Part II

Physically-based Rendering

In Part I, we discussed rasterization techniques. While the raterization technique provides the efficient performance based on rendering pipeline utilizing modern GPUs, its fundamental approach is not based on the physical interaction between lights and materials. Another large stream of rendering methods are based on such physical interactions and thus are known as physically-based rendering.

In this part, we discuss two different approaches, ray tracing and radiosity, of physically based rendering methods. Ray tracing and radiosity are two main building blocks of many interactive or physically based rendering techniques. We first discuss ray tracing in this chapter, followed by radiosity (Ch. 11. We then study radiometric quantities (Ch. 12) to measure different energy terms to describe the physical interaction, known as the rendering equation (Ch. 13.1).

The rendering equation is a high dimensional integral problem, and thus its analytic solutions in many cases are not available. As an effective solution to solving the equation, we study the Monte Carlo technique, a numerical approach in Ch. 14, and its integration with ray tracing in Ch. 15. In many practical problems, such Monte Carlo approaches are slow to converge to noise-free images. We therefore study importance sampling techniques in Ch. 14.3.

9.6 Available Tools

Physically based rendering has been studied for many decades, and many useful resources are available. Some of them are listed here:

- Physically Based Rendering: From Theory to Implementation [1]. This book also known as pbrt comes with concepts with their actual implementations. As a result, readers can get understanding on those concepts and actual implementation that they can play with. Since this book discusses such implementation, we strongly recommend you to play with their source codes, which are available at github.
- Embree [2] and Optix [3]. Embree and Optix are interactive ray tracing kernels that run on CPUs and GPUs, respectively. While source codes of Optix are unavailable, Embree comes with their source codes.
- Instant Radiosity. Instant radiosiy is widely used in many games, thanks to its high quality rendering results with reasonably fast performance. Unfortunately due to its importance in recent game industry, mature library or open source projects are not available. One of useful open source projects are from my graphics lab. It is available at: `http://sglab.kaist.ac.kr/~sungeui/ICG/student_presentations.html`.

[1] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010b. ISBN 0123750792, 9780123750792

[2] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 2014

[3] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29:66:1–66:13, 2010

10
Ray Tracing

Ray casting and tracing techniques have been introduced late 70's and early 80's to the computer graphics field as rendering techniques for achieving high-quality images.

Ray casting [1] shoots a ray from the camera origin to a pixel and compute the first intersection point between the ray and objects in the scene. Ray casting then computes the color from the intersection point and use it as the color of the pixel. It computes a direct illumination that has one bounce from the light to the eye. Its result is same to those of the basic rasterization considering only the direct illumination.

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968

Ray tracing [2] is an recursive version of the ray casting. In other words, once we have the intersection between the initial ray and objects, ray tracing generates another ray or rays to simulate the interaction between and the light and objects. A ray can be considered as a photon traveling in a straight line, and by simulating many rays in a physically correct way, we can achieve physically correct images. While the algorithm is extremely simple, we can support various effects by generating different rays (Fig. 10.1).

[2] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980

10.1 Basic algorithm

Ray tracing simulates how a photon interacts with objects.

The basic ray tracing algorithm is very simple, as shown in Algorithm 1. We first generate a ray from the eye to the scene. While a photon travels from a light source, we typically perform ray tracing in backward from the eye (Fig. 10.2). We then identify the first intersection point between the ray and the scene. This has been studied well, especially around the early stage of developing this technique. At this point, we simply assume that we can compute such intersection points and this is discussed in Sec. 10.2.

Suppose that we identify such an intersection point between the ray and the scene. We can then perform various shading operations based on the Phong illumination (Sec. 8.3). To see whether the point

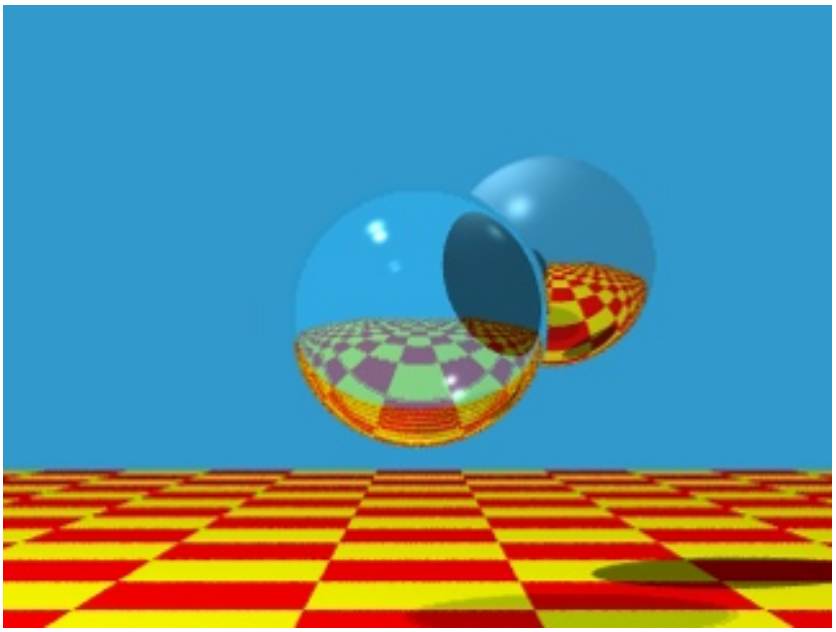


Figure 10.1: One of early images generated by ray tracing, i.e., Whitted style ray tracing. The image has reflection, refraction, and shadow effects. The image is excerpted from its original paper.

Algorithm 1 Basic ray tracing

```
Trace rays from the eye into the scene (backward ray tracing).
Identify the first intersection point and shade with it.
Generate additional, secondary rays needed for shading.
    Generate ray for reflections.
    Generate ray for refraction and transparency.
    Generate ray for shadows.
```

is under the shadow or not, we simple generate another ray, called shadow ray, to the light source (the bottom image of Fig. 10.2).

Reflection and refractions are handled in a similar manner by generating another secondary rays (Fig. 10.3). The main question that we need to address here is how we can construct the secondary rays for supporting reflection and refraction. For the mirror-like objects, we can apply the perfect-specular reflection and compute the reflection direction for the reflection ray, where the incoming angle is same to the outgoing angle. In other words, the origin of the reflection ray, R, is set to the hit point of the prior ray, and the direction of R is set as the reflection direction. Its exact equation is shown in Sec. 8.

