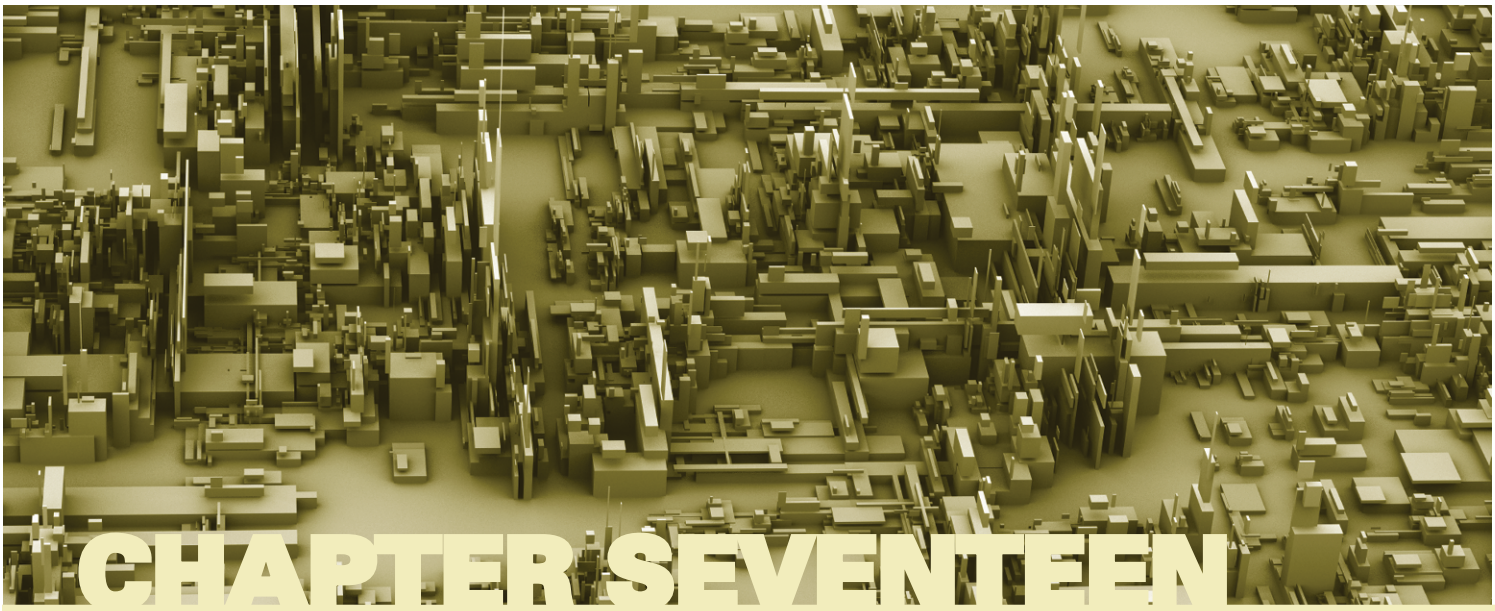


CHAPTER SEVENTEEN

★ 17 LIGHT TRANSPORT III: PRECOMPUTED LIGHT TRANSPORT

The rendering algorithms in the preceeding chapters generally take minutes to hours of computation to generate high-quality imagery for interesting scenes. This is in general the price to be paid for their robustness, flexibility, and generality. Of course, this computational cost has a number of disadvantages. Artists modeling scenes generally desire quick feedback, and many applications require not just faster rendering but also full interactivity.

It's desirable that the images generated by interactive rendering share in the benefits of the improvements in light transport algorithms of recent years. A variety of *precomputed light transport* algorithms have been developed to this end: the general idea is to separate the complete rendering computation into a portion that can be performed offline in a preprocess and stored in a data structure, such that the results can then be incorporated into the final rendering computation later—for example, in an interactive system. As a simple example of this idea of reusing precomputed data in rendering, consider that the photon map integrator of Chapter 15 could be modified to save the photon map it generated to disk. When rendering a walkthrough of a scene (where only the camera position changed from frame to frame, and the objects, materials, and lights stayed the same), the same photon map could be reused across all of the frames of the animation, saving the cost of recomputing it.

An early example of a precomputation-based approach to rendering is a lighting design developed by Dorsey et al. (1995). Their goal was to build a system for opera lighting designers that would allow them to interactively change the colors and intensities of the light sources illuminating a set until a desired result was achieved. (The positions of the light sources themselves were fixed.) Because they were using finite-element light

transport algorithms to compute global illumination and because of the slow computers of the time, they had to develop a system that didn't rerender the scene from scratch each time a lighting parameter changed.

The system they built took advantage of the linearity of light transport; recall from Section 5.4 that the image that results from lighting a scene with two light sources will be the same as the sum of the two images from lighting them with each source independently. Therefore, they rendered one image for each light source, at unit intensity. The interactive system loaded all of the images into memory and then allowed the user to change the color and intensity of each light source, computing the resulting image with a weighted sum of the individual light source images.

Another early technique in precomputed light transport is *ambient occlusion* (AO) (Zhukov et al. 1998; Landis 2002). The basic idea is to precompute at points on surfaces how much of the hemisphere above each point is occluded by other objects in the scene and how much is not occluded, using the resulting value in the range $[0, 1]$ to modulate a standard ambient lighting term. To compute the ambient occlusion, at each point of interest, we estimate the value of the integral

$$a(\mathrm{p}, \mathbf{n}, d) = \frac{1}{\pi} \int_{\mathcal{H}^2(\mathbf{n})} V(\mathrm{p}, \omega_\mathrm{i}, d)\, |\cos \theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}, \qquad [17.1]$$

where $V(\mathrm{p}, \omega, d)$ is a visibility funcion that returns one if the ray from p in direction ω is not occluded up to the distance d, and zero if it is occluded. The resulting value can then be stored with the scene geometry and then be used later for efficient rendering: Figure 17.1 shows the idea for an abstract structure scene; the image shows a visualization of the ambient occlusion values computed at each point.

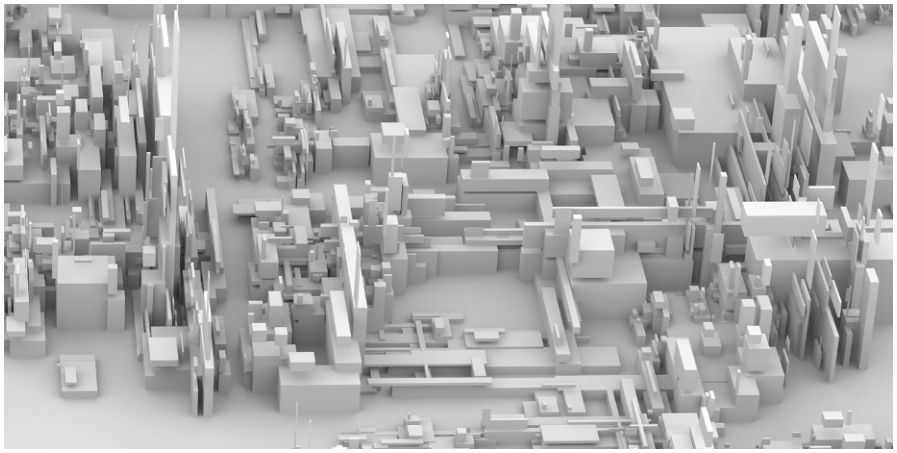


Figure 17.1: Ambient Occlusion. Abstract city-like scene rendered with ambient occlusion values, computed via Equation (17.1). Because the ambient occlusion value can be precomputed and the direct lighting can be computed very efficiently, a convincing facsimile of global illumination can be produced using AO values in shading calculations. (Model courtesy of Mark Schafer; this model was generated procedurally using *Structure Synth*.)

There are two main ways in which approaches to precomputed light transport can be categorized:

- What simplifications are made or which limitations are accepted, and hence which components of the simplified light transport model are precomputed?
- At what points in which domain are the precomputed values computed, and in what form are they stored?

For example, Dorsey et al.'s opera lighting algorithm limits the viewpoint to be a fixed position and requires that the scene geometry and light positions be fixed. These choices make it possible to precompute lighting values at the pixels of images and use them for rerendering.

A common use of ambient occlusion is for real-time rendering. In that setting, ambient occlusion values might be computed at points on surfaces of static geometry in the scene and stored in texture maps. When rendered in real time, the ambient occlusion values are retrieved from the textures and used for shading. Storing AO values in texture maps and using them for rendering in this manner does introduce the limitation that dynamic objects in the scene wouldn't affect the ambient occlusion of static objects and that dynamic objects won't have an ambient occlusion component of their own. More sophisticated approaches to ambient occlusion have been developed to address these sorts of shortcomings; see the "Further Reading" section for details.

Many different choices have been made with regard to the above two questions: for simplifications to the rendering problem, the viewpoint, geometry, materials, and light positions may or may not be fixed. Some of these may be constrained—for example, a limited set of viewpoints, a limited range of allowed motion of a light source, diffuse reflection only, etc. The precomputed values may in turn be stored in images, at polygon mesh vertices, texture maps, or 3D volumetric representations. Different requirements for quality and performance lead to different trade-offs in this space.

In the remainder of this chapter, we'll introduce some of the theory and a number of implementations of algorithms for precomputed light transport. This area of rendering has seen a substantial amount of research in recent years, so this chapter can only provide an introduction to some of the main ideas; the "Further Reading" section has many more references for this area. We will also side-step some of the important practical issues in representing the precomputed values (for example, parameterizing general triangle meshes for mapping (u, v) texture maps to them for storing precomputed values), instead focusing on how the precomputation is done by an offline renderer and showing how these values are then used for rendering without going into the details of their use in a particular real-time rendering environment.

17.1 BASIS FUNCTIONS: THEORY

To generalize a lighting design system, we might want to make it possible to move the position of light sources in the scene. For example, still assuming a fixed viewpoint, we'd ideally like to precompute a generalized image where each pixel stored a compact function of light position, such that evaluating the function for a given light position would give the color of the corresponding pixel for that light position.

The application of *basis functions* can make this sort of generalization possible. The idea here is to take a generally infinite-dimensional space of functions (pixel radiance as a function of light-source position) and represent them with a specific set of basis functions and corresponding basis function coefficients. Given a fixed preselected set of basis functions, the coefficient values at each pixel define the approximation to the general function in terms of the basis. The result will usually be an approximation, since a finite set of coefficients can't represent all possible functions. However, if the basis is chosen well, or the functions being approximated have properties like smoothness that can be taken advantage of, the results can be close enough for many applications.

Using basis functions in this manner makes it possible to represent potentially arbitrarily complex functions for rendering—for example, the light arriving at a point on a surface over the course of a day as the sun moves across the sky. There are three important advantages of representing functions like these with a basis that collectively make this technique extremely useful for interactive rendering.

The first advantage is that approximating a function with a basis makes it possible to map the infinite-dimensional space of functions to a finite-dimensional space of coefficients. While some functions can be represented precisely—for example, a constant function or the radiance arriving at a point from an unoccluded spherical light source—many others, like the directional distribution of incident radiance at a point in a complex environment, cannot. Representing complex functions like these with a basis transforms them to a form that is amenable to mathematical manipulation and evaluation in programs.

The second key advantage is the ability to compress the representation of the original function: if the error introduced from projection into the basis isn't objectionable, then basis functions can be used to substantially compress the representation of a function by converting it into a number of floating-point coefficients.

The third advantage is efficiency: certain operations may be much more efficiently performed with functions represented in a basis than with the original functions. In particular, *orthonormal* basis functions, which will be introduced in the following section, have important properties that make integration of products of functions in the basis very efficient. Another example of this idea is the use of the Fourier transform for image processing: operations like blurring an image with a very wide kernel may be much more efficient in the Fourier domain than in the original image domain.

There are many trade-offs to consider in choosing a set of basis functions for a particular application; how efficiently they can be evaluated, how easy it is to compute the representation of a given function in the basis, and how well the basis captures important features of the function, including discontinuities and smoothness.

17.1.1 A PIECEWISE-CONSTANT BASIS

A straightforward example of using basis functions to approximate an arbitrary function is to use a set of *piecewise-constant* basis functions, where we might decide to use four basis functions over the range [0, 1]. The first basis function, $B_1(x)$, has the value one between zero and 1/4 and is zero elsewhere; the next one, $B_2(x)$, is one between 1/4 and 2/4; and so forth.

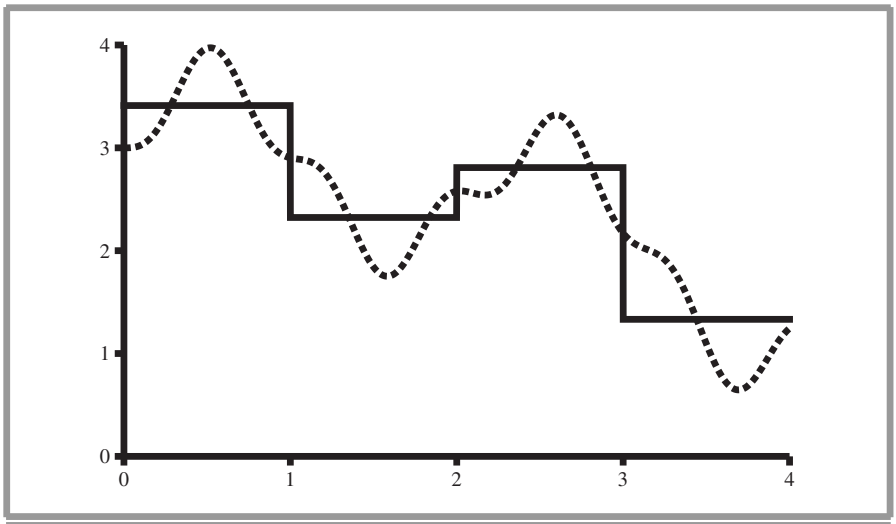


Figure 17.2: Representation of a Function in a Simple Piecewise-Constant Basis. A function (dashed lines) represented in the basis function defined by Equation (17.2) (solid lines).

$$B_i(x) = \begin{cases} 1 & \frac{i-1}{n} \leq x < \frac{i}{n} \\ 0 & \text{otherwise.} \end{cases} \tag{17.2}$$

To represent an arbitrary function $f(x)$ in this basis, we might evaluate it at a series of positions at the midpoint of each basis function's range, x_i, $x_0 = 1/8$, $x_1 = 3/8$, and so forth, giving a series of basis coefficient values $c_i = f(x_i)$. The representation of the original function in the basis, $\tilde{f}(x)$, is then found by a weighted sum of the basis functions (Figure 17.2):

$$f(x) \approx \tilde{f}(x) = \sum_i c_i B_i(x).$$

A more accurate approximation to f might be found by computing its average value over each subrange [0, 1/4), [1/4, 1/2), etc., or the approximation could be improved with a finer decomposition of [0, 1] with more basis functions. Alternatively, if we knew that f was a smooth function and that preserving the smoothness in the approximated function was important, we might use a different basis entirely (such as the Fourier basis).

In general, the representation of a function in a basis $\tilde{f}(x)$ and the original function $f(x)$ will be different. Given a particular basis, some functions can be represented exactly (for example, with the piecewise-constant basis used in the example above, the constant function $f(x) = c$). In general, however, the representation of the function in the basis will not match the original function exactly.

This notion of error in the approximation is made rigorous by defining a *norm* to measure the error between two functions in a space:

$$L_n[f(x), g(x)] = \left[\int (f(x) - g(x))^n \, \mathrm{d}x\right]^{\frac{1}{n}}.$$

Different choices of n for the L_n norm measure different types of error; for example, L_1 gives the average error, L_2 reduces the penalty for small errors, and the L_∞ norm gives the maximum error over the domain, penalizing isolated large errors.

17.1.2 PROJECTION ONTO A BASIS

In the general case, we are given some function $f(x)$ and have selected a set of N basis functions $B_i(x)$ defined over a domain D. (Here we are using the notation $f(x)$ to represent more than just functions over the 1D reals, but instead functions over arbitrary-dimensional domains.) We then want to compute the coefficients c_i that represent the function in the basis. The approximated function is given by the weighted sum of basis functions

$$\tilde{f}(x) = \sum_i^N c_i B_i(x).$$

In particular, we would like a set of c_i such that they minimize the difference between the original function $f(x)$ and the approximated function $\tilde{f}(x)$.

It can be shown that the optimal coefficients in terms of minimizing the least-squares error between $f(x)$ and $\tilde{f}(x)$ are given by integrating $f(x)$ with the *dual* of the basis, $\tilde{B}_i(x)$:

$$c_i = \int_D f(x)\tilde{B}_i(x)\,\mathrm{d}x, \tag{17.3}$$

where the dual of a basis function is defined as the function $\tilde{B}_i(x)$ such that

$$\int_D B_i(x)\tilde{B}_j(x)\,\mathrm{d}x = \delta_{i,j},$$

where $\delta_{i,j}$ is the Kronecker delta function,

$$\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise.} \end{cases}$$

17.1.3 ORTHONORMAL BASIS FUNCTIONS

An important type of basis functions are *orthonormal basis functions*. An orthonormal basis has two important properties. The first is *orthogonality*: given any two distinct basis functions $B_i(x)$ and $B_j(x)$ where $i \neq j$, the integral of the product of the two basis functions is zero:

$$\int_D B_i(x)B_j(x)\,\mathrm{d}x = 0.$$

Intuitively, this property can be understood to mean distinct basis functions account for distinct properties of the function being represented in the basis. Basis functions like the piecewise-constant basis used in Section 17.1.1 naturally have this property, since the product $B_i(x)B_j(x)$ is always zero for $i \neq j$ and thus the integral of their product is zero. Note, however, that many orthonormal bases do not have this property of non-overlapping domains although their products still integrate to zero over the entire domain.

The second property of an orthonormal basis is a normalization property—the integral of a particular basis function with itself over the domain is one:

$$\int_D B_i(x) B_i(x)\, \mathrm{d}x = 1.$$

These properties can be understood as generalizations of how the x, y, and z coordinate axes form an orthonormal basis for points in 3D space: given any particular point in 3D, there's a unique set of coordinates that represents it.

Given these definitions, the piecewise-constant basis defined above is orthogonal but not orthonormal (unless $n = 1$)—the value of the integral of each one's product with itself over the domain is $1/n$. However, the piecewise-constant basis above can be easily modified to be normalized

$$B_i(x) = \begin{cases} n & \frac{i-1}{n} \le x < \frac{i}{n} \\ 0 & \text{otherwise.} \end{cases}$$

Using an orthonormal basis to represent a space of functions has two important advantages compared to using a non-orthonormal basis: projecting a function into the basis is easier, and integration of the product of functions in the basis can be computed extremely efficiently.

The first advantage—easy projection of functions into the basis—comes from the useful feature of orthonormal basis functions that each basis function is equal to its dual:

$$B_i(x) = \tilde{B}_i(x).$$

The effect of this property is that coefficients of an arbitrary function in the basis are found by just integrating against the basis functions themselves:

$$c_i = \int_D f(x) B_i(x)\, \mathrm{d}x.$$

There's no need to find the dual basis functions for the chosen basis.

The second advantage, efficient integration, is an extremely useful one in practice. We will often want to compute the integral of the product of two functions represented in the basis (for example, a BRDF function and an incident radiance function). If the two functions $f(x)$ and $g(x)$ are represented by two sets of coefficients, c_i and c'_i, in the basis, the integral is

$$\int_D \tilde{f}(x) \tilde{g}(x)\, \mathrm{d}x = \int_D \left(\sum_i c_i B_i(x) \right) \left(\sum_j c'_j B_j(x) \right) \mathrm{d}x.$$

The sums and the coefficients can be pulled outside of the integral, giving a nested sum of integrals of products of basis functions:

$$\sum_i \sum_j c_i\, c'_j \int_D B_i(x) B_j(x)\, \mathrm{d}x.$$

Recall that orthonormality means that for $i \neq j$, $\int B_i(x)B_j(x)\mathrm{d}x = 0$. Therefore, many terms in the above sum are necessarily zero, and what remains is:

$$\sum_i c_i \, c_i' \int_D B_i(x)B_i(x) \, \mathrm{d}x.$$

Finally, because the integral of the product of an orthonormal basis function with itself is one, the integrals are all one, and we are left with:

$$\int_D \tilde{f}(x)\tilde{g}(x) \, \mathrm{d}x = \sum_i c_i \, c_i'. \tag{17.4}$$

This is an incredibly useful result: the integral of the two functions in the basis can be found as a simple inner product of their coefficients. Unfortunately, integrals of products of three or more functions in orthonormal bases aren't as efficient as products of just two functions; the "Further Reading" section has pointers and information about these triple-product integrals, which have many important applications in rendering but are substantially more computationally complex.

17.2 SPHERICAL HARMONICS

The spherical harmonic (SH) basis functions are an orthonomal basis defined over the unit sphere; they are essentially a generalization of the Fourier basis to that domain (Section 7.1.1). Recall that the Fourier transform makes it possible to express many functions as weighted sums of sinusoids; the SH basis can do the same for directionally varying functions on the sphere and are thus a useful basis for many rendering problems. The SH basis will be used for all of the precomputed light transport algorithms in the remainder of this chapter.

