Dynamically rendered cubic environment maps can be projected onto the SH basis [871, 897, 1458]. Since cubic environment maps are a discrete representation of the incoming radiance function, the integral over the sphere in Equation 10.21 becomes a sum over the cube map texels:

$$k_{Lj} = \sum_t f_j(\mathbf{r}[t])L[t]d\omega[t], \tag{10.40}$$

where t is the index of the current cube map texel, $\mathbf{r}[t]$ is the direction vector pointing to the current texel, $f_j(\mathbf{r}[t])$ is the jth SH basis function evaluated at $\mathbf{r}[t]$, $L[t]$ is the radiance stored in the texel, and $d\omega[t]$ is the solid angle subtended by the texel. Kautz [871], King [897], and Sloan [1656] describe how to compute $d\omega[t]$.

To convert the radiance coefficients k_{Lj} into irradiance coefficients, these need to be multiplied by the scaled coefficients of the clamped cosine function $\cos(\theta_i)^+$:

$$k_{Ej} = k'_{\cos^+ j} k_{Lj} = k'_{\cos^+ j} \sum_t f_j(\mathbf{r}[t])L[t]d\omega[t], \tag{10.41}$$

where k_{Ej} is the jth coefficient of the irradiance function $E(\mathbf{n})$, k_{Lj} is the jth coefficient of the incoming radiance function $L(\mathbf{l})$, and $k'_{\cos^+ j}$ is the jth coefficient of the clamped cosine function $\cos(\theta_i)^+$ scaled by $\sqrt{4\pi/(2l+1)}$ (l is the frequency band index).

Given t and the cube map resolution, the factor $k'_{\cos^+ j} f_j(\mathbf{r}[t])d\omega[t]$ is constant for each basis function $f_j()$. These basis factors can be precomputed offline and stored in cube maps, which should be of the same resolution as the dynamic environment maps that will be rendered. The number of textures used can be reduced by packing a separate basis factor in each color channel. To compute an irradiance coefficient for the dynamic cube map, the texels of the appropriate basis factor map are multiplied with those of the dynamic cube map, and the results are summed. Beyond information on dynamic irradiance cube maps, King [897] provides implementation details on GPU SH projection as well.

Dynamic light sources can be added into an existing SH irradiance environment map. This merge is done by computing the SH coefficients of the lights' irradiance contributions and adding them to the existing coefficients. Doing so avoids the need to recompute the entire irradiance environment map. It is a straightforward process, since simple analytical expressions exist for the coefficients of point, disk, and spherical lights [583, 871, 1656, 1690]. Summing the coefficients has the same effect as summing the irradiance. Usually these representations are given in zonal harmonics, for a light aligned with the z-axis, and then rotations can be applied to position the light toward an arbitrary direction. Zonal harmonics rotation is a special case of SH rotation (Section 10.3.2) and is much more efficient, requiring only a dot product instead of a full matrix transform. Coefficients for light sources with more complex shapes can be computed by drawing them into an image that is then numerically projected onto the SH basis [1690]. For the special case of a physical sky model, Habel [626] shows a direct expansion of the Preetham skylight in spherical harmonics.

The ease of projection of common analytical light sources into SH is important, as often environment lighting is used to stand in for distant or less-intense light sources. Fill lights are an important case. In rendering, these light source are placed to simulate the indirect light in the scene, i.e., light bouncing off surfaces. Specular contributions are often not computed for fill lights, especially as these lights can be physically large relative to the shaded object and relatively dim compared to other sources of illumination in the scene. These factors make their specular highlights more spread out and less noticeable. This type of light has a real-world analogy in lighting for film and video, where physical *fill lights* are often used to add lighting in shadows.

In spherical harmonic space it is also simple to do the opposite derivation, that is, to extract analytic light sources from radiance projected in SH. In his survey of SH techniques, Sloan [1656] shows how, given a directional light source with a known axis, it is easy to compute from a SH irradiance representation the intensity that the light should have to minimize the error between itself and the encoded irradiance.

In his previous work [1653] Sloan shows how to select a near-optimal direction by employing only the coefficients in the first (linear) band. The survey also includes a method for extracting multiple directional lights. This work shows that spherical harmonics are a practical basis for light summation. We can project multiple lights into SH and extract a smaller number of directional lights that can closely approximate the projected set. A principled approach to aggregating less important lights is provided by the *lightcuts* [1832] framework.

Although most commonly used for irradiance, SH projections can be employed to simulate glossy, view-dependent BRDF lighting. Ramamoorthi and Hanrahan [1459] describe one such technique. Instead of a single color, they store in a cube map the coefficients of a spherical harmonic projection encoding the view-dependence of the environment map. However, in practice this technique requires much more space than the prefiltered environment map approaches we have seen earlier. Kautz et al. [869] derive a more economical solution using two-dimensional tables of SH coefficients, but this method is limited to fairly low-frequency lighting.

10.6.2 Other Representations

Although cube maps and spherical harmonics are the most popular representations for irradiance environment maps, other representations are possible. See Figure 10.45. Many irradiance environment maps have two dominant colors: a sky color on the top and a ground color on the bottom. Motivated by this observation, Parker et al. [1356] present a *hemisphere lighting* model that uses just two colors. The upper hemisphere is assumed to emit a uniform radiance L_{sky}, and the lower hemisphere is assumed to emit a uniform radiance L_{ground}. The irradiance integral for this case is

$$E = \begin{cases} \pi \left(\left(1 - \frac{1}{2} \sin\theta \right) L_{\text{sky}} + \frac{1}{2} \sin\theta L_{\text{ground}} \right), & \text{where } \theta < 90^\circ, \\ \pi \left(\frac{1}{2} \sin\theta L_{\text{sky}} + \left(1 - \frac{1}{2} \sin\theta \right) L_{\text{ground}} \right), & \text{where } \theta \geq 90^\circ, \end{cases} \tag{10.42}$$

Figure 10.45. Various ways of encoding irradiance. From left to right: the environment map and diffuse lighting computed via Monte Carlo integration for the irradiance; irradiance encoded with an ambient cube; spherical harmonics; spherical Gaussians; and H-basis (which can represent only a hemisphere of directions, so backfacing normals are not shaded). *(Images computed via the* Probulator *open-source software by Yuriy O'Donnell and David Neubelt.)*

where θ is the angle between the surface normal and the sky hemisphere axis. Baker and Boyd propose a faster approximation (described by Taylor [1752]):

$$E = \pi \left(\frac{1 + \cos\theta}{2} L_{\text{sky}} + \frac{1 - \cos\theta}{2} L_{\text{ground}} \right), \tag{10.43}$$

which is a linear interpolation between sky and ground, using $(\cos\theta + 1)/2$ as the interpolation factor. The term $\cos\theta$ is generally fast to compute as a dot product, and in the common case where the sky hemisphere axis is one of the cardinal axes (e.g., the y- or z-axis), it does not need to be computed at all, since it is equal to one of the world-space coordinates of $\mathbf{n}$. The approximation is reasonably close and significantly faster, so it is preferable to the full expression for most applications.

Forsyth [487] presents an inexpensive and flexible lighting model called the *trilight*, which includes directional, bidirectional, hemispherical, and wrap lighting as special cases.

Valve originally introduced the ambient cube representation (Section 10.3.1) for irradiance. In general, all the spherical function representations we have seen in Section 10.3 can be employed for precomputed irradiance. For the low-frequency signals that irradiance functions represent, we know SH is a good approximation. We tend to create special methods to simplify or use less storage than spherical harmonics.

More complex representations for high frequencies are needed if we want to evaluate occlusions and other global illumination effects, or if we want to incorporate glossy reflections (Section 10.1.1). The general idea of precomputing lighting to account for all interactions is called *precomputed radiance transport* (PRT) and will be discussed in Section 11.5.3. Capturing high frequencies for glossy lighting is also referred to as *all-frequency* lighting. Wavelet representations are often used in this context [1059] as means of compressing environment maps and to devise efficient operators in a simi-

lar fashion to the ones we have seen for spherical harmonics. Ng et al. [1269, 1270] demonstrate the use of *Haar wavelets* to generalize irradiance environment mapping to model self-shadowing. They store both the environment map and the shadowing function, which varies over the object surface, in the wavelet basis. This representation is of note because it amounts to a transformation of an environment cube map, performing a two-dimensional wavelet projection of each of the cube faces. Thus, it can be seen as a compression technique for cube maps.

10.7 Sources of Error

To correctly perform shading, we have to evaluate integrals over non-punctual light sources. In practice, this requirement means there are many different techniques we can employ, based on the properties of the lights under consideration. Often real-time engines model a few important lights analytically, approximating integrals over the light area and computing occlusion via shadow maps. All the other light sources—distant lighting, sky, fill lights, and light bouncing over surfaces—are often represented by environment cube maps for the specular component, and spherical bases for diffuse irradiance.

Employing a mix of techniques for lighting means that we are never working directly with a given BRDF model, but with approximations that have varying degrees of error. Sometimes the BRDF approximation is explicit, as we fit intermediate models in order to compute lighting integrals—LTCs are an example. Other times we build approximations that are exact for a given BRDF under certain (often rare) conditions, but are subject to errors in general—prefiltered cube maps fall into this category.

An important aspect to take into consideration when developing real-time shading models is to make sure that the discrepancies between different forms of lighting are not evident. Having coherent light results from different representations might even be more important, visually, than the absolute approximation error committed by each.

Occlusions are also of key importance for realistic rendering, as light "leaking" where there should be none is often more noticeable than not having light where it should be. Most area light representations are not trivial to shadow. Today none of the existing real-time shadowing techniques, even when accounting for "softening" effects (Section 7.6), can accurately consider the light shape. We compute a scalar factor that we multiply to reduce the contribution of a given light when an object casts a shadow, which is not correct; we should take this occlusion into account while performing the integral with the BRDF. The case of environment lighting is particularly hard, as we do not have a defined, dominant light direction, so shadowing techniques for punctual light sources cannot be used.

Even if we have seen some fairly advanced lighting models, it is important to remember that these are not exact representations of real-world sources of illumination. For example, in the case of environment lighting, we assume infinitely distant

Figure 10.46. Production lighting. *(Trailer Park 5. Archival pigment print, 17x22 inches. Production stills from Gregory Crewdson's Beneath the Roses series. ©Gregory Crewdson. Courtesy Gagosian.)*

radiance sources, ones that are never possible. All the analytic lights we have seen work on an even stronger assumption, that the lights emit radiance uniformly over the outgoing hemisphere for each point on their surface. In practice, this assumption can be a source of error, as often real lights are strongly directional. In photographic and cinematic lighting, specially crafted masks and filters called *gobos*, *cuculoris*, or *cookies* are often employed for artistic effect. See for example the sophisticated cinematographic lighting in Figure 10.46, by photographer Gregory Crewdson. To restrict lighting angles while keeping a large area of emission, grids of shielding black material called *honeycombs* can be added in front of large light-emitting panels (so-called *softboxes*). Complex configurations of mirrors and reflectors can also be used in the light's housing, such as in interior lighting, automotive headlights, and flashlights. See Figure 10.47. These optical systems create one or more virtual emitters far from the physical center radiating light, and this offset should be considered when performing falloff computations.

Note that these errors should always be evaluated in a perceptual, result-oriented framework (unless our aim is to do predictive rendering, i.e., to reliably simulate the real-world appearance of surfaces). In the hands of the artists, certain simplifications, even if not realistic, can still result in useful and expressive primitives. Physical models are useful when they make it simpler for artists to create visually plausible images, but they are not a goal in their own.

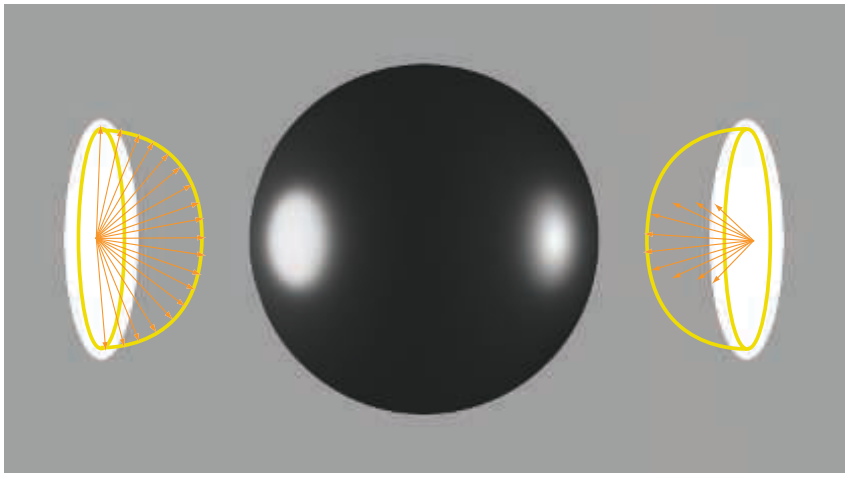

Figure 10.47. The same disk light with two different emission profiles. Left: each point on the disk emits light uniformly over the outgoing hemisphere. Right: emission is focused in a lobe around the disk normal.

Further Reading and Resources

The book *Light Science and Magic: An Introduction to Photographic Lighting* by Hunter [791] is a great reference for understanding real-world photographic lights. For movie lighting *Set Lighting Technician's Handbook: Film Lighting Equipment, Practice, and Electrical Distribution* [188] is a great introduction.

The work pioneered by Debevec in the area of image-based lighting is of great interest to anyone who needs to capture environment maps from real-world scenes. Much of this work is covered in a SIGGRAPH 2003 course [333], as well as in the book *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting* by Reinhard et al. [1479].

One resource that can help simulation are light profiles. The Illuminating Engineering Society (IES) publishes a handbook and file format standard for lighting measurements [960, 961]. Data in this format is commonly available from many manufacturers. The IES standard is limited to describing lights by only their angular emission profile. It does not fully model the effect on the falloff due to optical systems, nor the emission over the light surface area.

Szirmay-Kalos's [1732] state-of-the-art report on specular effects includes many references to environment mapping techniques.

Chapter 11
Global Illumination

"If it looks like computer graphics,
it is not good computer graphics."
—Jeremy Birn

Radiance is the final quantity computed by the rendering process. So far, we have been using the *reflectance equation* to compute it:

$$L_o(\mathbf{p}, \mathbf{v}) = \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_i(\mathbf{p}, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}, \tag{11.1}$$

where $L_o(\mathbf{p}, \mathbf{v})$ is the outgoing radiance from the surface location $\mathbf{p}$ in the view direction $\mathbf{v}$, Ω is the hemisphere of directions above $\mathbf{p}$, $f(\mathbf{l}, \mathbf{v})$ is the BRDF evaluated for $\mathbf{v}$ and the current incoming direction $\mathbf{l}$, $L_i(\mathbf{p}, \mathbf{l})$ is the incoming radiance into $\mathbf{p}$ from $\mathbf{l}$, and $(\mathbf{n} \cdot \mathbf{l})^+$ is the dot product between $\mathbf{l}$ and $\mathbf{n}$, with negative values clamped to zero.

