `Sampler` data structure with `NxN` samples. The `sh_functions` variable is initialized with `NULL` and, in a later step, memory will be allocated to store the values for the evaluated bands.

```
void GenerateSamples(Sampler* sampler, int N)
{
 Sample* samples = new Sample [N*N];
 sampler->samples = samples;
 sampler->number_of_samples = N*N;
 for (int i = 0; i < N; i++)
 {
   for (int j = 0; j < N; j++)
   {
     float a = ((float) i) + Random()) / (float) N;
     float b = ((float) j) + Random()) / (float) N;
     float theta = 2*acos(sqrt(1-a));
     float phi = 2*PI*b;
     float x = sin(theta)*cos(phi);
     float y = sin(theta)*sin(phi);
     float z = cos(theta);
     int k = i*N + j;
     sampler->samples[k].spherical_coord.theta = theta;
     sampler->samples[k].spherical_coord.phi = phi;
     sampler->samples[k].cartesian_coord.x = x;
     sampler->samples[k].cartesian_coord.y = y;
     sampler->samples[k].cartesian_coord.z = z;
     sampler->samples[k].sh_functions = NULL;
   }
 }
};

float Random()
{
 float random = (float) (rand() % 1000) / 1000.0f;
 return(random);
}
```

As it is a C based implementation, the `rand()` function works properly, a random seed must be initialized. So, at the early stages of the `main()` function, it can be initialized with the call showed below:

```
srand(time(NULL));
```

6.2.2 Spherical Harmonics Evaluation

In order to evaluate real spherical harmonic functions, one needs to be able to evaluate the associated Legendre polynomials, which can be done with the following code fragment.

```
float Legendre(int l, int m, float x)
{
 float result;
 if (l == m+1)
  result = x*(2*m + 1)*Legendre(m, m);
 else if (l == m)
  result = pow(-1, m)*DoubleFactorial(2*m-1)*pow((1-x*x), m/2);
 else
  result = (x*(2*l-1)*Legendre(l-1, m) - (l+m-1)*Legendre(l-2, m))/(l-m);
 return(result);
}

float DoubleFactorial(int n)
{
 if (n <= 1)
  return(1);
 else
  return(n * DoubleFactorial(n-2));
}
```

Now, the spherical harmonic functions can be evaluated

```
float SphericalHarmonic(int l, int m, float theta, float phi)
{
 float result;
 if (m > 0)
  result = sqrt(2) * K(l, m) * cos(m*phi) * Legendre(l, m, cos(theta));
 else if (m < 0)
  result = sqrt(2) * K(l, m) * sin(-m*phi) * Legendre(l, -m, cos(theta));
 else
  result = K(l, m) * Legendre(l, 0, cos(theta));
 return(result);
}
```

where the normalization factors can be computed as:

```
float K(int l, int m)
{
 float num = (2*l+1) * factorial(l-abs(m));
 float denom = 4*PI * factorial(l+abs(m));
 float result = sqrt(num/denom);
 return(result);
}
```

The presented implementations for both the associated Legendre polynomials and for the double factorial are recursive. They were used here for simplicity, as the mappings between their mathematical definitions and implementations are straightforward. Also, these functions are only used in the preprocessing stage. More efficient ways to implement these functions iteratively can be found in [20].

6.2.3 Precomputing the Spherical Harmonic Bands

It is useful to compute and store the SH bands at the samples to avoid evaluating them several times in further steps. The following code fragment does this.

```
void PrecomputeSHFunctions(Sampler* sampler, int bands)
{
 for (int i = 0; i < sampler->number_of_samples; i++)
 {
   float* sh_functions = new float [bands*bands];
   sampler->samples[i].sh_functions = sh_functions;
   float theta = sampler->samples[i].spherical_coord.theta;
   float phi = sampler->samples[i].spherical_coord.phi;
   for (int l = 0; l < bands; l++)
     for (int m = -l; m <= l; m++)
     {
       int j = l*(l+1) + m;
       sh_functions[j] = SphericalHarmonic(l, m, theta, phi);
     }
 }
}
```

This function takes two parameters: the sampler, which was created previously, and the number of spherical harmonic bands to be used (the more bands, the better the approximation). Recall that an approximation using l bands uses l^2 spherical harmonic functions. The code iterates over the bands and evaluates these functions. This process must be repeated for all samples.

6.2.4 Lighting Function Representation

There are essentially two ways to represent a lighting function: empirically and analytically. In this tutorial, light functions are represented empirically using *light probes* (Figure 5). A light probe is an omnidirectional image that records the incident illumination at a particular point in space and can be captured from real world scenes. There are several freely available light probe images on the Internet [14].