The spherical harmonic functions $Y_l^m(\theta, \phi)$ are defined over a series of $l_{\max}$ *bands*, where each band l has indices $m = -l_{\max}, \ldots, l_{\max}$. Thus, if we wanted to use bands up to $l_{\max} = 1$ to represent a function in the SH basis, we would have basis functions $Y_0^0(\theta, \phi)$, $Y_1^{-1}(\theta, \phi)$, $Y_1^0(\theta, \phi)$, and $Y_1^1(\theta, \phi)$. The basis functions are functions of direction (θ, ϕ) defined using the associated Legendre polynomials $P_l^m(x)$.

$$Y_l^m(\theta, \phi) = K_l^m \mathrm{e}^{\mathrm{i}m\phi} P_l^{|m|}(\cos\theta). \tag{17.5}$$

Here, K_l^m is a normalization factor,

$$K_l^m = \sqrt{\frac{2l+1}{4\pi} \frac{(l-|m|)!}{(l+|m|)!}}, \tag{17.6}$$

and the first few associated Legendre polynomials are $P_0^0(x) = 1$, $P_1^{-1}(x) = 1/2\sqrt{1-x^2}$, $P_1^0(x) = x$, and $P_1^1(x) = -\sqrt{1-x^2}$. In the implementation below, we will compute values of the associated Legendre polynomials using a series of recurrences that express the higher order polynomials in terms of the lower order ones.

For rendering applications, the *real spherical harmonics* are typically used; the imaginary factor i in Equation (17.5) is not necessary. The real spherical harmonics are defined as:

$$Y_l^m(\theta, \phi) = \begin{cases} \sqrt{2} K_l^m P_l^{-m}(\cos\theta) \sin(-m\phi) & m < 0 \\ K_l^m P_l^m(\cos\theta) & m = 0 \\ \sqrt{2} K_l^m P_l^m(\cos\theta) \cos(m\phi) & m > 0. \end{cases} \quad [17.7]$$

These equations can also be expressed as polynomials in Cartesian (x, y, z) coordinates on the unit sphere rather than (θ, ϕ) angles. The polynomial forms can be found by applying the identities we previously used for BRDF direction vectors on page 425. For example,

$$Y_1^1(x, y, z) = -\sqrt{\frac{3}{4\pi}} x.$$

The basis projection formula from Equation (17.3) gives us the formula to compute the coefficients of the projection of a given function $f(\omega)$ into the SH basis (recall that the SH basis is orthonormal, so the basis functions are equal to their duals):

$$c_l^m = \int_{S^2} f(\omega) Y_l^m(\omega) \, d\omega. \quad [17.8]$$

Given SH coefficients, the reconstructed function from the coefficients is

$$\tilde{f}(\omega) = \sum_{l=0}^{l_{\max}} \sum_{l=-m}^{m} c_l^m Y_l^m(\omega). \quad [17.9]$$

It's often useful to treat a set of SH coefficients or SH basis functions as a vector indexed by a one-dimensional value i rather than indexed by (l, m):

$$c_0 = c_0^0, \qquad c_1 = c_1^{-1}, \qquad c_2 = c_1^0, \qquad c_i = \ldots \quad [17.10]$$

In general, the coefficient (l, m) maps to the index $i = l^2 + l + m$.

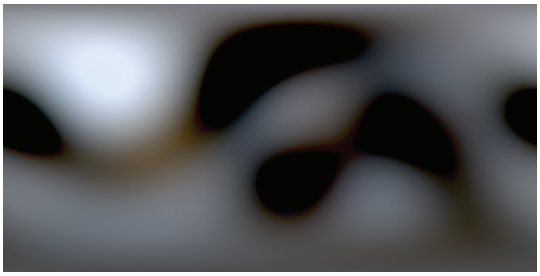
We will use Monte Carlo integration to project functions into the SH basis. Using the standard Monte Carlo estimator and the indexed notation, if N random samples are taken, we have the coefficients c_i given by

$$c_i \approx \frac{1}{N} \sum_{j=1}^{N} \frac{f(\omega_j) Y_i(\omega_j)}{p(\omega_j)}. \quad [17.11]$$

In implementations, a single set of samples ω_j will generally be generated for all c_i estimates, making it possible to reuse the computed values of $f(\omega_j)$ and $p(\omega_j)$ in all of their estimates, and only needing to evaluate $Y_i(\omega_j)$ for each of the coefficients c_i.

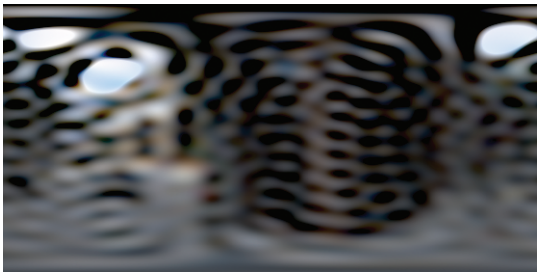
Figure 17.3 shows an example of representing an environment map in the SH basis for increasing numbers of basis function coefficients; note that as more coefficients are used, the approximation becomes closer to the original image. Further, note the smoothness of the function reconstructed from the SH basis. For many rendering tasks involving diffuse or very glossy reflection, very smooth representations of illumination such as these can be used without visible error. Spherical harmonics are in general only suitable for "low frequency" lighting effects, where the incident lighting distributions, visibility changes, and BSDFs are relatively blurry. In general, impractically large numbers of SH coefficients are needed to accurately represent higher frequency effects. Adaptive basis functions,

(a)



(b)

(c)



(d)

Figure 17.3: Environment Map Represented in the Spherical Harmonics Basis. (a) Original image, (b) Image represented with 5 SH bands, to $l_{\max} = 4$, (c) 10 SH bands, $l_{\max} = 9$, (d) 29 SH bands, $l_{\max} = 8$. Increasing the number of SH bands (and thus, coefficients) used to represent this complex spherical function improves the fidelity of the approximation. Even hundreds of SH coefficients can't represent the full complexity of this function, however.

such as nonlinear approximations based on wavelets, have been found to be better suited to "all frequency" lighting; see the "Further Reading" section for more details.

Functions for spherical harmonics are defined in the files `core/sh.h` and `core/sh.cpp`. Some additional functions related to rotating SH functions are defined in the file `core/shrots.cpp`; this file is quite large and so has been isolated from the main `sh.cpp` implementation file in order to reduce compile times when making changes to the `sh.cpp` file that don't involve the rotation functions.

17.2.1 EFFICIENT EVALUATION

It's useful to be able to efficiently evaluate the real SH functions up to arbitrary bands $l = l_{\max}$. For many real-time applications, it is worthwhile to precompute and tabularize the SH functions into cube map textures and then use fixed-function texture hardware to look up values from these cube maps. Here, we will develop a reasonably efficient SH evaluation routine that computes the basis function values for any given direction, though we will not use precomputed tables in the implementations below.

Two utility routines will be useful in the implementations of SH functions below. The first, `SHTerms()` gives the total number of basis functions (or equivalently, the total number of coefficients) that there are for a given $l_{\max}$ value. This function is used frequently for tasks like computing the size of arrays needed to hold SH coefficients.

⟨*Spherical Harmonics Declarations*⟩ ≡
```
inline int SHTerms(int lmax) {
    return (lmax + 1) * (lmax + 1);
}
```

Given a 1D array to be used for storing SH basis function coefficients, the `SHIndex()` computes the corresponding index into the array for a particular (l, m) basis function.

⟨*Spherical Harmonics Declarations*⟩ +≡
```
inline int SHIndex(int l, int m) {
    return l*l + l + m;
}
```

Now we can define the SH evaluation function, `SHEvaluate()`. `SHEvaluate()` takes a direction vector `w` and a maximum number of SH bands $l_{\max}$. It evaluates the real SH functions up to $l = l_{\max}$, storing the values of the functions in the provided `out` array, using the indexing scheme defined by `SHIndex()`. The length of `out` must be at least `SHTerms(lmax)`.

There are a few challenges in the implementation of this function. The first is efficiently bridging from the Cartesian (x, y, z) coordinates that `pbrt` uses for representing direction vectors to the spherical (θ, ϕ) coordinates that the real spherical harmonics are defined in terms of. The second will be applying a number of different recurrences to efficiently incrementally compute the associated Legendre polynomials and other required

SHIndex() 935

values. Using these recurrences is important: a naïve implementation based on evaluating each associated Legendre polynomials directly would be substantially less efficient.[1]

⟨*Spherical Harmonics Definitions*⟩ ≡
```
void SHEvaluate(const Vector &w, int lmax, float *out) {
    ⟨Compute Legendre polynomial values for cos θ 936⟩
    ⟨Compute K_l^m coefficients 939⟩
    ⟨Compute sin φ and cos φ values 940⟩
    ⟨Apply SH definitions to compute final (l, m) values 941⟩
}
```

The implementation starts by computing the associated Legendre polynomial values $P_l^m(\cos\theta)$, using the `legendrep()` utility routine. It stores the results in the `out` array; after these values have been stored, the implementation below will multiply by the remaining factors from Equation (17.7). Recall from Section 5.5.2 that in spherical coordinates, $\cos\theta = z$. Thus, we can just pass the z component of `w` in to the `legendrep()` function.

⟨*Compute Legendre polynomial values for* cos θ⟩ ≡ 936
```
legendrep(w.z, lmax, out);
```

There are four main steps in the implementation of `legendrep()`; each one uses a different recurrence to compute a different subset of the associated Legendre polynomial values for the given value. Figure 17.4 illustrates which subset of (l, m) values each step computes. The code here fills in the given `out` array where index `SHIndex(l,m)` holds the value of the associated Legendre polynomial for the given x value. It only computes the values of $P_l^m(x)$ for $m \geq 0$; the code in `SHEvaluate()` doesn't need the values for $m < 0$ in the end, so they aren't computed here.[2]

For the convenience of the code to follow, `legendrep()` starts by defining a macro `P()` that concisely indexes into the given output array.

⟨*Spherical Harmonics Local Definitions*⟩ ≡
```
static void legendrep(float x, int lmax, float *out) {
#define P(l,m) out[SHIndex(l,m)]
    ⟨Compute m = 0 Legendre values using recurrence 938⟩
    ⟨Compute m = l edge using Legendre recurrence 938⟩
    ⟨Compute m = l − 1 edge using Legendre recurrence 939⟩
    ⟨Compute m = 1, ..., l − 2 values using Legendre recurrence 939⟩
#undef P
}
```

First, the associated Legendre polynomials $P_l^0(x)$ are computed. The $m = 0$ values are the same as the regular Legendre polynomials $P_l(x)$. The first two of them are $P_0^0(x) = 1$

legendrep() 936
Vector 57

1 Even more efficient SH evaluation routines can be developed by writing a specialized code generator that generates C code to evaluate the recurrence relations directly.

2 There is commented-out code in the implementation to compute these values, using yet another recurrence, should those values be specifically needed for other applications.

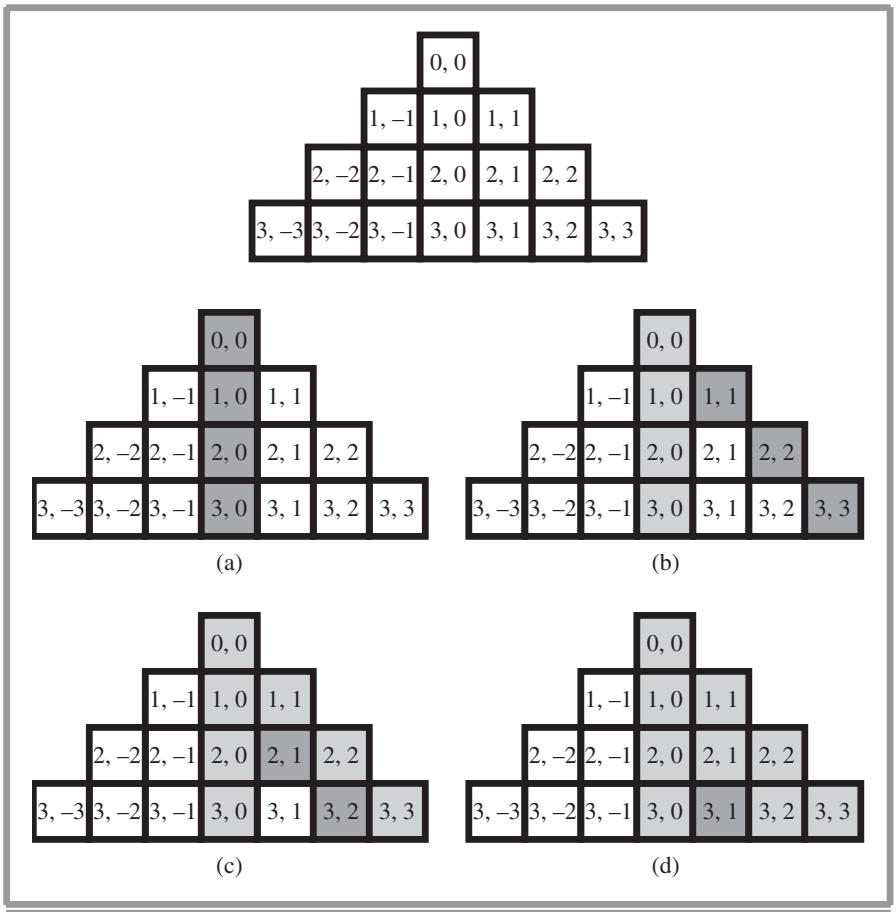


Figure 17.4: The Steps of Computing the Values of the Associated Legendre Polynomials. A series of recurrences is used to compute their values in the function `legendrep()`. At each step, the newly computed values have dark shading and the previously computed values have light shading. (a) The values along $m = 0$ are computed for all l bands, (b) the $m = l$ edge is computed from $l = 1$ to $l = l_{\max}$, (c) the $m = l - 1$ edge is computed for $l = 2$ to $l = l_{\max}$, (d) finally, remaining points $m = 1$ to $m = l - 2$ are computed for bands $l = 3$ and higher.

and $P_1^0(x) = x$. Subsequent ones are given by the recurrence

$$(l + 1)P_{l+1}(x) = (2l + 1)xP_l(x) - lP_{l-1}(x).$$

Rearranging terms and shifting the indexing by subtracting one from all of the l terms (which defines the same recurrence), we can see that if we want to compute the Legendre polynomial $P_l(x)$ and already know the values of the Legendre polynomials $P_{l-1}(x)$ and $P_{l-2}(x)$, then we have

$$P_l(x) = \frac{(2l - 1)xP_{l-1}(x) - (l - 1)P_{l-2}(x)}{l}.$$

Given this, the implementation is straightforward. (Note, however, that the loop indexing goes up to and includes $l = l_{\max}$.)

⟨*Compute $m = 0$ Legendre values using recurrence*⟩ ≡ 936

```
P(0,0) = 1.f;
P(1,0) = x;
for (int l = 2; l <= lmax; ++l)
    P(l, 0) = ((2*l-1)*x*P(l-1,0) - (l-1)*P(l-2,0)) / l;
```

After computing the $m = 0$ strip, the next step is to compute the values along the outer edge, where $m = l$. For these values we use following recurrence:

$$P_l^l(x) = -1^l(2l-1)!!(1-x^2)^{l/2}.$$

In this expression, !! denotes the double factorial, which is defined as

$$x!! = \begin{cases} 1 & x = 0 \text{ or } x = 1 \\ x(x-2)!! & \text{otherwise.} \end{cases}$$

The code in the fragment below incrementally computes this value for l values from 1 up to $l_{\max}$. The variable `neg` tracks the -1^l factor, `dfact` tracks the $(2l-1)!!$ factor, and `xpow` tracks the $(1-x^2)^{l/2}$ factor. These values are initialized with their appropriate values for $l = 1$, the initial condition going into the `for` loop.

After the value of $P_l^l(x)$ is updated in the loop, using the current values of these three variables, they are updated for the next pass through the loop. `neg` is negated, reflecting the effect of multiplication by another -1 factor, and the incremental effect of the next l value to the double factorial is incorporated in `dfact`. Finally, `xpow` is updated to account for the increment in l, which corresponds to an additional factor of $(1-x^2)^{1/2}$ using the value `xroot`, which stores this value.

⟨*Compute $m = l$ edge using Legendre recurrence*⟩ ≡ 936

```
float neg = -1.f;
float dfact = 1.f;
float xroot = sqrtf(max(0.f, 1.f - x*x));
float xpow = xroot;
for (int l = 1; l <= lmax; ++l) {
    P(l, l) = neg * dfact * xpow;
    neg *= -1.f;      // neg = (-1)^l
    dfact *= 2*l + 1; // dfact = (2*l-1)!!
    xpow *= xroot;    // xpow = powf(1.f - x*x, float(l) * 0.5f);
}
```

Next, the slice where $m = l - 1$ is computed. These values are found using the recurrence

$$P_{l+1}^l(x) = x(2l+1)P_l^l(x),$$

which is equivalent to

$$P_l^{l-1}(x) = x(2l-1)P_{l-1}^{l-1}(x),$$

if l is shifted by -1. Since we have computed $P_1^1(x)$ in the previous code, this loop starts with $l = 2$ and implements this recurrence directly.

⟨*Compute $m = l - 1$ edge using Legendre recurrence*⟩ ≡ 936
```
for (int l = 2; l <= lmax; ++l)
    P(l, l-1) = x * (2*l-1) * P(l-1, l-1);
```

We can now finally fill in the remainder of the values for $0 < m < l - 1$ using one last recurrence,

$$(l - m + 1)P_{l+1}^m(x) = (2l + 1)xP_l^m(x) - (l + m)P_{l-1}^m(x).$$

Once again we offset l by -1 and divide the right-hand side by the resulting $(l - m)$ factor in the implementation below to compute the value of $P_l^m(x)$. Note that the loop starting and ending points are chosen carefully so that they don't recompute values that the previous recurrences have already covered.

⟨*Compute $m = 1, \ldots, l - 2$ values using Legendre recurrence*⟩ ≡ 936
```
for (int l = 3; l <= lmax; ++l)
    for (int m = 1; m <= l-2; ++m)
        P(l, m) = ((2 * (l-1) + 1) * x * P(l-1,m) -
                   (l-1+m) * P(l-2,m)) / (l - m);
```

Next, the `SHEvaluate()` function needs to compute the K_l^m coefficients using Equation (17.6). A separate temporary array is allocated to store them here. Because the K_l^m values are fixed for given (l, m) values and because code that uses the spherical harmonics routines will generally only go up to a fixed $l = l_{\text{max}}$ value, a more optimized implementation might require the caller to compute an array holding these values once at startup time and then pass them into the SH routines. The implementation here doesn't impose this requirement, in the interests of simpler interfaces for the SH routines.[3]

⟨*Compute K_l^m coefficients*⟩ ≡ 936
```
float *Klm = ALLOCA(float, SHTerms(lmax));
for (int l = 0; l <= lmax; ++l)
    for (int m = -l; m <= l; ++m)
        Klm[SHIndex(l, m)] = K(l, m);
```

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static inline float K(int l, int m) {
    return sqrtf((2.f * l + 1.f) * INV_FOURPI * divfact(l, m));
}
```

The `divfact()` function efficiently computes the $(l - |m|)!/(l + |m|)$ term of the K_l^m function using the relationship

$$\frac{(a - |b|)!}{(a + |b|)!} = \frac{1}{(a - |b| + 1)(a - |b| + 2)\ldots(a + |b|)}$$

ALLOCA() 1009
divfact() 940
INV_FOURPI 1002
K() 939
SHIndex() 935
SHTerms() 935

3 A third alternative—probably the best—would be to have a local array initialized with precomputed K_l^m coefficients up to a large l_{max} value.

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static inline float divfact(int a, int b) {
    if (b == 0) return 1.f;
    float fa = a, fb = fabsf(b);
    float v = 1.f;
    for (float x = fa-fb+1.f; x <= fa+fb; x += 1.f)
        v *= x;
    return 1.f / v;
}
```

`SHEvaluate()` next computes the sine and cosine factors of Equation (17.7). From this equation, the real spherical harmonics have factors of the form $\sin(-m\phi)$ and $\cos(m\phi)$; thus, we need the values of $\sin -\phi$, $\sin(-2\phi)$, and so forth up to $\sin(-m\phi)$ for $m = l_{\max}$ and similarly for the cosine factors.

Space is allocated for the sine and cosine values in the `sins` and `coss` arrays. When these are filled in, the element `sins[i]` will hold the value $\sin(i\phi)$ and so forth. We use the relationships from page 425 to start this process; $\sin\phi = y/\sqrt{1-z^2}$ and $\cos\phi = x/\sqrt{1-z^2}$. If z is nearly one or negative one so that $\sqrt{1-z^2}$ is zero, then the direction is essentially toward the $z = 1$ or $z = -1$ pole and the sine and cosine values are set to arbitrary (but mutually consistent) values. Otherwise, the `sinCosIndexed()` routine computes all of the sine and cosine factors, given the values of the first ones computed and passed in here.