Most objects in practice do not support such perfect reflection. For simple cases such as rays bending in glasses or water, we apply the Snell's law to compute the outgoing angle for refraction. The Snell's

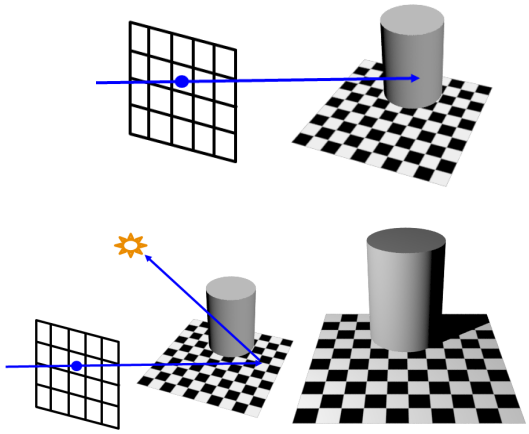


Figure 10.2: We generate a ray, primary ray, from the eye (top). To see whether the intersection point is in the shadow or not, we generate another ray, shadow ray, to the light source (bottom). These images are created by using 3ds Max.

law is described as follows:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}, \tag{10.1}$$

where θ_1 and θ_2 are incoming and outgoing angles given rays at the interface between two different objects (Fig. 10.4). n_1 and n_2 are refractive indices of those two objects. The refractive index of a material (e.g., water) is defined as $\frac{c}{v}$, where c is the velocity of the light in vacuum, while v is the speed of the light in that material. As a result, refractive indices of different materials are measured and can be used for simulating such materials within ray tracing.

Many objects used in practice consist of many different materials. As a result, the Snell's law designed for isotropic media may not be appropriate for such cases. For general cases, BRDF and BSSRDF have been proposed and are discussed in Ch. 12.

Physically based rendering techniques adopt many physical laws, as exemplified by adopting the Snell's law for computing refraction rays. This is one of main difference between rasterization and physically based rendering methods.

For various effects, ray tracing generate different types of rays, while rasterization adopts different types of texture maps.

Note that in rasterization techniques, to handle shadow, reflection, refraction, and many other rendering effects, we commonly generate some maps (e.g., shadow maps) accommodating such effects. As a result, handling texture mapping efficiently is one of key components for many rasterization techniques running on GPUs. On the other hand, ray tracing generates various rays for such effects, and handling rays efficiently is one of key components of ray tracing.

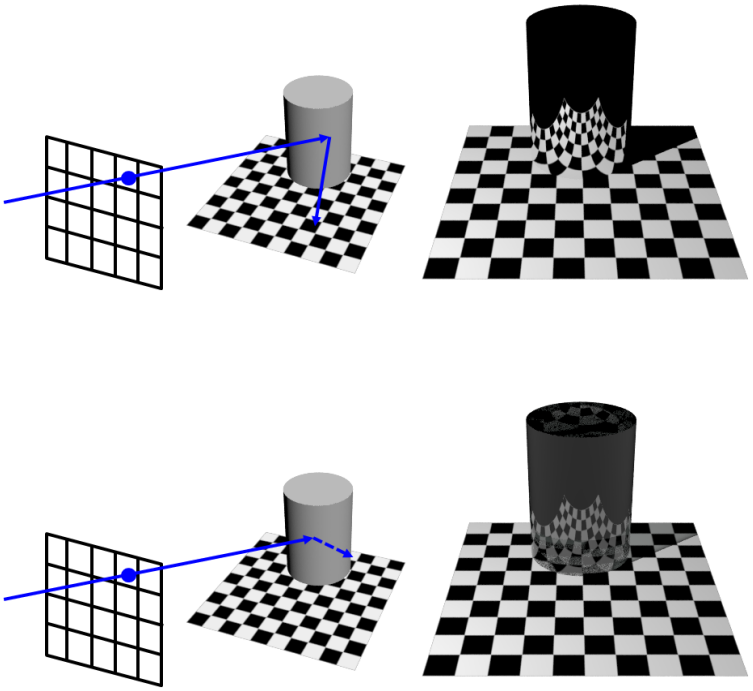


Figure 10.3: Handling reflection and refraction by generating secondary rays.

10.2 *Intersection Tests*

Performing intersection tests is one of main operations of ray tracing. Furthermore, they tend to become the main bottleneck of ray tracing and thus have been optimized for a few decades. In this section, we discuss basic ways of computing intersection tests between a ray and a few simple representations of a model.

Implicit forms of objects are commonly used for intersection tests.

Any points, $p(t)$, in a ray parameterized by a parameter t can be represented as follows:

$$p(t) = o + t\vec{d}, \tag{10.2}$$

where o and $\vec{d}$ are the origin and direction of the ray, respectively. A common way of approaching this problem is to first define an object in an implicit mathematical form, $f(p) = 0$, where p is any point on the object. We then compute the intersection point, t_i, satisfying $f(p(t_i)) = 0$.

We now look at a specific case of computing an intersection point between a ray and a plane. A well known implicit form of a plane is:

$$\vec{n}p - d = 0, \tag{10.3}$$

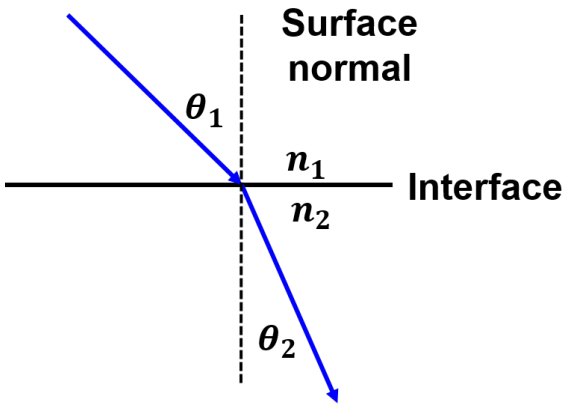


Figure 10.4: How a ray bends at an interface between simple objects, specifically, isotropic media such as water, air, and glass, is described by the Snell's law.

where $\vec{n}$ is a normalized normal vector of the plane and d is the distance from the origin to the plane. This implicit form of the plane equation is also known as the Hessian normal form [3].

[3] E. Weisstein. From mathworld–a wolfram web resource. URL http://mathworld.wolfram.com

By plugging the ray equation into the implicit of the plane equation, we get:

$$\begin{aligned} \vec{n}(o + t\vec{d}) - d &= 0 \\ t &= \frac{d - \vec{n}o}{\vec{n} \cdot \vec{d}}. \end{aligned} \tag{10.4}$$

We now discuss a ray intersection method against triangles, which are one of common representations of objects in computer graphics. There are many different ways of computing the intersection point with triangles. We approach the problem based on barycentric coordinates of points with a triangle.

Barycentric coordinates are computed based on non-orthogonal bases.

Barycentric coordinates are computed based on non-orthogonal bases unlike the Cartesian coordinate system, which uses orthogonal bases such as X, Y, and Z-axis. Suppose that p is an intersection point between a ray and a triangle consisting of three vertices, v_0, v_1, v_2 (Fig. 10.5). We can represent the point p as the following:

$$\begin{aligned} p =& v_0 + \beta(v_1 - v_0) + \gamma(v_2 - v_0) \\ =& (1 - \beta - \gamma)v_0 + \beta v_1 + \gamma v_2 \\ =& \alpha v_0 + \beta v_1 + \gamma v_2, \end{aligned} \tag{10.5}$$

where we use α to denote $1 - \beta - \gamma$. We can then see a constraint that $\alpha + \beta + \gamma = 1$, indicating that we have two degrees-of-freedom, while there are three parameters.