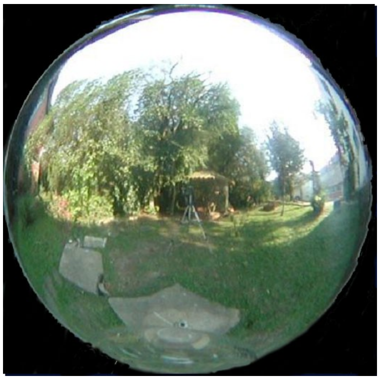

Figure 5: A light probe captured at one of the UFRGS campus.

Since a light probe is only a 2D image, one needs to convert spherical or Cartesian coordinates into image coordinates. This can be done using the following code fragment available from Paul Debevec's IBL Tutorial [5]. Basically, the function converts the 3D Cartesian coordinates (`direction`) into 2D texture coordinates (`tex_coord`) and, finally, converts them to the pixel offset in the image (`pixel_coord`), based on image dimensions. The output parameter `color` will have the RGB color associated to that direction, after its execution.

```
struct Color
{
 float r;
 float g;
 float b;
};
void LightProbeAccess(Color* color, Image* image, Vector3 direction)
{
 float d = sqrt(direction.x*direction.x + direction.y*direction.y);
 float r = (d == 0) ? 0.0f : (1.0f/PI/2.0f) * acos(direction.z) / d;
 float tex_coord [2];
 tex_coord[0] = 0.5f + direction.x * r;
 tex_coord[1] = 0.5f + direction.y * r;
 int pixel_coord [2];
 pixel_coord[0] = tex_coord[0] * image.width;
 pixel_coord[1] = tex_coord[1] * image.height;
 int pixel_index = pixel_coord[1]*image.width + pixel_coord[0];
 color->r = image.pixel[pixel_index][0];
 color->g = image.pixel[pixel_index][1];
 color->b = image.pixel[pixel_index][2];
}
```

The lighting function can also be represented as a cube map, but it is a lot easier to fetch the desired value from a spherical map, such as a light probe. Another way to evaluate a lighting function is to use an analytical representation. An example of such a function is shown below, which corresponds to two monochromatic light sources, at 90 degrees from each other [10]. Figure 6 provides an illustration for this lighting function.

$$light(\theta,\phi) = max(0,\, 5\cos(\theta)-4) + max(0,\, -4\sin(\theta-\pi)\cos(\phi-2.5)-3)$$

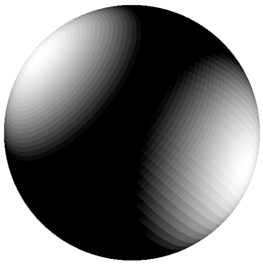

Figure 6: A 3D plotting of an analytical spherical lighting function that defines two monochromatic light sources, at 90 degrees from each other.

6.2.5 Projection into Spherical Harmonic basis using Monte Carlo Integration

Everything that is necessary to implement the projection operator is now available: the samples, the function, and the SH basis functions. It is time to take a look at Monte Carlo integration in action, and it is really simple to implement it. The weight function is just a constant and the number of samples is known. The following code fragment implements the projection operation using Monte Carlo integration:

```
void ProjectLightFunction(Color* coeffs, Sampler* sampler,
                          Image* light, int bands)
{
 for (int i = 0; i < bands*bands; i++)
 {
  coeffs[i].r = 0.0f;
  coeffs[i].g = 0.0f;
  coeffs[i].b = 0.0f;
 }

 for (int i = 0; i < sampler->number_of_samples; i++)
 {
  Vector3& direction = sampler->samples[i].cartesian_coord;
  for (int j = 0; j < bands*bands; i++)
  {
   Color color;
   LightProbeAccess(&color, light, &direction);
   float sh_function = sampler->samples[i].sh_functions[j];
   coeffs[j].r += (color.r * sh_function);
   coeffs[j].g += (color.g * sh_function);
   coeffs[j].b += (color.b * sh_function);
  }
 }

 float weight = 4.0f*PI;
 float scale = weight / sampler->number_of_samples;
 for (int i = 0; i < bands*bands; i++)
 {
  coeffs[i].r *= scale;
  coeffs[i].g *= scale;
  coeffs[i].b *= scale;
 }
}
```

The code initializes all coefficients with zero. The sum operates over all samples. For each sample, the directional Cartesian coordinates are used to access the light probe and the spherical harmonic precomputed values associated with each band are retrieved. They are combined and accumulated. At the end, all values are scaled using the weight function.

One important thing to note is that three sets of coefficients were generated independently, one for each color channel. For monochromatic lighting functions, one vector of coefficients is enough. The same principle applies when projecting transfer functions, which will be discussed next.