⟨*Compute* $\sin\phi$ *and* $\cos\phi$ *values*⟩ ≡ 936
```
float *sins = ALLOCA(float, lmax+1), *coss = ALLOCA(float, lmax+1);
float xyLen = sqrtf(max(0.f, 1.f - w.z*w.z));
if (xyLen == 0.f) {
    for (int i = 0; i <= lmax; ++i) sins[i] = 0.f;
    for (int i = 0; i <= lmax; ++i) coss[i] = 1.f;
}
else
    sinCosIndexed(w.y / xyLen, w.x / xyLen, lmax+1, sins, coss);
```

There are recurrence formulas that can be used to give the values of $\sin(i\phi)$ and $\cos(i\phi)$ in terms of $\sin((i-1)\phi)$ and $\cos((i-1)\phi)$:

$$\sin(x+y) = \sin x \cos y + \cos x \sin y$$
$$\cos(x+y) = \cos x \cos y - \sin x \sin y$$

The direct implementation of these, for $x = y = \phi$ and where $\sin\phi$ and $\cos\phi$ are passed in as parameters, is in the code below.

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static void sinCosIndexed(float s, float c, int n,
                          float *sout, float *cout) {
    float si = 0, ci = 1;
    for (int i = 0; i < n; ++i) {
        ⟨Compute sin iφ and cos iφ using recurrence 941⟩
    }
}
```

ALLOCA() 1009
sinCosIndexed() 940

⟨*Compute* sin $i\phi$ *and* cos $i\phi$ *using recurrence*⟩ ≡ 940
```
*sout++ = si;
*cout++ = ci;
float oldsi = si;
si = si * c + ci * s;
ci = ci * c - oldsi * s;
```

Given all of these components, the `SHEvaluate()` function can now finish computing the values from Equation (17.7). Going into this fragment, the `out` array holds the values of the associated Legendre polynomials for the corresponding (l, m) indices. For each term, then, we need to multiply in the K_l^m normalization factor and the sine or cosine factor. The implementation below uses the fact that $\sin -x = -\sin x$. For the $m < 0$ terms, note that the associated Legendre polynomial values for $m > 0$ are used as per the definition in Equation (17.7).

⟨*Apply SH definitions to compute final* (l, m) *values*⟩ ≡ 936
```
static const float sqrt2 = sqrtf(2.f);
for (int l = 0; l <= lmax; ++l) {
    for (int m = -l; m < 0; ++m)
        out[SHIndex(l, m)] = sqrt2 * Klm[SHIndex(l, m)] *
            out[SHIndex(l, -m)] * sins[-m];
    out[SHIndex(l, 0)] *= Klm[SHIndex(l, 0)];
    for (int m = 1; m <= l; ++m)
        out[SHIndex(l, m)] *= sqrt2 * Klm[SHIndex(l, m)] * coss[m];
}
```

17.2.2 PROJECTING LIGHT SOURCES

Now that we have routines for evaluating the spherical harmonics functions, we will start to add infrastructure to other parts of `pbrt` to support them. Given a light source and a particular point p in the scene, the light source defines a directionally varying incident radiance function $L_i(\mathrm{p}, \omega)$ at the point. For point light sources, the incident radiance comes from only a single direction, but for area or infinite area lights, many directions may carry incident illumination. In this section, we will add methods to the various light sources to project their incident radiance functions into SH coefficients.

The `Light::SHProject()` method computes the SH coefficients that represent the incident radiance function due to the light source in the SH basis up to the band $l = l_{\max}$. In this case, the coefficients aren't just scalars but are spectra; they are returned in the array `coeffs`. Here, we will define specific implementations of this method for point and infinite area light sources as well as a general implementation that uses quasi-Monte Carlo integration to compute SH coefficients at a point for the incident radiance function from arbitrary light sources.

Point 63
RNG 1003
Scene 22
SHIndex() 935
Spectrum 263

⟨*Light Interface*⟩ +≡ 606
```
virtual void SHProject(const Point &p, float pEpsilon, int lmax,
    const Scene *scene, bool computeLightVisibility, float time,
    RNG &rng, Spectrum *coeffs) const;
```

Point Lights

After initializing the coefficients to zero, the `PointLight::SHProject()` method checks to see if the point light source is visible from the given point. If not, the incident radiance is zero and its work is done. Note that the user can indicate whether occlusion of geometric objects in the scene should be ignored or not. For some applications, it's useful to use the unoccluded incident radiance function; for others, the effect of shadowing by geometry should be incorporated.

⟨*PointLight Method Definitions*⟩ +≡

```
void PointLight::SHProject(const Point &p, float pEpsilon, int lmax,
        const Scene *scene, bool computeLightVisibility, float time,
        RNG &rng, Spectrum *coeffs) const {
    for (int i = 0; i < SHTerms(lmax); ++i)
        coeffs[i] = 0.f;
    if (computeLightVisibility &&
        scene->IntersectP(Ray(p, Normalize(lightPos - p), pEpsilon,
                              Distance(lightPos, p), time)))
        return;
    ⟨Project point light source to SH 942⟩
}
```

Because point light sources are defined by a delta distribution, projecting their contribution into SH with Equation (17.8) is straightforward. If we define ω_i to be the normalized direction from the point p to the point light source position p_l and take advantage of the fact that the point light is defined by a delta distribution (recall Section 14.6.2), then the integral turns into a single term to evaluate:

$$\begin{aligned} c_l^m &= \int_{S^2} L_i(p, \omega) Y_l^m(\omega)\, d\omega \\ &= \int_{S^2} \delta(\omega - \omega_i) \frac{I}{||p_l - p||} Y_l^m(\omega)\, d\omega \\ &= \frac{I}{||p_l - p||} Y_l^m(\omega_i). \end{aligned}$$

⟨*Project point light source to SH*⟩ ≡ 942

```
float *Ylm = ALLOCA(float, SHTerms(lmax));
Vector wi = Normalize(lightPos - p);
SHEvaluate(wi, lmax, Ylm);
Spectrum Li = Intensity / DistanceSquared(lightPos, p);
for (int i = 0; i < SHTerms(lmax); ++i)
    coeffs[i] = Li * Ylm[i];
```

General Light Sources

For general light sources without specialized implementations of the method, the default implementation of `Light::SHProject()` uses quasi-Monte Carlo integration to estimate the SH coefficient values. The implementation here starts by computing random scramble values for generating low-discrepancy samples using the routines from Section 7.4.3

ALLOCA() 1009
Distance() 65
DistanceSquared() 65
Point 63
PointLight::Intensity 611
PointLight::lightPos 611
Ray 66
RNG 1003
Scene 22
Scene::IntersectP() 24
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Vector 57
Vector::Normalize() 63

and allocating a temporary array, `Ylm`, for holding values of the SH basis functions returned by `SHEvaluate()`.

⟨*Light Method Definitions*⟩ +≡
```
void Light::SHProject(const Point &p, float pEpsilon, int lmax,
        const Scene *scene, bool computeLightVisibility, float time,
        RNG &rng, Spectrum *coeffs) const {
    for (int i = 0; i < SHTerms(lmax); ++i)
        coeffs[i] = 0.f;
    uint32_t ns = RoundUpPow2(nSamples);
    uint32_t scramble1D = rng.RandomUInt();
    uint32_t scramble2D[2] = { rng.RandomUInt(), rng.RandomUInt() };
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (uint32_t i = 0; i < ns; ++i) {
        ⟨Compute incident radiance sample from light, update SH coeffs 943⟩
    }
}
```

For each sample to be taken of the estimator, a `LightSample` is created to sample the light source.

⟨*Compute incident radiance sample from* `light`*, update SH* `coeffs`⟩ ≡ 943
```
float u[2], pdf;
Sample02(i, scramble2D, u);
LightSample lightSample(u[0], u[1], VanDerCorput(i, scramble1D));
Vector wi;
VisibilityTester vis;
Spectrum Li = Sample_L(p, pEpsilon, lightSample, time, &wi, &pdf, &vis);
if (!Li.IsBlack() && pdf > 0.f &&
    (!computeLightVisibility || vis.Unoccluded(scene))) {
    ⟨Add light sample contribution to MC estimate of SH coefficients 943⟩
}
```

The values of the coefficients are found using Equation (17.11).

⟨*Add light sample contribution to MC estimate of SH coefficients*⟩ ≡ 943
```
SHEvaluate(wi, lmax, Ylm);
for (int j = 0; j < SHTerms(lmax); ++j)
    coeffs[j] += Li * Ylm[j] / (pdf * ns);
```

ALLOCA() 1009
InfiniteAreaLight 629
Light::Sample_L() 608
LightSample 710
Point 63
RNG 1003
RNG::RandomUInt() 1003
RoundUpPow2() 1002
Sample02() 372
Scene 22
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Spectrum::IsBlack() 265
VanDerCorput() 372
Vector 57
VisibilityTester 608
VisibilityTester::
  Unoccluded() 609

Infinite Area Lights

Recall that `InfiniteAreaLights` are represented by 2D texture maps that define emitted radiance as a function of (θ, ϕ) incident directions in spherical coordinates. Its `SHProject()` method here uses one of three different approaches, based on whether visibility from occluding objects should be included in the incident radiance function and based on the resolution of the image map.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
void InfiniteAreaLight::SHProject(const Point &p, float pEpsilon,
        int lmax, const Scene *scene, bool computeLightVis,
        float time, RNG &rng, Spectrum *coeffs) const {
    ⟨Project InfiniteAreaLight to SH using Monte Carlo if visibility needed 944⟩
    for (int i = 0; i < SHTerms(lmax); ++i)
        coeffs[i] = 0.f;
    int ntheta = radianceMap->Height(), nphi = radianceMap->Width();
    if (min(ntheta, nphi) > 50) {
        ⟨Project InfiniteAreaLight to SH from lat-long representation 945⟩
    }
    else {
        ⟨Project InfiniteAreaLight to SH from cube map sampling 949⟩
    }
}
```

If occlusion from objects in the scene should be included in the coefficients, Monte Carlo integration is a reasonable approach; although the other two methods for `InfiniteAreaLights` can be modified to handle this case, it's most efficiently handled here.

⟨*Project* `InfiniteAreaLight` *to SH using Monte Carlo if visibility needed*⟩ ≡ 944
```
if (computeLightVis) {
    Light::SHProject(p, pEpsilon, lmax, scene, computeLightVis,
                     time, rng, coeffs);
    return;
}
```

As an alternative to Monte Carlo integration, the incident radiance function from an infinite area light can be projected into the SH basis using Riemann integration, based on looping over all of the texels in the environment map. Relatively low-resolution environment maps are handled using a slightly different method that gives better accuracy in that case.

We can equivalently express the SH projection as integral over (θ, ϕ) directions, versus the definition over directions ω in Equation (17.8). Recall from Equation (5.4) that an extra $\sin\theta$ factor is introduced when transforming to an integral over (θ, ϕ), giving

$$c_l^m = \int_0^{2\pi} \int_0^{\pi} L_{\mathrm{i}}(\theta, \phi)\, Y_l^m(\theta, \phi) \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi.$$

We can then compute an approximation to this integral as a 2D Riemann sum over the elements of the environment map,

$$c_l^m \approx \frac{\pi}{N_\theta} \sum_t^{N_\theta} \frac{2\pi}{N_\phi} \sum_p^{N_\phi} L_{\mathrm{i}}(\theta_t, \phi_p)\, Y_l^m(\theta_t, \phi_p) \sin\theta_t, \qquad [17.12]$$

where (θ_t, ϕ_p) is the (θ, ϕ) direction corresponding to the environment map texel at (t, p) (recall Section 5.5.2).

Computing this sum involves looping over all of the texels and applying the formula above. The fragment ⟨*Precompute theta and phi values for lat-long map projection*⟩ (not

InfiniteAreaLight 629
InfiniteAreaLight::
radianceMap 631
Light::SHProject() 943
MIPMap::Height() 535
MIPMap::Width() 535
Point 63
RNG 1003
Scene 22
SHTerms() 935
Spectrum 263

included here) fills in the arrays sintheta, costheta, sinphi, and cosphi, of sizes ntheta and nphi such that sintheta[i] corresponds to the value of $\sin\theta$ for the ith texel in the theta direction and so forth.

⟨*Project* InfiniteAreaLight *to SH from lat-long representation*⟩ ≡ 944
```
⟨Precompute θ and φ values for lat-long map projection⟩
float *Ylm = ALLOCA(float, SHTerms(lmax));
for (int theta = 0; theta < ntheta; ++theta) {
    for (int phi = 0; phi < nphi; ++phi) {
        ⟨Add InfiniteAreaLight texel's contribution to SH coefficients 945⟩
    }
}
⟨Free memory used for lat-long theta and phi values⟩
```

For each texel, we compute the world-space direction vector ω that corresponds to its incident illumination direction. The running coeffs sum can then be updated with the contribution given by Equation (17.12).

⟨*Add* InfiniteAreaLight *texel's contribution to SH coefficients*⟩ ≡ 945
```
Vector w = Vector(sintheta[theta] * cosphi[phi],
                  sintheta[theta] * sinphi[phi],
                  costheta[theta]);
w = Normalize(LightToWorld(w));
Spectrum Le = Spectrum(radianceMap->Texel(0, phi, theta),
                       SPECTRUM_ILLUMINANT);
SHEvaluate(w, lmax, Ylm);
for (int i = 0; i < SHTerms(lmax); ++i)
    coeffs[i] += Le * Ylm[i] * sintheta[theta] *
        (M_PI / ntheta) * (2.f * M_PI / nphi);
```

For very-low-resolution environment maps, the Riemann approach can suffer from noticeable numeric error. Consider as an extreme the case of a single-texel environment map. This environment map represents uniform illumination from all directions, yet the Riemann sum will compute coefficients that are essentially the same as those for a distant light illuminating the receiving point from a single direction. The problem is that the factors of the sum other than the incident radiance function—the $Y_l^m(\theta, \phi)$ functions and the $\sin\theta$ factor—aren't being sampled at a high enough rate to capture enough of their variation to give a good result.

One option would be to resample the environment map to a higher resolution in this case, either explicitly or by taking more Riemann samples than there are texels. However, we will instead use a different approach based on the SHProjectCube() function, which projects a directionally varying function specified as a *cube map* into the SH basis. We will discuss the implementation of this function now before showing its use for InfiniteAreaLights.

Cube maps define a directionally varying function with six square two-dimensional texture maps, one along each of the $\pm x$, $\pm y$, and $\pm z$ coordinate axes. Cube maps are usually specified with their faces at unit distance from the origin. For example, consider the $-x$ face of a cube map at $x = -1$. The y and z coordinates of this face range from $[-1, 1]$.

ALLOCA() 1009
InfiniteAreaLight::
radianceMap 631
Light::LightToWorld 606
MIPMap::Texel() 535
M_PI 1002
SHEvaluate() 936
SHProjectCube() 946
SHTerms() 935
Spectrum 263
SPECTRUM_ILLUMINANT 277
Vector 57
Vector::Normalize() 63

Thus, given a texel at particular (y, z) coordinates on the $-x$ axis, the corresponding direction vector is $(-1, y, z)$. The ability to project functions defined by cube map faces into SH is useful in a number of other situations, most notably for environment maps already represented as cube maps. Rendering systems based on rasterization can more easily generate a cubical environment map than other parameterizations of the unit sphere of directions.

`SHProjectCube()` is a template function, parameterized by a function `func` that is called to sample the value of the underlying function in a particular direction. `func` should have the signature

```
Spectrum func(int u, int v, const Point &p, const Vector &w);
```

The `u` and `v` values passed to the callback function will be texel coordinates in the range [0, `res`), where `res` is the sampling resolution passed to `SHProjectCube()`, `p` is the point passed in, and `w` is the non-normalized direction vector from the origin to the point on the face.

The callback function can either be a class or `struct` with an `operator()` method, as the `InfiniteAreaCube` structure does below, or it may be a pointer to a regular function. Using an `operator()` is generally more efficient, since the callback can be expanded inline into the instantiation of `SHProjectCube()` if the method is defined inline to the class or structure.

The implementation of `SHProjectCube()` also does a Riemann sum for each face of the cube; in the general 2D setting, it is

$$\int_{x_a}^{x_b}\int_{y_a}^{y_b} f(x, y)\,\mathrm{d}x\,\mathrm{d}y \approx \frac{x_b - x_a}{N_x}\sum_i \frac{y_b - y_a}{N_y}\sum_j f(x_i, y_j).$$

The implementation of `SHProjectCube()` loops over the samples of a canonical face and updates the Riemann sum's contribution for all six faces of the cube map in each iteration.

⟨*Spherical Harmonics Declarations*⟩ +≡

```
template <typename Func>
void SHProjectCube(Func func, const Point &p, int res, int lmax,
                   Spectrum *coeffs) {
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (int u = 0; u < res; ++u) {
        float fu = -1.f + 2.f * (float(u) + 0.5f) / float(res);
        for (int v = 0; v < res; ++v) {
            float fv = -1.f + 2.f * (float(v) + 0.5f) / float(res);
            ⟨Incorporate results from +z face to coefficients 948⟩
            ⟨Incorporate results from other faces to coefficients⟩
        }
    }
}
```

ALLOCA() 1009
Point 63
SHTerms() 935
Spectrum 263

This task is made slightly more complex by the fact that we actually want to compute the integral of a function of directions on the unit sphere defined by the faces of a cube map,

rather than a simple 2D integral over a planar region. As such, a corrective factor has to be added to account for the distortion from the mapping from the planar faces of the cube map to the surface of the sphere. Intuitively, texels toward the edges of the cube map faces project to a smaller area on the unit sphere than those toward the middle. This factor is found by expressing the change of variables that corresponds to this situation and then computing the Jacobian of the resulting transformation function. In the general n-dimensional case, given a mapping between two domains $(y_1, \ldots, y_n) = g(x_1, \ldots, x_n)$, we have

$$\int f(x_1, \ldots, x_n)\, \mathrm{d}x_1 \ldots \mathrm{d}x_n = \int f(y_1, \ldots, y_n) \left| \frac{\partial(x_1, \ldots, x_n)}{\partial(y_1, \ldots, y_n)} \right| \mathrm{d}y_1 \ldots \mathrm{d}y_n,$$

where the partial derivatives term is the Jacobian of the mapping.

As a concrete example, consider integrating over the $+z$ face of a cube map. Based on the cube map definition, we can see see that the function g that maps points (x, y) on the $z = 1$ face to points on a sphere of radius r (x', y', z') is

$$(x', y', z') = g(x, y, r) = \left(\frac{rx}{\sqrt{x^2 + y^2 + 1}}, \frac{ry}{\sqrt{x^2 + y^2 + 1}}, \frac{r}{\sqrt{x^2 + y^2 + 1}}, \right).$$

Finding the entries of the Jacobian matrix and then the matrix determinant is a straightforward algebraic exercise; the end result is:

$$\frac{1}{(1 + x^2 + y^2)^{3/2}}.$$

This is the term that corrects for the varying density of cube map samples when mapped to the unit sphere. Because z was assumed to be 1 in the derivation above, more generally we can see that an appropriate correction term that can be used for all faces is:

$$\frac{1}{(x^2 + y^2 + z^2)^{3/2}}.$$

There is also an elegant geometric derivation of the change of variables term. Recall from Section 5.5.2 the relationship between differential area and projected solid angle,

$$\mathrm{d}\omega = \frac{\mathrm{d}A \cos\theta}{r^2}.$$

Given a differential area $\mathrm{d}A$ at a coordinate (x, y) on the $+z$ face of the cube map, it's easy to see that $r = \sqrt{x^2 + y^2 + 1}$ and thus $r^2 = x^2 + y^2 + 1$. We can also take advantage of the fact that the cosine of an angle in a right triangle is the ratio of the adjacent side to the hypotenuse: in Figure 17.5, note that the hypotenuse is r and the adjacent side has length 1. Therefore, $\cos\theta = 1/r$ and

$$\mathrm{d}\omega = \frac{\mathrm{d}A(1/r)}{r^2} = \frac{\mathrm{d}A}{r^3} = \frac{\mathrm{d}A}{(x^2 + y^2 + z^2)^{3/2}}.$$

Given all this work, computing the term for a sample on a face is straightforward. Here is the fragment for the $+z$ face; the others are similar and not included here. Recall that the integral must be computed independently for each of the `SHTerms(lmax)` coefficients; the final loop over `k` updates each of these.

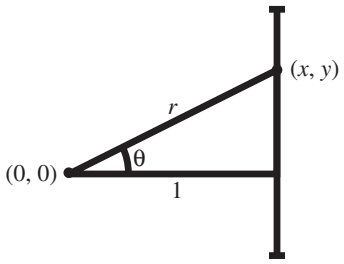


Figure 17.5: Geometric Setting for Computing the Relationship Between Differential Area on the Cube Map and on the Unit Sphere. The cosine of the angle θ is given by the ratio of the length of the adjacent side of the triangle, 1, and the hypotenuse, r.

⟨*Incorporate results from +z face to coefficients*⟩ ≡ 946
```
Vector w(fu, fv, 1);
SHEvaluate(Normalize(w), lmax, Ylm);
Spectrum f = func(u, v, p, w);
float dA = 1.f / powf(Dot(w, w), 3.f/2.f);
for (int k = 0; k < SHTerms(lmax); ++k)
    coeffs[k] += f * Ylm[k] * dA * (4.f / (res * res));
```

To use the `SHProjectCube()` function with infinite area lights, the `InfiniteAreaCube` structure is defined; it returns the appropriate color for the given direction from the infinite area light source environment map. Note that we don't create an explicit representation of the environment map in the form of a cube map here, but just compute the appropriate color for the sampled directions on cube map faces generated by `SHProjectCube()`.