Let's see in what ranges of these parameters the point p is inside the triangle. Consider edges along two vectors $v_0 - v_1$ and $v_2 - v_0$ (Fig. 10.5). Along those edges, β and γ should be in $[0, 1]$, when the

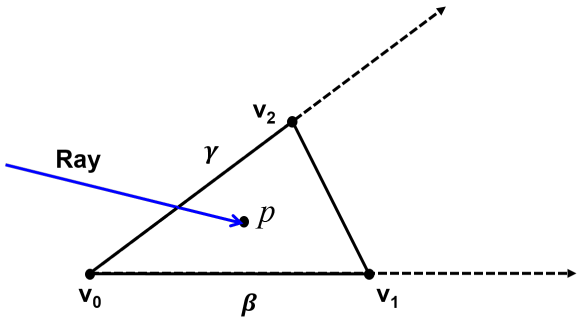


Figure 10.5: In the barycentric coordinate system, we represent the point p with β and γ coordinates with two non-orthogonal basis vectors, $v_1 - v_0$ and $v_2 - v_0$.

point is inside the triangle. Additionally, when we consider the other edge along the vector of $v_1 - v_2$, points on the edge satisfy $\gamma = 1 - \beta$[4]. When we plug the equation into the definition of α, we see α to be zero. On the other hand, on the point of v_0, β and γ should be zero, and thus α to be one. As a result, we have the following property:

$$0 \leq \alpha, \beta, \gamma \leq 1, \tag{10.6}$$

where these three coordinates are barycentric coordinates and $\alpha = 1 - \beta - \gamma$.

[4] When we consider a 2 D space whose basis vectors map to canonical vectors (e.g., X and Y axises) with β and γ coordinates, one can easily show that the relationship $\gamma = 1 - \beta$ is satisfied on the edge of $v_2 - v_1$.

Barycentric coordinates are also known as area coordinates, since they map to areas of sub-triangles associated with vertices.

There are many different ways of computing barycentric coordinates given points defined in the Cartesian coordinate system. An intuitive way is to associate barycentric coordinates with areas of sub-triangles of the triangle; as a result, barycentric coordinates are also known as area coordinates. For example, β associated with v_1 is equal to the ratio of the area of $\triangle p v_0 v_2$ to that of $\triangle v_0 v_1 v_2$.

Once we represent the intersection point p within the triangle with the barycentric coordinates, our goal is to find t of the ray that intersects with the triangle, denoted as the following:

$$o + t\vec{d} = (1 - \beta - \gamma)v_0 + \beta v_1 + \gamma v_2, \tag{10.7}$$

where unknown variables are t, β, γ. Since we have three different equations with X, Y, and Z coordinates of vertices and the ray, we can compute those three unknowns.

10.3 *Bounding Volume Hierarchy*

We have discussed how to perform intersection tests between a ray and implicit equations representing planes and triangles. Common models used in games and movies have thousands of or millions of triangles. A naive approach of computing the first intersection point between a ray and those triangles is to linearly scan those triangles and test the ray-triangle intersection tests. It, however, has a linear

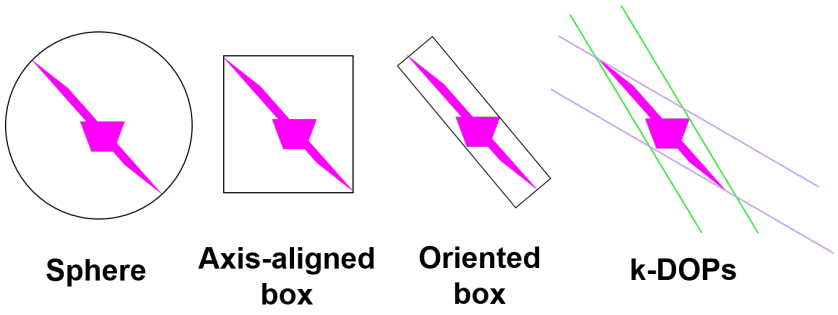


Figure 10.6: This figure shows different types of Bounding Volumes (BVs).

time complexity as a function of the number of triangles, and thus can take an excessive amount of computation time.

Many acceleration techniques have been proposed to reduce the time spent on ray intersection tests. Some of important techniques include optimized ray-triangle intersection tests using Barycentric coordinates [5]. In this section, we discuss an hierarchical acceleration technique that can improve the linear time complexity of the naive linear scan method.

Two hierarchical techniques have been widely used for accelerating the performance of ray tracing. They are kd-trees and bounding volume hierarchies (BVHs). kd-trees are constructed by partitioning the space of a scene and thus are classified as spatial partitioning trees. On the other hand, BVHs are constructed by partitioning underlying primitives (e.g., triangles) and thus known as object partitioning trees. They have been demonstrated to work well in most cases [6]. We focus on explaining BVHs in this chapter thanks to its simplicity and wide acceptance in related fields such as collision detection.

[5] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 1997

Bounding volume hierarchies are simple to use and have been widly adopted in related applications including collision detection.

[6] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 2014

10.3.1 Bounding Volumes

We first discuss bounding volumes (BVs). A BV is an object that encloses triangles. Also, the BV should be efficient for performing an intersection test between a ray and the BV. Given this constraint, simple geometric objects have been proposed. BVs commonly used in practice are sphere, Axis-Aligned Bounding Box (AABB), Oriented Bounding Box (OBB), k-DOPs (Discrete Oriented Polytopes), etc. (Fig. 10.6).

Spheres and AABBs are fast for checking intersection tests against a ray. Furthermore, constructing these BVs can be done quite quickly. For example, to compute a AABB from a soup of triangles, we just need to traverse those triangles and compute mim and max values of x, y, and z coordinates of triangles. We then compute the AABB

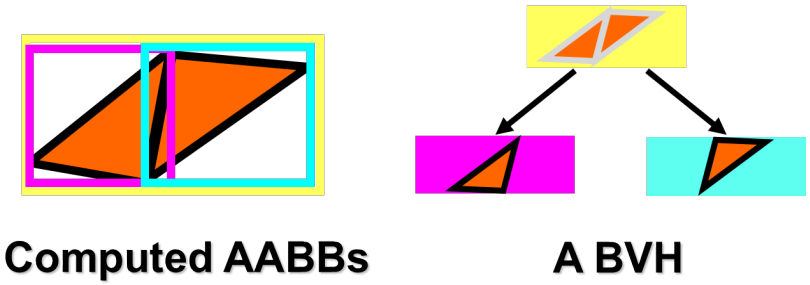


Figure 10.7: This figure shows a BVH with its nodes and AABBs given a model consisting of three triangles. Note that two child AABBs have a spatial overlap, while their nodes have different triangles. As a result, BVHs are classified into an object partitioning tree.

out of those computed min and max values. Since many man made artifacts have box-like shapes, AABB works well for those types. Nonetheless, spheres and AABBs may be too lose BVs, especially when the underlying object is not aligned into such canonical directions or is elongated along a non-canonical direction (Fig. 10.6).