11.1 The Rendering Equation

The reflectance equation is a restricted special case of the full *rendering equation*, presented by Kajiya in 1986 [846]. Different forms have been used for the rendering equation. We will use this version:

$$L_o(\mathbf{p}, \mathbf{v}) = L_e(\mathbf{p}, \mathbf{v}) + \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}, \tag{11.2}$$

where the new elements are $L_e(\mathbf{p}, \mathbf{v})$, which is the emitted radiance from the surface location $\mathbf{p}$ in direction $\mathbf{v}$, and the following replacement:

$$L_i(\mathbf{p}, \mathbf{l}) = L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l}). \tag{11.3}$$

This term means that the incoming radiance into location $\mathbf{p}$ from direction $\mathbf{l}$ is equal to the outgoing radiance from some other point in the opposite direction $-\mathbf{l}$. In this case, the "other point" is defined by the *ray casting function* $r(\mathbf{p}, \mathbf{l})$. This function returns the location of the first surface point hit by a ray cast from $\mathbf{p}$ in direction $\mathbf{l}$. See Figure 11.1.

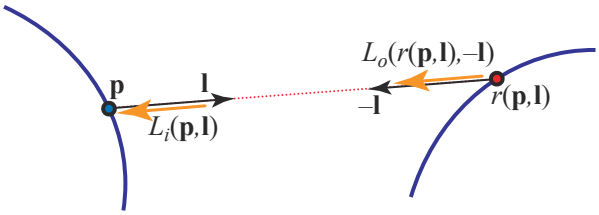

Figure 11.1. The shaded surface location $\mathbf{p}$, lighting direction $\mathbf{l}$, ray casting function $r(\mathbf{p},\mathbf{l})$, and incoming radiance $L_i(\mathbf{p},\mathbf{l})$, also represented as $L_o(r(\mathbf{p},\mathbf{l}),-\mathbf{l})$.

The meaning of the rendering equation is straightforward. To shade a surface location $\mathbf{p}$, we need to know the outgoing radiance L_o leaving $\mathbf{p}$ in the view direction $\mathbf{v}$. This is equal to the emitted radiance L_e plus the reflected radiance. Emission from light sources has been studied in previous chapters, as has reflectance. Even the ray casting operator is not as unfamiliar as it may seem. The z-buffer computes it for rays cast from the eye into the scene, for example.

The only new term is $L_o(r(\mathbf{p},\mathbf{l}),-\mathbf{l})$, which makes explicit the fact that the incoming radiance into one point must be outgoing from another point. Unfortunately, this is a recursive term. That is, it is computed by yet another summation over outgoing radiance from locations $r(r(\mathbf{p},\mathbf{l}),\mathbf{l}')$. These in turn need to compute the outgoing radiance from locations $r(r(r(\mathbf{p},\mathbf{l}),\mathbf{l}'),\mathbf{l}'')$, ad infinitum. It is amazing that the real world can compute all this in real time.

We know this intuitively, that lights illuminate a scene, and the photons bounce around and at each collision are absorbed, reflected, and refracted in a variety of ways. The rendering equation is significant in that it sums up all possible paths in a simple-looking equation.

An important property of the rendering equation is that it is *linear* with respect to the emitted lighting. If we make the lights twice as strong, the result of the shading will be two times brighter. The response of the material for each light is also independent from other sources. That is, one light's presence does not affect the interaction of another light with the material.

In real-time rendering, it is common to use just a local lighting model. Only the surface data at visible points are needed to compute the lighting—and that is exactly what GPUs can most efficiently provide. Primitives are processed and rasterized independently, after which they are discarded. Results of the lighting calculations at point $\mathbf{a}$ cannot be accessed when performing calculations at point $\mathbf{b}$. Transparency, reflections, and shadows are examples of *global illumination* algorithms. They use information from other objects than the one being illuminated. These effects contribute greatly to increasing the realism in a rendered image, and provide cues that help the viewer to understand spatial relationships. At the same time, they are also complex to simulate and might require precomputations or rendering multiple passes that compute some intermediate information.

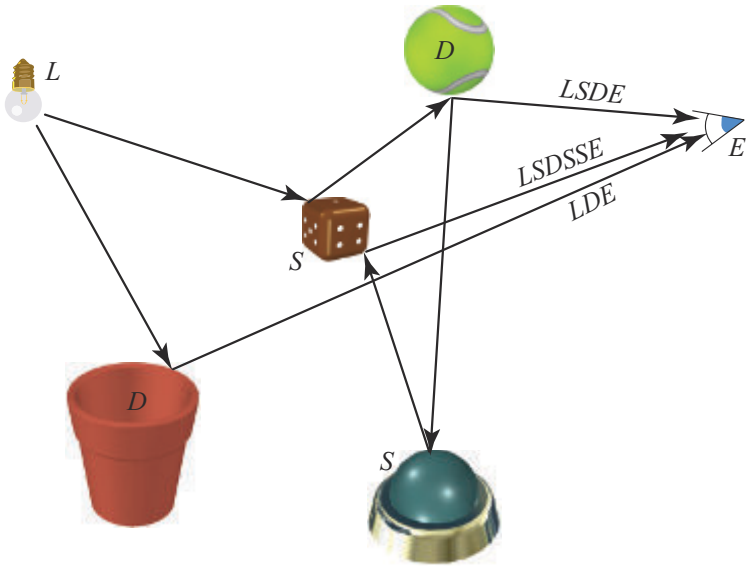

Figure 11.2. Some paths and their equivalent notation as they reach the eye. Note that two paths are shown continuing from the tennis ball.

One way to think of the problem of illumination is by the paths the photons take. In the local lighting model, photons travel from the light to a surface (ignoring intervening objects), then to the eye. Shadowing techniques take into account these intervening objects' direct occlusion effects. An environment map captures illumination traveling from light sources to distant objects, which is then applied to local shiny objects, which mirror-reflect this light to the eye. An irradiance map also captures the effect of light on distant objects, integrated in each direction over a hemisphere. The light reflected from all these objects is weighted and summed to compute the illumination for a surface, which in turn is seen by the eye.

Thinking about the different types and combinations of light transport paths in a more formal manner is helpful in understanding the various algorithms that exist. Heckbert [693] has a notational scheme that is useful for describing the paths simulated by a technique. Each interaction of a photon along its trip from the light (L) to the eye (E) can be labeled as diffuse (D) or specular (S). Categorization can be taken further by adding other surface types, such as "glossy," meaning shiny but not mirror-like. See Figure 11.2. Algorithms can be briefly summarized by regular expressions, showing what types of interactions they simulate. See Table 11.1 for a summary of basic notation.

Photons can take various paths from light to eye. The simplest path is LE, where a light is seen directly by the eye. A basic z-buffer is $L(D|S)E$, or equivalently, $LDE|LSE$. Photons leave the light, reach a diffuse or specular surface, and then

Operator	Description	Example	Explanation
$*$	zero or more	$S*$	zero or more specular bounces
$+$	one or more	$D+$	one or more diffuse bounces
?	zero or one	$S?$	zero or one specular bounces
|	either/or	$D\|SS$	either a diffuse or two specular bounces
()	group	$(D\|S)*$	zero or more of diffuse or specular

Table 11.1. Regular expression notation.

arrive at the eye. Note that with a basic rendering system, point lights have no physical representation. Giving lights geometry would yield a system $L(D|S)?E$, in which light can also then go directly to the eye.

If environment mapping is added to the renderer, a compact expression is a little less obvious. Though Heckbert's notation reads from light to eye, it is often easier to build up expressions going the other direction. The eye will first see a specular or diffuse surface, $(S|D)E$. If the surface is specular, it could also then, optionally, reflect a (distant) specular or diffuse surface that was rendered into the environment map. So, there is an additional potential path: $((S|D)?S|D)E$. To count in the path where the eye directly sees the light, add in a ? to this central expression to make it optional, and cap with the light itself: $L((S|D)?S|D)?E$.

This expression could be expanded to $LE|LSE|LDE|LSSE|LDSE$, which shows all the possible paths individually, or the shorter $L(D|S)?S?E$. Each has its uses in understanding relationships and limits. Part of the utility of the notation is in expressing algorithm effects and being able to build off of them. For example, $L(S|D)$ is what is encoded when an environment map is generated, and SE is the part that then accesses this map.

The rendering equation itself can be summarized by the simple expression $L(D|S)*E$, i.e., photons from the light can hit zero to nearly infinite numbers of diffuse or specular surfaces before reaching the eye.

Global illumination research focuses on methods for computing light transport along some of these paths. When applying it to real-time rendering, we are often willing to sacrifice some quality or correctness for efficient evaluation. The two most common strategies are to simplify and to precompute. For instance, we could assume that all the light bounces before the one reaching the eye are diffuse, a simplification that can work well for some environments. We could also precalculate some information about inter-object effects offline, such as generating textures that record illumination levels on surfaces, then in real time perform only basic calculations that rely on these stored values. This chapter will show examples of how these strategies can be used to achieve a variety of global illumination effects in real time.

Figure 11.3. Path tracing can generate photorealistic images, but is computationally expensive. The above image uses over two thousand paths per pixel, and each path is up to 64 segments long. It took over two hours to render, and it still exhibits some minor noise. *("Country Kitchen" model by Jay-Artist, Benedikt Bitterli Rendering Resources, licensed under CC BY 3.0 [149]. Rendered using the Mitsuba renderer.)*

11.2 General Global Illumination

Previous chapters have focused on various ways of solving the reflectance equation. We assumed a certain distribution of incoming radiance, L_i, and analyzed how it affected the shading. In this chapter we present algorithms that are designed to solve the full rendering equation. The difference between the two is that the former ignores where the radiance comes from—it is simply given. The latter states this explicitly: Radiance arriving at one point is the radiance that was emitted or reflected from other points.

Algorithms that solve the full rendering equation can generate stunning, photorealistic images (Figure 11.3). Those methods, however, are too computationally expensive for real-time applications. So, why discuss them? The first reason is that in static or partially static scenes, such algorithms can be run as a preprocess, storing the results for later use during rendering. This is a common approach in games, for example, and we will discuss different aspects of such systems.

The second reason is that global illumination algorithms are built on rigorous theoretical foundations. They are derived directly from the rendering equation, and any approximation they make is meticulously analyzed. A similar type of reasoning can and should be applied when designing real-time solutions. Even when we make certain shortcuts, we should be aware what the consequences are and what is the

correct way. As graphics hardware becomes more powerful, we will be able to make fewer compromises and create real-time rendered images that are closer to correct physical results.

Two common ways of solving the rendering equation are finite element methods and Monte Carlo methods. Radiosity is an algorithm based on the first approach; ray tracing in its various forms uses the second. Of the two, ray tracing is far more popular. This is mainly because it can efficiently handle general light transport—including effects such as volumetric scattering—all within the same framework. It also scales and parallelizes more easily.

We will briefly describe both approaches, but interested readers should refer to any of the excellent books that cover the details of solving the rendering equation in a non-real-time setting [400, 1413].

11.2.1 Radiosity

Radiosity [566] was the first computer graphics technique developed to simulate bounced light between diffuse surfaces. It gets its name from the quantity that is computed by the algorithm. In the classic form, radiosity can compute interreflections and soft shadows from area lights. There have been whole books written about this algorithm [76, 275, 1642], but the basic idea is relatively simple. Light bounces around an environment. You turn a light on and the illumination quickly reaches equilibrium. In this stable state, each surface can be considered as a light source in its own right. Basic radiosity algorithms make the simplifying assumption that all indirect light is from diffuse surfaces. This premise fails for places with polished marble floors or large mirrors on the walls, but for many architectural settings this is a reasonable approximation. Radiosity can follow an effectively unlimited number of diffuse bounces. Using the notation introduced at the beginning of this chapter, its light transport set is $LD*E$.

Radiosity assumes that each surface is composed of some number of patches. For each of these smaller areas, it computes a single, average radiosity value, so these patches need to be small enough to capture all the details of the lighting (e.g., shadow edges). However, they do not need to match the underlying surface triangles one to one, or even be uniform in size.

Starting from the rendering equation, we can derive that the radiosity for patch i is equal to

$$B_i = B_i^e + \rho_{ss} \sum_j F_{ij} B_j, \tag{11.4}$$

where B_i denotes the radiosity of patch i, B_i^e is the radiant *exitance*, i.e., the radiosity emitted by patch i, and ρ_{ss} is the subsurface albedo (Section 9.3). Emission is nonzero only for light sources. F_{ij} is the *form factor* between patches i and j. The form factor is defined as

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} V(\mathbf{i}, \mathbf{j}) \frac{\cos\theta_i \cos\theta_j}{\pi d_{ij}^2} da_i da_j, \tag{11.5}$$

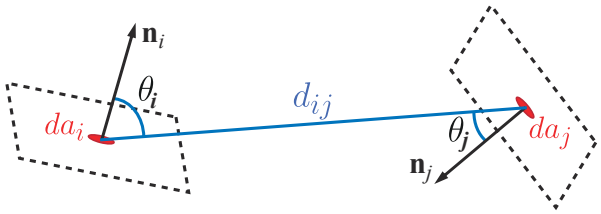

Figure 11.4. The form factor between two surface points.

where A_i is the area of patch i, and $V(\mathbf{i}, \mathbf{j})$ is the visibility function between points **i** and **j**, equal to one if there is nothing blocking light between them and zero otherwise. The values θ_i and θ_j are the angles between the two patch normals and the ray connecting points **i** and **j**. Finally, d_{ij} is the length of the ray. See Figure 11.4.

The form factor is a purely geometric term. It is the fraction of uniform diffuse radiant energy leaving patch i that is incident upon patch j [399]. The area, distance, and orientations of both patches, along with any surfaces that come between them, affect their form factor value. Imagine a patch represented by, say, a computer monitor. Every other patch in the room will directly receive some fraction of any light emitted from the monitor. This fraction could be zero, if the surface is behind or cannot "see" the monitor. These fractions all add up to one. A significant part of the radiosity algorithm is accurately determining the form factors between pairs of patches in the scene.

With the form factors computed, equations for all the patches (Equation 11.4) are combined into a single linear system. The system is then solved, resulting in a radiosity value for every patch. As the number of patches rises, the costs of reducing such a matrix is considerable, due to high computational complexity.

Because the algorithm scales poorly and has other limitations, classical radiosity is rarely used for generating lighting solutions. However, the idea of precomputing the form factors and using them at runtime to perform some form of light propagation is still popular among modern real-time global illumination systems. We will talk about these approaches later in the chapter (Section 11.5.3).

11.2.2 Ray Tracing

Ray casting is the process of firing a ray from a location to determine what objects are in a particular direction. *Ray tracing* uses rays to determine the light transport between various scene elements. In its most basic form, rays are shot from the camera through the pixel grid into the scene. For each ray, the closest object is found. This intersection point is then checked for whether it is in shadow by shooting a ray to each light and finding if any objects are in between. Opaque objects block the light; transparent objects attenuate it. Other rays can be spawned from an intersection point. If a surface is shiny, a ray is generated in the reflection direction. This ray

picks up the color of the first object intersected, which in turn has its intersection point tested for shadows. Rays can also be generated in the direction of refraction for transparent solid objects, again recursively evaluated. This basic mechanism is so simple that functional ray tracers have been written that fit on the back of a business card [696].