6.2.6 Transfer Functions

Recall that the transfer function, as defined within Equation (5), is the product of three other functions: the surface scattering function ρ, a visibility function V, and a geometric term G that expresses the relationship between the emitter and the receiving point. These intuitively simple functions are surprisingly the most difficult to implement in the whole PRT technique.

The function ρ can be as complex as we want, handling phenomena such as caustics, refraction, subsurface scattering, and so on. For this tutorial, a simple scattering function will be used, assuming that the surfaces are ideal diffusers (*i.e.*, Lambertian surfaces), meaning that they reflect the incident light with equal intensity in all directions. Using such a simplification, the dependence between the transfer function and an outgoing direction disappears. Also, the scattering function will be a constant value given by [3]:

$$\rho(x, \vec{\omega}_i, \vec{\omega}_o) = \frac{\rho_d}{\pi}$$

The ρ_d term in the equation above is a reflectance coefficient associated to the point x where the transfer function will be evaluated, just like the diffuse coefficient used in the Phong Illumination Model [19].

For the geometric term G, one can use the one known as *Lambert's cosine law* (Equation 22). Here, x' is the emitter, $\vec{\omega}_i$ is the normalized incident light direction, and $\vec{n}_x$ is the normalized surface normal at x. For some incident directions, the dot product in Equation (22) can be negative, representing light arriving from behind the surface. Equation (22) clamps such values to zero.

$$G(x, x') = max(0, \vec{n}_x \cdot \vec{\omega}_i) \qquad (22)$$

By blocking some light paths, the visibility function is responsible for a great deal of realism. Despite its simple definition – returns 0, if the light is blocked; 1, otherwise – it is an intricate function to implement. In fact, the visibility function is implemented as a ray-casting procedure. Before going any further into the details about visibility function implementation, it is worthy to take a look at some kinds of transfer function variations.

6.2.6.1 Unshadowed Transfer Function

By simply ignoring the visibility function, one can produce renderings similar to that obtained with the use of local illumination models, such as Phong and Cook-Torrence models [19] [4], but with one advantage: the use of any number of light sources without any additional rendering cost. Figure 7 exemplifies how unshadowed transfer works:

Before implementing the transfer function, it is necessary to have a data structure to handle the scene information (*e.g.*, objects, vertices, triangles, materials). The following data structure will be used to describe the scene:

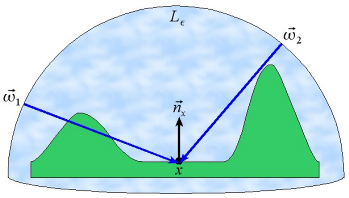

Figure 7: For the unshadowed transfer function, occlusions are ignored. As a result, $\vec{\omega}_1$ will be considered when illumination is evaluated for x.

```
struct Triangle
{
  int a;
  int b;
  int c;
};

struct Scene
{
  Vector3* vertices;
  Vector3* normals;
  int* material;
  Triangle* triangles;
  Color* albedo;
  int number_of_vertices;
};
```

Consider a scene composed of triangles, each with three indices to vertices and the same indices also used for the normals. Each vertex has an associated material, which is an index to a color (albedo).

```
void ProjectUnshadowed(Color** coeffs, Sampler* sampler,
                       Scene* scene, int bands)
{
  for (int i = 0; i < scene->number_of_vertices; i++)
  {
    for (int j = 0; j < bands*bands; j++)
    {
      coeffs[i][j].r = 0.0f;
      coeffs[i][j].g = 0.0f;
      coeffs[i][j].b = 0.0f;
    }
  }

  for (int i = 0; i < scene->number_of_vertices; i++)
  {
    for (int j = 0; j < sampler->number_of_samples; j++)
    {
      Sample& sample = sampler->samples[j];
      float cosine_term = dot(&scene->normals[i], &sample.cartesian_coord);
      for (int k = 0; k < bands*bands; k++)
      {
        float sh_function = sample.sh_functions[k];
        int materia_idx = scene->material[i];
        Color& albedo = scene->albedo[materia_idx];
        coeffs[i][k].r += (albedo.r * sh_function * cosine_term);
        coeffs[i][k].g += (albedo.g * sh_function * cosine_term);
        coeffs[i][k].b += (albedo.b * sh_function * cosine_term);
      }
    }
  }

  float weight = 4.0f*PI;
  float scale = weight / sampler->number_of_samples;
  for (int i = 0; i < scene->number_of_vertices; i++)
    for (int j = 0; j < bands*bands; j++)
    {
      coeffs[i][j].r *= scale;
      coeffs[i][j].g *= scale;
      coeffs[i][j].b *= scale;
    }
}
```

The projection of unshadowed transfer functions into SH basis is pretty similar to light projection. The main difference is that each vertex will have a transfer vector (thus the double indirection `Color**` is necessary for the `coeffs` variable). Inside the sum, the geometric factor must be evaluated, which is the dot product of the vertex normal and the sample direction. Since each vertex may have a different material, this must be retrieved for each vertex. The associated spherical harmonic function is retrieved just as it was for light projection. At the end, the resulting value is scaled as before.