A single BV type is not always better than others, but AABBs work reasonably well and are easy to use.

On the other hand, OBBs and k-DOPs tend to provide tighter bounding, but to require more complex and thus slow intersection tests. Given these trade-offs, an overhead of computing a BV, tightness of bounding, and time spent on intersection tests between a ray and a BV, it is hard to say which BV shows the best performance among all those BVs. Nonetheless, AABBs work reasonably well in models used for games and CAD industry, thanks to its simplicity and reasonable bounding power on those models.

10.3.2 Construction

Let's think about how we can construct a bounding volume hierarchy out of triangles. A simple approach is a top-down construction method, where we partition the input triangles into two sets in a recursive way, resulting in a binary tree. For simplicity, we use AABBs as BVs.

We first construct a root node with its AABB containing all the input triangles. We then partition those triangles into left and right child nodes. To partition those triangles associated with a current node, a simple method is to use a 2 D plane that partitions the longest edge of the current AABB of the node. Once we compute triangle sets for two child nodes, we recursively perform the process until each node has a fixed number of triangles (e.g., 1 or 2).

In the aforementioned method, we explained a simple partitioning method. More advanced techniques have been proposed including optimization techniques with Surface Area Heuristic (SAH) [7]. The SAH method estimates the probability that a BV intersects with random rays, and we can estimate the quality of a computed BVH. It

[7] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs. In *IEEE Symp. on Interactive Ray Tracing*, pages 39–46, 2006

has been demonstrated that this kind of optimizations can be slower than the simple method, but can show shorter traversal time spent on performing ray-BVH intersection tests.

BVHs suits well for dynamic models, since it can be refitted or re-computed from scratch efficiently.

Dynamic models. Many applications (e.g., games) use dynamic or animated models. As a result, it is important to build or update BVHs of models as they are changing. This is one of main benefits of using BVHs for ray tracing, since it is easy to update the BVH of a model, as the model changes its positions or is animated.

One of the most simple methods is to refit the existing BVH in a bottom-up manner, as the model is changing. Each leaf node is associated with a few triangles. As they change their positions, we re-compute the min and max values of the node and update the AABB of the node. We then merge those re-computed AABBs of two child nodes for their parent node by traversing the BVH in a bottom-up manner. This process has the linear time complexity in terms of the number of triangle. Nonetheless, this refitting approach can result in a poor quality, when the underlying objects deform significantly.

To address those problems, many techniques have been proposed. Some of them is to build BVHs from scratch every frame by using many cores [8] and to selectively identify a sub-BVH with poor quality and rebuild only those regions, known as selective restructuring [9]. At an extreme case, the topology of models can change due to fracturing of models. BVH construction methods even for fracturing cases have been proposed [10].

[8] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum (EG)*, 28(2):375–384, 2009

[9] Sungeui Yoon, Sean Curtis, and Dinesh Manocha. Ray tracing dynamic scenes using selective restructuring. *Eurographics Symp. on Rendering*, pages 73–84, 2007

[10] Jae-Pil Heo, Joon-Kyung Seong, DukSu Kim, Miguel A. Otaduy, Jeong-Mo Hong, Min Tang, and Sung-Eui Yoon. FASTCD: Fracturing-aware stable collision detection. In *SCA '10: Proceedings of the 2010 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2010

10.3.3 Traversing a BVH

Once we build a BVH, we now traverse the BVH for ray-triangle intersection tests. Since an AABB BVH provides AABBs, bounding boxes, on the scene in a hierarchical manner, we traverse the BVH in the hierarchical manner.

Given a ray, we first perform an intersection test between the ray and the AABB of the root node. If there is no intersection, it guarantees that there are no intersections between the ray and triangles contained in the AABB. As a result, we skip traversing its sub-tree. If there is an intersection, we traverse its sub-trees by accessing its two child nodes. Among two nodes, it is more desirable to access a node which is located closer to the ray origin, since we aim to identify the first intersection point along the ray starting from the ray origin.

Suppose that we decide to access the left node first. We then store the right node in a stack to process it later. We continue this process until we reach a leaf node containing primitives (e.g., triangles). Once we reach a leaf node, we perform ray-triangle intersection

tests for identifying an intersection point. If it is guaranteed that the intersection point is the closest to the ray origin, we terminate the process. Otherwise, we contribute to traverse the tree, by fetching and accessing nodes in the stack.

Many types of BVHs do not provide a strict ordering between two child nodes given a ray. This characteristic can result in traversing many parts of BVHs, leading to lower performance. Fortunately, this issue has been studied, and improvements such as identifying near and far child nodes have been proposed [11].

[11] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs. In *IEEE Symp. on Interactive Ray Tracing*, pages 39–46, 2006

10.4 Visibility Algorithms

In this chapter, we discussed different aspects of ray tracing. At a higher level, ray casting, a module of ray tracing, is one type of visibility algorithms, since it essentially tells us whether we can see a triangle or not given a ray. In this section, we would like to briefly discuss other visibility algorithms.

The Z-buffer method, an fundamental technique for rasterization (Part I), is another visibility algorithm. The Z-buffer method is an image-space method, which identifies a visible triangle at each pixel of an image buffer by considering the depth value, i.e., Z values of fragments of triangles (Ch. 7.4). Many different visibility or hidden-surface removal techniques have been proposed. Old, but well-known techniques have been discussed in a famous survey [12]. Interestingly, the Z-buffer method was mentioned as a brute-force method in the survey, because of its high memory requirement. Nonetheless, it has been widely adopted and used for many graphics applications, thanks to its simple method, resulting in an easy adoption in GPUs.

While the Z-buffer method was considered as a brute-force method, it is the de-factor standard in the rasterization method thanks to its adoption in modern GPU architectures.

[12] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, 1974

Compared with the Z-buffer, ray casting and ray tracing is much slower, since it uses a hierarchical data structure, and has many incoherent memory access. Ray casting based approaches, however, become more widely accepted in movies and games, because modern GPUs allow to support such complicated operations, and many algorithmic advances such as ray beams utilizing coherence have been prosed. It is hard to predict future exactly, but ray casting based approaches will be supported more and can be adopted as an interactive solution at some point in future.

11
Radiosity

In the last chapter, we discussed ray tracing techniques. While ray tracing techniques can support various rendering effects such as shadow and transparency, their performance was identified too slow to be used for interactive graphics applications. Some of issues of ray tracing is that we generate many rays whenever we change viewpoints. Furthermore, processing those rays take high computation time, and they tend to have random access patterns on underlying data structures (e.g., meshes and bounding volume hierarchy), resulting in high cache misses and lower computational performance.

On the other hand, radiosity emerges as an alternative rendering method that works for special cases with high performance [1]. While radiosity is not designed for handling various rendering effects, it has been widely used to complement other rendering techniques, since radiosity shows high rendering performance of specific material types such as diffuse materials. In other words, radiosiy as well as ray tracing are two common building blocks of designing other advanced rendering techniques, and we thus study this technique in this chapter.