Classical ray tracing can provide only limited set of effects: sharp reflections and refractions, and hard shadows. However, the same underlying principle can be used to solve the full rendering equation. Kajiya [846] realized that the mechanism of shooting rays and evaluating how much light they carry can be used to compute the integral in Equation 11.2. The equation is recursive, which means that for every ray, we need to evaluate the integral again, at a different position. Fortunately, solid mathematical foundations for handling this problem already existed. *Monte Carlo* methods, developed for physics experiments during the Manhattan Project, were designed specifically to deal with this class of problems. Instead of directly computing the value of the integral in each shading point via quadrature rules, the integrand is evaluated at a number of random points from the domain. These values are then used to compute the estimate for the value of the integral. The more sampling points, the higher the accuracy. The most important property of this method is that only point evaluations of the integrand are needed. Given enough time, we can compute the integral with arbitrary precision. In the context of rendering, this is exactly what ray tracing provides. When we shoot the ray, we point-sample the integrand from Equation 11.2. Even though there is another integral to be evaluated at the intersection point, we do not need its final value, we can just point-sample it again. As the ray bounces across the scene, a *path* is built. The light carried along each path provides one evaluation of the integrand. This procedure is called *path tracing* (Figure 11.5).

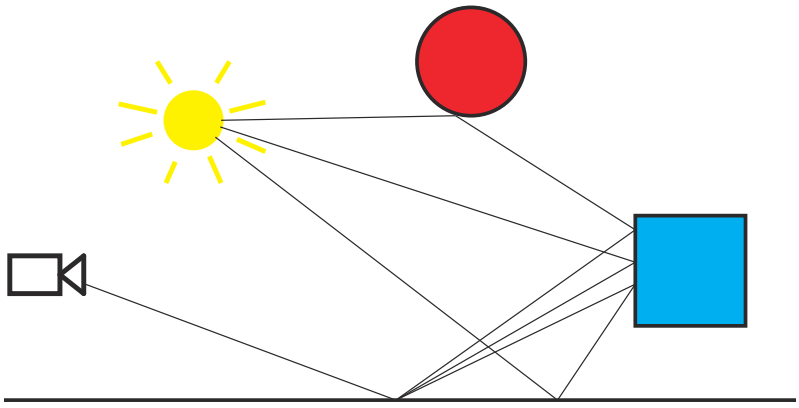

Figure 11.5. Example paths generated by a path tracing algorithm. All three paths pass through the same pixel in the film plane and are used to estimate its brightness. The floor at the bottom of the figure is highly glossy and reflects the rays within a small solid angle. The blue box and red sphere are diffuse and so scatter the rays uniformly around the normal at the point of intersection.

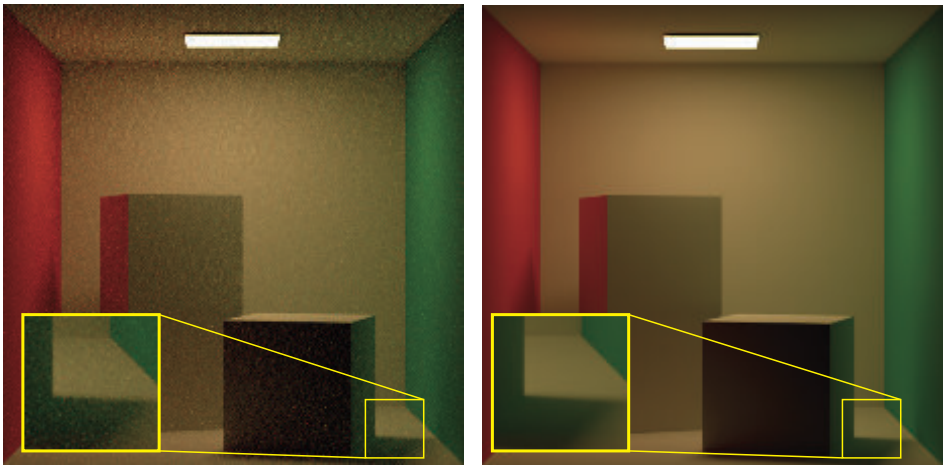

Figure 11.6. Noise resulting from using Monte Carlo path tracing with an insufficient number of samples. The image on the left was rendered with 8 paths per pixel, and the one on the right with 1024 paths per pixel. *("Cornell Box" model from Benedikt Bitterli Rendering Resources, licensed under CC BY 3.0 [149]. Rendered using the Mitsuba renderer.)*

Tracing paths is an extremely powerful concept. Paths can be used for rendering glossy or diffuse materials. Using them, we can generate soft shadows and render transparent objects along with caustic effects. After extending path tracing to sample points in volumes, not just surfaces, it can handle fog and subsurface scattering effects.

The only downside of path tracing is the computational complexity required to achieve high visual fidelity. For cinematic quality images, billions of paths may need to be traced. This is because we never compute the actual value of the integral, only its estimate. If too few paths are used, this approximation will be imprecise, sometimes considerably so. Additionally, results can be vastly different even for points that are next to each other, for which one would expect the lighting to be almost the same. We say that such results have *high variance*. Visually, this manifests itself as noise in the image (Figure 11.6). Many methods have been proposed to combat this effect without tracing additional paths. One popular technique is *importance sampling*. The idea is that variance can be greatly reduced by shooting more rays in the directions from which most of the light comes.

Many papers and books have been published on the subject of path tracing and related methods. Pharr et al. [1413] provide a great introduction to modern offline ray-tracing-based techniques. Veach [1815] lays down the mathematical foundations for modern reasoning about light transport algorithms. We discuss ray and path tracing at interactive rates at the end of this chapter, in Section 11.7.

11.3 Ambient Occlusion

The general global illumination algorithms covered in the previous section are computationally expensive. They can produce a wide range of complex effects, but it can take hours to generate an image. We will begin our exploration of the real-time alternatives with the simplest, yet still visually convincing, solutions and gradually build up to more complex effects throughout the chapter.

One basic global illumination effect is *ambient occlusion* (AO). The technique was developed in the early 2000s by Landis [974], at Industrial Light & Magic, to improve the quality of the environment lighting used on the computer-generated planes in the movie *Pearl Harbor*. Even though the physical basis for the effect includes a fair number of simplifications, the results look surprisingly plausible. This method inexpensively provides cues about shape when lighting lacks directional variation and cannot bring out object details.

11.3.1 Ambient Occlusion Theory

The theoretical background for ambient occlusion can be derived directly from the reflectance equation. For simplicity, we will first focus on Lambertian surfaces. The outgoing radiance L_o from such surfaces is proportional to the surface irradiance E. Irradiance is the cosine-weighted integral of the incoming radiance. In general, it depends on the surface position $\mathbf{p}$ and surface normal $\mathbf{n}$. Again, for simplicity, we will assume that the incoming radiance is constant, $L_i(\mathbf{l}) = L_A$, for all incoming directions $\mathbf{l}$. This results in the following equation for computing irradiance:

$$E(\mathbf{p}, \mathbf{n}) = \int_{\mathbf{l} \in \Omega} L_A (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} = \pi L_A, \tag{11.6}$$

where the integration is performed over the hemisphere Ω of possible incoming directions. With the assumption of constant uniform illumination, the irradiance (and, as a result, the outgoing radiance) does not rely on the surface position or normal, and is constant across the object. This leads to a flat appearance.

Equation 11.6 does not take any visibility into account. Some directions may be blocked by other parts of the object or by other objects in the scene. These directions will have different incoming radiance, not L_A. For simplicity, we will assume that the incoming radiance from the blocked directions is zero. This ignores all the light that might bounce off other objects in the scene and eventually reach point $\mathbf{p}$ from such blocked directions, but it greatly simplifies the reasoning. As a result, we get the following equation, first proposed by Cook and Torrance [285, 286]:

$$E(\mathbf{p}, \mathbf{n}) = L_A \int_{\mathbf{l} \in \Omega} v(\mathbf{p}, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}, \tag{11.7}$$

where $v(\mathbf{p}, \mathbf{l})$ is a visibility function that equals zero if a ray cast from $\mathbf{p}$ in the direction of $\mathbf{l}$ is blocked, and one if it is not.

Figure 11.7. An object rendered with only constant ambient lighting (left) and with ambient occlusion (right). Ambient occlusion brings out object details even when the lighting is constant. *("Dragon" model by Delatronic, Benedikt Bitterli Rendering Resources, licensed under CC BY 3.0 [149]. Rendered using the Mitsuba renderer.)*

The normalized, cosine-weighted integral of the visibility function is called ambient occlusion:

$$k_A(\mathbf{p}) = \frac{1}{\pi} \int_{\mathbf{l} \in \Omega} v(\mathbf{p}, \mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}. \tag{11.8}$$

It represents a cosine-weighted percentage of the unoccluded hemisphere. Values range from zero, for fully occluded surface points, to one, for locations with no occlusion. It is worth noting that convex objects, such as spheres or boxes, do not cause occlusion on themselves. If no other objects exist in the scene, a convex object will have an ambient occlusion value of one everywhere. If the object has any concavities, occlusion will be less than one in these areas.

Once k_A is defined, the equation for ambient irradiance in the presence of occlusion is

$$E(\mathbf{p}, \mathbf{n}) = k_A(\mathbf{p})\pi L_A. \tag{11.9}$$

Note that now the irradiance does change with surface location, because k_A does. This leads to much more realistic results, as seen in Figure 11.7 on the right. Surface locations in sharp creases will be dark since their value of k_A is low. Compare surface locations $\mathbf{p}_0$ and $\mathbf{p}_1$ in Figure 11.8. The surface orientation also has an effect, since the visibility function $v(\mathbf{p}, \mathbf{l})$ is weighted by a cosine factor when integrated. Compare $\mathbf{p}_1$ to $\mathbf{p}_2$ in the left side of the figure. Both have an unoccluded solid angle of about the same size, but most of the unoccluded region of $\mathbf{p}_1$ is around its surface normal, so the cosine factor is relatively high, as can be seen by the brightness of the arrows. In contrast, most of the unoccluded region of $\mathbf{p}_2$ is off to one side of the surface normal, with correspondingly lower values for the cosine factor. For this reason, the value of k_A is lower at $\mathbf{p}_2$. From here on, for brevity we will cease to explicitly show dependence on the surface location $\mathbf{p}$.

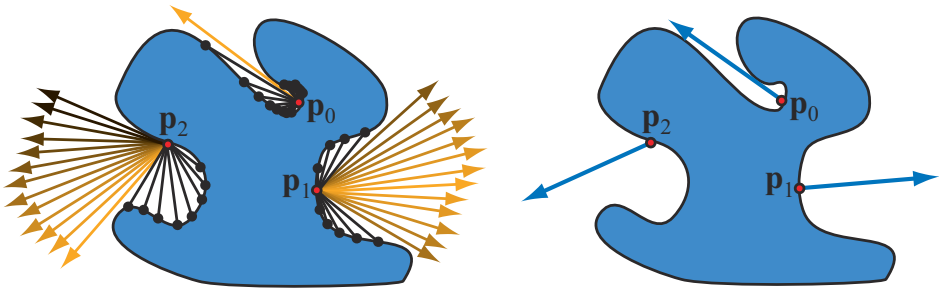

Figure 11.8. An object under ambient illumination. Three points ($\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$) are shown. On the left, blocked directions are shown as black rays ending in intersection points (black circles). Unblocked directions are shown as arrows, colored according to the cosine factor, so that those closer to the surface normal are lighter. On the right, each blue arrow shows the average unoccluded direction or bent normal.

In addition to k_A, Landis [974] also computes an average unoccluded direction, known as a *bent normal*. This direction vector is computed as a cosine-weighted average of unoccluded light directions:

$$\mathbf{n}_{\text{bent}} = \frac{\int_{\mathbf{l}\in\Omega} \mathbf{l}\, v(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}}{\left\|\int_{\mathbf{l}\in\Omega} \mathbf{l}\, v(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}\right\|}. \tag{11.10}$$

The notation $\|\mathbf{x}\|$ indicates the length of the vector $\mathbf{x}$. The result of the integral is divided by its own length to produce a normalized result. See the right side of Figure 11.8. The resulting vector can be used instead of the geometric normal during shading to provide more accurate results, at no extra performance cost (Section 11.3.7).

11.3.2 Visibility and Obscurance

The visibility function $v(\mathbf{l})$ used to compute the ambient occlusion factor k_A (Equation 11.8) needs to be carefully defined. For an object, such as a character or vehicle, it is straightforward to define $v(\mathbf{l})$ based on whether a ray cast from a surface location in direction $\mathbf{l}$ intersects any other part of the same object. However, this does not account for occlusion by other nearby objects. Often, the object can be assumed to be placed on a flat plane for lighting purposes. By including this plane in the visibility calculation, more realistic occlusion can be achieved. Another benefit is that the occlusion of the ground plane by the object can be used as a contact shadow [974].

Unfortunately, the visibility function approach fails for enclosed geometry. Imagine a scene consisting of a closed room containing various objects. All surfaces will have a k_A value of zero, since all rays from surfaces will hit something. Empirical approaches, which attempt to reproduce the look of ambient occlusion without necessarily simu-

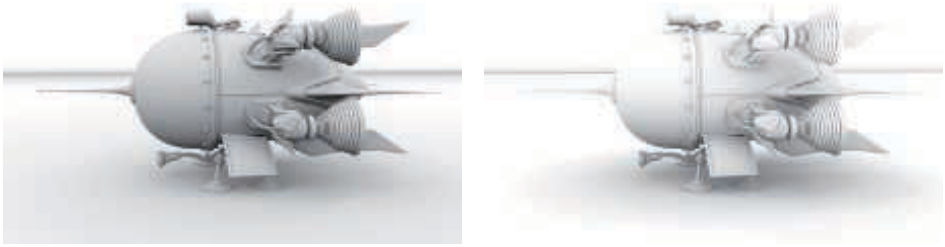

Figure 11.9. Difference between ambient occlusion and obscurance. Occlusion for the spaceship on the left was calculated using rays of infinite length. The image on the right uses finite length rays. *("4060.b Spaceship" model by thecali, Benedikt Bitterli Rendering Resources, licensed under CC BY 3.0 [149]. Rendered using the Mitsuba renderer.)*

lating physical visibility, often work better for such scenes. Some of these approaches were inspired by Miller's concept of *accessibility shading* [1211], which models how nooks and crannies in a surface capture dirt or corrosion.

Zhukov et al. [1970] introduced the idea of *obscurance*, which modifies the ambient occlusion computation by replacing the visibility function $v(\mathbf{l})$ with a distance mapping function $\rho(\mathbf{l})$:

$$k_A = \frac{1}{\pi} \int_{\mathbf{l} \in \Omega} \rho(\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}. \tag{11.11}$$

Unlike $v(\mathbf{l})$, which has only two valid values, 1 for no intersection and 0 for an intersection, $\rho(\mathbf{l})$ is a continuous function based on the distance the ray travels before intersecting a surface. The value of $\rho(\mathbf{l})$ is 0 at an intersection distance of 0 and 1 for any intersection distance greater than a specified distance $d_{\max}$, or when there is no intersection at all. Intersections beyond $d_{\max}$ do not need to be tested, which can considerably speed up the computation of k_A. Figure 11.9 shows the difference between ambient occlusion and ambient obscurance. Notice how the image rendered using ambient occlusion is considerably darker. This is because intersections are detected even at large distances and so affect the value of k_A.

Despite attempts to justify it on physical grounds, obscurance is not physically correct. However, it often gives plausible results that match the viewer's expectations. One drawback is that the value of $d_{\max}$ needs to be set by hand to achieve pleasing results. This type of compromise is often the case in computer graphics, where a technique has no direct physical basis but is "perceptually convincing." The goal is usually a believable image, so such techniques are fine to use. That said, some advantages of methods based on theory are that they can work automatically and can be improved further by reasoning about how the real world works.