⟨*InfiniteAreaLight Utility Classes*⟩ ≡
```
struct InfiniteAreaCube {
    ⟨InfiniteAreaCube Public Methods 948⟩
    const InfiniteAreaLight *light;
    const Scene *scene;
    float time, pEpsilon;
    bool computeVis;
};
```

⟨*InfiniteAreaCube Public Methods*⟩ ≡ 948
```
InfiniteAreaCube(const InfiniteAreaLight *l, const Scene *s,
                 float t, bool cv, float pe)
    : light(l), scene(s), time(t), pEpsilon(pe), computeVis(cv) { }
```

When the callback is called, the infinite light source's color is easily determined using existing methods.

InfiniteAreaCube 948
InfiniteAreaLight 629
Scene 22
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Vector 57
Vector::Normalize() 63

⟨*InfiniteAreaCube Public Methods*⟩ +≡ 948
```
Spectrum operator()(int, int, const Point &p, const Vector &w) {
    Ray ray(p, w, pEpsilon, INFINITY, time);
    if (!computeVis || !scene->IntersectP(ray))
        return light->Le(RayDifferential(ray));
    return 0.f;
}
```

For projecting low-resolution latitude-longitude maps into SH, a sampling resolution of 200×200 on each of the cube map faces is used, giving good directional sampling even if the original environment map is low resolution.

⟨*Project* `InfiniteAreaLight` *to SH from cube map sampling*⟩ ≡ 944
```
SHProjectCube(InfiniteAreaCube(this, scene, time, computeLightVis,
                               pEpsilon),
              p, 200, lmax, coeffs);
```

17.2.3 PROJECTING INCIDENT RADIANCE FUNCTIONS

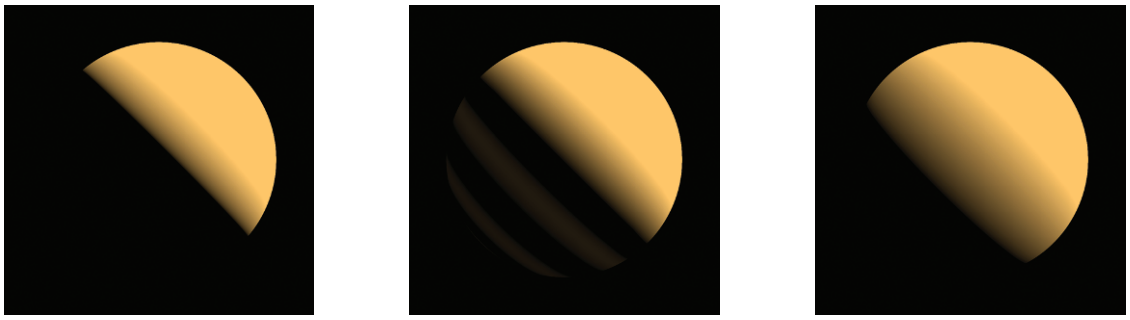
`pbrt` provides two utility functions for computing the complete incident radiance function at a point and projecting the function to the spherical harmonic basis. The first of them computes direct lighting from all light sources; the second computes direct and indirect lighting. Both will be used in the next section for the `CreateRadianceProbes` `Renderer`.

`SHProjectIncidentDirectRadiance()` loops over all of the lights in the scene and accumulates the sum of their SH coefficients into the `c_d` output array. Its only unusual feature is the call to `SHReduceRinging()`; this function (which will be introduced shortly) reduces some of the artifacts that can arise from projecting high-frequency functions into the SH basis when, as is commonly the case, there aren't enough SH coefficients to perfectly reconstruct the original function.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHProjectIncidentDirectRadiance(const Point &p, float pEpsilon,
        float time, MemoryArena &arena, const Scene *scene,
        bool computeLightVis, int lmax, RNG &rng, Spectrum *c_d) {
    ⟨Loop over light sources and sum their SH coefficients 949⟩
    SHReduceRinging(c_d, lmax);
}
```

⟨*Loop over light sources and sum their SH coefficients*⟩ ≡ 949
```
Spectrum *c = arena.Alloc<Spectrum>(SHTerms(lmax));
for (uint32_t i = 0; i < scene->lights.size(); ++i) {
    Light *light = scene->lights[i];
    light->SHProject(p, pEpsilon, lmax, scene, computeLightVis, time,
                     rng, c);
    for (int j = 0; j < SHTerms(lmax); ++j)
        c_d[j] += c[j];
}
```

CreateRadianceProbes 958
InfiniteAreaCube 948
InfiniteAreaCube::computeVis 948
InfiniteAreaCube::light 948
InfiniteAreaCube::pEpsilon 948
InfiniteAreaCube::scene 948
InfiniteAreaCube::time 948
INFINITY 1002
Light 606
Light::Le() 631
Light::SHProject() 943
MemoryArena 1015
Point 63
Ray 66
RayDifferential 69
Renderer 24
RNG 1003
Scene 22
Scene::IntersectP() 24
SHProjectCube() 946
SHReduceRinging() 951
Spectrum 263
Vector 57



(a) (b) (c)

Figure 17.6: Ringing From Projecting a Function with High Frequencies into the SH Basis. (a) Sphere illuminated with standard direct lighting algorithms from a point light source. (b) When projected into SH, the point source's infinite frequency content causes ringing in the function in the basis, leading to shading errors. (c) If the coefficients are scaled using the `SHReduceRinging()` function, the result is substantially improved.

We won't include the implementation of `SHProjectIncidentIndirectRadiance()` here; it computes a Monte Carlo estimate of Equation (17.11), computing incident radiance with the `Li()` method of the `Renderer` passed into it.

⟨*Spherical Harmonics Declarations*⟩ +≡
```
void SHProjectIncidentIndirectRadiance(const Point &p, float pEpsilon,
    float time, const Renderer *renderer, Sample *origSample,
    const Scene *scene, int lmax, RNG &rng, int nSamples, Spectrum *c_i);
```

17.2.4 REDUCING RINGING

When projecting signals with high-frequency directional variation over the unit sphere (such as the incident radiance function from a point light or a small area light), an artifact that often occurs is *ringing* in the function that is reconstructed from the coefficients and the SH basis functions. This is exactly the same problem we encountered when using perfect reconstruction with sinc functions to reconstruct signals with discontinuities in Section 7.1.6.

Figure 17.6 shows the effect of ringing with spherical harmonics. On the left is a sphere illuminated by a point light source rendered using the `DirectLightingIntegrator`. In the middle is the result of the usual approach of projecting the incident radiance function for the point light into SH and then using it to compute illumination arriving at the sphere. The right shows the effect of implementing the technique described here.

One way to solve this problem is to prefilter (i.e., blur) the signal before projection into SH, so that it doesn't have any excessively high-frequency information in it. A simpler approach, which we will implement here, was suggested by Sloan (2008). His observation was that we don't necessarily want to strictly minimize the least-squares error of the projection of a function into the SH basis, but that it's also worthwhile to have an error metric that penalizes oscillations in the result.

DirectLightingIntegrator 742
Point 63
Renderer 24
RNG 1003
Sample 343
Scene 22
SHReduceRinging() 951
Spectrum 263

Sloan derived an approach to minimize the Laplacian of the reconstructed function, which reduces these oscillations. The result gives a scale factor for each coefficient that depends on a scaling factor λ and the coefficient's band l:

$$\tilde{c}_l^m = \frac{c_l^m}{1 + \lambda l^2 (l+1)^2}. \tag{17.13}$$

Note that if $\lambda = 0$ then this is the same as regular least-squares projection. Small values of λ generally work well; the default value of .005 was used in Figure 17.6(c). See Sloan's report for details on the derivation as well as more discussion about how to compute λ values.

The `SHReduceRinging()` function applies Equation (17.13) to a given set of SH coefficients.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHReduceRinging(Spectrum *c, int lmax, float lambda) {
    for (int l = 0; l <= lmax; ++l) {
        float scale = 1.f / (1.f + lambda * l * l * (l + 1) * (l + 1));
        for (int m = -l; m <= l; ++m)
            c[SHIndex(l, m)] *= scale;
    }
}
```

17.2.5 ROTATIONS

Many precomputed light transport algorithms need to rotate functions in spherical harmonics; for example, if both illumination and a reflection function are represented in the SH basis but illumination is represented in world-space coordinates and reflection is represented in local tangent-frame coordinates, then one of the two must be rotated to be in the same coordinate system as the other if reflected light is to be computed.

A very useful property of the spherical harmonic basis for rendering is that it is *rotationally invariant*; this means that, given some function $f(\omega)$ defined on the sphere with SH coefficients c_i, if one wants to compute the new SH coefficients that result from rotating the original function on the sphere, they can be computed directly from the original coefficients c_i and a SH rotation matrix $\mathbf{S}$:

$$c_i' = \mathbf{S} c_i.$$

Using a SH rotation matrix in this manner gives the same result as if the original function had been rotated and then projected into the SH basis. More specifically, no error is introduced by rotating with the coefficients directly versus rotating the original function and reprojecting into spherical harmonics.

The fact that coefficients for the rotated function can be computed directly from the coefficients of the original function is an extremely useful one for rendering for two reasons. First, computing the new coefficients c_i' from the original ones is generally much more efficient than rotating the original function and then re-computing the coefficients. Second, because no error is introduced in the rotation, visual artifacts aren't introduced in animations where objects are moving or rotating and SH functions involved in rendering them need to be rotated.

SHIndex() 935
Spectrum 263

Although there are closed-form expressions for the SH rotation matrices **S** that correspond to a given 3×3 coordinate-system rotation matrix **R**, these matrices are dense, complex, and computationally inefficient.[4] More efficient implementations are possible by decomposing the given rotation matrix into Euler angles and successively applying matrices that rotate the SH coefficients around individual coordinate axes. This section will develop this approach.

The basic setting is that we want to find new coefficients c_i' given the rotation matrix **R** and the original coefficients c_i. One way to derive this transformation is to use c_i to reconstruct the function in 3D, apply the 3×3 rotation matrix to rotate the function, and then project the resulting function back into SH. This is perhaps best understood with the $l = 1$ case. (When rotating functions in the SH basis, coefficients within any given l band don't affect coefficients in other bands—this can be shown from orthonormality. Therefore, we can consider the rotation of each band independently.)

We have coefficients c_1^{-1}, c_1^0, and c_1^1; the unrotated function in the SH basis is

$$\hat{f}(x, y, z) = c_1^{-1} Y_1^{-1}(x, y, z) + c_1^0 Y_1^0(x, y, z) + c_1^1 Y_1^1(x, y, z).$$

Using the Cartesian form of the basis functions in Equation (17.7), we have

$$\hat{f}(x, y, z) = c_1^{-1}\left(-\sqrt{\frac{3}{4\pi}}y\right) + c_1^0\left(-\sqrt{\frac{3}{4\pi}}z\right) + c_1^1\left(-\sqrt{\frac{3}{4\pi}}x\right).$$

We can rotate this function with the 3×3 rotation matrix **R**. Define the rotated coordinates as usual by

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} r_{0,0} & r_{0,1} & r_{0,2} \\ r_{1,0} & r_{1,1} & r_{1,2} \\ r_{2,0} & r_{2,1} & r_{2,2} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix},$$

and then the rotated version of $\hat{f}$ can be written in terms of x', y', and z' by subsituting $x' = r_{0,0}x + r_{0,1}y + r_{0,2}z$, and so forth. We can now reproject the rotated version of $\hat{f}$ into the SH basis. Consider, for example, how to compute the new coefficient $\tilde{c}_1^{-1}$:

$$\tilde{c}_1^{-1} = \int_{S^2} \hat{f}(x', y', z')\, Y_1^{-1}(x, y, z)\, \mathrm{d}\omega.$$

If we expand $\hat{f}$ and work through the integral, the result turns out to be a linear combination of original coefficients with entries from the original rotation matrix:

$$\begin{aligned} \tilde{c}_1^{-1} &= r_{1,1}c_1^{-1} + -r_{2,1}c_1^0 + r_{0,1}c_1^1 \\ \tilde{c}_1^0 &= -r_{1,2}c_1^{-1} + -r_{2,2}c_1^0 - r_{0,2}c_1^1 \\ \tilde{c}_1^1 &= r_{1,0}c_1^{-1} + -r_{2,0}c_1^0 + r_{0,0}c_1^1. \end{aligned}$$

This rotation can equivalently be expressed in the following 3×3 SH rotation matrix. (The SH rotation matrix for the $l = 1$ case happens to be 3×3, but recall that higher SH

4 In this section, we will use R to denote 3×3 coordinate system matrices and S to denote SH rotation matrices, which are in general of dimension $(l_{\max} + 1) \times (l_{\max} + 1)$.

bands will have larger rotation matrices.)

$$\begin{pmatrix} \tilde{c}_1^{-1} \\ \tilde{c}_1^{0} \\ \tilde{c}_1^{1} \end{pmatrix} = \begin{pmatrix} r_{11} & -r_{21} & r_{01} \\ -r_{12} & r_{22} & -r_{02} \\ r_{10} & -r_{20} & r_{00} \end{pmatrix} \begin{pmatrix} c_1^{-1} \\ c_1^{0} \\ c_1^{1} \end{pmatrix}.$$

While this general approach works well for low-order SH (up to $l = 2$ or so), higher order bands lead to dense matrices with computationally complex individual entries—much worse than the simple result in the $l = 1$ case. Therefore, we will instead use an approach that builds arbitrary rotations through a series of individual rotations around coordinate axes.

SH Rotations via Euler Decomposition

All 3×3 rotation matrices can be equivalently represented as three rotations about coordinate axes (where any pair of successive rotations is about different axes). An efficient approach to spherical harmonic rotation can be developed by decomposing arbitrary rotation matrices into three Euler angles, representing rotations about the z, y, and then again z axis. Specifically, given a rotation matrix $\mathbf{R}$, we can write it in terms of rotations about angles α, β, and γ such that

$$\mathbf{R} = \mathbf{R}_z(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_z(\gamma). \quad [17.14]$$

The rotation about the y axis, R_y, can further be expressed as a rotation about the x axis by -90 degrees, a rotation about the z axis, and a rotation about the x axis by 90 degrees:

$$\mathbf{R}_y(\beta) = \mathbf{R}_x(-90)\mathbf{R}_z(\beta)\mathbf{R}_x(90).$$

Expressing R_y in this manner makes it possible for the implementation below to only need to handle arbitrary rotations about z. The final rotation is thus given by

$$\mathbf{R} = \mathbf{R}_z(\alpha)\mathbf{R}_x(-90)\mathbf{R}_z(\beta)\mathbf{R}_x(90)\mathbf{R}_z(\gamma).$$

The `SHRotate()` function takes a 3×3 rotation matrix; performs an Euler angle decomposition to find the angles α, β, and γ; and then performs the corresponding set of rotations to the given SH coefficients.

⟨*Spherical Harmonics Definitions*⟩ +≡

```
void SHRotate(const Spectrum *c_in, Spectrum *c_out, const Matrix4x4 &m,
              int lmax, MemoryArena &arena) {
    float alpha, beta, gamma;
    toZYZ(m, &alpha, &beta, &gamma);
    Spectrum *work = arena.Alloc<Spectrum>(SHTerms(lmax));
    SHRotateZ(c_in, c_out, gamma, lmax);
    SHRotateXPlus(c_out, work, lmax);
    SHRotateZ(work, c_out, beta, lmax);
    SHRotateXMinus(c_out, work, lmax);
    SHRotateZ(work, c_out, alpha, lmax);
}
```

Matrix4x4 1021
MemoryArena 1015
MemoryArena::Alloc() 1016
SHRotateXMinus() 956
SHRotateXPlus() 956
SHRotateZ() 955
SHTerms() 935
Spectrum 263
toZYZ() 953

We won't include `toZYZ()` here; it is based on Shoemake's implementation (1994b). The basic idea is that the angles for the Euler decomposition of a given matrix can be found using the values of the elements of the matrix. If we expand out the product of the three

rotation matrices from Equation (17.14), we can find expressions for each of the entries of $\mathbf{R}$. For example, $r_{0,2} = \cos\alpha \sin\beta$, $r_{2,2} = \cos\beta$, and so forth. These expressions can in turn be used to solve for the angles.

In order to be able to implement `SHRotateZ()`, we next need to find the matrices that rotate SH coefficients a given angle about the z axis. Thanks to the particular form of the SH functions, rotating a function represented in the spherical harmonic basis about the z axis is particularly straightforward. Given a particular rotation angle α about z, we can find the rotated function by taking the original coefficients for the function $\hat{f}$ and rotating the basis functions by α, using the (θ, ϕ) form of the SH functions. The rotated function is:

$$\hat{f}(\theta, \phi + \alpha) = \sum_{l}^{l_{\max}} \sum_{m=-l}^{l} c_l^m Y_l^m(\theta, \phi + \alpha).$$

Given the rotated function, we can project it back into the SH basis, integrating with the basis functions to find the SH coefficients of the rotated function,

$$\tilde{c}_l^m = \sum_{l}^{l_{\max}} \sum_{m'=-l}^{l} \int_0^{2\pi} \int_0^{\pi} (c_l^{m'} Y_l^{m'}(\theta, \phi + \alpha))\, Y_l^m(\theta, \phi)\, \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi.$$

We can ignore all bands $l' \neq l$ when computing the coefficients $\tilde{c}_l^m$ since the coefficients from any l band don't affect other bands' coefficients under rotation, as mentioned earlier. We can also pull the coefficients $c_l^{m'}$ out of the integrand, giving:

$$\tilde{c}_l^m = \sum_{m'=-l}^{l} c_l^{m'} \int_0^{2\pi} \int_0^{\pi} Y_l^{m'}(\theta, \phi + \alpha)\, Y_l^m(\theta, \phi)\, \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi.$$

The values of these integrals can be found ahead of time, allowing us to express the rotation as a matrix multiplication.

For the z axis rotation case, these integrals of the products $Y_l^m(\theta, \phi) Y_l^{m'}(\theta, \phi + \alpha)$ have a particularly simple form. Again, for $l = 1$, working out the integrals gives

$$\begin{pmatrix} \tilde{c}_1^{-1} \\ \tilde{c}_1^0 \\ \tilde{c}_1^1 \end{pmatrix} = \begin{pmatrix} \cos -\alpha & 0 & \sin -\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{pmatrix} \begin{pmatrix} c_1^{-1} \\ c_1^0 \\ c_1^1 \end{pmatrix}.$$

As l increases, the matrix takes on a regular form. For $l = 2$, it is:

$$\begin{pmatrix} \cos -2\alpha & 0 & 0 & 0 & \sin -2\alpha \\ 0 & \cos -\alpha & 0 & \sin -\alpha & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & \sin\alpha & 0 & \cos\alpha & 0 \\ \sin 2\alpha & 0 & 0 & 0 & \cos 2\alpha \end{pmatrix},$$

SHRotateZ() 955

and so forth. For arbitrary l, the rotation matrix elements $s_{i,j}$ are:

$$s_{i,j} = \begin{cases} \cos((j-l)\alpha) & i = j, i \neq l, j \neq l \\ \sin((l-j)\alpha) & i + j = 2l, i \neq l, j \neq l \\ 1 & i = j = l \\ 0 & \text{otherwise.} \end{cases} \quad (17.15)$$

The SHRotateZ() implements this rotation for Spectrum-valued spherical harmonic coefficients. It rotates all of the bands by the angle α, up to the given $l_{\max}$ band, storing the results in the memory pointed to by c_out. Note that the first coefficient, representing the constant $l = 0$ term, is unchanged by rotation.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHRotateZ(const Spectrum *c_in, Spectrum *c_out, float alpha,
               int lmax) {
    c_out[0] = c_in[0];
    if (lmax == 0) return;
    ⟨Precompute sine and cosine terms for z-axis SH rotation 955⟩
    for (int l = 1; l <= lmax; ++l) {
        ⟨Rotate coefficients for band l about z 955⟩
    }
}
```

This function first computes all of the sine and cosine values needed for all of the SH bands up to $l_{\max}$ using the efficient sinCosIndexed() function.

⟨*Precompute sine and cosine terms for z-axis SH rotation*⟩ ≡ 955
```
float *ct = ALLOCA(float, lmax+1);
float *st = ALLOCA(float, lmax+1);
sinCosIndexed(sinf(alpha), cosf(alpha), lmax+1, st, ct);
```

The rotations are then performed by directly implementing the matrix/vector multiplication from the matrices given by Equation (17.15). Generating the complete rotation matrix and using a general matrix/vector multiplication routine would be substantially less efficient, given the sparsity of the full rotation matrices.

⟨*Rotate coefficients for band* l *about z*⟩ ≡ 955
```
for (int m = -l; m < 0; ++m)
    c_out[SHIndex(l, m)] =
        ( ct[-m] * c_in[SHIndex(l,  m)] +
         -st[-m] * c_in[SHIndex(l, -m)]);
c_out[SHIndex(l, 0)] = c_in[SHIndex(l, 0)];
for (int m = 1; m <= l; ++m)
    c_out[SHIndex(l, m)] =
        (ct[m] * c_in[SHIndex(l,  m)] +
         st[m] * c_in[SHIndex(l, -m)]);
```

ALLOCA() 1009
SHIndex() 935
sinCosIndexed() 940
Spectrum 263

The last step is to find the x axis rotations by ± 90 degrees in order to implement SHRotateXPlus() and SHRotateXMinus(). These also turn out to be relatively simple matrices. For example, following the same approach working through the integrals of products of SH basis functions, the SH rotation matrix for rotating by 90 degrees about

the x axis in the $l = 2$ case is given by

$$\begin{pmatrix} \tilde{c}_2^{-2} \\ \tilde{c}_2^{-1} \\ \tilde{c}_2^{0} \\ \tilde{c}_2^{1} \\ \tilde{c}_2^{2} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1/2 & 0 & -\sqrt{3}/2 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{3}/2 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} c_2^{-2} \\ c_2^{-1} \\ c_2^{0} \\ c_2^{1} \\ c_2^{2} \end{pmatrix}.$$

Similar to the z rotation case, rotating a given set of coefficients with this matrix can be implemented extremely efficiently because many of the entries are zero, one, or negative one. The terms with square roots can be computed ahead of time, leading to an implementation requiring just a few multiplies and adds.

The rotation matrices for other l bands as well as the rotation matrices for -90 degree rotations around the x axis are similarly sparse; we won't include them here. The routines `SHRotateXPlus()` and `SHRotateXMinus()` implement these rotations.

⟨*Spherical Harmonics Declarations*⟩ +≡