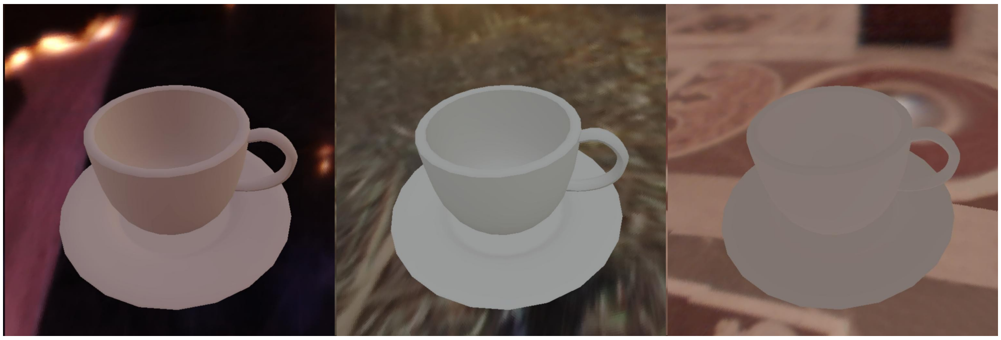

Figure 8: A Teacup rendered using an unshadowed transfer function under three different lighting functions.

The previous code fragment implements the projection of the transfer function into SH basis. The transfer function is evaluated using the scattering function (constant) and Lambert's cosine law (Equation 22) for the geometric term. Obviously, the visibility function is omitted. It is important to notice that the transfer function will be evaluated and projected for every vertex in the model, and each vertex will have as many transfer coefficients as the number of basis functions used. Recall that this number is the square of the number of SH bands used, just like when projecting the lighting function.

Figure 8 illustrates the use of the unshadowed transfer function for rendering a teacup using three different lighting functions. Note the influence of the lighting function in the perceived color of the object.

6.2.6.2 Shadowed Transfer Function

In order to produce a shadowed transfer function, the visibility term must be taken into account, which can be implemented using ray-casting. The code fragment below implements the Möller et al. algorithm for ray-triangle intersection [16]:

```
bool RayIntersectsTriangle(Vector3* p, Vector3* d,
                           Vector3* v0, Vector3* v1, Vector3* v2)
{
 float e1 [3] = { v1->x - v0->x, v1->y - v0->y, v1->z - v0->z };
 float e2 [3] = { v2->x - v0->x, v2->y - v0->y, v2->z - v0->z };
 float h [3];
 cross(h, d, e2);
 float a = dot(e1, h);
 if (a > -0.00001f && a < 0.00001f)
  return(false);
 float f = 1.0f / a;
 float s [3] = { p->x - v0->x, p->y - v0->y, p->z - v0->z };
 float u = f * dot(s, h);
 if (u < 0.0f || u > 1.0f)
  return(false);
 float q [3];
 cross(q, s, e1);
 float v = f * dot(d, q);
 if (v < 0.0f || u + v > 1.0f)
  return(false);
 float t = dot(e2, q)*f;
 if (t < 0.0f)
  return(false);
 return(true);
}
```

This code is called inside the visibility function. Since this is an unoptimized code, it will check all other triangles of the mesh against the ray. If no triangle is intercepted, the direction is free from obstacles, and the visibility function returns 1; otherwise, it returns zero. The code fragment below implements the visibility function.

```
bool Visibility(Scene* scene, int vertexidx, Vector3* direction)
{
 bool visible (true);
 Vector3& p = scene->vertices[vertexidx];
 for (int i = 0; i < scene->number_of_triangles; i++)
 {
  Triangle& t = scene->triangles[i];
  if ((vertexidx != t.a) && (vertexidx != t.b) && (vertexidx != t.c))
  {
   Vector3& v0 = scene->vertices[t.a];
   Vector3& v1 = scene->vertices[t.b];
   Vector3& v2 = scene->vertices[t.c];
   visible = !RayIntersectsTriangle(&p, direction, &v0, &v1, &v2);
   if (!visible)
    break;
  }
 }
 return(visible);
}
```

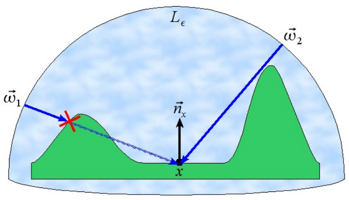

Figure 9: For the shadowed transfer function, occlusions are considered, producing shadows. The direction $\vec{\omega}_1$ has been occluded by other elements of the scene and will not be considered when illuminating x.