Figure 11.10. Difference between ambient occlusion without and with interreflections. The left image uses only information about visibility. The image on the right also uses one bounce of indirect illumination. *("Victorian Style House" model by MrChimp2313, Benedikt Bitterli Rendering Resources, licensed under CC BY 3.0 [149]. Rendered using the Mitsuba renderer.)*

11.3.3 Accounting for Interreflections

Even though the results produced by ambient occlusion are visually convincing, they are darker than those produced by a full global illumination simulation. Compare the images in Figure 11.10.

A significant source of the difference between ambient occlusion and full global illumination are the interreflections. Equation 11.8 assumes that the radiance in the blocked directions is zero, while in reality interreflections will introduce a nonzero radiance from these directions. The effect of this can be seen as darkening in the creases and pits of the model on the left in Figure 11.10, compared to the model on the right. This difference can be addressed by increasing the value of k_A. Using the obscurance distance mapping function instead of the visibility function (Section 11.3.2) can also mitigate this problem, since the obscurance function often has values greater than zero for blocked directions.

Tracking interreflections in a more accurate manner is expensive, since it requires solving a recursive problem. To shade one point, other points must first be shaded, and so on. Computing the value of k_A is significantly less expensive than performing a full global illumination calculation, but it is often desirable to include this missing light in some form, to avoid overdarkening. Stewart and Langer [1699] propose an inexpensive, but surprisingly accurate, method to approximate interreflections. It is based on the observation that for Lambertian scenes under diffuse illumination, the surface locations visible from a given location tend to have similar radiance. By assuming that the radiance L_i from blocked directions is equal to the outgoing radiance L_o from the currently shaded point, the recursion is broken and an analytical expression can be found:

$$E = \frac{\pi k_A}{1 - \rho_{\text{ss}}(1 - k_A)} L_i, \tag{11.12}$$

where ρ_{ss} is the subsurface albedo, or diffuse reflectance. This is equivalent to replacing

the ambient occlusion factor k_A with a new factor k'_A:

$$k'_A = \frac{k_A}{1 - \rho_{\text{ss}}(1 - k_A)}. \tag{11.13}$$

This equation will tend to brighten the ambient occlusion factor, making it visually closer to the result of a full global illumination solution, including interreflections. The effect is highly dependent on the value of ρ_{ss}. The underlying approximation assumes that the surface color is the same in the proximity of the shaded point, to produce an effect somewhat like color bleeding. Hoffman and Mitchell [755] use this method to illuminate terrain with sky light.

A different solution is presented by Jimenez et al. [835]. They perform full, offline path tracing for a number of scenes, each lit by a uniformly white, infinitely distant environment map to obtain occlusion values that properly take interreflections into account. Based on these examples, they fit cubic polynomials to approximate the function f that maps from the ambient occlusion value k_A and subsurface albedo ρ_{ss} to the occlusion value k'_A, which is brightened by the interreflected light. Their method also assumes that the albedo is locally constant, and the color of the incident bounced light can be derived based on the albedo at a given point.

11.3.4 Precomputed Ambient Occlusion

Calculation of the ambient occlusion factors can be time consuming and is often performed offline, prior to rendering. The process of precomputing any lighting-related information, including ambient occlusion, is often called *baking*.

The most common way of precomputing ambient occlusion is via Monte Carlo methods. Rays are cast and checked for intersections with the scene and Equation 11.8 is evaluated numerically. For example, say we pick N random directions $\mathbf{l}$ uniformly distributed over the hemisphere around the normal $\mathbf{n}$, and trace rays in these directions. Based on the intersection results, we evaluate the visibility function v. The ambient occlusion can then be computed as

$$k_A = \frac{1}{N} \sum_{i}^{N} v(\mathbf{l}_i)(\mathbf{n} \cdot \mathbf{l}_i)^+. \tag{11.14}$$

When calculating ambient obscurance, the cast rays can be restricted to a maximum distance, and the value of v is based on the intersection distance found.

The computation of ambient occlusion or obscurance factors includes a cosine weighting factor. While it can be included directly, as in Equation 11.14, the more efficient way to incorporate this weighting factor is by means of importance sampling. Instead of casting rays uniformly over the hemisphere and cosine-weighting the results, the distribution of ray directions is cosine weighted. In other words, rays are more likely to be cast closer to the direction of the surface normal, since the results from such directions are likely to matter more. This sampling scheme is called *Malley's method*.

Ambient occlusion precomputations can performed on the CPU or on the GPU. In both cases, libraries that accelerate ray casting against complex geometry are available. The two most popular are Embree [1829], for the CPU, and OptiX [951], for the GPU. In the past, results from the GPU pipeline, such as depth maps [1412] or occlusion queries [493], have also been used for computing ambient occlusion. With the growing popularity of more general ray casting solutions on GPUs, their use is less common today. Most commercially available modeling and rendering software packages provide an option to precompute ambient occlusion.

The occlusion data are unique for every point on the object. They are typically stored in textures, in volumes, or on mesh vertices. The characteristics and problems of the different storage methods are similar, regardless of the type of signal stored. The same methodologies can be used for storing ambient occlusion, directional occlusion, or precomputed lighting, as described in Section 11.5.4.

Precomputed data can also be used to model the ambient occlusion effects of objects on each other. Kontkanen and Laine [924, 925] store the ambient occlusion effect of an object on its surroundings in a cube map, called an *ambient occlusion field*. They model how the ambient occlusion value changes with distance from the object with a reciprocal of a quadratic polynomial. Its coefficients are stored in a cube map, to model directional variation of the occlusion. At runtime, the distance and relative position of the occluding object are used to fetch proper coefficients and reconstruct the occlusion value.

Malmer et al. [1111] show improved results by storing the ambient occlusion factors, and optionally the bent normal, in a three-dimensional grid called an *ambient occlusion volume*. The computation requirements are lower, since the ambient occlusion factor is read directly from the texture, not computed. Fewer scalars are stored compared to Kontkanen and Laine's approach, and the textures in both methods have low resolutions, so the overall storage requirements are similar. Hill [737] and Reed [1469] describe implementations of Malmer et al.'s method in commercial game engines. They discuss various practical aspects of the algorithm as well as useful optimizations. Both methods are meant to work for rigid objects, but they can be extended to articulated objects with small numbers of moving parts, where each part is treated as a separate object.

Whichever method we choose for storing ambient occlusion values, we need to be aware that we are dealing with a continuous signal. When we shoot rays from a particular point in space, we *sample*, and when we interpolate a value from these results before shading, we *reconstruct*. All tools from the signal processing field can be used to improve the quality of the sampling-reconstruction process. Kavan et al. [875] propose a method they call *least-squares baking*. The occlusion signal is sampled uniformly across the mesh. Next, the values for the vertices are derived, so that the total difference between the interpolated and sampled ones is minimized, in the least-squares sense. They discuss the method specifically in the context of storing data at the vertices, but the same reasoning can be used for deriving values to be stored in textures or volumes.

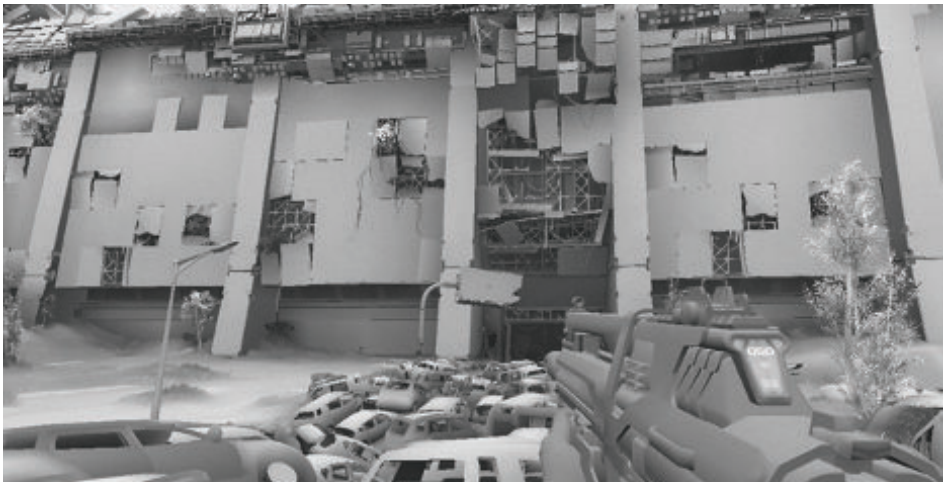

Figure 11.11. *Destiny* uses precomputed ambient occlusion in its indirect lighting calculations. The solution was used on game versions for two different hardware generations, providing high quality and performance. *(Image ©2013 Bungie, Inc. all rights reserved.)*

Destiny is an example of an acclaimed game that uses precomputed ambient occlusion as a basis for its indirect lighting solution (Figure 11.11). The game shipped during the transition period between two generations of console hardware and needed a solution that balanced the high quality expected on new platforms with the limitations, both in performance and memory use, of the older ones. The game features dynamic time of day, so any precomputed solution had to take this correctly into account. The developers chose ambient occlusion for its believable look and low cost. As ambient occlusion decouples the visibility calculations from lighting, the same precomputed data could be used regardless of the time of day. The complete system, including the GPU-based baking pipeline, is described by Sloan et al. [1658].

The *Assassin's Creed* [1692] and *Far Cry* [1154] series from Ubisoft also use a form of precomputed ambient occlusion to augment their indirect illumination solution. They render the world from a top-down view and process the resulting depth map to compute large-scale occlusion. Various heuristics are used to estimate the value based on the distribution of the neighboring depth samples. The resulting world-space AO map is applied to all objects, by projecting their world-space position to texture space. They call this method *World AO*. A similar approach is also described by Swoboda [1728].

11.3.5 Dynamic Computation of Ambient Occlusion

For static scenes, the ambient occlusion factor k_A and bent normal $\mathbf{n}_{\text{bent}}$ can be precomputed. However, for scenes where objects are moving or changing shape, better results can be achieved by computing these factors on the fly. Methods for doing so

can be grouped into those that operate in object space, and those that operate in screen space.

Offline methods for computing ambient occlusion usually involve casting a large number of rays, dozens to hundreds, from each surface point into the scene and checking for intersection. This is a costly operation, and real-time methods focus on ways to approximate or avoid much of this computation.

Bunnell [210] computes the ambient occlusion factor k_A and bent normal $\mathbf{n}_{\text{bent}}$ by modeling the surface as a collection of disk-shaped elements placed at the mesh vertices. Disks were chosen since the occlusion of one disk by another can be computed analytically, avoiding the need to cast rays. Simply summing the occlusion factors of a disk by all the other disks leads to overly dark results due to double-shadowing. That is, if one disk is behind another, both will be counted as occluding the surface, even though only the closer of the two should be. Bunnell uses a clever two-pass method to avoid this problem. The first pass computes ambient occlusion including double-shadowing. In the second pass, the contribution of each disk is reduced by its occlusion from the first pass. This is an approximation, but in practice it yields results that are convincing.

Computing occlusion between each pair of elements is an order $O(n^2)$ operation, which is too expensive for all but the simplest scenes. The cost can be reduced by using simplified representations for distant surfaces. Bunnell constructs a hierarchical tree of elements, where each node is a disk that represents the aggregation of the disks below it in the tree. When performing inter-disk occlusion computations, higher-level nodes are used for more distant surfaces. This reduces the computation to order $O(n \log n)$, which is much more reasonable. Bunnell's technique is quite efficient and produces high-quality results. For example, it was used in final renders for the *Pirates of the Caribbean* films [265].

Hoberock [751] proposes several modifications to Bunnell's algorithm that improve quality at a higher computational expense. He also presents a distance attenuation factor, which yields results similar to the obscurance factor proposed by Zhukov et al. [1970].

Evans [444] describes a dynamic ambient occlusion approximation method based on *signed distance fields* (SDF). In this representation, an object is embedded in a three-dimensional grid. Each location in the grid stores the distance to the closest surface of the object. This value is negative for points inside any object and positive for points outside all of them. Evans creates and stores an SDF for a scene in a volume texture. To estimate the occlusion for a location on an object, he uses a heuristic that combines values sampled at some number of points, progressively farther away from the surface, along the normal. The same approach can also be used when the SDF is represented analytically (Section 17.3) instead of stored in a three-dimensional texture, as described by Quílez [1450]. Although the method is non-physical, the results are visually pleasing.

Using signed distance fields for ambient occlusion was further extended by Wright [1910]. Instead of using an ad hoc heuristic to generate occlusion values,

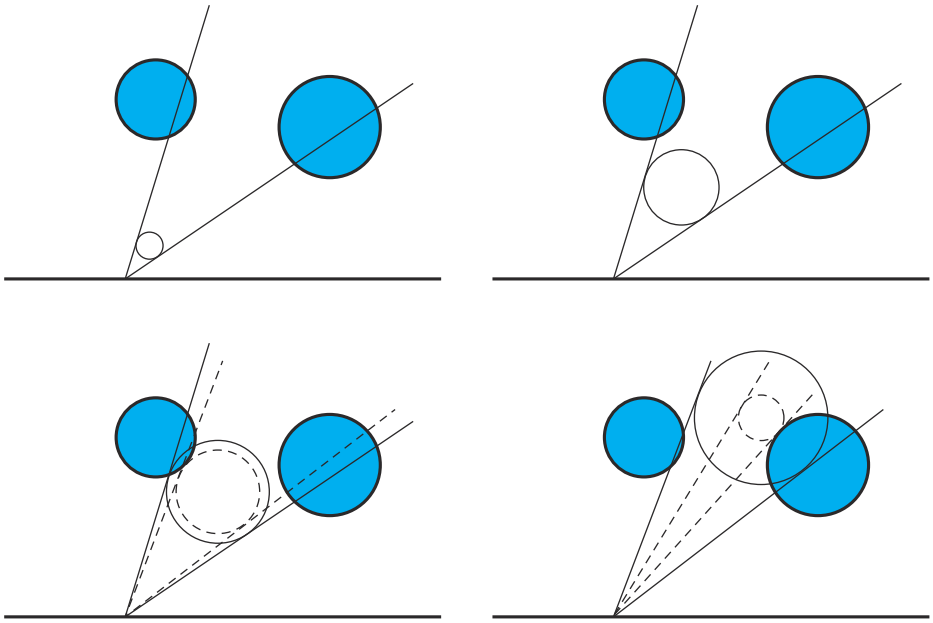

Figure 11.12. Cone tracing is approximated by performing a series of intersections between scene geometry and spheres of increasing radius. The size of the sphere corresponds to the radius of the cone at a given distance from the trace origin. In each step, the cone angle is reduced, to account for occlusion by the scene geometry. The final occlusion factor is estimated as a ratio of the solid angle subtended by the clipped cone to the solid angle of the original cone.

Wright performs *cone tracing*. Cones originate at the location that is being shaded, and are tested for intersections, with the scene representation encoded in the distance field. Cone tracing is approximated by performing a set of steps along the axis and checking for intersections of the SDF with a sphere of increasing radius at each step. If the distance to the nearest occluder (the value sampled from the SDF) is smaller than the sphere's radius, that part of the cone is occluded (Figure 11.12). Tracing a single cone is imprecise and does not allow incorporation of the cosine term. For these reasons, Wright traces a set of cones, covering the entire hemisphere, to estimate the ambient occlusion. To increase visual fidelity, his solution uses not only a global SDF for the scene, but also local SDFs, representing individual objects or logically connected sets of objects.