```
void SHRotateXMinus(const Spectrum *c_in, Spectrum *c_out, int lmax);
void SHRotateXPlus(const Spectrum *c_in, Spectrum *c_out, int lmax);
```

17.3 RADIANCE PROBES

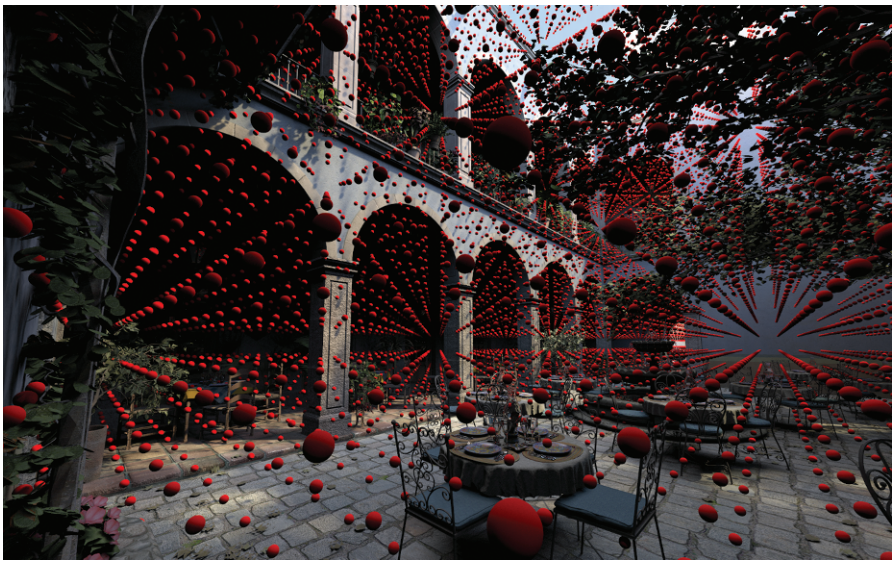
It's often possible to partition the geometry of complex environments into two subsets: static geometry with fixed position and material properties and dynamic geometry with arbitrary positions, materials, and shapes. (Consider, for example, a game level with fixed buildings and vegetation and characters moving through it.) For these sorts of scenes, one approach for interactive rendering is to precompute a representation of directionally varying incident radiance at a set of points in the scene, taking into account light interreflection with the static geometry. When the scene is later rendered, these sampled precomputed radiance functions are used to approximate incident illumination. In this section, we will refer to the technique as *radiance probes*, though it's sometimes also described as *volumetric lighting* or *irradiance volumes*. Figure 17.7 shows a scene rendered with this technique.

There are a few obvious sources of error from this approach. First, incident radiance computed at one point may not be an accurate approximation to incident radiance at other nearby points. (Consider, for example, the difference between the incident radiance function near a wall versus the incident radiance function at a point inside the wall.) Second, the effect of the dynamic objects in the scene isn't represented in the radiance functions used for shading; for example, interreflection between the static objects and the dynamic objects is not accounted for. In spite of these shortcomings, this approach has seen wide use for interactive rendering, as it allows the incorporation of global lighting effects with relatively little computational cost.

17.3.1 CREATING RADIANCE PROBES

Spectrum 263

One advantage of using a physically based renderer like `pbrt` for creating radiance probes is that global illumination algorithms can be used to compute the distribu-



(a)

(b)

Figure 17.7: Rendering with Radiance Probes. (a) San Miguel scene (rendered with a standard direct lighting calculation), showing points at which radiance probes were computed with red spheres. (b) Scene rendered using the radiance probes to approximate the incident illumination. Global lighting effects like indirect lighting from the walls are captured by the radiance probes and low-frequency shadows like those under the tables are visible. However, more detailed illumination effects can not be accurately represented with this sampling density. *(Model courtesy of Guillermo M. Leal Llaguno.)*

tion of illumination in the scene. We will first describe the implementation of the CreateRadianceProbes Renderer, which computes incident radiance and projects it into the spherical harmonics at a grid of points in the scene. It is defined in the files renderers/createprobes.h and renderers/createprobes.cpp. The computed SH coefficients include the effect of *all* of the lights in the scene; thus, there is no incremental performance cost for a large number of lights at final rendering time. Of course, a limited number of SH coefficients can represent only a limited amount of lighting complexity. However, for low-frequency (i.e., diffuse or only slightly glossy) BRDFs, the error from this approximation is often acceptable.

This renderer then writes the SH coefficients out to a file. CreateRadianceProbes is thus a renderer that doesn't create an image as its output but instead computes a series of measurements of the scene.

Section 17.3.2 demonstrates one method for using radiance probes for rendering with the UseRadianceProbes integrator. In general, the greatest benefit from radiance probes comes from their use in an interactive renderer rather than an offline renderer like pbrt; however, here we show their use in pbrt in the interests of simplicity.

The parameters to the CreateRadianceProbes constructor manage the radiance probe creation process. Radiance probes are created with no more than probeSpacing distance between samples in each dimension, in a regular grid with extent given by the provided bounding box. (If an explicit bounding box isn't provided by the parameters in the scene description file, then the scene bounds are used for the bounding box.) The incident radiance functions are projected into spherical harmonics using lmax SH bands and stored in the given file. The camera and integrators from the scene description file are passed to the constructor, and the number of Monte Carlo samples to use when estimating SH coefficients of indirect illumination is given by nIndirSamples.

⟨*CreateRadianceProbes Public Methods*⟩ ≡
```
CreateRadianceProbes(SurfaceIntegrator *surf, VolumeIntegrator *vol,
    const Camera *camera, int lmax, float probeSpacing, const BBox &bbox,
    int nIndirSamples, bool includeDirect, bool includeIndirect,
    float time, const string &filename);
```

The implementation of the constructor is straightforward and isn't included here—the member variables here are set directly from the provided parameters.

⟨*CreateRadianceProbes Private Data*⟩ ≡
```
SurfaceIntegrator *surfaceIntegrator;
VolumeIntegrator *volumeIntegrator;
const Camera *camera;
int lmax, nIndirSamples;
BBox bbox;
bool includeDirectInProbes, includeIndirectInProbes;
float time, probeSpacing;
string filename;
```

BBox 70
Camera 302
CreateRadianceProbes 958
Renderer 24
SurfaceIntegrator 740
UseRadianceProbes 965
VolumeIntegrator 876

As an implementation of the Renderer abstract base class, CreateRadianceProbes must implement the Li() and Transmittance() methods. Their implementations, not in-

cluded here, are essentially the same as the corresponding SamplerRenderer methods defined in Section 1.3.4, with the surfaceIntegrator and volumeIntegrator used to compute the returned values. These methods of CreateRadianceProbes will be called by the SHProjectIncidentIndirectRadiance() function, which is used computing the SH coefficients for the probes below.

After some general preparation, the bulk of the work done by the Render() method is farmed out to tasks that run in parallel; one task is launched for each of the points at which a radiance probe is to be computed. After the tasks have all finished, the implementation here writes the SH coefficients of the probes to a file.

⟨*CreateRadianceProbes Method Definitions*⟩ ≡
```
void CreateRadianceProbes::Render(const Scene *scene) {
    ⟨Compute scene bounds and initialize probe integrators 959⟩
    ⟨Compute sampling rate in each dimension 959⟩
    ⟨Allocate SH coefficient vector pointers for sample points 960⟩
    ⟨Compute random points on surfaces of scene 962⟩
    ⟨Launch tasks to compute radiance probes at sample points⟩
    ⟨Write radiance probe coefficients to file⟩
}
```

If a bounding box for the radiance probes isn't provided in the scene description file, the BBox is initialized to a degenerate bounding box, which has the respective minimum position components greater than the maximum components. In that case, the bounding box for the probes is initialized here from the scene bounding box. Next, the integrators have a chance to do precomputation now that the Scene is available.

⟨*Compute scene bounds and initialize probe integrators*⟩ ≡ 959
```
if (bbox.pMin.x > bbox.pMax.x)
    bbox = scene->WorldBound();
surfaceIntegrator->Preprocess(scene, camera, this);
volumeIntegrator->Preprocess(scene, camera, this);
Sample *origSample = new Sample(NULL, surfaceIntegrator, volumeIntegrator,
                                scene);
```

BBox 70
BBox::pMax 71
BBox::pMin 71
Ceil2Int() 1002
CreateRadianceProbes::probeSpacing 958
CreateRadianceProbes::surfaceIntegrator 958
CreateRadianceProbes::volumeIntegrator 958
Integrator::Preprocess() 740
Sample 343
SamplerRenderer 25
Scene 22
Scene::WorldBound() 24
SHProjectIncidentIndirectRadiance() 950
Vector 57

Now that the bounding box is known, it is possible to determine how many probes to compute. The number of probes in each dimension is set so that the probes are no farther apart than the given probeSpacing.

⟨*Compute sampling rate in each dimension*⟩ ≡ 959
```
Vector delta = bbox.pMax - bbox.pMin;
int nProbes[3];
for (int i = 0; i < 3; ++i)
    nProbes[i] = max(1, Ceil2Int(delta[i] / probeSpacing));
```

Given the number of probes, it's now possible to allocate space for the SH coefficients. The tasks will initialize these values for each probe.

(a) (b)

Figure 17.8: Artifacts From Bad Placement of Radiance Probes. (a) If the incident radiance functions for radiance probes are computed at the centers of their respective grid cells, then some probes will often end up being completely inside closed objects and will be completely black. (b) When these probes are used for rendering, characteristic dark splotches appear when those probes are used to shade points outside the closed objects. When the alternative probe placement approach implemented in this section is used, we get the better result shown in Figure 17.7(b). *(Model courtesy of Guillermo M. Leal Llaguno.)*

⟨*Allocate SH coefficient vector pointers for sample points*⟩ ≡ 959

```
int count = nProbes[0] * nProbes[1] * nProbes[2];
Spectrum **c_in = new Spectrum *[count];
for (int i = 0; i < count; ++i)
    c_in[i] = new Spectrum[SHTerms(lmax)];
```

The most straightforward approach for computing each radiance probe would be to compute the incident radiance function at a single point at the center of the probe's grid cell, as shown in Figure 17.8(a). However, as that figure illustrates, this approach often suffers from sampling errors, since some probes would end up located inside solid objects in the scene. The incident radiance functions for these points would be completely black, and artifacts would occur during rendering when the radiance functions at these probes were used for shading objects that are actually outside the solid objects. Figure 17.8(b) shows these artifacts with the scene from Figure 17.7(b) rendered with the probes placed at the center of each grid cell.

To avoid this issue, `pbrt` implements an approach inspired by Kontkanen and Laine (2006), who investigated this problem but came up with a somewhat different solution than the one implemented here. For each radiance probe to be computed, we first compute the extent of the grid cell of the overall scene bounding box that it represents. We then select a sequence of points inside the cell; for each one, we determine whether it is *indirectly visible* from the camera position. (This is the only requirement for the camera position for this renderer; we just require that it be in the same part of the scene where

Spectrum 263

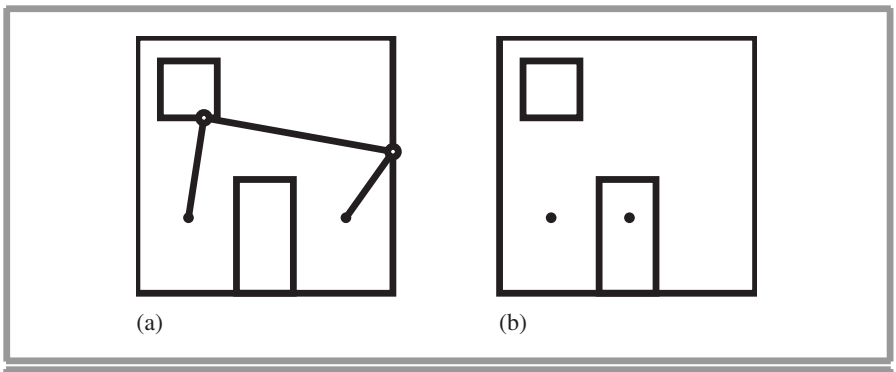


Figure 17.9: Indirect Visibility Between Two Points. (a) Two points (filled circles) are defined to be indirectly visible to each other if there is a path of one or more unobstructed ray segments between them. Here, the vertices of the path on other surfaces of the scene are shown with open circles. (b) These two points aren't indirectly visible to each other since there is no path between them that doesn't pass through the walls of the box that encloses the one on the right.

final rendering will be done.) We define two points as being indirectly visible if there is a path with zero or more vertices on scene surfaces between them; see Figure 17.9. The points in grid cells that are indirectly visible to the camera are assumed to thus not be inside scene objects and can be used to compute the incident radiance function.

This approach doesn't eliminate all sampling errors—see, for example, the case illustrated in the upper right corner of Figure 17.8(a) where a closed piece of geometry splits a cell in half. In this case, the incident radiance functions on the two sides of it are likely be very different. A single radiance probe can't represent this case accurately; using adaptive refinement rather than the regular grid here can reduce the effect of this problem.

In order to compute indirect visibility from the camera position while computing probes, a short precomputation is performed before the tasks are launched. This computation finds a number of points on surfaces in the scene by tracing random paths starting from the camera position; these points are stored in the `surfacePoints` array. If any one of these points has an unobstructed path between it and a candidate radiance probe point, then we know by construction that there is an indirect visibility path from the camera to the point and that the point is a safe one at which to compute a radiance probe (Figure 17.10). Note that this method for computing indirect visibility isn't perfect: there may be some points that are in fact indirectly visible but that are not found by this sampling algorithm. However, we have found this approach to work reasonably well in practice.

The implementation of the fragment ⟨*Generate random path from camera and deposit surface points*⟩ is very similar to the fragment ⟨*Follow ray through scene and attempt to deposit candidate sample points*⟩ defined in Section 16.5.1 (though it doesn't check to make sure that surfaces have BSSRDFs before storing points and doesn't try to enforce a Poisson sphere distribution of points). Therefore, this fragment is not included here.

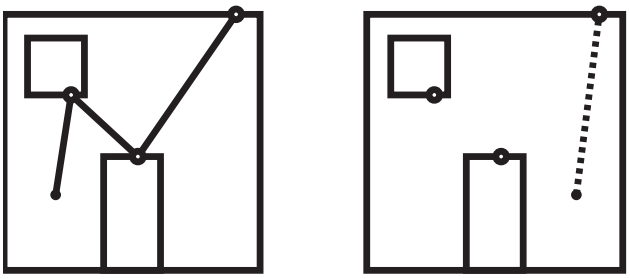


Figure 17.10: Efficiently Computing Indirect Visibility. (a) In a preprocessing step, random paths are followed from the camera (filled circle), and intersection points on scene surfaces (open circles) are recorded in an array. (b) Given a candidate probe location (filled circle), we can efficiently test to see if it is indirectly visible to the camera by tracing shadow rays between it and all of the stored intersection points. If any of the rays is not occluded, the point and the camera are indirectly visible.

⟨*Compute random points on surfaces of scene*⟩ ≡ 959
```
    ⟨Create scene bounding sphere to catch rays that leave the scene 891⟩
    vector<Point> surfacePoints;
    uint32_t nPoints = 32768, maxDepth = 32;
    surfacePoints.reserve(nPoints + maxDepth);
    Point pCamera = camera->CameraToWorld(camera->shutterOpen,
                                          Point(0, 0, 0));
    surfacePoints.push_back(pCamera);
    RNG rng;
    while (surfacePoints.size() < nPoints) {
        ⟨Generate random path from camera and deposit surface points⟩
    }
```

After these points are found, tasks are launched to compute the radiance probes. One task is launched for each probe by the fragment ⟨*Launch tasks to compute radiance probes at sample points*⟩, which we won't include in the text here. After the tasks complete, the fragment ⟨*Write radiance probe coefficients to file*⟩ (also not included here) writes a text file that stores the overall bounding box, the number of SH bands, and the SH coefficients for each probe in turn. This file can be read by the `UseRadianceProbes Integrator`, defined shortly.

The task for each probe first computes the bounding box of the subregion of the scene for which it is responsible, initializes a few common variables, and then selects a number of candidate points within the bounding box of its region. For all of the points that are indirectly visible from the given camera position, it samples the incident radiance function at the point and projects it into SH coefficients. In the end, it returns the average SH coefficients for the radiance function for all accepted candidate points. Because the SH functions used here are a linear basis, averaging their coefficients for the probe points used gives the average of the projected incident radiance functions at the points. No more

Camera::CameraToWorld 302
CreateRadianceProbes::camera 958
Integrator 740
Point 63
RNG 1003
UseRadianceProbes 965

than 256 points within the cell are tested, but once 32 points that are indirectly visible from the camera are found the task exits.

⟨*CreateRadianceProbes Method Definitions*⟩ +≡
```
void CreateRadProbeTask::Run() {
    ⟨Compute region in which to compute incident radiance probes 963⟩
    ⟨Initialize common variables for CreateRadProbeTask::Run() 963⟩
    for (int i = 0; i < 256; ++i) {
        if (nFound == 32) break;
        ⟨Try to compute radiance probe contribution at ith sample point 963⟩
    }
    ⟨Compute final average value for probe and cleanup 965⟩
}
```

The bounding box for the cell that corresponds to this probe is found by first computing the integer (sx, sy, sz) coordinates for the cell with respect to the overall probe sampling rate and then linearly interpolating from the overall bounding box corners.

⟨*Compute region in which to compute incident radiance probes*⟩ ≡ 963
```
int sx = pointNum % nProbes[0];
int sy = (pointNum / nProbes[0]) % nProbes[1];
int sz = pointNum / (nProbes[0] * nProbes[1]);
float tx0 = float(sx) / nProbes[0], tx1 = float(sx+1) / nProbes[0];
float ty0 = float(sy) / nProbes[1], ty1 = float(sy+1) / nProbes[1];
float tz0 = float(sz) / nProbes[2], tz1 = float(sz+1) / nProbes[2];
BBox b(bbox.Lerp(tx0, ty0, tz0), bbox.Lerp(tx1, ty1, tz1));
```

A number of variables will be needed in the following code. First is a random number generator, uniquely seeded based on the radiance probe we're generating. The c_probe variable stores the SH coefficients computed at a single point before they're added to the average coefficients for all of the probes in this cell, and the MemoryArena is used by the incident radiance projection routines. nFound tracks the number of found probe points that are indirectly visible to the camera, and lastVisibleOffset, described further below, is used to accelerate the indirect visibility tests.

⟨*Initialize common variables for* CreateRadProbeTask::Run()⟩ ≡ 963
```
RNG rng(pointNum);
Spectrum *c_probe = new Spectrum[SHTerms(lmax)];
MemoryArena arena;
uint32_t nFound = 0, lastVisibleOffset = 0;
```

After computing the position of each candidate sample point within the probe's bounds, the code checks to see if the point is indirectly visible from the given camera point; if so, it computes the SH coefficients for the incident radiance function at the point.

⟨*Try to compute radiance probe contribution at* ith *sample point*⟩ ≡ 963
```
⟨Compute ith candidate point p in cell's bounding box 964⟩
⟨Skip point p if not indirectly visible from camera 964⟩
⟨Compute SH coefficients of incident radiance at point p 964⟩
```

BBox 70
BBox::Lerp() 73
MemoryArena 1015
RNG 1003
SHTerms() 935
Spectrum 263

To choose candidate points, points from a 3D Halton sequence (Section 7.4.2) are used to interpolate between the corners of the cell's bounding box.

⟨*Compute* i*th candidate point* p *in cell's bounding box*⟩ ≡ 963
```
float dx = RadicalInverse(i+1, 2);
float dy = RadicalInverse(i+1, 3);
float dz = RadicalInverse(i+1, 5);
Point p = b.Lerp(dx, dy, dz);
```

For each candidate point p, we need to see if it's indirectly visible from the camera: if any of the points in the surfacePoints array and the candidate point are mutually visible, then the candidate point can be used. The implementation here keeps track of the offset to the last point from surfacePoints that had an unoccluded path to a previous candidate point in this probe's grid cell in lastVisibleOffset. It then tests visibility between that point and the next candidate point first. It's often the case that this point will also have an unoccluded path to the next candidate point, so testing it first can substantially speed up this part of the computation.

⟨*Skip point* p *if not indirectly visible from camera*⟩ ≡ 963
```
if (scene->IntersectP(Ray(surfacePoints[lastVisibleOffset],
                          p - surfacePoints[lastVisibleOffset],
                          1e-4f, 1.f, time))) {
    uint32_t j;
    ⟨See if point is visible to any element of surfacePoints 964⟩
    if (j == surfacePoints.size())
        continue;
}
++nFound;
```

⟨*See if point is visible to any element of* surfacePoints⟩ ≡ 964
```
for (j = 0; j < surfacePoints.size(); ++j)
    if (!scene->IntersectP(Ray(surfacePoints[j], p - surfacePoints[j],
                               1e-4f, 1.f, time))) {
        lastVisibleOffset = j;
        break;
    }
```

Given a valid point inside the probe's grid cell, the utility functions defined in Section 17.2.3 are used to compute SH coefficients for the incident radiance function at the point, accumulating the results into the c_in array.

⟨*Compute SH coefficients of incident radiance at point* p⟩ ≡ 963
```
if (includeDirectInProbes) {
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_probe[i] = 0.f;
    SHProjectIncidentDirectRadiance(p, 0.f, time, arena, scene,
                                    true, lmax, rng, c_probe);
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_in[i] += c_probe[i];
}
```

BBox::Lerp() 73
MemoryArena::FreeAll() 1017
Point 63
RadicalInverse() 362
Ray 66
Scene::IntersectP() 24
SHProjectIncidentDirectRadiance() 949
SHProjectIncidentIndirectRadiance() 950
SHTerms() 935