[1] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 212–22, July 1984

11.1 Two Assumptions

Radiosity has two main assumptions (Fig. 11.1):

- **Diffuse material.** We assume that the material type we handle for radiosiy is diffuse or close to the diffuse materials. The ideal diffuse material reflects incoming light into all the possible outgoing directions with the equal amount of light energy, i.e., the same radiance, which is one of radiometric quantity discussed in Sec. 12. Thanks to this diffuse material assumption, any surface looks the same and has the same amount of illumination level given the view point. This in turn simplifies many computations.

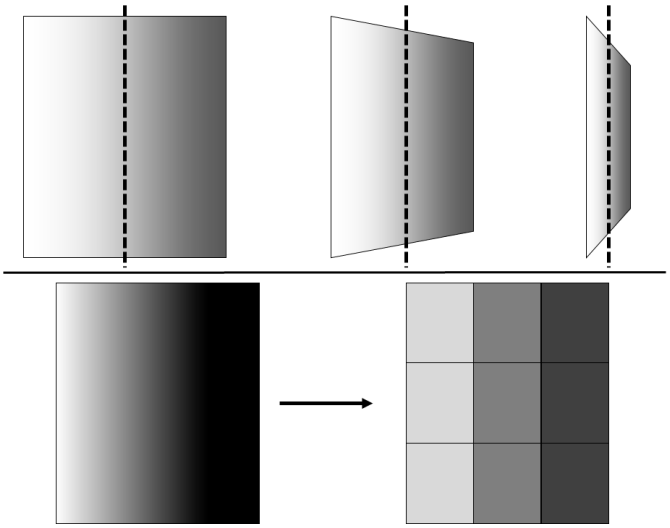


Figure 11.1: Radiosity has the diffuse material assumption (top) and constant illumination per surface element (bottom).

- **Constant radiance per each surface element.** Take a look at a particular surface (e.g., a wall or a desk in your room). The illumination level typically varies smoothly depending on the configuration between a point in the surface and position of light sources. To support this phenomenon, radiosity treats that each surface is decomposed into surface elements such as triangles. It then assumes for simplicity that each surface element has a single value related to the illumination level, especially, radiosity value (Ch. 12). Simply speaking, radiosity is the total incoming (or outgoing) energy arriving in a unit area in a surface.

We will see how these assumptions lead to a simple solution to the rendering problem.

Relationship with finite element method (FEM). As you will see, radiosity can generate realistic rendering results with an interactive performance, while dealing only with diffuse materials and light sources. This was excellent results, when radiosity was proposed back at 1984. Furthermore, approaches and solution for radiosity were novel at the graphics community at that time. Nonetheless, those techniques were originally introduced for simulating heat transfers and have been well established as Finite Element Methods (FEM). FEM was realizing its potential benefits around 1960s and 70s, and was applied even to a totally different problem, physically based rendering. This is a very encouraging story to us. By studying and adopting recently developing techniques into our own problem, we can design very creative techniques in our own field!

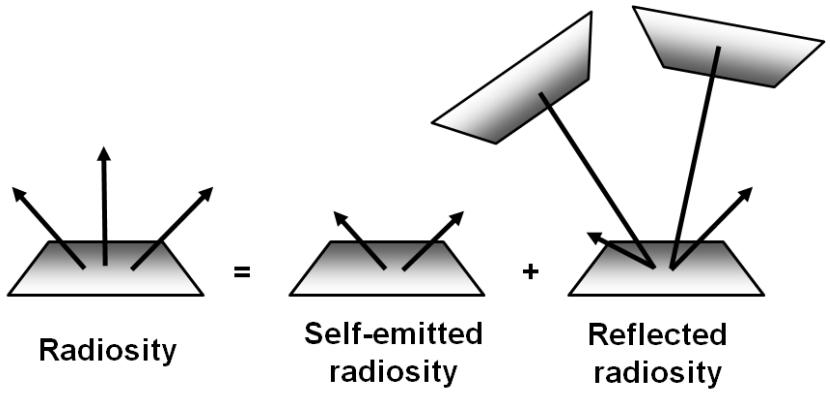


Figure 11.2: The radiosity of a patch is computed by the sum of the self-emitted radiosity from itself and the radiosity reflected and received from other patches.

11.2 *Radiosity Equation*

An input scene to radiosity is commonly composed of triangles. We first subdivide the scene into smaller triangles such that our assumption of the constant radiance per each subdivided triangle is valid. Suppose that there are n different surface elements. We use B_i to denote radiosity of a patch i. Some of such patches can be light sources and thus emit some energy. Since we also assume the light sources to be diffuse emitters, we also use radiosity for such self-emitting patches, and their emitting energy is denoted by $B_{e,i}$.

Intuitively speaking, the radiosity of the patch i is the sum of the self-emitting energy from the patch itself, $B_{e,i}$, and the energy reflected from the patch i by receiving energy from all the other patches (Fig. 11.2. We can then model the interaction between the patch i and different patches as the following:

$$B_i = B_{e,i} + \rho_i \sum_j B_j F(i \to j), \tag{11.1}$$

where j is another index to access all the surface elements in the scene, $F(i \to j)$ is a form factor that describes how much the energy from the patch i arrives at another patch j, and ρ_i is a reflectivity of the patch i.

$B_{e,i}$ and ρ_i are input parameters to the equation and given by a scene designer. The form factor is a term that we can compute depending on the geometric configuration between two patches i and j. The form factor can be understood by the area integration of the rendering equation, which is more general than the radiosity equation. This is discussed in Sec. 13.2. As a result, the unknown terms of the equation is the radiosity B_i of n different patches. Our goal is then to compute such unknown terms. We discuss them in the next section, followed by the overall algorithm of the radiosity

rendering method.

11.3 Radiosity Algorithm

Given the radiosity equation (Eq. 11.1), the unknown term is the radiosity, B_i, per each patch, resulting in n different unknown radiosity values for n patches. Since we can setup n different equation for each patch based on the radiosity equation, overall we have n different equations and unknowns. When we represent such n different equations, we have the following matrix representation:

$$\begin{bmatrix} 1-\rho_1 F(1\to 1) & -\rho_1 F(1\to 2) & \dots & -\rho_1 F(1\to n) \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F(n\to 1) & -\rho_n F(n\to 2) & \dots & 1-\rho_n F(n\to n) \end{bmatrix} \begin{bmatrix} B_1 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} B_{e,1} \\ \vdots \\ B_{e,n} \end{bmatrix} \tag{11.2}$$

The above matrix has the form of $AX = B$, where $X = [B_1 \dots B_n]^T$ is a 1 by n matrix containing unknowns.

To compute the unknown X, we can apply many matrix inversion algorithms including Gaussian elimination that has $O(n^3)$ time complexity [2]. This approach, however, can be very expensive to be used for interactive applications, since the number of surface elements can be hundreds of thousands in practice.