A similar approach is described by Crassin et al. [305] in the context of a voxel representation of the scene. They use a sparse voxel octree (Section 13.10) to store the voxelization of a scene. Their algorithm for computing ambient occlusion is a special case of a more general method for rendering full global illumination effects (Section 11.5.7).

Figure 11.13. Ambient occlusion effects are blurry and do not reveal details of the occluders. AO calculations can use much simpler representation of the geometry and still achieve plausible effects. The armadillo model (left) is approximated with a set of spheres (right). There is practically no difference in the occlusion cast by the models on the wall behind them. *(Model courtesy of the Stanford Computer Graphics Laboratory.)*

Ren et al. [1482] approximate the occluding geometry as a collection of spheres (Figure 11.13). The visibility function of a surface point occluded by a single sphere is represented using spherical harmonics. The aggregate visibility function for occlusion by a group of spheres is the result of multiplying the individual sphere visibility functions. Unfortunately, computing the product of spherical harmonic functions is an expensive operation. Their key idea is to sum the logarithms of the individual spherical harmonic visibility functions, and exponentiate the result. This produces the same end result as multiplying the visibility functions, but summation of spherical harmonic functions is significantly less expensive than multiplication. The paper shows that, with the right approximations, logarithms and exponentiations can be performed quickly, yielding an overall speedup.

The method computes not just an ambient occlusion factor, but rather a full spherical visibility function, represented with spherical harmonics (Section 10.3.2). The first (order 0) coefficient can be used as the ambient occlusion factor k_A, and the next three (order 1) coefficients can be used to compute the bent normal $\mathbf{n}_{\text{bent}}$. Higher-order coefficients can be used to shadow environment maps or circular light sources. Since geometry is approximated as bounding spheres, occlusion from creases and other small details is not modeled.

Sloan et al. [1655] perform the accumulation of the visibility functions described by Ren in screen space. For each occluder, they consider a set of pixels that are within some prescribed world-space distance from its center. This operation can be achieved by rendering a sphere and either performing the distance test in the shader or using stencil testing. For all the affected screen areas, a proper spherical harmonic value is added to an offscreen buffer. After accumulating visibility for all the occluders, the values in the buffer are exponentiated to get the final, combined visibility functions

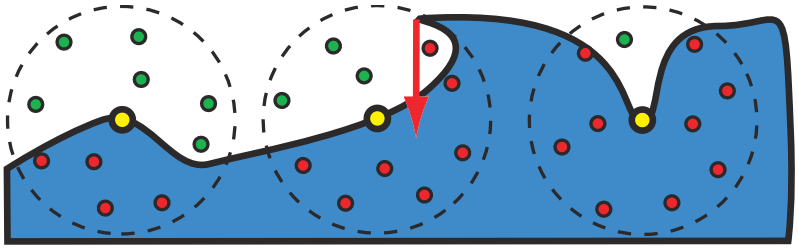

Figure 11.14. Crytek's ambient occlusion method applied to three surface points (the yellow circles). For clarity, the algorithm is shown in two dimensions, with the camera (not shown) above the figure. In this example, ten samples are distributed over a disk around each surface point (in actuality, they are distributed over a sphere). Samples failing the z-test, i.e., those beyond the stored z-buffer values, are shown in red, and samples passing are shown in green. The value of k_A is a function of the ratio of the passing samples to the total samples. We ignore the variable sample weights here for simplicity. The point on the left has 6 passing samples out of 10 total, resulting in a ratio of 0.6, from which k_A is computed. The middle point has three passing samples. One more is outside the object but fails the z-test, as shown by the red arrow. This gives a k_A of 0.3. The point on the right has one passing sample, so k_A is 0.1.

for each screen pixel. Hill [737] uses the same method, but restricts the spherical harmonics visibility functions to only second-order coefficients. With this assumption, the spherical harmonic product is just a handful of scalar multiplies and can be performed by even the GPU's fixed-function blending hardware. This allows us to use the method even on console hardware with limited performance. Because the method uses low-order spherical harmonics, it cannot be used to generate hard shadows with more defined boundaries, rather only mostly directionless occlusion.

11.3.6 Screen-Space Methods

The expense of object-space methods is proportional to scene complexity. However, some information about occlusion can be deduced purely from screen-space data that is already available, such as depth and normals. Such methods have a constant cost, not related to how detailed the scene is, but only to the resolution used for rendering.[1]

Crytek developed a dynamic *screen-space ambient occlusion* (SSAO) approach used in *Crysis* [1227]. They compute ambient occlusion in a full-screen pass, using the z-buffer as the only input. The ambient occlusion factor k_A of each pixel is estimated by testing a set of points, distributed in a sphere around the pixel's location, against the z-buffer. The value of k_A is a function of the number of samples that are in front of the corresponding values in the z-buffer. A smaller number of passing samples results in a lower value for k_A. See Figure 11.14. The samples have weights that decrease with distance from the pixel, similarly to the obscurance factor [1970]. Note that since the samples are not weighted by a $(\mathbf{n} \cdot \mathbf{l})^+$ factor, the resulting ambient occlusion is

[1] In practice, the execution time will depend on the distribution of the data in the depth or normal buffers, as this dispersal affects how effectively the occlusion calculation logic uses the GPU caches.

Figure 11.15. The effect of screen-space ambient occlusion is shown in the upper left. The upper right shows the albedo (diffuse color) without ambient occlusion. In the lower left, the two are shown combined. Specular shading and shadows are added for the final image, in the lower right. *(Images from "Crysis" courtesy of Crytek.)*

incorrect. Instead of considering only those samples in the hemisphere above a surface location, all are counted and factored in. This simplification means that samples below the surface are counted when they should not be. Doing so causes a flat surface to be darkened, with edges being brighter than their surroundings. Nonetheless, the results are often visually pleasing. See Figure 11.15.

A similar method was simultaneously developed by Shanmugam and Arikan [1615]. In their paper they describe two approaches. One generates fine ambient occlusion from small, nearby details. The other generates coarse ambient occlusion from larger objects. The results of the two are combined to produce the final ambient occlusion factor. Their fine-scale ambient occlusion method uses a full-screen pass that accesses the z-buffer, along with a second buffer containing the surface normals of the visible pixels. For each shaded pixel, nearby pixels are sampled from the z-buffer. The sampled pixels are represented as spheres, and an occlusion term is computed for the shaded pixel, taking its normal into account. Double-shadowing is not accounted for, so the result is somewhat dark. Their coarse occlusion method is similar to the object-space method of Ren et al. [1482], discussed on page 456, in that the occluding geometry is approximated as a collection of spheres. However, Shanmugam and Arikan accumulate occlusion in screen space, using screen-aligned billboards covering the

"area of effect" of each occluding sphere. Double-shadowing is also not accounted for in the coarse occlusion method, unlike the method of Ren et al.

The extreme simplicity of these two methods was quickly noticed by both industry and academia, and spawned numerous follow-up works. Many methods, such as that used in the game *Starcraft II* by Filion et al. [471] and *scalable ambient obscurance* by McGuire et al. [1174], use ad hoc heuristics to generate the occlusion factor. These kinds of methods have good performance characteristics and expose some parameters that can be hand-tweaked to achieve the desired artistic effect.

Other methods aim to provide more principled ways of calculating the occlusion. Loos and Sloan [1072] noticed that Crytek's method could be interpreted as Monte Carlo integration. They call the calculated value *volumetric obscurance* and define it as

$$v_A = \int_{\mathbf{x} \in X} \rho(d(\mathbf{x}))o(\mathbf{x})d\mathbf{x}, \tag{11.15}$$

where X is a three-dimensional, spherical neighborhood around the point, ρ is the distance-mapping function, analogous to that in Equation 11.11, d is the distance function, and $o(\mathbf{x})$ is the *occupancy function*, equal to zero if $\mathbf{x}$ is not occupied and one otherwise. They note that the $\rho(d)$ function has little impact on the final visual quality and so use a constant function. Given this assumption, volumetric obscurance is an integral of the occupancy function over the neighborhood of a point. Crytek's method samples the three-dimensional neighborhood randomly to evaluate the integral. Loos and Sloan compute the integral numerically in the xy-dimensions, by randomly sampling the screen-space neighborhood of a pixel. The z-dimension is integrated over analytically. If the spherical neighborhood of the point does not contain any geometry, the integral is equal to the length of the intersection between the ray and a sphere representing X. In the presence of geometry, the depth buffer is used as an approximation of the occupancy function, and the integral is computed over only the unoccupied part of each line segment. See the left side of Figure 11.16. The method generates results of quality comparable to Crytek's, but using fewer samples, as integration over one of the dimensions is exact. If the surface normals are available, the method can be extended to take them into account. In that version, the evaluation of the line integral is clamped at the plane defined by the normal at the evaluation point.

Szirmay-Kalos et al. [1733] present another screen-space approach that uses normal information, called *volumetric ambient occlusion*. Equation 11.6 performs the integration over a hemisphere around the normal and includes the cosine term. They propose that this type of integral can be approximated by removing the cosine term from the integrand and clamping the integration range with the cosine distribution. This transforms the integral to be over a sphere instead of a hemisphere, one with half the radius and shifted along the normal, to be fully enclosed within the hemisphere. The volume of its unoccupied portion is calculated just as in the method by Loos and Sloan, by randomly sampling the pixel neighborhood and analytically integrating the occupancy function in the z-dimension. See the right side of Figure 11.16.

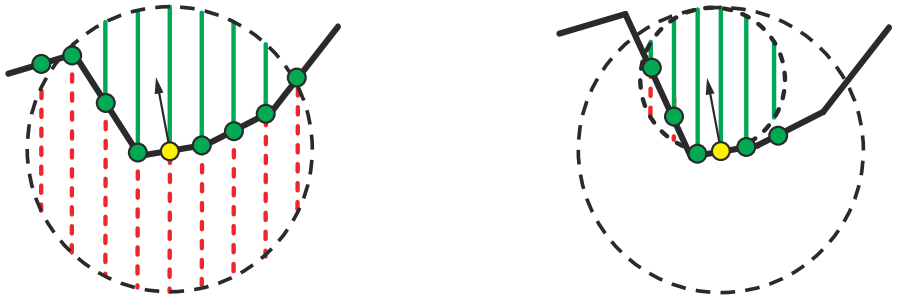

Figure 11.16. Volumetric obscurance (left) estimates the integral of the unoccupied volume around the point using line integrals. Volumetric ambient occlusion (right) also uses line integrals, but to compute the occupancy of the sphere tangent to the shading point, which models the cosine term from the reflectance equation. In both cases, the integral is estimated from the ratio of the unoccupied volume of the sphere (marked with solid green lines) to the total volume of the sphere (the sum of unoccupied and occupied volume, marked with dashed red lines). For both figures the camera is viewing from above. The green dots represent samples read from the depth buffer, and the yellow one is the sample for which the occlusion is being calculated.

A different approach to the problem of estimating local visibility was proposed by Bavoil et al. [119]. They draw inspiration from the *horizon mapping* technique by Max [1145]. Their method, called *horizon-based ambient occlusion* (HBAO), assumes that the data in the z-buffer represents a continuous heightfield. The visibility at a point can be estimated by determining *horizon angles*, the maximum angles above the tangent plane that are occluded by the neighborhood. That is, given a direction from a point, we record the angle of the highest object visible. If we ignore the cosine term, the ambient occlusion factor then can be computed as the integral of the unoccluded portions over the horizon, or, alternatively, as one minus the integral of the occluded parts under the horizon:

$$k_A = 1 - \frac{1}{2\pi} \int_{\phi=-\pi}^{\pi} \int_{\alpha=t(\phi)}^{h(\phi)} W(\omega) \cos(\theta) d\theta d\phi, \tag{11.16}$$

where $h(\phi)$ is the horizon angle above the tangent plane, $t(\phi)$ is the *tangent angle* between the tangent plane and the view vector, and $W(\omega)$ is the attenuation function. See Figure 11.17. The $\frac{1}{2\pi}$ term normalizes the integral so that the result is between zero and one.

By using a linear falloff on the distance to the point that defines the horizon for a given ϕ, we can compute the inner integral analytically:

$$k_A = 1 - \frac{1}{2\pi} \int_{\phi=-\pi}^{\pi} \big(\sin(h(\phi)) - \sin(t(\phi))\big) W(\phi) d\phi. \tag{11.17}$$

The remaining integral is computed numerically, by sampling a number of directions and finding horizon angles.

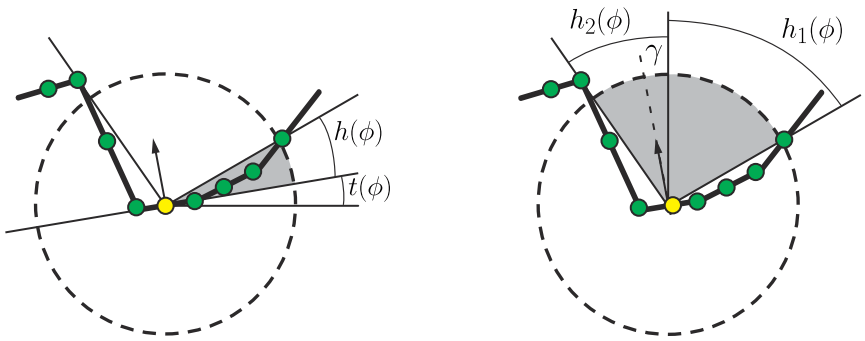

Figure 11.17. Horizon-based ambient occlusion (left) finds the horizon angles h above the tangent plane and integrates the unoccluded angle between them. The angle between the tangent plane and the view vector is denoted as t. Ground-truth ambient occlusion (right) uses the same horizon angles, h_1 and h_2, but also uses the angle between the normal and the view vector, γ, to incorporate the cosine term into the calculations. In both figures the camera is viewing the scene from above. The figures show cross sections, and horizon angles are functions of ϕ, the angle around the view direction. The green dots represent samples read from the depth buffer. The yellow dot is the sample for which the occlusion is being calculated.

The horizon-based approach is also used by Jimenez et al. [835] in a method they call *ground-truth ambient occlusion* (GTAO). Their aim is to achieve ground-truth results, matching those obtained from ray tracing, assuming that the only available information is the heightfield formed by the z-buffer data. Horizon-based ambient occlusion does not include the cosine term in its definition. It also adds the ad hoc attenuation, not present in Equation 11.8, so its results, even though close to those that are ray traced, are not the same. GTAO introduces the missing cosine factor, removes the attenuation function, and formulates the occlusion integral in the reference frame around the view vector. The occlusion factor is defined as

$$k_A = \frac{1}{\pi}\int_0^{\pi}\int_{h_1(\phi)}^{h_2(\phi)} \cos(\theta-\gamma)^+ |\sin(\theta)| d\theta d\phi, \tag{11.18}$$

where $h_1(\phi)$ and $h_2(\phi)$ are the left and right horizon angles for a given ϕ, and γ is the angle between the normal and the view direction. The normalization term $\frac{1}{\pi}$ is different than for HBAO because the cosine term is included. This makes the open hemisphere integrate to π. Not including the cosine term in the formula makes it integrate to 2π. This formulation matches Equation 11.8 exactly, given the heightfield assumption. See Figure 11.17. The inner integral can still be solved analytically, so only the outer one needs to be computed numerically. This integration is performed the same way as in HBAO, by sampling a number of directions around the given pixel.