```
if (includeIndirectInProbes) {
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_probe[i] = 0.f;
    SHProjectIncidentIndirectRadiance(p, 0.f, time, renderer,
        origSample, scene, lmax, rng, nIndirSamples, c_probe);
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_in[i] += c_probe[i];
}
arena.FreeAll();
```

The final coefficient values in `c_in` for this cell are found by dividing the summed coefficient values by the number of points at which probes were computed.

⟨*Compute final average value for probe and cleanup*⟩ ≡ 963

```
if (nFound > 0)
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_in[i] /= nFound;
delete[] c_probe;
```

17.3.2 USING RADIANCE PROBES

The `UseRadianceProbes` `SurfaceIntegrator` reads in a text file of radiance probes generated by the `CreateRadianceProbes` `Renderer` and uses the probes to illuminate the scene. Because the lighting computations with radiance probes can be performed extremely efficiently and because looking up radiance probes stored in a regular grid is straightforward, the corresponding lighting calculations can reasonably be performed in interactive rendering systems.

The `UseRadianceProbes` constructor (not included here) reads in the values of its member variables from the given radiance probe file.

⟨*UseRadianceProbes Private Data*⟩ ≡

```
BBox bbox;
int lmax, includeDirectInProbes, includeIndirectInProbes;
int nProbes[3];
Spectrum *c_in;
```

BBox 70
CreateRadianceProbes 958
Renderer 24
SHTerms() 935
Spectrum 263
SurfaceIntegrator 740
UniformSampleAllLights() 745

The structure of the `UseRadianceProbes::Li()` method is similar to that of most other `SurfaceIntegrators`. Here, we'll only include the fragments that handle illumination from radiance probes. If the probes only include indirect lighting, then the implementation performs the regular direct lighting calculation with `UniformSampleAllLights()`. (Interactive renderers that use radiance probes often use conventional methods like shadow maps to compute direct lighting and only use the radiance probes for indirect illumination. Or, they may include the contribution of some direct lights in the radiance probes and compute the effect of others explicitly.)

⟨*Compute reflection for radiance probes integrator*⟩ ≡
```
if (!includeDirectInProbes)
    L += UniformSampleAllLights(scene, renderer, arena, p, n,
            wo, isect.rayEpsilon, ray.time, bsdf, sample, rng,
            lightSampleOffsets, bsdfSampleOffsets);
```
⟨*Compute reflected lighting using radiance probes* 966⟩

To use the radiance probes for lighting calculations, the first step is to look up the probes around the point being shaded and to trilinearly interpolate their coefficients to get coefficients that approximate the incident radiance function at the lookup point. In this section, we'll implement an efficient method that computes the incident irradiance from this interpolated incident radiance function and then uses it to shade the surface, approximating its full BSDF with a diffuse term.

⟨*Compute reflected lighting using radiance probes*⟩ ≡ 966
⟨*Compute probe coordinates and offsets for lookup point* 966⟩
⟨*Get radiance probe coefficients around lookup point* 966⟩
⟨*Compute incident radiance from radiance probe coefficients* 967⟩
⟨*Convolve incident radiance to compute irradiance function* 968⟩
⟨*Evaluate irradiance function and accumulate reflection* 969⟩

The integer coordinates of the lookup point `p` are found using the same calculation as is used to convert from continuous to discrete pixel coordinates (recall the discussion in Section 7.1.7). Given the discrete coordinates, floating-point offsets `dx`, `dy`, and `dz` are found for trilinear interpolation.

⟨*Compute probe coordinates and offsets for lookup point*⟩ ≡ 966
```
Vector offset = bbox.Offset(p);
float voxx = (offset.x * nProbes[0]) - 0.5f;
float voxy = (offset.y * nProbes[1]) - 0.5f;
float voxz = (offset.z * nProbes[2]) - 0.5f;
int vx = Floor2Int(voxx), vy = Floor2Int(voxy), vz = Floor2Int(voxz);
float dx = voxx - vx, dy = voxy - vy, dz = voxz - vz;
```

The `c_inXYZ()` utility function, not included here in the book text, clamps the given discrete coordinates to the valid range and returns a pointer to the first `Spectrum` in the array that holds the SH coefficients at the given grid cell.

⟨*Get radiance probe coefficients around lookup point*⟩ ≡ 966
```
const Spectrum *b000 = c_inXYZ(lmax, vx,   vy,   vz);
const Spectrum *b100 = c_inXYZ(lmax, vx+1, vy,   vz);
const Spectrum *b010 = c_inXYZ(lmax, vx,   vy+1, vz);
const Spectrum *b110 = c_inXYZ(lmax, vx+1, vy+1, vz);
const Spectrum *b001 = c_inXYZ(lmax, vx,   vy,   vz+1);
const Spectrum *b101 = c_inXYZ(lmax, vx+1, vy,   vz+1);
const Spectrum *b011 = c_inXYZ(lmax, vx,   vy+1, vz+1);
const Spectrum *b111 = c_inXYZ(lmax, vx+1, vy+1, vz+1);
```

BBox::Offset() 73
Floor2Int() 1002
Intersection::rayEpsilon 186
Ray::time 67
Spectrum 263
UniformSampleAllLights() 745
UseRadianceProbes::includeDirectInProbes 965
UseRadianceProbes::lmax 965
UseRadianceProbes::nProbes 965
Vector 57

Given the pointers to the eight sets of coefficients around the point, we'll now compute the trilinearly interpolated SH coefficients that represent the incident radiance at p and store them in the temporary c_inp array.

⟨*Compute incident radiance from radiance probe coefficients*⟩ ≡ 966
```
Spectrum *c_inp = arena.Alloc<Spectrum>(SHTerms(lmax));
for (int i = 0; i < SHTerms(lmax); ++i) {
    ⟨Do trilinear interpolation to compute SH coefficients at point 967⟩
}
```

⟨*Do trilinear interpolation to compute SH coefficients at point*⟩ ≡ 967
```
Spectrum c00 = Lerp(dx, b000[i], b100[i]);
Spectrum c10 = Lerp(dx, b010[i], b110[i]);
Spectrum c01 = Lerp(dx, b001[i], b101[i]);
Spectrum c11 = Lerp(dx, b011[i], b111[i]);
Spectrum c0 = Lerp(dy, c00, c10);
Spectrum c1 = Lerp(dy, c01, c11);
c_inp[i] = Lerp(dz, c0, c1);
```

Given the incident radiance function in SH coefficients, we now need to compute the reflected radiance due to illumination at the point being shaded. Here, we will show how to compute the incident irradiance function from the incident radiance function using efficient methods for convolution of functions in the spherical harmonic basis. The incident irradiance is in turn used to compute the reflected radiance under the simplification of treating all surfaces as diffuse. (Recall from Equation (15.13) that the reflected radiance from a diffuse surface is equal to the irradiance arriving at the surface times its diffuse reflectance.) This general approach is frequently used with radiance probes in interactive applications.

Ramamoorthi (2002) showed that the reflection equation, Equation (5.8), can equivalently be written as a convolution of a signal (the incident radiance function) with a filter (the cosine weighted BSDF). If both of these are represented in the SH basis, then the convolution can be performed very efficiently directly in the SH basis, similarly to how convolutions in 2D Cartesian coordinates can be performed very efficiently with the product of Fourier basis coefficients (Equation (7.3)). This approach leads to a number of new efficient algorithms for computing reflection.

Here, we would like to compute the incident irradiance function on one side of a surface, which is given as a function of position and normal by the integral

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i})(\cos\theta_\mathrm{i})\, \mathrm{d}\omega_\mathrm{i} = \int_{\mathcal{S}^2} L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i}) \max(0, \cos\theta_\mathrm{i})\, \mathrm{d}\omega_\mathrm{i}.$$

Ramamoorthi showed that, if the BRDF can be reparameterized to be a 1D function of θ, the SH coefficients of the convolution of the BRDF and the incident radiance function function have a particularly simple form. Given a function of direction $f(\omega)$ and a circularly symmetric 1D function $g(\theta)$, with respective SH coefficients f_l^m and g_l^m, then it's possible to directly compute the SH coefficients of the function that is the result of their convolution $(f * g)$,

$$(f * g)_l^m = \Lambda_l f_l^m g_l^0, \tag{17.16}$$

Lerp() 1000
MemoryArena::Alloc() 1016
SHTerms() 935
Spectrum 263

where

$$\Lambda_l = \sqrt{\frac{4\pi}{2l+1}}.$$

(Note that $g_l^m = 0$ for $m \neq 0$ thanks to the 1D symmetry condition.)

For computing irradiance, no reparameterization is necessary, since the filter function, $g(\theta) = \max(0, \cos\theta)$, is already 1D. The `SHConvolveCosTheta()` function implements the convolution of Equation (17.16) for the $\max(0, \cos\theta)$ function, returning the SH coefficients c_l^m for incident irradiance from the SH coefficients of incident radiance. The resulting irradiance function is a function of the surface normal direction:

$$E(\mathrm{p}, \mathbf{n}) \approx \sum_{l=0}^{l_{\max}} \sum_{m=-l}^{l} c_l^m Y_l^m(\mathbf{n}). \tag{17.17}$$

⟨*Convolve incident radiance to compute irradiance function*⟩ ≡ 966
```
Spectrum *c_E = arena.Alloc<Spectrum>(SHTerms(lmax));
SHConvolveCosTheta(lmax, c_inp, c_E);
```

The SH coefficients g_l^m for the projected $\max(0, \cos\theta)$ function can be computed in closed form. For computational efficiency, we have precomputed them for l up to 17; this is substantially many more terms than are necessary in practice—Ramamoorthi has shown that just the bands up to $l = 2$ can be used to represent irradiance with very low error. All coefficients for $l > 17$ are therefore clamped to zero here.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHConvolveCosTheta(int lmax, const Spectrum *c_in,
                        Spectrum *c_out) {
    static const float c_costheta[18] = { 0.8862268925, 1.0233267546,
        0.4954159260, 0.0000000000, -0.1107783690, 0.0000000000,
        0.0499271341, 0.0000000000, -0.0285469331, 0.0000000000,
        0.0185080823, 0.0000000000, -0.0129818395, 0.0000000000,
        0.0096125342, 0.0000000000, -0.0074057109, 0.0000000000 };
    for (int l = 0; l <= lmax; ++l)
        for (int m = -l; m <= l; ++m) {
            int o = SHIndex(l, m);
            if (l < 18) c_out[o] = lambda(l) * c_in[o] * c_costheta[l];
            else        c_out[o] = 0.f;
        }
}
```

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static inline float lambda(float l) {
    return sqrtf((4.f * M_PI) / (2.f * l + 1.));
}
```

Now that we have the SH coefficients of the irradiance function `c_E`, we need to compute the value of the irradiance function for the normal $\mathbf{n}$ using Equation (17.17). The diffuse reflectance ρ times the reconstructed irradiance gives the final contribution to outgoing radiance.

lambda() 968
MemoryArena::Alloc() 1016
M_PI 1002
SHConvolveCosTheta() 968
SHIndex() 935
SHTerms() 935
Spectrum 263

⟨*Evaluate irradiance function and accumulate reflection*⟩ ≡ 966

```
Spectrum rho = bsdf->rho(wo, rng, BSDF_ALL_REFLECTION);
float *Ylm = ALLOCA(float, SHTerms(lmax));
SHEvaluate(Vector(Faceforward(n, wo)), lmax, Ylm);
Spectrum E = 0.f;
for (int i = 0; i < SHTerms(lmax); ++i)
    E += c_E[i] * Ylm[i];
L += rho * INV_PI * E.Clamp();
```

It's also possible to convolve incident radiance with the (nonphysical) Phong BRDF

$$f_r(p, \omega_o, \omega_i) = \frac{n+1}{2\pi}(\omega' \cdot \omega_i)^n,$$

(where ω' is the direction of the outgoing vector ω_o reflected about the surface normal $\mathbf{n}$) to get a reflected radiance function as a function of the specular reflected direction. Though this functionality isn't currently used in `pbrt`, the `SHConvolvePhong()` function, not included in the text here, implements this computation for $g(\theta) = (\cos\theta)^n$.

17.4 PRECOMPUTED DIFFUSE TRANSFER

Precomputed radiance transfer (PRT) methods generally try to precompute information about how objects reflect and shadow light for use at rendering time, given arbitrary incident lighting distributions. For example, given a static geometric model of a tree, one might want to store a representation of how much light reaches the leaves in the interior of the tree as a function of the illumination arriving from outside of it. Ideally, this representation would encode all shadowing and inter-reflection effects that the light undergoes.

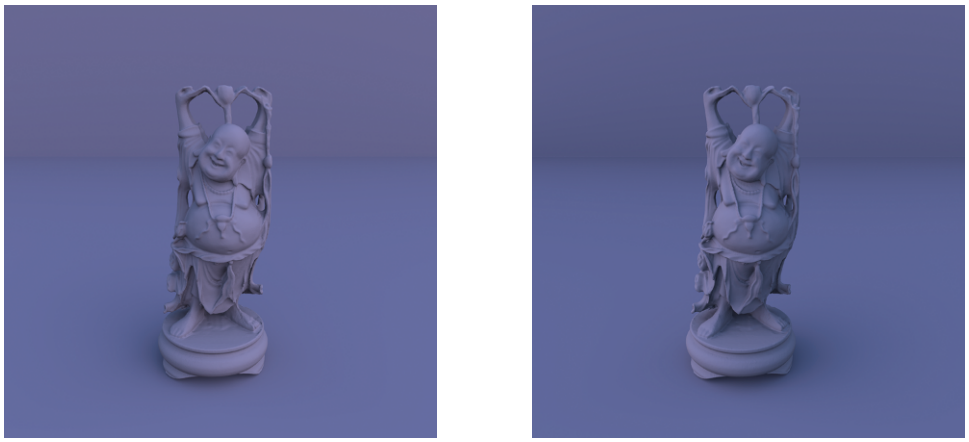
A wide variety of PRT techniques have been developed. Different techniques make different simplifying assumptions and in turn place different limitations on the lighting effects that they can handle. This section and the following section will describe the implementations of two such methods.

In general, PRT techniques assume that some representation of the directionally varying incident radiance function is available for each object in the scene to be shaded. This lighting information is generally represented in some basis; here, we will continue to use the spherical harmonics. The incident lighting information comes from one of a number of sources:

- A single infinite area light source (in which case all objects in the scene have the same incident lighting)
- A collection of radiance probes, such as were implemented in the previous section
- A cubical environment map of a dynamic scene rendered with the center of the object as its origin

Hybrid approaches are possible as well, where a single infinite area light source might be used to illuminate the entire scene, but where a custom incident radiance function is computed for each object to be shaded by modifying the incident radiance function

ALLOCA() 1009
BSDF::rho() 482
BSDF_ALL_REFLECTION 428
Faceforward() 66
INV_PI 1002
SHConvolvePhong() 969
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Spectrum::Clamp() 265
Vector 57



(a) (b)

Figure 17.11: Buddha Model Rendered Using Diffuse Precomputed Radiance Transfer. A view of the Buddha model illuminated by an infinite area light source, where the light source has been rotated between the two frames. Rendering was performed with the `DiffusePRTIntegrator`, which implements algorithms that make it possible to re-render images like these extremely efficiently after the initial precomputation has been performed.

to account for nearby objects that block illumination from the infinite light source. The "Further Reading" section has more details on these approaches.

The `DiffusePRTIntegrator` implements one of the simplest PRT methods, described by Sloan et al. (2002). It is limited to diffuse surfaces and is based on encoding how a geometric model diffusely reflects arbitrary incident illumination, accounting for self-shadowing (and optionally interreflection). This approach is extremely efficient, requiring only a small amount of computation to evaluate the approximation to the scattering equation during rendering.

Figure 17.11 shows two images rendered with the `DiffusePRTIntegrator`. While these images are not any more complex than those rendered with offline ray tracing by other integrators in `pbrt`, remember that once the offline precomputation has been performed images like these can easily be rendered interactively at hundreds of times per second, with both the illumination and viewing position varying from frame to frame.

Diffuse PRT is based on a few simplifications to the scattering equation, Equation (5.8). If we ignore for now multiply reflected light, we have

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) \, |\cos \theta_i| \, d\omega_i.$$

With diffuse PRT, diffuse reflection is assumed, so that scattered radiance isn't directionally varying and the BSDF, ρ/π, is constant and can be moved outside the integral:

`DiffusePRTIntegrator` 972

$$L_o(p) = \frac{\rho(p)}{\pi} \int_{S^2} L_i(p, \omega_i) \, |\cos \theta_i| \, d\omega_i.$$

If we assume an infinitely distant environment light source, then the integrand can be described by the product of a directionally varying incident radiance term $L_{\text{env}}(\omega)$ that ignores occlusion between the point and the light and a transfer function T that represents visiblility and the cosine factor, giving

$$L_{\text{o}}(\text{p}) = \frac{\rho(\text{p})}{\pi} \int_{\mathbb{S}^2} L_{\text{env}}(\omega_{\text{i}})\, T(\text{p}, \omega_{\text{i}})\, \text{d}\omega_{\text{i}}, \tag{17.18}$$

where the *transfer function* T is defined by

$$T(\text{p}, \omega_{\text{i}}) = V(\text{p}, \omega_{\text{i}})\, |\cos \theta_{\text{i}}|. \tag{17.19}$$

Here, $V(\text{p}, \omega_{\text{i}})$ is a visibility term that is one if the ray from p in direction ω is unoccluded and zero otherwise. The transfer function essentially describes how incident radiance from the light is mapped to incident radiance at a point.[5]

Note that the transfer function depends purely on the geometry of the model being rendered and can thus be precomputed independently of the actual lighting environment. Furthermore, the $L_{\text{env}}(\omega)$ term depends purely on the incident lighting environment at rendering time. Sloan et al. observed that if both the incident radiance function and the transfer function are represented in SH coefficients (L_l^m and T_l^m, respectively), then the integral in Equation (17.18) can be evaluated simply by summing the products of their respective basis function coefficients (recall Equation (17.4)):

$$L_{\text{o}}(\text{p}) \approx \frac{\rho(\text{p})}{\pi} \sum_{l=0}^{l_{\text{max}}} \sum_{m=-l}^{l} L_l^m T_l^m. \tag{17.20}$$

To apply this approach to interactive rendering generally requires that the transfer function's SH coefficients be precomputed in an offline process. They are generally computed at the vertices of polygonal models, or at the texels of a texture mapped to the model. At runtime, the transfer function's SH coefficients are looked up, the incident radiance function's SH coefficients are computed (generally only once per model or frame), and the efficient computation of Equation (17.20) is applied. In the implementation here in the `DiffusePRTIntegrator`, the transfer function coefficients are found at each point immediately before it is shaded with Equation (17.20). Thus, the efficiency benefits aren't experienced when using this integrator, but this simplifies the presentation here.

The `DiffusePRTIntegrator`'s constructor takes the number of SH bands to use in its lighting calculations and a sample count used in the Monte Carlo routines that compute SH coefficients. A single set of SH coefficients is used for the incident radiance function for all objects in the implementation here; they are initialized in the `Preprocess()` method below.

`DiffusePRTIntegrator` 972

5 The transfer function can be generalized to account for light that is reflected by other parts of the model before arriving at the current point; this extension is left as an exercise at the end of the chapter.

⟨*DiffusePRTIntegrator Method Definitions*⟩ ≡
```
DiffusePRTIntegrator::DiffusePRTIntegrator(int lm, int ns)
    : lmax(lm), nSamples(RoundUpPow2(ns)) {
    c_in = new Spectrum[SHTerms(lmax)];
}
```

⟨*DiffusePRTIntegrator Private Data*⟩ ≡
```
const int lmax, nSamples;
Spectrum *c_in;
```

The SH coefficients for incident radiance from the lights in the scene are computed at a single point at the center of the scene's bounding box, ignoring occlusion between that point and the light sources. (Recall that the transfer function includes a visibility factor.)

⟨*DiffusePRTIntegrator Method Definitions*⟩ +≡
```
void DiffusePRTIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    BBox bbox = scene->WorldBound();
    Point p = .5f * bbox.pMin + .5f * bbox.pMax;
    RNG rng;
    MemoryArena arena;
    SHProjectIncidentDirectRadiance(p, 0.f, camera->shutterOpen, arena,
                                    scene, false, lmax, rng, c_in);
}
```

The `Li()` method starts by declaring the usual set of common variables and computing the BSDF. The heart of its work is performed by the ⟨*Compute reflected radiance using diffuse PRT*⟩ fragment, which both computes the information needed for diffuse PRT at a point in the scene and then uses this information for the lighting calculation.

⟨*Compute reflected radiance using diffuse PRT*⟩ ≡
⟨*Project diffuse transfer function at point to SH* 972⟩
⟨*Compute integral of product of incident radiance and transfer function* 973⟩

This computation of the SH coefficients of the transfer function is encapsulated in a utility function under the expectation that other implementations may want to use this to precompute values at vertices of the model or at other sampling rates. Here, the coefficients are computed at the ray's intersection point.

⟨*Project diffuse transfer function at point to SH*⟩ ≡ 972
```
Spectrum *c_transfer = arena.Alloc<Spectrum>(SHTerms(lmax));
SHComputeDiffuseTransfer(p, Faceforward(n, wo), isect.rayEpsilon,
                         scene, rng, nSamples, lmax, c_transfer);
```

Computing the SH coefficients of the transfer function at a given point can be done easily with Monte Carlo integration via Equation (17.11), similarly to how we've previously projected other functions into the SH basis.

BBox 70
Camera 302
Camera::shutterOpen 302
DiffusePRTIntegrator 972
DiffusePRTIntegrator::c_in 972
DiffusePRTIntegrator::lmax 972
DiffusePRTIntegrator::nSamples 972
Faceforward() 66
Intersection::rayEpsilon 186
MemoryArena 1015
MemoryArena::Alloc() 1016
Point 63
Renderer 24
RNG 1003
RoundUpPow2() 1002
Scene 22
Scene::WorldBound() 24
SHComputeDiffuseTransfer() 973
SHProjectIncidentDirectRadiance() 949
SHTerms() 935
Spectrum 263

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHComputeDiffuseTransfer(const Point &p, const Normal &n,
        float rayEpsilon, const Scene *scene, RNG &rng, int nSamples,
        int lmax, Spectrum *c_transfer) {
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_transfer[i] = 0.f;
    uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (int i = 0; i < nSamples; ++i) {
        ⟨Sample ith direction and compute estimate for transfer coefficients 973⟩
    }
}
```