[2] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 2nd edition, 1993

Instead of using exact approaches of computing the linear equations, we can use other numerical approaches such as Jacobi and Gauss-Seidel iteration methods. Jacobi iteration works as the following:

- **Initial values.** Start with initial guesses on radiosity values to surface patches. For example, we can use the direct illumination results using Phong illumination considering the light sources as the initial values for surface patches.
- **Update step.** We plug those values, i.e., old values, into the right term of the radiosity equation (Eq. 11.1), and get new values on B_i. We perform this procedure to all the other patches.
- **Repeat until converge.** We repeat the update step until radiosity values converge.

The Jacobi iteration method has been studied well in numerical analysis, and its properties related to convergence have been well known [3].

[3] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 2nd edition, 1993

One numerical iteration simulates one bounce of the light energy from a patch to another patch.

Effects of numerical iteration. Instead, we discuss how it works in the context of rendering. While performing the update step of the

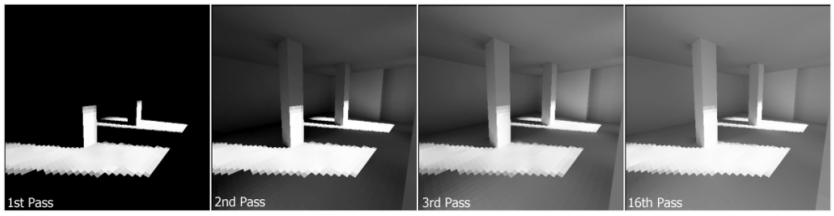


Figure 11.3: This shows a sequence of images computed by different updates, i.e., light bounces, during the radiosity iteration process. This is the courtesy of the wikipedia.

Jacobi iteration, we compute a new radiosity value for each patch from old values. In this process, we compute the new radiosity value received and reflected from other patches. Intuitively, the update step supports one bounce of the light energy from a patch to another patch.

Fig. 11.3 visualizes how radiosity values change as we have different number of update steps, i.e., passes. While only surface elements that are directly visible from the light source are lit in the first pass, other surface elements get brighter as we perform multiple update steps and thus multiple bounces. In a way, this also visualizes how the incoming light energy is distributed across the scene. In the end, we see only the converged result, which is the equilibrium state of the light and material interaction described in the radiosity equation.

Overall algorithm. In summary, we subdivide triangles of the input scene into smaller surface elements, i.e., patches. We then compute radiosity values per each patch by solving the linear equations given by the radiosity equation. For static models, we perform this process only a single time. At runtime, when a viewer changes a view point, we then project those triangles whose color. This projection process is efficiently performed by using the rasterization process in GPUs.

Radiosity is commonly accelerated by adopting the rasterization method

So far, we did not consider view points given by users while computing radiosity values. This is unnecessary, because we do not need to consider view-dependent information for radiosity computation process; note that radiosity algorithm assumes the diffuse materials and emitters and thus we get the same radiance value for any view directions. This is one of the main features of the radiosity algorithm, leading to its strength and weakness of the method.

The basic radiosity method does not support glossy materials.

Drawbacks of the basic radiosity method. The main benefit of the basic radiosity method is that we can re-use the pre-computed radiosity values, even though the user changes the viewpoint. Nonetheless, it has also drawbacks. First of all, the radiosity assumes different materials and emitters, while various scenes have other materials such as glossy materials. Also, when we have dynamic models, we

cannot re-use pre-computed radiosity values and thus re-compute them.

11.4 Light Path Expressions

The radiosity method does support light reflections between diffuse materials, but does not support interactions between glossy materials. Can we represent such light paths that the radiosity method supports?

Regular expressions are used to denote different types of light paths.

Heckbert proposed to use the regular expression to characterize light paths [4]. This approach considers light paths starting from the eye, noted E, to the light, denoted, L. Diffuse, specular, and glossy materials are denoted as D, S, and G, respectively. We also adopt various operations of regular expressions such as | (or), * (zero or more), and + (one or more).

The light paths that radiosity method are then characterized by LD^*E. On the other hand, the classic ray tracing method (Ch. 10) supports $L(DS^*)E$, since it generates secondary rays when a ray hits specular or refractive objects.

[4] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 145–154, August 1990

12
Radiometry

One of important aspects of physically-based rendering is to simulate physical interactions between lights and materials in a correct manner. To explain these physical interactions, we discuss various physical models of light in this chapter. Most rendering effects that we observe can be explained by a simple, geometric optics. Based on this simple light model, we then explain radiometric quantities that are important for computing colors. Finally, we explain basic material models that are used for simulating the physical interaction with lights.

12.1 Physics of Light

Understanding light has drawn major human efforts in physics and resulted in many profound progress on optics and related fields. Light or visible light is a type of electromagnetic radiations or waves that we can see through our eyes. The most general physical model is based on quantum physics and explains the duality of wave and particle natures of light.

While the quantum physics explains the mysterious wave-particle duality, it is rather impossible to simulate the quantum physics for making our applications, i.e., games and movies, at the current computing hardware. One of simpler light models is the wave model that treats light like sound. Such wave characteristics become prominent, when the wavelength of light is similar to sizes of interacting materials, and diffraction is one of such phenomena. For example, when we see sides of CD, we can see rainbow-like color patterns, which are created by small features of the CD surface.

The most commonly used light model used in computer graphics so far is the geometric optics, which treats light propagation as rays. This model assumes that object sizes are much bigger than the wavelength of light, and thus wave characteristics disappear mostly. This geometric optics can support reflection, refraction, etc.

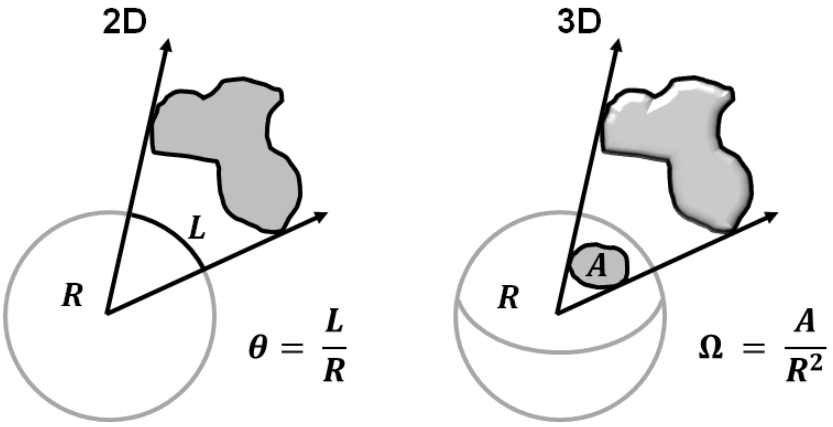


Figure 12.1: Solid angles in 2 D and 3 D cases.

Many rendering methods based on ray tracing assumes the geometric optics, and we also assume this model unless mentioned otherwise.

Our goal is then to measure the amount of energy that a particular ray carries or that a particular location receives from. Along this line, we commonly use a hemisphere, specifically, hemispherical coordinates, to parameterize rays that can arrive at a particular location in a surface. We discuss hemispherical coordinates before we move on to studying radiometry.