The most expensive part of the process in horizon-based methods is sampling the depth buffer along screen-space lines to determine the horizon angles. Timonen [1771] presents a method that aims specifically to improve the performance characteristics

of this step. He points out that samples used for estimating the horizon angles for a given direction can be heavily reused between pixels that lay along straight lines in screen space. He splits the occlusion calculations into two steps. First, he performs line traces across the whole z-buffer. At each step of the trace, he updates the horizon angles as he moves along the line, accounting for the prescribed maximum influence distance, and writes this information out to a buffer. One such buffer is created for each screen-space direction used in horizon mapping. The buffers do not need to be the same size as the original depth buffer. Their size depends on the spacing between the lines, and the distance between the steps along the lines, and there is some flexibility in choosing these parameters. Different settings affect the final quality.

The second step is the calculation of the occlusion factors, based on the horizon information stored in the buffers. Timonen uses the occlusion factor defined by HBAO (Equation 11.17), but other occlusion estimators, such as GTAO (Equation 11.18), could be used instead.

A depth buffer is not a perfect representation of the scene, since only the closest object is recorded for a given direction and we do not know what occurs behind it. Many of the methods use various heuristics to try to infer some information about the thickness of the visible objects. These approximations suffice for many situations, and the eye is forgiving of inaccuracies. While there are methods that use multiple layers of depth to mitigate the problem, they never gained wider popularity, due to complex integration with rendering engines and high runtime cost.

The screen-space approaches rely on repeatedly sampling the z-buffer to form some simplified model of the geometry around a given point. Experiments indicate that to achieve a high visual quality, as many as a couple hundred samples are needed. However, to be usable for interactive rendering, no more than 10 to 20 samples can be taken, often even less than that. Jimenez et al. [835] report that, to fit into the performance budget of a 60 FPS game, they could afford to use only one sample per pixel! To bridge the gap between theory and practice, screen-space methods usually employ some form of spatial dithering. In the most common form, every screen pixel uses a slightly different, random set of samples, rotated or shifted radially. After the main phase of AO calculations, a full-screen filtering pass is performed. Joint bilateral filtering (Section 12.1.1) is used to avoid filtering across surface discontinuities and preserve sharp edges. It uses the available information about depth or normals to restrict filtering to use only samples belonging to the same surface. Some of the methods use a randomly changing sampling pattern and experimentally chosen filtering kernel; others use a fixed-size screen-space pattern (for example, 4×4 pixels) of repeating sample sets, and a filter that is restricted to that neighborhood.

Ambient occlusion calculations are also often supersampled over time [835, 1660, 1916]. This process is commonly done by applying a different sampling pattern every frame and performing exponential averaging of the occlusion factors. Data from the previous frame is reprojected to the current view, using the last frame's z-buffer, camera transforms, and information about the motion of dynamic objects. It is then blended with the current frame results. Heuristics based on depth, normal, or velocity

are usually used to detect situations when the data from the last frame is not reliable and should be discarded (for instance, because some new object has come into view). Section 5.4.2 explains temporal supersampling and antialiasing techniques in a more general setting. The cost of temporal filtering is small, it is straightforward to implement, and even though it is not always fully reliable, in practice most problems are not noticeable. This is mainly because ambient occlusion is never visualized directly, it only serves as one of the inputs to the lighting calculations. After combining this effect with normal maps, albedo textures, and direct lighting, any minor artifacts are masked out and no longer visible.

11.3.7 Shading with Ambient Occlusion

Even though we have derived the ambient occlusion value in the context of constant, distant illumination, we can also apply it to more complex lighting scenarios. Consider the reflectance equation again:

$$L_o(\mathbf{v}) = \int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v}) L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \tag{11.19}$$

The above form contains the visibility function $v(\mathbf{l})$, as introduced in Section 11.3.1.

If we are dealing with a diffuse surface, we can replace $f(\mathbf{l},\mathbf{v})$ with the Lambertian BRDF, which is equal to the subsurface albedo ρ_{ss} divided by π. We get

$$L_o = \int_{\mathbf{l}\in\Omega} \frac{\rho_{\text{ss}}}{\pi} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} = \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l}\in\Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \tag{11.20}$$

We can reformulate the above equation to get

$$\begin{aligned} L_o =& \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l}\in\Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \\ =& \frac{\rho_{\text{ss}}}{\pi} \frac{\int_{\mathbf{l}\in\Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}}{\int_{\mathbf{l}\in\Omega} v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}} \int_{\mathbf{l}\in\Omega} v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \\ =& \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l}\in\Omega} L_i(\mathbf{l}) \frac{v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+}{\int_{\mathbf{l}\in\Omega} v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}} d\mathbf{l} \int_{\mathbf{l}\in\Omega} v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \end{aligned} \tag{11.21}$$

If we use the definition of the ambient occlusion from Equation 11.8, the above simplifies to

$$L_o = k_A \rho_{\text{ss}} \int_{\mathbf{l}\in\Omega} L_i(\mathbf{l}) K(\mathbf{n},\mathbf{l}) d\mathbf{l}, \tag{11.22}$$

where

$$K(\mathbf{n},\mathbf{l}) = \frac{v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+}{\int_{\mathbf{l}\in\Omega} v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}}. \tag{11.23}$$

This form gives us a new perspective on the process. The integral in Equation 11.22 can be thought of as applying a directional filtering kernel K to the incoming radiance L_i.

The filter K changes both spatially and directionally in a complicated way, but it has two important properties. First, it covers, at most, the hemisphere around the normal at point $\mathbf{p}$, due to the clamped dot product. Second, its integral over the hemisphere is equal to one, due to the normalization factor in the denominator.

To perform shading, we need to compute an integral of a product of two functions, the incident radiance L_i and the filter function K. In some cases, it is possible to describe the filter in a simplified way and compute this *double product integral* at a fairly low cost, for example, when both L_i and K are represented using spherical harmonics (Section 10.3.2). Another way of dealing with the complexity of this equation is approximating the filter with a simpler one that has similar properties. The most common choice is the normalized cosine kernel H:

$$H(\mathbf{n}, \mathbf{l}) = \frac{(\mathbf{n} \cdot \mathbf{l})^+}{\int_{\mathbf{l} \in \Omega} (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}}. \tag{11.24}$$

This approximation is accurate when there is nothing blocking the incoming lighting. It also covers the same angular extents as the filter we are approximating. It completely ignores visibility, but the ambient occlusion k_A term is still present in Equation 11.22, so there will be some visibility-dependent darkening on the shaded surface.

With this choice of filtering kernel, Equation 11.22 becomes

$$L_o = k_A \rho_{\text{ss}} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) \frac{(\mathbf{n} \cdot \mathbf{l})^+}{\int_{\mathbf{l} \in \Omega} (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}} d\mathbf{l} = \frac{k_A}{\pi} \rho_{\text{ss}} E. \tag{11.25}$$

This means that, in its simplest form, shading with ambient occlusion can be performed by computing irradiance and multiplying it by the ambient occlusion value. The irradiance can come from any source. For instance, it can be sampled from an irradiance environment map (Section 10.6). The accuracy of the method depends only on how well the approximated filter represents the correct one. For lighting that changes smoothly across the sphere, the approximation gives plausible results. It is also fully accurate if L_i is constant across all possible directions, i.e., as if the scene were lit by an all-white environment map representing the illumination.

This formulation also gives us some insight into why ambient occlusion is a poor approximation of visibility for punctual or small area light sources. They subtend only a small solid angle over the surface—infinitesimally small in case of punctual lights—and the visibility function has an important effect on the value of the lighting integral. It controls the light contribution in an almost binary fashion, i.e., it either enables or disables it entirely. Ignoring visibility, as we did in Equation 11.25, is a significant approximation and, in general, does not yield expected results. Shadows lack definition and do not exhibit any expected directionality, that is, do not appear to be produced by particular lights. Ambient occlusion is not a good choice for modeling visibility of such lights. Other methods, such as shadow maps, should be used instead. It is worth noting, however, that sometimes small, local lights are used to model indirect illumination. In such cases, modulating their contribution with an ambient occlusion value is justified.

Until now we assumed that we are shading a Lambertian surface. When dealing with a more complex, non-constant BRDF, this function cannot be pulled out of the integral, as we did in Equation 11.20. For specular materials, K depends not only on the visibility and normal but also on the viewing direction. A lobe of a typical microfacet BRDF changes significantly across the domain. Approximating it with a single, predetermined shape is too coarse to yield believable results. This is why using ambient occlusion for shading makes most sense for diffuse BRDFs. Other methods, discussed in the upcoming sections, are better for more complex material models.

Using the bent normal (see Equation 11.10 on page 448) can be seen as a way of approximating the filter K more precisely. The visibility term is still not present in the filter, but its maximum matches the average unoccluded direction, which makes it a slightly better approximation to Equation 11.23 overall. In cases where the geometric normal and bent normal do not match, using the latter will give more accurate results. Landis [974] uses it not only for shading with environment maps but also for some of the direct lights, instead of regular shadowing techniques.

For shading with environment maps, Pharr [1412] presents an alternative that uses the GPU's texture filtering hardware to perform the filtering dynamically. The shape of filter K is determined on the fly. Its center is the direction of the bent normal, and its size depends on the value of k_A. This provides an even more precise match to the original filter from Equation 11.23.

11.4 Directional Occlusion

Even though using ambient occlusion alone can increase the visual quality of the image tremendously, it is a greatly simplified model. It gives a poor approximation for visibility when dealing with even large area-light sources, not to mention small or punctual ones. It also cannot correctly deal with glossy BRDFs or more complex lighting setups. Consider a surface lit by a distant dome light, with color changing from red to green across the dome. This might represent ground illuminated by light coming from the sky—given the colors, probably on some distant planet. See Figure 11.18. Even though ambient occlusion will darken the lighting at points **a** and **b**, they will still be illuminated by both red and green portions of the sky. Using bent normals helps mitigate this effect but it is also not perfect. The simple model that we presented before is not flexible enough to deal with such situations. One solution is to describe the visibility in some more expressive way.

We will focus on methods that encode the entire spherical or hemispherical visibility, i.e., ways of describing which directions are blocking the incoming radiance. While this information can be used for shadowing punctual lights, it is not its main purpose. Methods targeting those specific types of lights—discussed extensively in Chapter 7—are able to achieve much better quality, as they need to encode visibility for just a single location or direction for the source. Solutions described here are meant to be used mainly for providing occlusion for large area lights or environment lighting, where

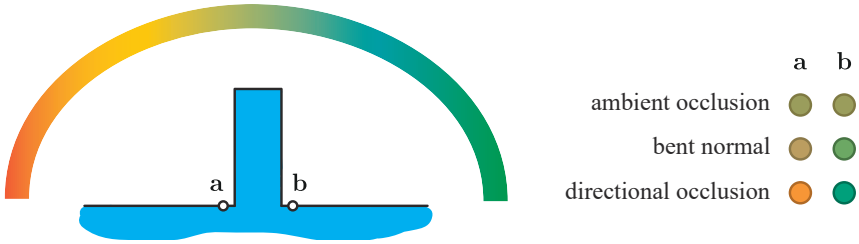

Figure 11.18. Approximated colors of the irradiance in points **a** and **b** under complex lighting conditions. Ambient occlusion does not model any directionality, so the color is the same at both points. Using a bent normal effectively shifts the cosine lobe toward the unoccluded portions of the sky, but because the integration range is not restricted in any way, it is not enough to provide accurate results. Directional methods are able to correctly eliminate lighting coming from the occluded parts of the sky.

the generated shadows are soft, and artifacts caused by approximating visibility are not noticeable. Additionally, these methods can also be used to provide occlusion in cases where regular shadowing techniques are not feasible, such as for self-shadowing of bump map details and for shadowing of extremely large scenes, where shadow maps do not have enough resolution.

11.4.1 Precomputed Directional Occlusion

Max [1145] introduced the concept of *horizon mapping* to describe self-occlusion of a heightfield surface. In horizon mapping, for each point on the surface, the altitude angle of the horizon is determined for some set of azimuth directions, e.g., eight: north, northeast, east, southeast, on around.

Instead of storing horizon angles for some given compass directions, the set of unoccluded three-dimensional directions as a whole can be modeled as an elliptical [705, 866] or circular [1306, 1307] aperture. The latter technique is called *ambient aperture lighting* (Figure 11.19). These techniques have lower storage requirements than horizon maps, but may result in incorrect shadows when the set of unoccluded directions does not resemble an ellipse or circle. For example, a flat plane from which tall spikes protrude at regular intervals should have a star-shaped direction set, which does not map well to the scheme.

There are many variations for occlusion techniques. Wang et al. [1838] use a *spherical signed distance function* (SSDF) to represent visibility. It encodes a signed distance to the boundary of the occluded regions on the sphere. Any of the spherical or hemispherical bases discussed in Section 10.3 can also be used to encode visibility [582, 632, 805, 1267]. Just as with ambient occlusion, directional visibility information can be stored in textures, mesh vertices, or volumes [1969].

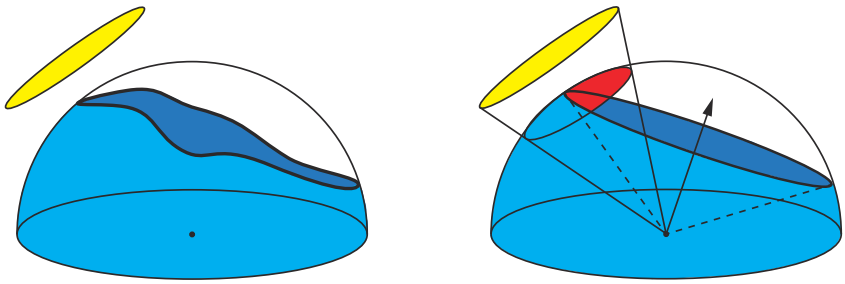

Figure 11.19. Ambient aperture lighting approximates the actual shape of unoccluded regions above the shaded point with a cone. On the left, the area light source is shown in yellow and the visible horizon for the surface location is shown in blue. On the right, the horizon is simplified to a circle, which is the edge of a cone projecting up and to the right from the surface location, shown with dashed lines. Occlusion of the area light is then estimated by intersecting its cone with the occlusion cone, yielding the area shown in red.

11.4.2 Dynamic Computation of Directional Occlusion

Many of the methods used for generating ambient occlusion can also be used to generate directional visibility information. Spherical harmonic exponentiation by Ren et al. [1482], and its screen-space variant by Sloan et al. [1655] generate visibility in a form of spherical harmonic vectors. If more than one SH band is used, these methods natively provide directional information. Using more bands allows encoding visibility with more precision.

Cone tracing methods, such as those from Crassin et al. [305] and Wright [1910], provide one occlusion value for every trace. For quality reasons, even the ambient occlusion estimations are performed using multiple traces, so the available information is already directional. If visibility in a particular direction is needed, we can trace fewer cones.

Iwanicki [806] also uses cone tracing, but he restricts it to just one direction. The results are used to generate soft shadows cast onto static geometry by dynamic characters who are approximated with a set of spheres, similar to Ren et al. [1482] and Sloan et al. [1655]. In this solution, lighting for the static geometry is stored using AHD encoding (Section 10.3.3). Visibility for ambient and directional components can be handled independently. Occlusion of the ambient part is computed analytically. A single cone is traced and intersected with the spheres to calculate the attenuation factor for the directional component.