The Monte Carlo estimate of the SH projection integral is based on uniform sampling of the sphere. The implementation here implicitly ignores transmissive surfaces by checking that the sampled direction `w` and the surface normal `n` are in the same hemisphere; a more general implementation would not perform this check if the surface was transmissive.

⟨*Sample* `i`*th direction and compute estimate for transfer coefficients*⟩ ≡ 973
```
float u[2];
Sample02(i, scramble, u);
Vector w = UniformSampleSphere(u[0], u[1]);
float pdf = UniformSpherePdf();
if (Dot(w, n) > 0.f && !scene->IntersectP(Ray(p, w, rayEpsilon))) {
    ⟨Accumulate contribution of direction ω to transfer coefficients 973⟩
}
```

If the visibility function is nonzero, then the SH functions are evaluated for the sampled direction and the coefficients are updated using the Monte Carlo estimator. A more efficient approach to implementing this function that would be appropriate for many applications would be to precompute an array of sampled directions as well as the corresponding values of the SH basis functions for each of the directions, reusing these values each time this function was called rather than calling `UniformSampleSphere()` and `SHEvaluate()` each time.

⟨*Accumulate contribution of direction ω to transfer coefficients*⟩ ≡ 973
```
SHEvaluate(w, lmax, Ylm);
for (int j = 0; j < SHTerms(lmax); ++j)
    c_transfer[j] += (Ylm[j] * AbsDot(w, n)) / (pdf * nSamples);
```

The final fragment implements the actual shading calculation, applying Equation (17.20) to compute the integral of the transfer function and the incident radiance function. It scales the result by the diffuse reflectance of the surface.

⟨*Compute integral of product of incident radiance and transfer function*⟩ ≡ 972
```
Spectrum Kd = bsdf->rho(wo, rng, BSDF_ALL_REFLECTION) * INV_PI;
Spectrum Lo = 0.f;
for (int i = 0; i < SHTerms(lmax); ++i)
    Lo += c_in[i] * c_transfer[i];
return L + Kd * Lo.Clamp();
```

AbsDot() 61
ALLOCA() 1009
BSDF::rho() 482
BSDF_ALL_REFLECTION 428
INV_PI 1002
Normal 65
Point 63
Ray 66
RNG 1003
RNG::RandomUInt() 1003
Sample02() 372
Scene 22
Scene::IntersectP() 24
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Spectrum::Clamp() 265
UniformSampleSphere() 664
UniformSpherePdf() 664
Vector 57

17.5 PRECOMPUTED GLOSSY TRANSFER

The restriction to diffuse reflection imposed by the diffuse PRT approach of the previous section is a significant one; most interesting surfaces exhibit some form of directionally varying reflection. This section describes a more complex PRT method that supports more general BSDFs than Lambertian; because it is based on using a relatively small number of SH bands to represent illumination and reflection, it works well only with relatively low-frequency reflection functions. This method is more computationally intensive than diffuse PRT, though it is still practical for interactive rendering with modern graphics processors. Discussing the implementation of this approach also makes it possible to introduce a number of useful additional techniques for using spherical harmonics in rendering.

The PRT algorithm implemented in this section goes through three steps to transform the coefficients that describe the incident radiance function to coefficients that describe the scattered radiance function at a point. When $l_{\max}$ SH bands are used, each of these steps can be represented with a $(l_{\max}+1)^2 \times (l_{\max}+1)^2$ matrix that transforms one set of SH coefficients to another. The three steps are:

1. A transfer matrix $\mathbf{T}$ converts the incident radiance function arriving at the object to the incident radiance function at the particular point being shaded, accounting for shadowing by the object's geometry.
2. A rotation matrix $\mathbf{R}$ takes these SH coefficients of the incident radiance function, which is defined in the world coordinate frame, and rotates them so that the radiance function is defined in the local coordinate system at the point, where the normal vector points in the $+z$ direction.
3. The BSDF matrix $\mathbf{B}$ transforms the incident radiance function (in local coordinates) to an outgoing radiance function (still in local coordinates) accounting for how the BSDF scatters the incident illumination.

The final coefficients describe the directional distribution of outgoing radiance at the point. Given them, we can evaluate the SH basis functions for the outgoing direction to compute the outgoing radiance. Figure 17.12 shows the Buddha model rendered with the glossy PRT algorithm implemented in this section as well as a reference image rendered with the `DirectLightingIntegrator`. As long as enough SH coefficients are used, the PRT approach gives a close match.

Thus, to do the final shading calculation, given the three matrices and the coefficients for the incident radiance c_{i}, we will compute outgoing radiance coefficients c_{o} by

$$c_{\mathrm{o}} = \mathbf{B}(\mathbf{R}(\mathbf{T}c_{\mathrm{i}})),$$

and then use Equation (17.9) to compute outgoing radiance for the direction ω_{o}:

$$L_{\mathrm{o}}(\omega_{\mathrm{o}}) = \sum_i c_{\mathrm{o},i}\, Y_i(\omega_{\mathrm{o}})$$

How to rotate functions represented with SH coefficients was discussed in Section 17.2.5; there, we also saw that rotating them with a series of sparse rotation matrices was a more efficient approach than computing a dense matrix that encoded the entire rotation. Thus, the implementation to follow will perform the rotation without explicitly computing $\mathbf{R}$.

DirectLightingIntegrator 742

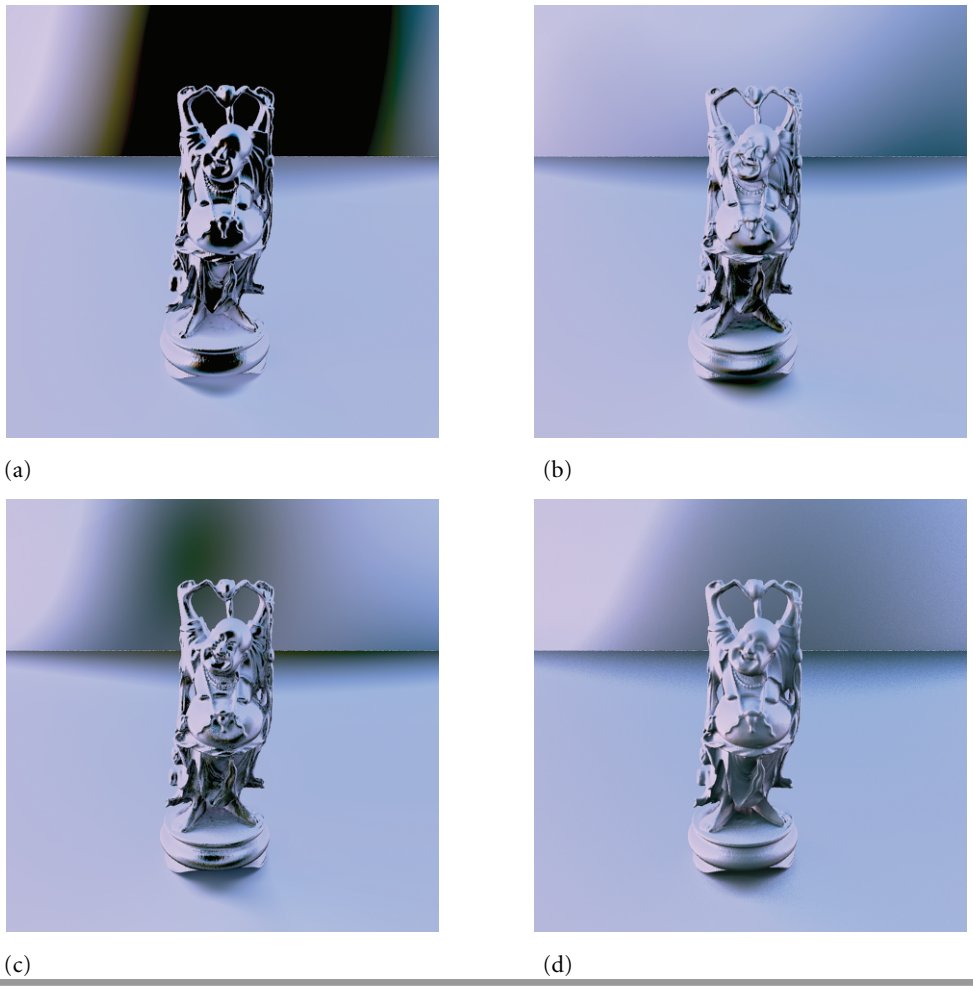


Figure 17.12: Buddha Model Rendered Using Glossy Precomputed Radiance Transfer. (a) A glossy Buddha illuminated by an infinite area light source, rendered with the `GlossyPRTIntegrator` with $l_{\max} = 3$, (b) `GlossyPRTIntegrator` with $l_{\max} = 5$, (c) `GlossyPRTIntegrator` with $l_{\max} = 8$, (d) reference image rendered with the `DirectLightingIntegrator`. The approximations introduced by representing illumination in the SH accurately render the object as long as enough SH coefficients are used.

We'll first show how the transfer and BSDF matrices are computed in the two following sections.

17.5.1 THE TRANSFER MATRIX

`DirectLightingIntegrator` 742

Given an unoccluded incident radiance function $L_{\text{env}}(\text{p}, \omega)$ (unknown until rendering time) and a transfer function $T(\text{p}, \omega)$ (which can be evaluated at precomputation time), we would like to be able to efficiently compute the SH coefficients of the directionally varying incident radiance function given by their product,

$$L_{\text{i}}(\text{p}, \omega) = L_{\text{env}}(\text{p}, \omega)\, T(\text{p}, \omega),$$

once the SH coefficients of the incident radiance function are known. Here, the transfer function is given by $T(\mathrm{p}, \omega_\mathrm{i}) = V(\mathrm{p}, \omega_\mathrm{i})$, and doesn't include the cosine term in Equation (17.19). (The cosine factor will be incorporated as part of the BSDF matrix.)

More generally, we would like to be able to multiply an unknown function represented in SH (the incident radiance function) with a known function (the transfer function) and compute the coefficients of the SH projection of the result.

If we use the indexed notation for SH coefficients from Equation (17.10), c_i are the coefficients for the incident radiance function and c'_i are the SH coefficients we'd like to compute for the product of the incident radiance and visibility functions. Application of the spherical harmonics projection formula says that the ith coefficient of c'_i is

$$c'_i = \int_{\mathbb{S}^2} \left(\sum_j c_j Y_j(\omega) \right) T(\mathrm{p}, \omega)\, Y_i(\omega)\, \mathrm{d}\omega.$$

We can interchange the integral and the sum and pull the constant c_j terms out of the integral, giving

$$c'_i = \sum_j c_j \int_{\mathbb{S}^2} Y_j(\omega)\, T(\mathrm{p}, \omega)\, Y_i(\omega)\, \mathrm{d}\omega.$$

The remaining terms in the integral can now all be evaluated at preprocessing time. They can be stored in the form of a matrix that can be used to compute new SH c'_i coefficients given coefficients c_i:

$$\begin{pmatrix} c'_0 \\ c'_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} t_{0,0} & t_{0,1} & \cdots \\ t_{1,0} & \ddots & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \end{pmatrix}$$

where

$$t_{i,j} = \int_{\mathbb{S}^2} Y_j(\omega)\, T(\mathrm{p}, \omega)\, Y_i(\omega)\, \mathrm{d}\omega. \tag{17.21}$$

Note that this matrix is symmetric.

The `SHComputeTransferMatrix()` method computes the transfer matrix for a given position in the scene, returning it in an $(l_{\max}+1)^2 \times (l_{\max}+1)^2$ array `T` allocated by the caller.

⟨*Spherical Harmonics Definitions*⟩ +≡

```
void SHComputeTransferMatrix(const Point &p, float rayEpsilon,
        const Scene *scene, RNG &rng, int nSamples, int lmax,
        Spectrum *T) {
    for (int i = 0; i < SHTerms(lmax)*SHTerms(lmax); ++i)
        T[i] = 0.f;
    uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (int i = 0; i < nSamples; ++i) {
        ⟨Compute Monte Carlo estimate of i th sample for transfer matrix 977⟩
    }
}
```

Point 63
RNG 1003
RNG::RandomUInt() 1003
Scene 22
Spectrum 263

Monte Carlo integration is used to estimate the values of the integrals in Equation (17.21). Each sample ω is used to update the estimates of all of the $(l_{\max}+1)^2 \times (l_{\max}+1)^2$ integrals that need to be computed; doing so lets us reuse the result of the relatively expensive shadow ray test, improving efficiency.

⟨*Compute Monte Carlo estimate of i th sample for transfer matrix*⟩ ≡ 976

```
float u[2];
Sample02(i, scramble, u);
Vector w = UniformSampleSphere(u[0], u[1]);
float pdf = UniformSpherePdf();
if (!scene->IntersectP(Ray(p, w, rayEpsilon))) {
    ⟨Update transfer matrix for unoccluded direction 977⟩
}
```

If the transfer function for the sampled direction `w` is one, then the estimates of the integrands for all of the elements of the matrix are updated.

⟨*Update transfer matrix for unoccluded direction*⟩ ≡ 977

```
SHEvaluate(w, lmax, Ylm);
for (int j = 0; j < SHTerms(lmax); ++j)
    for (int k = 0; k < SHTerms(lmax); ++k)
        T[j*SHTerms(lmax)+k] += (Ylm[j] * Ylm[k]) / (pdf * nSamples);
```

17.5.2 THE BSDF MATRIX

Given a known BSDF, $f(\omega_o, \omega_i)$, we'd like to compute the SH coefficients for the outgoing radiance function L_o that results from integrating the product of the BSDF and $\cos\theta$ term with an incident radiance function defined in SH.

The scattering equation, Equation (5.8), describes the value to be computed in the general case; if the incident radiance function is described by SH coefficients c_j, we have

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) \left(\sum_j c_j Y_j(\omega_i) \right) |\cos\theta_i| \, d\omega_i.$$

We'd like to project the outgoing radiance function, $L_o(p, \omega_o)$, into the spherical harmonic basis to get coefficients c'_k for the outgoing radiance function. Using the SH projection equation, Equation (17.8), we have

$$\begin{aligned} c'_k &= \int_{S^2} L_o(p, \omega_o) \, Y_k(\omega_o) \, d\omega_o \\ &= \int_{S^2} \int_{S^2} f(p, \omega_o, \omega_i) \left(\sum_j c_j \, Y_j(\omega_i) \right) |\cos\theta_i| \, Y_k(\omega_o) \, d\omega_i \, d\omega_o \end{aligned}$$

The sum and the constant terms c_j can be moved outside of the integral, giving:

$$c'_k = \sum_j c_j \int_{S^2} \int_{S^2} f(\omega_o, \omega_i) \, |\cos\theta_i| \, Y_j(\omega_i) \, Y_k(\omega_o) \, d\omega_i \, d\omega_o.$$

Similar to the transfer function in Section 17.5.1, the integrand can be evaluated if the BSDF is known.

Ray 66
Sample02() 372
Scene::IntersectP() 24
SHEvaluate() 936
SHTerms() 935
UniformSampleSphere() 664
UniformSpherePdf() 664
Vector 57

We can therefore represent the values of each of these double integrals for all (j, k) pairs in a matrix $\mathbf{B}$, where the elements of the matrix $b_{j,k}$ are given by

$$b_{j,k} = \int_{S^2} \int_{S^2} f(\omega_o, \omega_i) \, |\cos \theta_i| \, Y_j(\omega_i) \, Y_k(\omega_o) \, d\omega_i \, d\omega_o. \quad [17.22]$$

The spherical harmonic coefficients of the outgoing radiance function can then be found with a matrix multiplication of $\mathbf{B}$ with the SH coefficients of the incident radiance function,

$$c_i' = \mathbf{B} \, c_i$$

The utility function `SHComputeBSDFMatrix()` computes this matrix up to a given l_{max} band, for a specific parameterized BSDF model (diffuse and glossy reflection). Generalizing the function to handle arbitrary BSDFs would be straightforward. The BSDF matrix is returned in the `B` array, which must be at least $(l_{max} + 1)^2 \times (l_{max} + 1)^2$ elements large.

```
⟨Spherical Harmonics Definitions⟩ +≡
void SHComputeBSDFMatrix(const Spectrum &Kd, const Spectrum &Ks,
        float roughness, RNG &rng, int nSamples, int lmax, Spectrum *B) {
    for (int i = 0; i < SHTerms(lmax)*SHTerms(lmax); ++i)
        B[i] = 0.f;
    ⟨Create BSDF for computing BSDF transfer matrix⟩
    ⟨Precompute directions ω and SH values for directions 978⟩
    ⟨Compute double spherical integral for BSDF matrix 979⟩
    ⟨Free memory allocated for SH matrix computation⟩
}
```

The ⟨*Create* `BSDF` *for computing BSDF transfer matrix*⟩ fragment (not included here) does some straightforward plumbing to compute a `BSDF` object, `bsdf`, that represents to the BSDF for the provided reflection parameters.

Because it has to compute a double integral over the sphere of directions, the implementation here precomputes a set of well-distributed direction vectors on the sphere and the corresponding values of the SH basis functions for each direction. These values will be reused many times in the code that estimates the integral below.

```
⟨Precompute directions ω and SH values for directions⟩ ≡ 978
float *Ylm = new float[SHTerms(lmax) * nSamples];
Vector *w = new Vector[nSamples];
uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
for (int i = 0; i < nSamples; ++i) {
    float u[2];
    Sample02(i, scramble, u);
    w[i] = UniformSampleSphere(u[0], u[1]);
    SHEvaluate(w[i], lmax, &Ylm[SHTerms(lmax)*i]);
}
```

BSDF 478
RNG 1003
RNG::RandomUInt() 1003
Sample02() 372
SHComputeBSDFMatrix() 978
SHEvaluate() 936
SHTerms() 935
Spectrum 263
UniformSampleSphere() 664
Vector 57

Now the Monte Carlo estimate of the integrals can be computed, using estimators of the form:

$$\int_{S^2}\int_{S^2} f(\omega,\omega')\,d\omega\,d\omega' \approx \frac{1}{N_i}\frac{1}{N_j}\sum_i^{N_i}\sum_j^{N_j}\frac{f(\omega_i,\omega_j)}{p(\omega_i)\,p(\omega_j)}. \quad [17.23]$$

The implementation here uses the `w` array to find direction samples ω_i and ω_j for each of the estimates.