For the case of the shadowed transfer function (Figure 9), the projection code is identical to that presented for unshadowed transfer, except for the addition of the binary visibility function.

```
void ProjectShadowed(Color** coeffs, Sampler* sampler,
                     Scene* scene, int bands)
{
 ...
 for (int i = 0; i < scene->number_of_vertices; i++)
 {
  for (int j = 0; j < sampler->number_of_samples; j++)
  {
   Sample& sample = sampler->samples[j];
   if (Visibility(scene, i, &sample.cartesian_coord))
   {
    float cosine_term = dot(&scene->normals[i], &sample.cartesian_coord);
    for (int k = 0; k < bands*bands; k++)
    {
     float sh_function = sample.sh_functions[k];
     int materia_idx = scene->material[i];
     Color& albedo = scene->albedo[materia_idx];
     coeffs[i][k].r += (albedo.r * sh_function * cosine_term);
     coeffs[i][k].g += (albedo.g * sh_function * cosine_term);
     coeffs[i][k].b += (albedo.b * sh_function * cosine_term);
    }
   }
  }
 }
 ...
}
```

Figure 10 shows the result of using the shadowed transfer function for rendering the teacup from the same viewpoint shown in Figure 8 and under the same lighting functions. Note the considerable increase in realism due to the presence of soft shadows, when compared to the images shown in Figure 8.

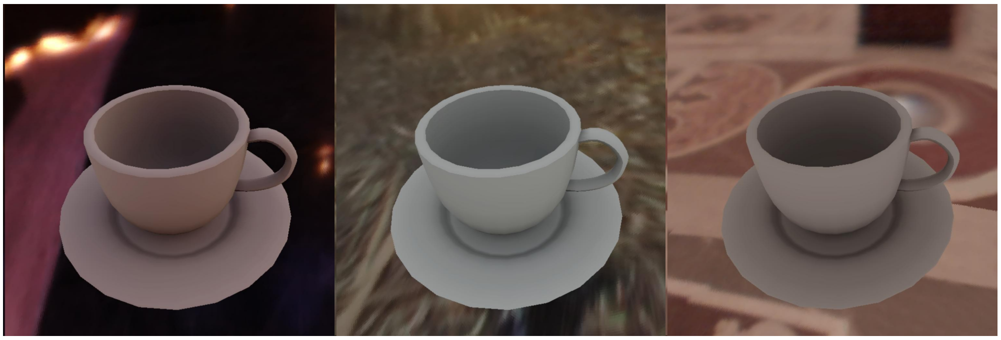

Figure 10: The teacup rendered using a shadowed transfer function under three different lighting functions. Notice the soft shadows that greatly increases the realism of the scene.

One should notice, however, that the plain shadowed transfer function only computes one level of transport (*i.e.*, the direct lighting transport level). For additional bounces, some considerations should be done, and will be discussed in the next section.

6.2.6.3 Interreflected Transfer Function

Given the shadowed transfer function, it is possible to consider some refinements to perform indirect lighting and make the renderings even more realistic. However, one should note that any additional level of indirect lighting implies more transfer coefficients that need to be precomputed, stored and then handled during runtime. However, as it was shown previously, these transfer coefficients can be used to compute the next bounce (Equation 3) and, when all bounces were performed, they can be merged, resulting in a final transfer vector (Equation 2), which handles all bounces plus the one for direct transfer, obtained from the shadowed transfer vector.

To implement interreflected transfer, first the transfer coefficients for shadowed transfer should be computed. By doing so, each vertex of the scene will have a transfer vector associated to it. The next step is to perform the additional bounces, obtaining the terms step by step and adding them at the end.

Even if it is easy to understand the basics, implementing such a transfer function may be challenging (and complicated to debug). But it is just a matter of brute force. For more details on how to implement the interreflected transfer function, please see [10].

6.3 Real Time Step

Once computed, the light and transfer coefficients are ready to be used for real-time rendering. This is the easiest part of the entire technique, which consists of computing a per-vertex dot product -- one that can have lots of coefficients, but still a dot product.