Many of the screen-space methods can also be extended to provide directional occlusion information. Klehm et al. [904] use the z-buffer data to compute *screen-space bent cones*, which are in fact circular apertures, much like those precomputed offline by Oat and Sander [1307]. When sampling the neighborhood of a pixel, they sum the unoccluded directions. The length of the resulting vector can be used to

estimate the apex angle of the visibility cone, and its direction defines the axis of this cone. Jimenez et al. [835] estimate the cone axis direction based on the horizon angles, and derive the angle from the ambient occlusion factor.

11.4.3 Shading with Directional Occlusion

With so many different ways of encoding directional occlusion, we cannot provide a single prescription for how to perform shading. The solution will depend on what particular effect we want to achieve.

Let us consider the reflectance equation again, in a version with incoming radiance split into distant lighting L_i and its visibility v:

$$L_o(\mathbf{v}) = \int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_i(\mathbf{l})v(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \tag{11.26}$$

The simplest operation we can do is use the visibility signal to shadow punctual lights. Because of the simplicity of most ways of encoding visibility, the quality of the result will often be unsatisfactory, but it will allow us to follow the reasoning on a basic example. This method can also be used in situations where traditional shadowing methods fail because of insufficient resolution, and the precision of the results is less important than achieving any form of occlusion at all. Examples of such situations include extremely large terrain models, or small surface details represented with bump maps.

Following the discussion in Section 9.4, when dealing with punctual lights, Equation 11.26 becomes

$$L_o(\mathbf{v}) = \pi f(\mathbf{l}_c,\mathbf{v})\mathbf{c}_{\text{light}}v(\mathbf{l}_c)(\mathbf{n}\cdot\mathbf{l}_c)^+, \tag{11.27}$$

where $\mathbf{c}_{\text{light}}$ is the radiance reflected from a white, Lambertian surface facing the light, and $\mathbf{l}_c$ is the direction toward the light. We can interpret the above equation as computing a response of the material to the unoccluded light and multiplying the result by the value of the visibility function. If the light direction falls under the horizon (when using horizon mapping), outside the visibility cone (when using ambient aperture lighting), or in the negative area of the SSDF, the visibility function is equal to zero and so any contribution from the light should not be taken into account. It is worth mentioning that even though visibility is defined as binary functions,[2] many representations can return an entire range of values, not only zero or one. Such values convey partial occlusion. Spherical harmonics or the H-basis can even reconstruct negative values, due to ringing. These behaviors might be unwanted, but are just an inherent property of the encoding.

We can perform similar reasoning for lighting with area lights. In this case L_i is equal to zero everywhere, except for within the solid angle subtended by the light,

[2] In most situations, at least. There are cases when we want the visibility function to take values other than zero or one, but still in that range. For example, when encoding occlusion caused by a semitransparent material, we might want to use fractional occlusion values.

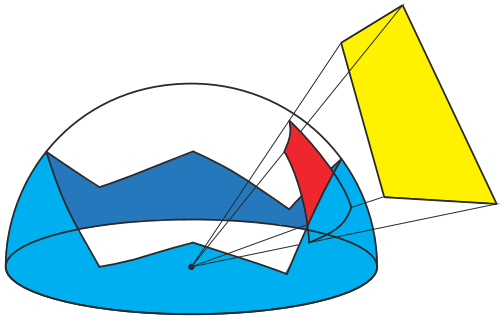

Figure 11.20. A yellow polygonal light source can be projected onto a unit hemisphere over the shaded point to form a spherical polygon. If the visibility is described using horizon mapping, that polygon can be clipped against it. The cosine-weighted integral of the clipped, red polygon can be computed analytically using Lambert's formula.

where it is equal to the radiance emitted by that source. We will refer to it as L_l and assume that it is constant over the light's solid angle. We can replace integration over the entire sphere, Ω, with integration over the solid angle of the light, Ω_l:

$$L_o(\mathbf{v}) = L_l \int_{\mathbf{l}\in\Omega_l} v(\mathbf{l}) f(\mathbf{l},\mathbf{v})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \tag{11.28}$$

If we assume that the BRDF is constant—so we are dealing with a Lambertian surface—it can also be pulled out from under the integral:

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} L_l \int_{\mathbf{l}\in\Omega_l} v(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \tag{11.29}$$

To determine the occluded lighting, we need to compute the integral of the visibility function multiplied by the cosine term over the solid angle subtended by the light. There are cases when this can be done analytically. Lambert [967] derived a formula to compute the integral of a cosine over a spherical polygon. If our area light is polygonal, and we can clip it against the visibility representation, we then need to use only Lambert's formula to get a precise result (Figure 11.20). This is possible when, for example, we choose horizon angles as our visibility representation. However, if for any reason we settled on another encoding, such as bent cones, the clipping would produce circular segments, for which we can no longer use Lambert's formula. The same applies if we want to use non-polygonal area lights.

Another possibility is to assume that the value of the cosine term is constant across the integration domain. If the size of the area light is small, this approximation is fairly precise. For simplicity, we can use the value of the cosine evaluated in the direction of the center of the area light. This leaves us with the integral of the visibility term over the solid angle of the light. Our options on how to proceed depend, again, on our

choice of visibility representation and type of area light. If we use spherical lights and visibility represented with bent cones, the value of the integral is the solid angle of the intersection of the visibility cone and the cone subtended by the light. It can be computed analytically, as shown by Oat and Sander [1307]. While the exact formula is complex, they provide an approximation that works well in practice. If visibility is encoded with spherical harmonics, the integral can also be computed analytically.

For environment lighting we cannot restrict the integration range, because illumination comes from all directions. We need to find a way to calculate the full integral from Equation 11.26. Let us consider the Lambertian BRDF first:

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}. \tag{11.30}$$

The type of integral in this equation is called a *triple product integral*. If the individual functions are represented in certain ways—for instance, as spherical harmonics or wavelets—it can be computed analytically. Unfortunately, this is too expensive for typical real-time applications, though such solutions have been shown to run at interactive frame rates in simple setups [1270].

Our particular case is slightly simpler, though, as one of the functions is the cosine. We can instead write Equation 11.30 as

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} \overline{L_i}(\mathbf{l}) v(\mathbf{l}) d\mathbf{l} \tag{11.31}$$

or

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) \overline{v}(\mathbf{l}) d\mathbf{l}, \tag{11.32}$$

where

$$\overline{L_i}(\mathbf{l}) = L_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+,$$
$$\overline{v}(\mathbf{l}) = v(\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+.$$

Both $\overline{L_i}(\mathbf{l})$ and $\overline{v}(\mathbf{l})$ are spherical functions, like $L_i(\mathbf{l})$ and $v(\mathbf{l})$. Instead of trying to compute a triple product integral, we first multiply the cosine by either L_i (Equation 11.31) or v_i (Equation 11.32). Doing so makes the integrand a product of only two functions. While this might look like just a mathematics trick, it significantly simplifies calculations. If the factors are represented using an orthonormal basis such as spherical harmonics, the double product integral can be trivially computed. It is a dot product of their coefficient vectors (Section 10.3.2).

We still need to compute $\overline{L_i}(\mathbf{l})$ or $\overline{v}(\mathbf{l})$, but because they involve the cosine, this is simpler than for the fully general case. If we represent the functions using spherical harmonics, the cosine projects to zonal harmonics (ZH), a subset of spherical harmonics for which only one coefficient per band is nonzero (Section 10.3.2). The coefficients for this projection have simple, analytic formulae [1656]. The product of an SH and a ZH is much more efficient to compute than a product of an SH and another SH.

If we decide to multiply the cosine by v first (Equation 11.32), we can do it offline and instead store just the visibility. This is a form of *precomputed radiance transfer*, as described by Sloan et al. [1651] (Section 11.5.3). In this form, however, we cannot apply any fine-scale modification of the normal, as the cosine term, which is controlled by the normal, is already fused with the visibility. If we want to model fine-scale normal details, we can multiply the cosine by L_i first (Equation 11.31). Since we do not know the normal direction up front, we can either precompute this product for different normals [805] or perform the multiplication at runtime [809]. Precomputing the products of L_i and cosine offline means that, in turn, any changes to the lighting are restricted, and allowing the lighting to change spatially would require prohibitive amounts of memory. On the other hand, computing the product at runtime is computationally expensive. Iwanicki and Sloan [809] describe how to reduce this cost. Products can be computed at a lower granularity—on vertices, in their case. The result is convolved with the cosine term, projected onto a simpler representation (AHD), then interpolated and reconstructed with the per-pixel normal. This approach allows them to use the method in performance-demanding 60 FPS games.

Klehm et al. [904] present a solution for lighting represented with an environment map and visibility encoded with a cone. They filter the environment map with different-sized kernels that represent the integral of a product of visibility and lighting for different cone openings. They store the results for increasing cone angles in the mip levels of a texture. This is possible because prefiltered results for large cone angles vary smoothly over the sphere and do not need to be stored with high angular resolution. During prefiltering, they assume that the direction of the visibility cone is aligned with the normal, which is an approximation, but in practice gives plausible results. They provide an analysis on how this approximation affects the final quality.

The situation is even more complex if we are dealing with glossy BRDFs and environment lighting. We can no longer pull the BRDF from under the integral, as it is not constant. To handle this, Green et al. [582] suggest approximating the BRDF itself with a set of *spherical Gaussians*. These are radially symmetric functions that can be compactly represented with just three parameters: direction (or mean) $\mathbf{d}$, standard deviation μ, and amplitude w. The approximate BRDF is defined as a sum of spherical Gaussians:

$$f(\mathbf{l}, \mathbf{v}) \approx \sum_k w_k(\mathbf{v}) G(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}), \tag{11.33}$$

where $G(\mathbf{d}, \mu, \mathbf{l})$ is the spherical Gaussian lobe, oriented along direction $\mathbf{d}$, with sharpness μ (Section 10.3.2), and w_k in the amplitude of the kth lobe. For an isotropic BRDF, the shape of the lobe depends on only the angle between the normal and the view direction. Approximations can be stored in a one-dimensional lookup table and interpolated.

With this approximation, we can write Equation 11.26 as

$$\begin{aligned} L_o(\mathbf{v}) &\approx \int_{\mathbf{l}\in\Omega} \sum_k w_k(\mathbf{v}) G(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}) L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \\ &= \sum_k w_k(\mathbf{v}) \int_{\mathbf{l}\in\Omega} G(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}) L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \end{aligned} \tag{11.34}$$

Green et al. also assume that the visibility function is constant across the entire support of each spherical Gaussian, which allows them to pull it out from under the integral. They evaluate the visibility function in the direction of the lobe center:

$$L_o(\mathbf{v}) \approx \sum_k w_k(\mathbf{v}) v_k(\mathbf{d}_k(\mathbf{v})) \int_{\mathbf{l}\in\Omega} G(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}) L_i(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \tag{11.35}$$

The remaining integral represents incoming lighting convolved with a spherical Gaussian oriented in a given direction and with a given standard deviation. The results of such convolutions can be precomputed and stored in an environment map, with convolutions for larger μ's stored in lower mip levels. Visibility is encoded with low-order spherical harmonics, but any other representation could also be used, as it is only point-evaluated.

Wang et al. [1838] approximate the BRDF in similar fashion, but handle visibility in a more precise way. Their representation allows them to calculate the integral of a single spherical Gaussian over the support of the visibility function. They use this value to introduce a new spherical Gaussian, one with the same direction and standard deviation, but a different amplitude. They use this new function during lighting calculations.

For certain applications this method may be too expensive. It requires multiple samples from the prefiltered environment maps, and texture sampling is often already a bottleneck during rendering. Jimenez et al. [835] and El Garawany [414] present simpler approximations. To compute the occlusion factor, they represent the entire BRDF lobe with a single cone, ignoring its dependence on viewing angle and considering only parameters such as material roughness (Figure 11.21). They approximate visibility as a cone and compute the solid angle of the intersection of the visibility and BRDF cones, much as done for ambient aperture lighting. The scalar result is used to attenuate lighting. Even though it is a significant simplification, the results are believable.

11.5 Diffuse Global Illumination

The next sections cover various ways of simulating not only occlusion but also full light bounces in real time. They can be roughly divided into algorithms that assume that, right before reaching the eye, light bounces off either a diffuse or specular surface. The

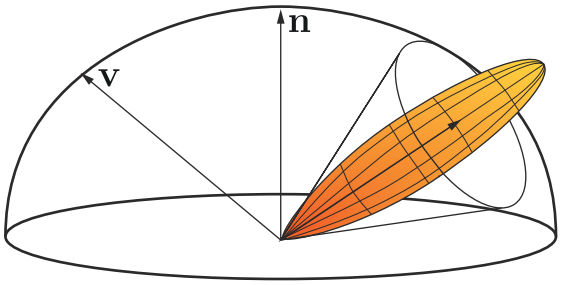

Figure 11.21. For the purpose of computing occlusion, the specular lobe of a glossy material can be represented as a cone. If visibility is approximated as another cone, the occlusion factor can be calculated as a solid angle of the intersection of the two, in a same way as done for ambient aperture lighting (Figure 11.19). The image shows the general principle of representing a BRDF lobe with a cone, but is only meant as an illustration. In practice, to produce plausible occlusion results, the cone would need to be wider.

corresponding light paths can be written as $L(D|S)*DE$ or $L(D|S)*SE$, respectively, with many of the methods placing some constraints on the types of earlier bounces. Solutions in the first group assume that the incoming lighting changes smoothly across the hemisphere above the shading point, or ignore that change entirely. Algorithms in the second group assume a high rate of change across the incident directions. They rely on the fact that the lighting will be accessed within only a relatively small solid angle. Because of these vastly different constraints, it is beneficial to handle these two groups separately. We cover methods for diffuse global illumination in this section, specular in the next, then promising unified approaches in the final section.

11.5.1 Surface Prelighting

Both radiosity and path tracing are designed for offline use. While there have been efforts to use them in real-time settings, the results are still too immature to be used in production. Currently the most common practice is to use them to precompute lighting-related information. The expensive, offline process is run ahead of time, and its results are stored and later used during display to provide high-quality lighting. As mentioned in Section 11.3.4, precomputing in this way for a static scene is referred to as baking.

This practice comes with certain restrictions. If we perform lighting calculations ahead of time, we cannot change the scene setup at runtime. All the scene geometry, lights, and materials need to remain unchanged. We cannot change time of day or blow a hole in a wall. In many cases this limitation is an acceptable trade-off. Architectural visualizations can assume that the user only walks around virtual environments. Games place restrictions on player actions, too. In such applications we can classify geometry into *static* and *dynamic* objects. The static objects are used in the precomputation process and they fully interact with the lighting. Static walls

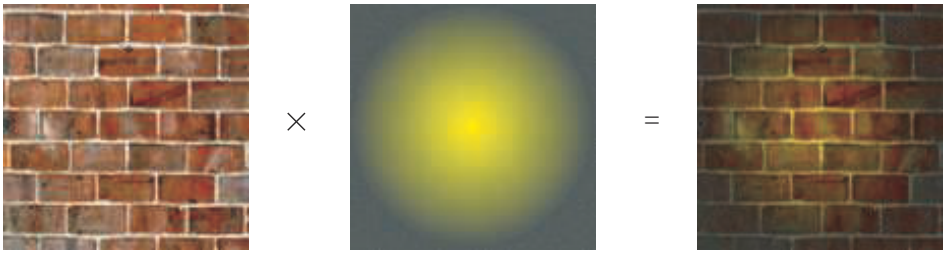

Figure 11.22. Given a Lambertian surface with a known normal, its irradiance can be precomputed. At runtime this value is multiplied by the actual surface color (for instance, from a texture) to obtain the reflected radiance. Depending on the exact form of the surface color, additional division by π might be needed to ensure energy conservation.

cast shadows and static red carpets bounce red light. Dynamic objects act only as receivers. They do not block light, and they do not generate indirect illumination effects. In such scenarios, dynamic geometry is usually restricted to be relatively small, so its effect on the rest of the lighting can be either ignored or modeled with other techniques, with minimal loss of quality. Dynamic geometry can, for example, use screen-space approaches to generate occlusion. A typical set of dynamic objects includes characters, decorative geometry, and vehicles.