⟨*Compute double spherical integral for BSDF matrix*⟩ ≡ 978
```
for (int osamp = 0; osamp < nSamples; ++osamp) {
    const Vector &wo = w[osamp];
    for (int isamp = 0; isamp < nSamples; ++isamp) {
        const Vector &wi = w[isamp];
        ⟨Update BSDF matrix elements for sampled directions 979⟩
    }
}
```

The BSDF is evaluated for each pair of directions; if its value is nonzero, then the MC estimates for all of the elements of the matrix are updated; recall from Equation (17.22) that each element of the BSDF matrix is given by an integral of the product of this term with two SH basis functions.

⟨*Update BSDF matrix elements for sampled directions*⟩ ≡ 979
```
Spectrum f = bsdf->f(wo, wi);
if (!f.IsBlack()) {
    float pdf = UniformSpherePdf() * UniformSpherePdf();
    f *= fabsf(CosTheta(wi)) / (pdf * nSamples * nSamples);
    for (int i = 0; i < SHTerms(lmax); ++i)
        for (int j = 0; j < SHTerms(lmax); ++j)
            B[i*SHTerms(lmax)+j] += f * Ylm[isamp*SHTerms(lmax)+j] *
                                        Ylm[osamp*SHTerms(lmax)+i];
}
```

Another approach to implementing this function would be to uniformly sample the outgoing direction ω_o and then use the `BSDF::Sample_f()` method to sample an incident direction ω_i according to the BSDF's sampling distribution. However, doing so would mean that we wouldn't be able to reuse the precomputed `Ylm` values for each incident direction ω_i, but would need to evaluate the SH basis functions from scratch for each sampled direction. For highly glossy or specular BSDFs, using their sampling methods would likely be more efficient, though given that spherical harmonics aren't an efficient representation for the high-frequency lighting effects of highly glossy or specular BSDFs, this approach likely isn't worthwhile in practice.

BSDF::f() 481
BSDF::Sample_f() 706
CosTheta() 426
DiffusePRTIntegrator 972
SHTerms() 935
Spectrum 263
Spectrum::IsBlack() 265
UniformSpherePdf() 664
Vector 57

17.5.3 GlossyPRTIntegrator IMPLEMENTATION

The `GlossyPRTIntegrator` puts the approach outlined at the start of this section together to implement the described PRT algorithm. Like the `DiffusePRTIntegrator` it does some parts of the computation that would normally be done at precomputation time in its `Li()` method so that the approach can be easily implemented in the context of an offline rendering system like `pbrt`. In an interactive rendering system, the stages would normally be done at preprocessing time.

The constructor takes material parameters, the number of SH bands to use, and the number of samples to take when computing the transfer matrix. Material parameters are needed here since this implementation requires that all surfaces in the scene have the same BSDF so a single BSDF matrix can be computed. The computation of the BSDF matrix is computationally intensive, such that we would prefer to not do it at all points in the scene. Other glossy PRT implementations may be willing to pay this price or could compute a set of representative BSDF matrices and then represent BSDF matrices for arbitrary BSDFs as weighted sums of them. (For example, a single BSDF matrix could be computed for diffuse reflection, assuming reflectance of one, and then its terms could be scaled by the surfaces' actual reflectances.)

⟨*GlossyPRTIntegrator Public Methods*⟩ ≡
```
GlossyPRTIntegrator(const Spectrum &kd, const Spectrum &ks,
                    float rough, int lm, int ns)
    : Kd(kd), Ks(ks), roughness(rough), lmax(lm),
      nSamples(RoundUpPow2(ns)) {
}
```

⟨*GlossyPRTIntegrator Private Data*⟩ ≡
```
const Spectrum Kd, Ks;
const float roughness;
const int lmax, nSamples;
```

Like the `DiffusePRTIntegrator`, a single incident radiance function is used for rendering: the scene's incident lighting at the center of the scene is projected into SH, ignoring visibility, and stored in the `c_in` member variable. This computation is implemented by the ⟨*Project direct lighting into SH for* `GlossyPRTIntegrator`⟩, not included here.

⟨*GlossyPRTIntegrator Method Definitions*⟩ ≡
```
void GlossyPRTIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    ⟨Project direct lighting into SH for GlossyPRTIntegrator⟩
    ⟨Compute glossy BSDF matrix for PRT 980⟩
}
```

⟨*GlossyPRTIntegrator Private Data*⟩ +≡
```
Spectrum *c_in;
Spectrum *B;
```

After the light's SH coefficients have been stored in the `c_in` member variable, the BSDF matrix is computed and stored in `B`.

⟨*Compute glossy BSDF matrix for PRT*⟩ ≡ 980
```
B = new Spectrum[SHTerms(lmax)*SHTerms(lmax)];
SHComputeBSDFMatrix(Kd, Ks, roughness, rng, 1024, lmax, B);
```

The `GlossyPRTIntegrator::Li()` method computes outgoing radiance at a point in the scene given a ray–object intersection, using the glossy PRT algorithm. We won't include the standard parts of its implementation, focusing instead on its main fragment, ⟨*Compute reflected radiance with glossy PRT at point*⟩.

Camera 302
DiffusePRTIntegrator 972
GlossyPRTIntegrator 980
GlossyPRTIntegrator::B 980
GlossyPRTIntegrator::Kd 980
GlossyPRTIntegrator::Ks 980
GlossyPRTIntegrator::roughness 980
Renderer 24
Scene 22
SHComputeBSDFMatrix() 978
Spectrum 263

As outlined above, there are four main steps. The incident radiance function is converted to incident radiance at the point being shaded, using the transfer matrix to account for visibility between the point and the environment. Then the incident radiance function is rotated to local coordinates and an outgoing radiance function is computed using the previously computed BSDF matrix. Finally, the value of the outgoing radiance function is computed for the outgoing direction given by the incident ray.

⟨*Compute reflected radiance with glossy PRT at point*⟩ ≡
  ⟨*Compute SH radiance transfer matrix at point and SH coefficients* 981⟩
  ⟨*Rotate incident SH lighting to local coordinate frame* 981⟩
  ⟨*Compute final coefficients* c_o *using BSDF matrix* 982⟩
  ⟨*Evaluate outgoing radiance function for* ω_o *and add to* L 982⟩

⟨*Compute SH radiance transfer matrix at point and SH coefficients*⟩ ≡ 981
```
Spectrum *c_t = arena.Alloc<Spectrum>(SHTerms(lmax));
Spectrum *T = arena.Alloc<Spectrum>(SHTerms(lmax)*SHTerms(lmax));
SHComputeTransferMatrix(p, isect.rayEpsilon, scene, rng, nSamples,
                        lmax, T);
SHMatrixVectorMultiply(T, c_in, c_t, lmax);
```

The SHMatrixVectorMultiply() utility function multiplies the given vector of SH coefficients by the matrix and returns the result. Its implementation is straightforward and so won't be included here.

⟨*Spherical Harmonics Declarations*⟩ +≡
```
void SHMatrixVectorMultiply(const Spectrum *M, const Spectrum *v,
                            Spectrum *vout, int lmax);
```

Now that we have the SH coefficients of the incident radiance function at the point after accounting for occlusion by the object, we now need to rotate them to the local coordinate system defined by the surface normal and tangent vectors at the point; this rotation is necessary because the BSDF matrix was computed with respect to the local coordinate system. We find the rotation matrix by determining how the coordinate system basis vectors are transformed by the local-to-world transformations and then use the SHRotate() utility function to rotate the incident radiance function to the local coordinate system.

⟨*Rotate incident SH lighting to local coordinate frame*⟩ ≡ 981
```
Vector r1 = bsdf->LocalToWorld(Vector(1,0,0));
Vector r2 = bsdf->LocalToWorld(Vector(0,1,0));
Normal nl = Normal(bsdf->LocalToWorld(Vector(0,0,1)));
Matrix4x4 rot(r1.x, r2.x, nl.x, 0,
              r1.y, r2.y, nl.y, 0,
              r1.z, r2.z, nl.z, 0,
                 0,    0,    0, 1);
Spectrum *c_l = arena.Alloc<Spectrum>(SHTerms(lmax));
SHRotate(c_t, c_l, rot, lmax, arena);
```

Given the incident radiance function's SH coefficients in c_l and the BSDF matrix, the product of the matrix and the coefficients gives the SH coefficients of the outgoing radiance function.

BSDF::LocalToWorld() 480
GlossyPRTIntegrator::
  lmax 980
Intersection::rayEpsilon 186
Matrix4x4 1021
MemoryArena::Alloc() 1016
Normal 65
SHComputeTransferMatrix() 976
SHMatrixVectorMultiply() 981
SHRotate() 953
SHTerms() 935
Spectrum 263
Vector 57

⟨*Compute final coefficients* c_o *using BSDF matrix*⟩ ≡ 981
```
Spectrum *c_o = arena.Alloc<Spectrum>(SHTerms(lmax));
SHMatrixVectorMultiply(B, c_l, c_o, lmax);
```

Finally, to compute outgoing radiance along the ray, we just need to evaluate the SH basis functions for the outgoing direction (in local coordinates).

⟨*Evaluate outgoing radiance function for* ω_o *and add to* L⟩ ≡ 981
```
Vector woLocal = bsdf->WorldToLocal(wo);
float *Ylm = ALLOCA(float, SHTerms(lmax));
SHEvaluate(woLocal, lmax, Ylm);
Spectrum Li = 0.f;
for (int i = 0; i < SHTerms(lmax); ++i)
    Li += Ylm[i] * c_o[i];
L += Li.Clamp();
```

Implementations of glossy PRT in interactive renderers generally find the product of the **B**, **R**, and **T** matrices at each point where the precomputation is being performed (mesh vertices or samples in a texture map, etc.). However, storing even a single dense SH transformation matrix for each of a large number of points requires a substantial amount of storage. Furthermore, the computational expense of even a single full SH matrix/vector multiply at each point being shaded can also be significant.

As such, a number of compression techniques have been developed that reduce both the storage and computational requirements of storing these matrices at a large number of points. The "Further Reading" section has further details. For example, Sloan et al. (2003) suggest computing all of these matrices in the pre-processing step and then applying clustered principal component analysis to compute a small number of representative matrices from the original set. Each of the original matrices is then represented as a weighted sum of the representative matrices. Thus, the expensive matrix/SH vector multiplication only needs to be performed for the representative matrices. Each time a result is needed at a point, a weighted sum of the transformed SH vectors corresponding to the matrices gives an approximation to the transformed SH vector for the outgoing radiance function at the point; this is a substantially more efficient computation.

FURTHER READING

Early work on image-based lighting design systems was done by Airey et al. (1990), Dobashi et al. (1995), Dorsey et al. (1995), Nimeroff et al. (1994), and Teo et al. (1997), who laid the groundwork for much of the subsequent work on more general modes of precomputed light transport. More recently, Pellacini et al. (2005) and Ragan-Kelley et al. (2007) described more flexible and more efficient systems for lighting design. Kristensen et al. (2005) described a lighting design system based on precomputed light transport algorithms that consequently allows changing viewpoints, and Ben-Artzi et al. (2008) developed precomputation-based algorithms for editing BRDFs in complex scenes with fixed lighting and camera positions.

Ambient occlusion was first developed by Zhukov et al. (1998), though Miller's accessibility shading also introduced related concepts (Miller 1994). Landis (2002) described

ALLOCA() 1009
BSDF::WorldToLocal() 480
GlossyPRTIntegrator::B 980
GlossyPRTIntegrator::lmax 980
SHEvaluate() 936
SHMatrixVectorMultiply() 981
SHTerms() 935
Spectrum 263
Spectrum::Clamp() 265
Vector 57

its application for offline rendering for movies, and Kontkanen and Laine (2005) and Kontkanen and Aila (2006) described ambient occlusion representations that are applicable to animated and moving objects. See also the survey article by Méndez-Feliu and Sbert (2009) for more information about techniques based on and related to ambient occlusion.

Patmore precomputed incident directional radiance on a grid for illuminating foliage (Patmore 1993), and Greger et al. (1998) introduced the *irradiance volume*, where directional irradiance functions are stored on an adaptive grid and used for illuminating objects in a scene. Both of these approaches laid the foundation for radiance probes based on spherical harmonic coefficients. Kontkanen and Laine (2006) developed methods for reducing rendering artifacts due to radiance probes being computed inside solid objects; the approach used in Section 17.3 is inspired by their insights. Nijasure et al. (2005) developed a real-time global illumination algorithm for dynamic scenes based on dynamically generated radiance probes exchanging illumination with their neighbors.

Kajiya and Von Herzen (1984) first introduced spherical harmonics to graphics. Cabral et al. (1987) used them to represent BRDFs, and BRDF matrices of the form used in Section 17.5 were used to represent complex simulated reflection functions for Monte Carlo path tracing by Westin et al. (1992). Kautz et al. (2002) tabularized spherical harmonic coefficients of 2D slices of the 4D BRDF for a set of outgoing directions and then interpolated between them, making it possible to efficiently compute glossy reflection by using the efficient inner product method of Equation (17.4). Génevaux et al. (2006) described an approach for interactive rendering of specular refraction where precomputed refraction paths are stored in the spherical harmonics basis.

Sloan's notes (Sloan 2008) and Snyder's technical report (Snyder 2006) collected a variety of useful information about using spherical harmonics in practice. Krivánek et al. (2006) developed an approximate method for rotation SH functions that is more efficient than the approach used in this chapter. For functions defined only on the hemisphere, a variant of the SH basis that only covers the hemisphere was introduced by Guatron et al. (2004). Jarosz et al. (2009) showed how to generate samples from functions represented in SH using importance sampling.

Ramamoorthi has formalized and developed the approach of expressing reflection in terms of convolution, deriving a number of important new algorithms for rendering. His Ph.D. thesis (Ramamoorthi 2002) is a valuable reference, with pointers to a series of publications in this area. See in particular his papers with Hanrahan (Ramamoorthi and Hanrahan 2001a, 2001b) for the development of the theory and its application to computing irradiance from radiance functions (which we applied in Section 17.3.2). Basri and Jacobs (2001) simultaneously derived this approach for irradiance functions.

The first modern precomputed radiance transport techniques were introduced by Sloan and collaborators (Sloan et al. 2002, 2003a, 2003b). The diffuse PRT algorithm implemented in Section 17.4 was introduced in their 2002 paper, and the glossy PRT algorithm from Section 17.5 is similar to the one in Sloan et al. (2003a), though we have not implemented the compression technique they introduced here. Other early work includes Lehtinen and Kautz (2003), who used the general BRDF matrix approach for PRT, though they also introduced an even more efficient approach for run-time

evaluation than we used here. See Lehtinen's M.S. thesis and paper for a solid mathematical framework for describing PRT algorithms in terms of standard rendering theory (Lehtinen 2004, 2007). Ramamoorthi has published an extensive survey of precomputed light transport algorithms; it is an excellent resource for more information and pointers to other work on this topic (Ramamoorthi 2007).

Since the introduction of initial PRT algorithms, a substantial amount of work has been done by researchers to increase the generality of scenes that can be rendered using PRT approaches. For example, James and Fatahalian (2003) and Sloan et al. (2005) introduced methods to support animated models, and Wang et al. (2005) applied PRT to subsurface scattering. Zhou et al. (2005) and Ren et al. (2006) developed techniques for precomputing the effect of occlusion from complex objects for more efficient real-time rendering; these techniques can also be used for computing more accurate incident radiance functions for objects when using other PRT algorithms. Annen et al. (2004) showed how SH illumination from local light sources (rather than infinitely far away light sources) can be efficiently incorporated.

A particularly effective technique for efficiently handling general BRDFs with PRT has been to use *separable approximations*. Kautz and McCool developed a (non-PRT-based) approach for interactive rendering that approximated 4D BRDFs $f(\omega_o, \omega_i)$ as products of two 2D functions $g(\omega_i)h(\omega_o)$ (Kautz and McCool 1999). This approach was first used for PRT by Liu et al. (2004) and Wang et al. (2004). Mahajan et al. (2008) analyzed the error from these factorization approaches and provided pointers to other recent applications of them.

One disadvantage of the spherical harmonics basis is that it can't represent sharp reflections and shadows accurately. Ng et al. (2003) demonstrated that non-linear approximation using wavelets could effectively capture these all-frequency lighting effects for PRT. Wang et al. (2006b) applied wavelets and a separable BRDF approximation to rendering glossy objects, and Wang et al. (2006a) introduced an algorithm for rotating functions in wavelet bases; wavelets don't rotate as simply as spherical harmonics do.

For many desirable applications of PRT, it is necessary to be able to compute a *triple product* integral, the integral of the product of three functions represented in a basis. (For example, we might have an incident radiance function, a BRDF function, and a visibility function all represented in a particular basis.) Unfortunately, computing integrals of triple products of functions is much less efficient than the double product of Equation (17.4). Ng et al. (2004) investigated triple product integrals in depth and presented a new $O(n)$ algorithm for triple products for functions in the Haar wavelet basis, with n the number of basis function coefficients. (In contrast, triple product integrals for spherical harmonics require $O(n^{5/2})$ operations.)

EXERCISES

❷ 17.1 Use `pbrt` to compute radiance probes for a scene and write an interactive rendering system that loads these probes and uses them for shading dynamic objects. Experiment with computing probes at different sampling frequencies;

how fine a spacing is necessary for artifact-free rendering? What sort of visual errors result from too low a sampling frequency?

● 17.2 Read Ramamoorthi and Hanrahan's paper (2004), which includes closed-form expressions for projecting a variety of symmetric BRDFs into spherical harmonics, thus making it possible to use them in the convolution framework introduced in Section 17.3.2. Implement these functions and use them to develop an extended radiance probe integrator that supports a wider variety of materials.

■ 17.3 Generalize the radiance probe approach for volume light transport with multiple scattering. First, modify the probe computation process to account for volume light transport and scattering, taking these factors into account when computing the incident radiance function at a point. Next, modify the rendering process to do ray marching through `VolumeRegions` and use the radiance probes to compute scattering.

The Henyey–Greenstein phase function can be projected into spherical harmonics in closed-form; for a given asymmetry parameter g, the coefficients are given by:

$$c_l^0 = 2g^l\sqrt{(2l+1)\pi},$$

with $c_l^m = 0$ for $m \neq 0$. Use these coefficients to apply the convolution approach to computing reflected light at points in the volume.

■ 17.4 Implement an interactive rendering system that uses `pbrt` to precompute PRT coefficients and then uses them for hardware-accelerated real-time rendering. You may want to implement a new `Renderer` that takes as input a number of points at which to precompute coefficients (for example, the vertices of a triangle mesh), and then stores the computed values in a file for use by the real-time renderer.

● 17.5 Read Sloan et al.'s paper on precomputed radiance transfer (Sloan et al. 2002) and modify the computation in the `DiffusePRTIntegrator` to account for multiple scattering rather than just direct lighting in the transfer function, as the current implementation does. Render images demonstrating the differences from including this factor.

■ 17.6 A shortcoming of using a single incident radiance function from an infinitely far away light source for lighting in PRT algorithms is that occlusion due to other objects in the scene that weren't included in the computation of the transfer function isn't accounted for. A more accurate approach is to compute customized incident radiance functions that account for occlusion—for example, for each object in the scene.

Ren et al. (2006) showed how to approximate complex shadow casting objects as collections of spheres, projecting the spherical blockers into the spherical harmonic basis. They then showed how to efficiently compute products of functions in the SH basis (incident lighting and blocker occlusion) by taking the

`DiffusePRTIntegrator` 972
`Renderer` 24
`VolumeRegion` 587

SH logarithm, adding, and then performing SH exponentiation, similar to how multiplication of real numbers can be converted into addition via logarithms and exponentiation. (The mathematics for SH logs and exponents are much more complex, however.)

Implement an approach like Ren et al.'s and show the image differences that result from accounting for shadowing of incident radiance. What is the resulting computational overhead?

3 17.7 The glossy PRT approach in Section 17.5 is relatively inefficient in both memory and computation: dense SH matrices are computed at each point and a number of matrix multiplications are performed. Sloan et al. (2003b) introduced a compression technique that addresses both the storage and computation problems. In a preprocess, at each point where PRT computation is to be performed, the product of the transfer, rotation, and BRDF matrices is computed and stored. Then, using clustered principal component analysis (CPCA), a relatively small number of representative matrices is computed based on the initial set; each original matrix is then represented as a weighted sum of the matrices. At rendering time, the expensive matrix product with the incident radiance function is only performed for the representative matrices. Shading is just a matter of computing outgoing radiance SH coefficients as a weighted sum of the coefficients computed from the representative matrices and using these coefficients to compute the outgoing radiance value.

Implement this approach, either in a modified `GlossyPRTIntegrator` or in a hybrid system that uses `pbrt` for precomputation and a real-time renderer for rendering. Measure the difference in computation time and storage requirements. For your test scene, how many representative matrices are needed for good results? What sort of artifacts do you see with too few matrices?