Solid angles. We use the concept of solid angles for various integration on the hemisphere. The solid angle is used to measure how much an objected located in 3 D space affects a point in a surface. This metric is very useful for computing shadow and other factors related to visibility. In the 2 D case (the left figure of Fig. 12.1), a solid angle, Ω, of an object is measured by $\frac{L}{R}$, where L is the length of the arc, where the object is projected to in the 2 D hemisphere (or sphere). R is the radius of the sphere; we typically use a unit sphere, where $R = 1$. The unit of the solid angle in the 2 D case is measured by radians. The solid angle mapping to the full circle is 2π radians.

The solid angle in the 3 D case is computed by $\frac{A}{R^2}$, whose unit is steradians (the right figure of Fig. 12.1). A indicates the area subtended by the 3 D object in the hemisphere. For example, the full sphere has 4π steradians.

Hemispherical coordinates. A hemisphere is two dimensional surface and thus we can represent a point on the hemisphere with two parameters such as latitude, θ, and longitude, φ (Fig. 12.2), where $\theta \in [0, \frac{\pi}{2}]$ and $\varphi \in [0, 2\pi]$. Now let's see how we can compute the differential area, dA, on the hemisphere controlled by $d\phi$ and $d\theta$.

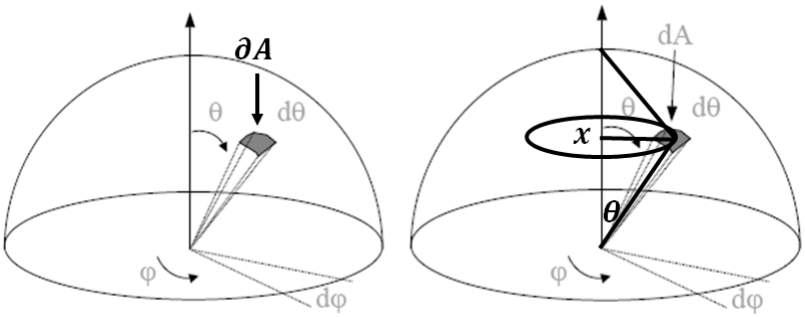


Figure 12.2: Hemispherical coordinates (θ, φ). These images from slides of Kavita Bala.

In infinitely small differential angles, we can treat that the area is approximated by a rectangular shape, whose area can be computed by multiplying its height and width. Its height is given by $d\theta$. On the other hand, its width varies depending on θ; its largest and minimum occur at $\theta = \pi/2$ and $\theta = 0$, respectively.

To compute the width, we consider a virtual circle that touches the rectangular shape of the hemisphere. Let x be the radius of the ch ircle. The radius is then compute by $\sin\theta = \frac{x}{r}, x = r\sin\theta$, where r is the radius of the hemisphere. The width is then computed by applying the concept of the solid angle, and is $r\sin\theta d\phi$. We then have the following differentials:

$$dA = (r\sin\theta d\phi)(rd\theta). \tag{12.1}$$

Based on this equation, we can easily derive differential solid angles, dw:

$$dw = \frac{dA}{r^2} \tag{12.2}$$

$$= \sin\theta d\phi d\theta. \tag{12.3}$$

We use these differential units to define the rendering equation (Ch. 13.1).

12.2 *Radiometry*

In this section, we study various radiometric quantities that are important for rendering. Human perception on brightness and colors depends on various factors such as the sensitivity of photoreceptor cells in our eyes. Nonetheless, those photoreceptor cells receive photons and trigger biological signals. As a result, measuring photons, i.e., energy, is the first step for performing the rendering process.

Power or flux. Power, P, is a total amount of energy consumed per unit time, denoted by dW/dt, where W indicates watt. In our

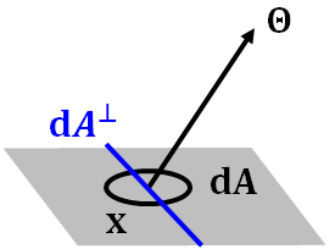


Figure 12.3: Radiance is measured per unit projected area, $dA^{\perp}$, while we receive the energy on the surface A.

rendering context, it is the total amount of energy arriving at (or passing through) a surface per unit time, and also called radiant flux. Its unit is Watt, which is joules per second. For example, we say that a light source emits 50 watts of radiant power or 20 watts of radiant power is incident on a table.

Irradiance or radiosity. Irradiance is power or radiant flux arriving at a surface per unit area, denoted by dW/dA with the unit of W/m^2. Radiant exitance is the radiant flux emited by a surface per unit area, while radiosity is the radiant flux emitted, reflected, or transmitted from a surface per unit area; that is why the radiosity algorithm has its name (Ch. 11). For example, when we have a light source emitting $100W$ of area $0.1m^2$, we say that the radiant existance of the light is $1000W/m^2$.

Radiance is one of the most important radiometric quantity used for physically-based rendering.

Radiance. In terms of computing rendering images, computing the radiance for a ray is the most important radiometric measure. The radiance is radiant flux emitted, reflected, or received by a surface per unit solid angle and per unit projected area, $dA^{\perp}$, whose normal is aligned with the center of the solid angle (Fig. 12.3):

$$L(x \to \Theta) = \frac{d^2P}{d\Theta dA^{\perp}} \tag{12.4}$$

$$= \frac{d^2P}{d\Theta dA\cos\theta}. \tag{12.5}$$

$\cos\theta$ is introduced for considering the projected area.

Diffuse emitter. Suppose that we have an ideal diffuse emitter that emits the equal radiance, L, in any possible direction. Its irradiance on a location is measured as the following:

$$\begin{aligned} E &= \int_{\Theta} L\cos\theta dw_{\Theta}, \\ &= \int_0^{2\pi}\int_0^{\frac{\pi}{2}} L\cos\theta\sin\theta d\theta d\phi = \int_0^{2\pi} d\phi \int_0^{\frac{\pi}{2}} L\cos\theta\sin\theta d\theta \\ &= 2\pi L\frac{1}{2} = L\pi. \end{aligned} \tag{12.6}$$

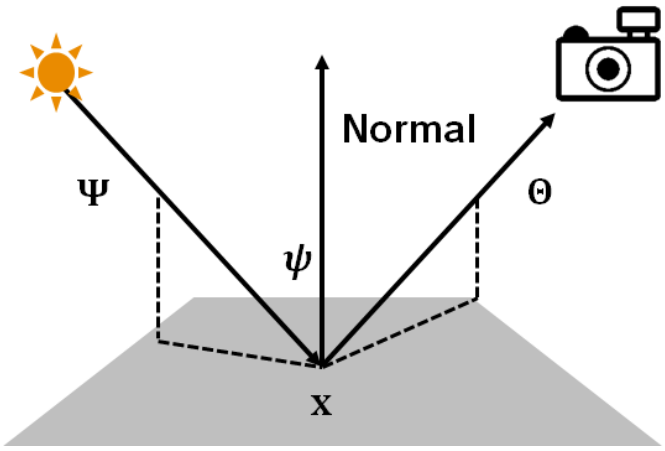


Figure 12.4: A configuration setting for measuring the BRDF is shown. Ψ and Θ are incoming and outgoing directions, while ψ is the angle between the surface normal and Ψ.

where Θ is the hemispherical coordinates, (θ, ϕ).