The simplest form of lighting information that can be precomputed is irradiance. For flat, Lambertian surfaces, together with surface color it fully describes the material's response to lighting. Because the effect of a source of illumination is independent of any others, dynamic lights can be added on top of the precomputed irradiance (Figure 11.22).

Quake in 1996 and *Quake II* in 1997 were the first commercial interactive applications to make use of precomputed irradiance values. *Quake* was precomputing the direct contribution from static lights, mainly as a way of improving performance. *Quake II* also included an indirect component, making it the first game that used a global illumination algorithm to generate more realistic lighting. It used a radiosity-based algorithm, since this technique was well suited to computing irradiance in Lambertian environments. Also, memory constraints of the time restricted the lighting to be relatively low resolution, which matched well with the blurry, low-frequency shadows typical of radiosity solutions.

Precomputed irradiance values are usually multiplied with diffuse color or albedo maps stored in a separate set of textures. Although the exitance (irradiance times diffuse color) could in theory be precomputed and stored in a single set of textures, many practical considerations rule out this option in most cases. The color maps are usually quite high frequency, they make use of various kinds of tiling, and their parts are often reused across the model, all to keep the memory usage reasonable. The irradiance values are usually much lower frequency and cannot easily be reused. Keeping lighting and surface color separate consumes much less memory.

Precomputed irradiance is rarely used today, except for the most restrictive hardware platforms. Since, by definition, irradiance is computed for a given normal direction, we cannot use normal mapping to provide high-frequency details. This also means that irradiance can only be precomputed for flat surfaces. If we need to use baked lighting on dynamic geometry, we need other methods to store it. These limitations have motivated a search for ways to store precomputed lighting with a directional component.

11.5.2 Directional Surface Prelighting

To use prelighting together with normal mapping on Lambertian surfaces, we want a way to represent how the irradiance changes with the surface normal. To provide indirect lighting for dynamic geometry, we also need its value for every possible surface orientation. Happily, we already have tools to represent such functions. In Section 10.3 we described various ways of determining lighting dependent on the normal's direction. These include specialized solutions for cases where the function domain is hemispherical and the values on the lower half of the sphere do not matter, as is the case for opaque surfaces.

The most general method is to store full spherical irradiance information, for example by using spherical harmonics. This scheme was first presented by Good and Taylor [564] in the context of accelerating photon mapping, and used in a real-time setting by Shopf et al. [1637]. In both cases directional irradiance was stored in textures. If nine spherical harmonic coefficients are used (third-order SH), the quality is excellent, however the storage and bandwidth costs are high. Using just four coefficients (a second-order SH) is less costly, but many subtleties get lost, the lighting has less contrast, and normal maps are less pronounced.

Chen [257] uses a variation of the method for *Halo 3*, developed to achieve the quality of a third-order SH at a reduced cost. He extracts the most dominant light out of the spherical signal and stores it separately, as a color and a direction. The residue is encoded using a second-order SH. This reduces the number of coefficients from 27 to 18, with little quality loss. Hu [780] describes how these data can be compressed further. Chen and Tatarchuk [258] provide further information on their GPU-based baking pipeline used in production.

The *H-basis* by Habel et al. [627] is another alternative solution. Since it encodes only hemispherical signals, fewer coefficients can provide the same precision as spherical harmonics. Quality comparable to a third-order SH can be obtained with just six coefficients. Because the basis is defined only for a hemisphere, we need some local coordinate system on the surface to properly orient it. Usually, a tangent frame resulting from uv-parameterization is used for this purpose. If H-basis components are stored in a texture, its resolution should be high enough to adapt to the changes of the underlying tangent space. If multiple triangles with significantly different tangent spaces cover the same texel, the reconstructed signal will not be precise.

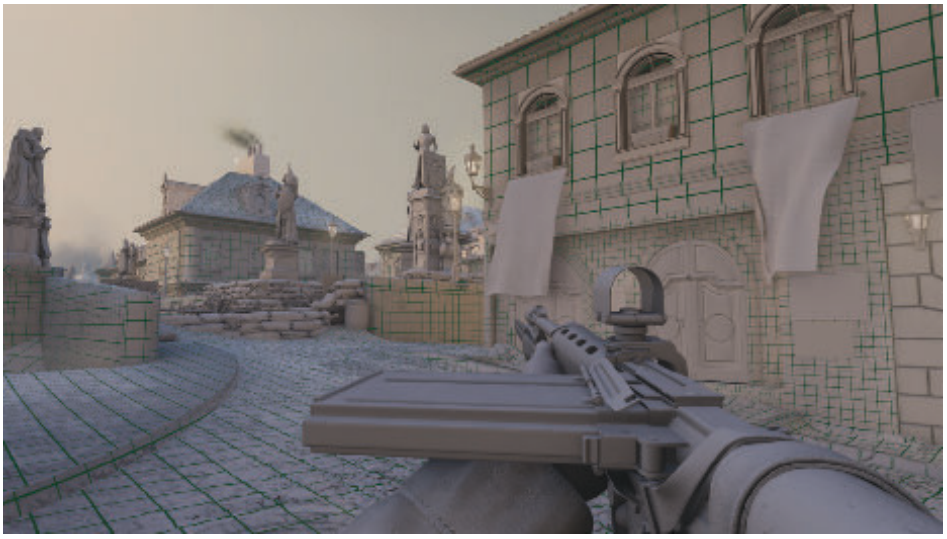

Figure 11.23. *Call of Duty: WWII* uses the *AHD* representation to encode the directional variation of the lighting in the light map. The grid is used to visualize the light map density in debug mode. Each square corresponds to a single light map texel. *(Image courtesy of Activision Publishing, Inc. 2018.)*

One problem with both spherical harmonics and the H-basis is that they can exhibit ringing (Section 10.6.1). While prefiltering can mitigate this effect, it also smoothes the lighting, which might not always be desirable. Additionally, even the less-expensive variants still have relatively high cost, both in terms of storage and computation. This expense might be prohibitive in more restrictive cases, such as on low-end platforms or when rendering for virtual reality.

Costs are why simple alternatives are still popular. *Half-Life 2* uses a custom, hemispherical basis (Section 10.3.3) that stores three color values, for a total of nine coefficients per sample. The ambient/highlight/direction (AHD) basis (Section 10.3.3) is also a popular choice despite its simplicity. It has been used in games such as the *Call of Duty* [809, 998] series and *The Last of Us* [806]. See Figure 11.23.

A variation was used by Crytek in the game *Far Cry* [1227]. The Crytek representation consists of an average light direction in tangent space, an average light color, and a scalar directionality factor. This last value is used to blend between the ambient and directional components, which both use the same color. This reduces the storage to six coefficients per sample: three values for the color, two for the direction, and one for the directionality factor. The *Unity* engine also uses similar methods in one of its modes [315].

This type of representation is nonlinear, which means that, technically, linearly interpolating individual components, either between texels or vertices, is not mathe-

Figure 11.24. *The Order: 1886* stores incident radiance projected onto a set of spherical Gaussian lobes in its light maps. At runtime, the radiance is convolved with the cosine lobe to compute the diffuse response (left) and with a properly shaped anisotropic spherical Gaussian to generate the specular response (right). *(Image courtesy of Ready at Dawn Studios, copyright Sony Interactive Entertainment.)*

matically correct. If the direction of the dominant light changes rapidly, for instance on shadow boundaries, visual artifacts might appear in shading. Despite these inaccuracies, the results look visually pleasing. Because of the high contrast between the ambient and directionally lit areas, the effects of normal maps are accentuated, which is often desirable. Additionally, the directional component can be used when calculating the specular response of the BRDF to provide a low-cost alternative to environment maps for low-gloss materials.

On the opposite end of the spectrum are methods designed for high visual quality. Neubelt and Pettineo [1268] use texture maps storing coefficients for spherical Gaussians in the game *The Order: 1886* (Figure 11.24). Instead of irradiance, they store incoming radiance, which is projected to a set of Gaussian lobes (Section 10.3.2), defined in a tangent frame. They use between five and nine lobes, depending on the complexity of the lighting in a particular scene. To generate a diffuse response, the spherical Gaussians are convolved with a cosine lobe oriented along the surface normal. The representation is also precise enough to provide low-gloss specular effects, by convolving Gaussians with the specular BRDF lobe. Pettineo describes the full system in detail [1408]. He also provides source code to an application capable of baking and rendering different lighting representations.

If we need information about the lighting in an arbitrary direction, not just within a hemisphere above the surface (for example, to provide indirect lighting for dynamic geometry), we can use methods that encode a full spherical signal. Spherical harmonics are a natural fit here. When memory is less of a concern, third-order SH (nine coefficients per color channel) is the popular choice; otherwise, second-order is used (four coefficients per color channel, which matches the number of components in an RGBA texture, so a single map can store coefficients for one color channel). Spherical Gaussians also work in a fully spherical setting, as the lobes can be distributed across either the entire sphere or only the hemisphere around the normal. However, since

the solid angle that needs to be covered by the lobes is twice as large for spherical techniques, we might need to use more lobes to retain the same quality.

If we want to avoid dealing with ringing, but cannot afford to use a high number of lobes, the ambient cube [1193] (Section 10.3.1) is a viable alternative. It consists of six clamped $\cos^2$ lobes, oriented along the major axes. The cosine lobes each cover just one hemisphere, as they have *local support*, meaning they have nonzero values on only a subset of their spherical domain. For this reason, only the three visible lobes out of the six stored values are needed during the reconstruction. This limits the bandwidth cost of lighting calculations. The quality of reconstruction is similar to second-order spherical harmonics.

Ambient dice [808] (also Section 10.3.1) can be used for higher quality than ambient cubes. This scheme uses twelve lobes oriented along the vertices of an icosahedron that are a linear combination of $\cos^2$ and $\cos^4$ lobes. Six values out of the twelve stored are used during a reconstruction. The quality is comparable to third-order spherical harmonics. These and other similar representations (for example, a basis consisting of three $\cos^2$ lobes and a cosine lobe, warped to cover a full sphere) have been used in many commercially successful games, such as *Half-Life 2* [1193], the *Call of Duty* series [766, 808], *Far Cry 3* [533], *Tom Clancy's The Division* [1694], and *Assassin's Creed 4: Black Flag* [1911], to name a few.

11.5.3 Precomputed Transfer

While precomputed lighting can look stunning, it is also inherently static. Any change to geometry or lighting can invalidate the entire solution. Just as in the real world, opening the curtains (local change to the geometry in the scene) may flood the entire room with light (global change to the lighting). Much research effort has been spent on finding solutions that allow for certain types of changes.

If we make the assumption that the scene's geometry does not change, only the lighting, we can precompute how light interacts with the models. Inter-object effects such as interreflections or subsurface scattering can be analyzed up front to a certain degree and the results stored, without operating on actual radiance values. The function that takes the incoming lighting and turns it into a description of the radiance distribution throughout the scene is called a *transfer function*. Solutions that precompute this are called *precomputed transfer* or *precomputed radiance transfer* (PRT) approaches.

As opposed to fully baking the lighting offline, these techniques do have a noticeable runtime cost. When displaying the scene on screen, we need to calculate radiance values for a particular lighting setup. To do this, the actual amount of direct light is "injected" into the system, then the transfer function is applied to propagate it across the scene. Some of the methods assume that this direct lighting comes from an environment map. Other schemes allow the lighting setup to be arbitrary and to change in a flexible fashion.

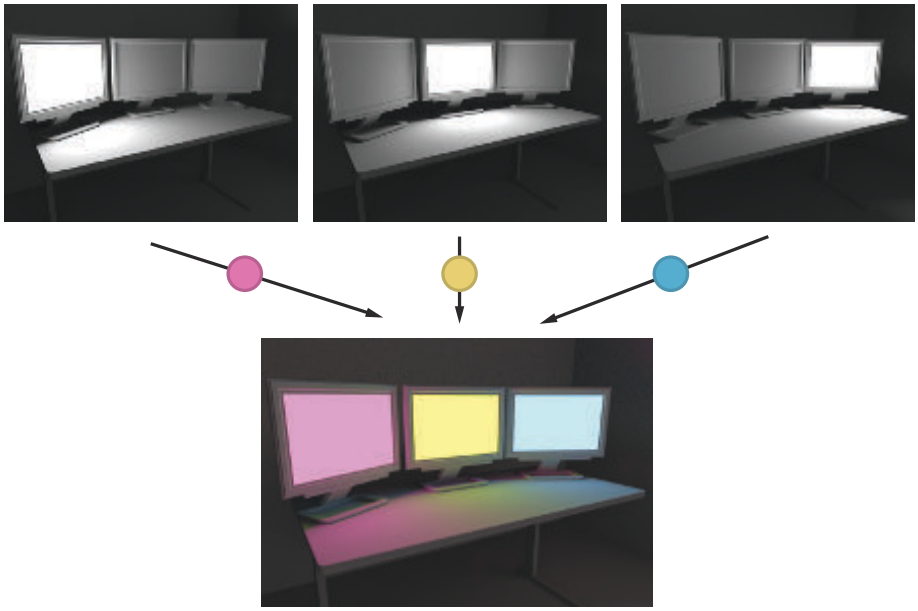

Figure 11.25. Example of rendering using precomputed radiance transfer. Full transport of lighting from each of the three monitors was precomputed separately, obtaining a "unit" response. Because of the linearity of the light transport, these individual solutions can be multiplied by the colors of the screens (pink, yellow, and blue in the example here) to obtain the final lighting.

The concept of precomputed radiance transfer was introduced to graphics by Sloan et al. [1651]. They describe it in terms of spherical harmonics, but the method does not have to use SH. The basic idea is simple. If we describe the direct lighting using some (preferably low) number of "building block" lights, we can precalculate how the scene is lit by each one of them. Imagine a room with three computer monitors inside, and assume that each can display only a single color, but with varying intensity. Consider the maximum brightness of each screen as equal to one, a normalized "unit" lightness. We can independently precompute the effect each monitor has on the room. This process can be done using the methods we covered in Section 11.2. Because light transport is linear, the result of illuminating the scene with all three monitors will be equal to the sum of the light coming from each, directly or indirectly. The illumination from each monitor does not affect the other solutions, so if we set one of the screens to be half as bright, doing so will change only its own contribution to the total lighting.

This allows us to quickly compute the full, bounced lighting within the entire room. We take each precomputed light solution, multiply it by the actual brightness of the screen, and sum the results. We can turn the monitors on and off, make them brighter or darker, even change their colors, and all that is required to get the final lighting are these multiplies and additions (Figure 11.25).