6.3.1 Evaluating the PRT Equation in the CPU

To determine the final shading of a point (vertex), it is necessary to evaluate the PRT equation (Equation 8 or 9, depending on the existence of indirect lighting) for it by computing the dot product between its transfer vector and the lighting vector. The following code fragment computes the dot product for each color channel and renders the results using OpenGL (with Gouraud shading).

```
void Render(Color* light, Color** coeffs, Scene* scene, int bands)
{
 glBegin(GL_TRIANGLES);

 for (int i = 0; i < scene->number_of_triangles; i++)
 {
  Triangle& t = &scene->triangles[i];
  Vector3& v0 = scene->vertices[t.a];
  Vector3& v1 = scene->vertices[t.b];
  Vector3& v2 = scene->vertices[t.c];

  Color c0 = { 0.0f, 0.0f, 0.0f };
  Color c1 = { 0.0f, 0.0f, 0.0f };
  Color c2 = { 0.0f, 0.0f, 0.0f };
  for (int k = 0; k < bands*bands; k++)
  {
   c0.r += (light[k].r * coeffs[t->a][k].r);
   c0.g += (light[k].g * coeffs[t->a][k].g);
   c0.b += (light[k].b * coeffs[t->a][k].b);
   c1.r += (light[k].r * coeffs[t->b][k].r);
   c1.g += (light[k].g * coeffs[t->b][k].g);
   c1.b += (light[k].b * coeffs[t->b][k].b);
   c2.r += (light[k].r * coeffs[t->c][k].r);
   c2.g += (light[k].g * coeffs[t->c][k].g);
   c2.b += (light[k].b * coeffs[t->c][k].b);
  }

  glColor3f(c0.r, c0.g, c0.b);
  glVertex3f(v0.x, v0.y, v0.z);

  glColor3f(c1.r, c1.g, c1.b);
  glVertex3f(v1.x, v1.y, v1.z);

  glColor3f(c2.r, c2.g, c2.b);
  glVertex3f(v2.x, v2.y, v2.z);
 }

 glEnd();
}
```

6.3.2 Evaluating the PRT Equation in the GPU

Evaluating the PRT equation in the GPU is pretty straightforward. All it takes is to declare one *uniform parameter* (*i.e.*, a parameter that will remain constant over the processing of all vertices), which contains the lighting function coefficients. It is still necessary to declare the transfer vector, which varies for each vertex. Given that, the vertex program only needs to compute the dot product and forward the result to the rasterizer.

The following code implements the vertex shader for PRT:

```
struct app2vertex
{
 float4 f4Position           : POSITION;
 float4 f4Color              : COLOR;
 float3 vf3Transfer [N*N];
};

struct vertex2fragment
{
 float4 f4ProjPos            : POSITION;
 float4 f4Color              : COLOR;
};

vertex2fragment VertexShader
(
 app2vertex IN,
 uniform float3 vf3Light [N*N],
 uniform float4x4 mxModelViewProj
)
{
 vertex2fragment OUT;

 OUT.f4ProjPos = mul(mxModelViewProj, IN.f4Position);

 OUT.f4Color = float4(0.0f, 0.0f, 0.0f, 1.0f);
 for (int i = 0; i < N*N; i++)
 {
  OUT.f4Color.r += (IN.vf3Transfer[i].r * vf3Light[i].r);
  OUT.f4Color.g += (IN.vf3Transfer[i].g * vf3Light[i].g);
  OUT.f4Color.b += (IN.vf3Transfer[i].b * vf3Light[i].b);
 }

 return(OUT);
}
```

The following code implements the pixel shader for PRT:

```
struct fragment2screen
{
 float4 f4Color              : COLOR;
};

vertex2fragment PixelShader
(
 vertex2fragment IN
)
{
 fragment2screen OUT;
 OUT.f4Color = IN.f4Color;
 return(OUT);
}
```

7 Extensions to the Plain PRT Rendering

PRT provides a global illumination solution for scenes containing only diffuse surfaces. To increase the realism of the generated images, it is possible to add specular effects to the final rendering. Sloan et al. [24] described a method to render glossy reflections using PRT. It is also possible to add high frequency specular reflections using reflection mapping [2] [8]. In this case, the results will not be physically correct, but the technique is very easy to implement, very fast to evaluate in real time, and produces more sophisticated images. Figure 11 shows the result of integrating diffuse PRT with reflection mapping. On the left, pure diffuse and specular renderings are shown. On the right, both techniques have been combined, producing a more realistic image. A code fragment showing how to combine the diffuse and specular contributions is shown below.

The reflection mapping implementation should use an environment mapping that is consistent with lighting function (light probe). To implement reflection mapping on the CPU, one can use the `EXT_texture_cube_map` OpenGL extension, for which a tutorial can be found at [18]. For a GPU implementation of reflection mapping, refer to the NVIDIA SDK [17].

When using high dynamic range images for the environment texture, a blooming effect [17] and tone mapping [22] [27] can also be used to enhance the the image quality even further. These techniques were used to synthesize Figure 11 (right). A comprehensive discussion on how to combine diffuse PRT, reflection mapping and high dynamic range images in real time can be found in [26].

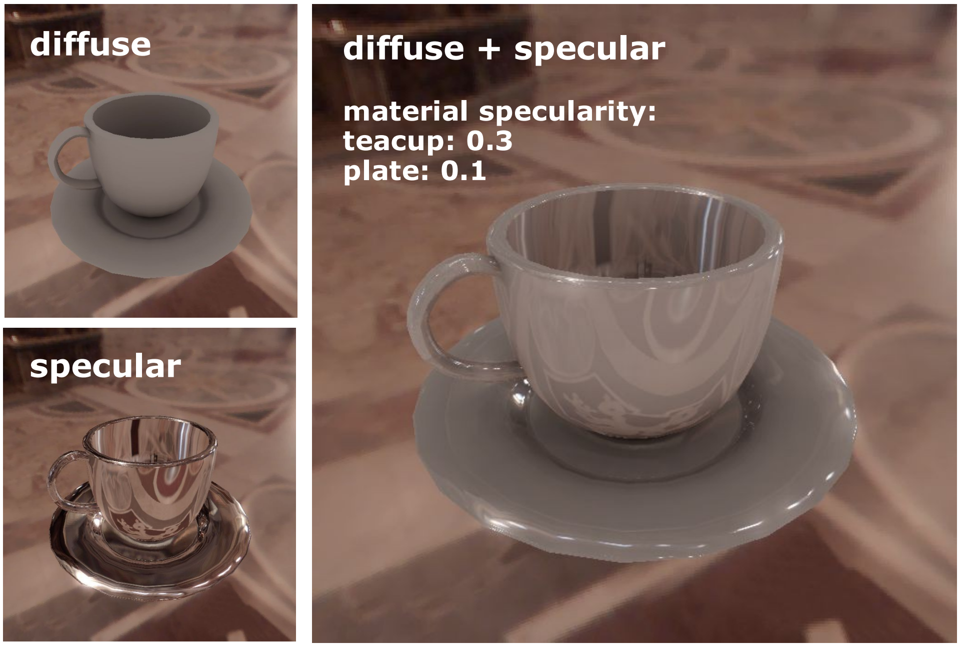

Figure 11: A teacup rendered using both diffuse PRT and reflection mapping. An high dynamic range image was used for lighting function and environment texture, so tone mapping was employed to properly exhibits the generated image.

Once both diffuse PRT result and reflection mapping color have been computed, one can combine them using the following formula, where specular factor is equivalent to the specular coefficient used in the Phong model [19]. Remember that the diffuse coefficient was already encoded in the PRT diffuse color, as explained on section 6.2.6:

$$FinalColor = PRTDiffuseColor + SpecularFactor * SpecularColor$$

8 Conclusion

Precomputed radiance transfer is capable of rendering highly realistic images of static diffuse environments in real time. It assumes an infinitely distant light source, and consists of factoring the rendering equation into incident light and light transport terms. By projecting these terms into a family of orthonormal basis functions, the resulting simplification of the rendering equation can be estimated using dot products. PRT has a big advantage over the commonly used local illumination models since it is able to render scenes using any number of light sources without compromising performance. It is also capable of producing soft shadows that highly increases the realism of the scenes. By taking extra light bounces into account, one can simulate diffuse interreflection among the objects in the scene, increasing the realism even further.

PRT can produce glossy reflections [24], and specular ones can be easily combined with PRT renderings by using reflection mapping. In this case, the resulting images tend to look quite realistic even though the approach is not physically correct.

Currently, PRT cannot be used with dynamic scenes. Although, spherical harmonic functions allow the lighting function or the scene to be rotated, preserving the precomputed light transport, it does not allow object translation, since this would change the visibility function. Some research has been done to try to remove this limitation [1] [13] [23] [25] [28], but no definitive solution has been found yet.

The quality of the renderings obtained with PRT depends on the basis functions chosen to perform the projections. Using a small number of coefficients, spherical harmonic functions are acceptable just to approximate low frequency characteristics of the lighting function. One way to support high-frequency approximation of the lighting function with less coefficients is to use Haar Wavelets [21].

This tutorial focused on a gentle introduction to PRT, using diffuse reflection and direct lighting only. More sophisticated transfer functions are possible supporting, for instance, interreflections, caustics, refractions, and subsurface scattering. PRT has proved to be a very interesting and promising rendering technique.

Acknowledgments

The teacup model was created by Antônio César de Boer from Singular Studios. Rafael Huff helped creating the light probe shown in Figure 5. This work was partially supported by Microsoft Brazil.

References

[1] Annen, T., Kautz, J., Durand F., and Seidel H. 2004. “Spherical harmonic gradients for mid-range illumination”. In EuroGraphics Symposium on Rendering, 2004.

[2] Blinn, J.F. and Newell, M.E. 1976. “Texture and reflection in computer generated images”. Communications of the ACM 19, 542–546.

[3] Cohen, M. F. and Wallace, J. R. 1993. “Radiosity and Realistic Image Synthesis”. Morgan Kaufmann, August 4, 1993, pp 32-33.

[4] Cook, R. L. and Torrence, K. E. 1981. “A Reflectance Model for Computer Graphics” Computer Graphics (SIGGRAPH '81 Proceedings), pp. 307-316, July 1981.

[5] Debevec, P. 2002. “Image-Based Lighting. Computer Graphics and Applications”. March/April 2002. Pages 26-34.

[6] Dempski, K. and Viale, E. 2005. “Advanced Lighting and Materials with Shaders”. Wordware Publishing, Inc., 2005, pp 157-210.

[7] DirectX SDK. Precomputed Radiance Transfer demos.
http://www.microsoft.com/directx

[8] Miller G. and Hoffman C. 1984. "Illumination and reflection maps: Simulated objectsin simulated and real environments". SIGGRAPH 84 Advanced Computer Graphics Animation seminar notes, 1984.

[9] Goral, C., Torrance, K.E., Greenberg D.P. and Battaile, B. 1984. “Modelling the interaction of light between diffuse surfaces”. Computer Graphics (Proceedings of SIGGRAPH 1984) 18 (3), 213–222.

[10] Green, R. 2003. “Spherical Harmonic Lighting: The Gritty Details”. Game Developers Conference 2003.

[11] Hammerslay, J. M., Handscomb, D. C., 1964. “Monte Carlo Methods”.

[12] Kajiya, J.T. 1986. The rendering equation. Computer Graphics (Proceedings of SIGGRAPH 1986) 20 (4), 143–150.

[13] Kristensen, A. W., Akenine-Möller, T. and Jensen, H. W. 2005. “Precomputed Local Radiance Transfer for Real-Time Lighting Design”. To appear in Proceedings of ACM SIGGRAPH 2005, Los Angeles, USA, July 31 - August 4, 2005.

[14] Light Probe Image Gallery.
http://www.debevec.org/Probes/

[15] Mark. W. R. 2003. "Cg: A System for Programming Graphics Hardware in a C-like Language". ACM Transactions on Graphics, 22(3):896--907, July 2003.

[16] Möller, T., and Trumbore, B. 1997. “Fast, minimum storage raytriangle intersection”. Journal of graphics tools, 2(1):21-28, 1997.

[17] NVIDIA Corporation. “NVIDIA Software Development Kit (SDK)”.
http://download.developer.nvidia.com/developer/SDK

[18] NVIDIA Corporation. 1999. "Cube Map OpenGL Tutorial".
http://developer.nvidia.com/object/cube_map_ogl_tutorial.html

[19] Phong, B.T. 1973. “Ilumination for Computer Generated Surfaces”. Doctoral Thesis, University of Utah, 1973.

[20] Press, W.H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T. "Numerical Recipes in C: The Art of Scientific Computing". Cambridge University Press, 1992, pp 252-254.

[21] Ramamoorthi, R., Ng, R., and Hanrahan, P. 2003. “All-Frequency Shadows Using Non-linear Wavelet Lighting Approximation”. SIGGRAPH`03.

[22] Reinhard, E., Stark, M., Shirley, P., and Ferwerda, J. 2002. "Photographic tone reproduction for digital images". ACM Transactions on Graphics 21, 3, 267-276.

[23] Sloan PP. 2006. “Normal Mapping for Precomputed Radiance Transfer”. Proceedings of ACM Symposium on Interactive 3D Graphics and Games.

[24] Sloan, PP., Kautz, J., and Snyder, J. 2002. “Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments”. ACM Transactions on Graphics 21, 3, 527–536.

[25] Sloan, PP., Luna, B. and Snyder, J. 2005. “Local, Deformable Precomputed Radiance Transfer”. SIGGRAPH 2005, July, 2005.

[26] Slomp, M.P.B. 2005. “Rendering em Tempo Real com Iluminação Baseada em Imagens em Alta Faixa Dinâmica e Harmônicas Esféricas”. (in portuguese).

[27] Tumblin, J., and Rushmeier, H. 1991. "Tone reproduction for realistic computer generated images". Technical Report GIT-GVU-91-13, Graphics, Visualization, and Useability Center, Georgia Institute of Technology.

[28] Zhou, K., Hu, Y., Lin, S., Guo, B. and Shum, H. 2005. “Precomputed Shadow Fields for Dynamic Scenes”. ACM SIGGRAPH 2005, 1196-1201.