12.3 *Materials*

We discussed the Snell's law to support the ideal specular (Sec. 10.1. Phong illumination supports ideal diffuse and a certain class of glossy materials (Ch. 8). However, some materials have complex appearances that are not captured by those ideal specular, ideal diffuse, and glossy materials. In this section, we discuss Bidirectional Reflectance Distribution Function (BRDF) that covers a wide variety of materials.

Our idea is to measure an appearance model of a material and to use it within physically based rendering methods. Suppose the light and camera settings shown in Fig. 12.4. We would like to measure how the material reflects incoming radiance with a direction of Ψ into outgoing radiance with a direction of Θ. As a result, BRDF, $f_r(x, \Psi \rightarrow \Theta)$, at a particular location x is a four dimensional function, defined as the following:

$$f_r(x, \Psi \rightarrow \Theta) = \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \Psi)} = \frac{dL(x \rightarrow \Theta)}{L(x \leftarrow \Psi) \cos \psi dw_\Psi}, \tag{12.7}$$

where ψ is the angle between the normal of the surface at x and the incoming direction Ψ, and dw_Ψ is the differential of the solid angle for the light. The main reason why we use differential units, not non-differential units, is that we want to cancel existing light energy in addition to the light used for measuring the BRDF.

The BRDF satisfies the following properties:

1. Reciprocity. Simply speaking, when we switch locations of the camera and light, we still get the same BRDF. In other words, $f_r(x, \Psi \rightarrow \Theta) = f_r(x, \Theta \rightarrow \Psi)$.

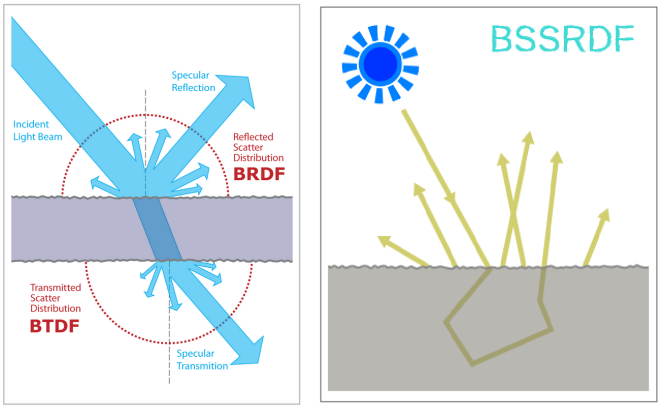


Figure 12.5: These images show interactions between the light and materials that BRDF, BTDF, and BSSRDF. These images are excepted from Wiki.

2. Energy conservation. $\int_\Theta f_r(x, \Psi \rightarrow \Theta) \cos\theta dw_\Theta \leq 1$.

To measure a BRDF of a material, a measuring device, called gonioreflectometer, is used. Unfortunately, measuring the BRDF takes long time, since we have to scan different incoming and outgoing angles. Computing BRDFs in an efficient manner is an active research area.

Material appearance varies depending on wavelengths of lights. To support such material appearance depending on wavelengths of lights, we can measure BRDFs as a function of wavelengths, and use a BRDF given a wavelengths of the light.

12.3.1 Other Distribution Functions

So far, we mainly considered BRDF. BRDF, however, cannot support many other rendering effects such as subsurface scattering.

BRDF considered reflection at a particular point, x. For translucent models, lights can pass through the surface and are reflected in the other side of the surface. To capture such transmittance, BTDF (Bi-direction Transmittance Distribution Function) is designed (Fig. 12.5). Furthermore, light can be emitted from points other than the point x that we receive the light. This phenomenon occurs as a result of transmittance and reflection within a surface of translucent materials. BSSRDF (Bidirectional Surface Scattering Reflection Distribution Function) captures such complex phenomenon. Capturing and rendering these complex appearance models is very important topics and still an active research area.

13
Rendering Equation

In this chapter, we discuss the rendering equation that mathematically explains how the light is reflected given incoming lights. The radiosity equation (Ch. 11) is a simplified model of this rendering equation assuming diffuse reflectors and emitters.

Nonetheless, the rendering equation does not explain all the light and material interactions. Some aspects that the rendering equation does not capture include subsurface scattering and transmissions.

13.1 Rendering Equation

The rendering equation explains how the light interacts with materials. In particular, it assumes geometric optics (Sec. 12.1) and the light and material interaction in an equilibrium status.

The inputs to the rendering equation are scene geometry, light information, material appearance information (e.g., BRDF), and viewing information. The output of the rendering equation is radiance values transferred, i.e., reflected and emitted, from a location to a particular direction. Based on those radiance values for primary rays generated from the camera location, we can compute the final rendered image.

Suppose that we want to compute the radiance, $L(x \rightarrow \Theta)$, from a location x in the direction of Θ [1]. To compute the radiance, we need to sum the emitted radiance, $L_e(x \rightarrow \Theta)$, and the reflected radiance, $L_r(x \rightarrow \Theta)$ (Fig. 13.1). The emitted radiance can be easily given by the input light configurations. To compute the reflected radiance, we need to consider incoming radiance to the location x and the BRDF of the object at the location x. The incoming radiance can come to x in any possible directions, and thus we introduce an integration with the hemispherical coordinates. In other words, the reflected radiance is computed as the following:

$$L_r(x \rightarrow \Theta) = \int_{\Psi} L(x \leftarrow \Psi) f_r(x, \Psi \rightarrow \Theta) \cos\theta_x dw_{\Psi}, \tag{13.1}$$

[1] For simplicity, we use a vector Θ for representing a direction based on the hemispherical coordinates.

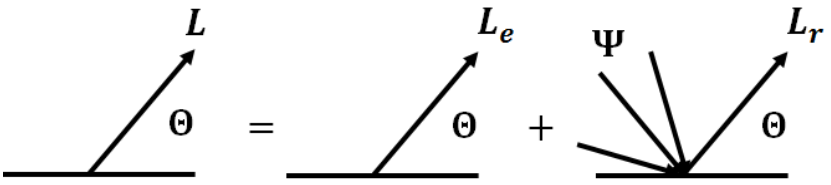


Figure 13.1: The radiance, $L(x \rightarrow \Theta)$, is computed by adding the emitted radiance, $L_e(x \rightarrow \Theta)$, and the reflected radiance, $L_r(x \rightarrow \Theta)$.

where $L(x \leftarrow \Psi)$ is a radiance arriving at x from the incoming direction, Ψ, $\cos\theta_x$ is used to consider the angle between the incoming direction and the surface normal, and the BRDF $f_r(\cdot)$ returns the outgoing radiance given its input.

The rendering equation can be represented in different manners including hemispherical or area integration.

We use the hemispherical coordinates to derive the rendering equation shown in Eq. 13.1, known as hemispherical integration. In some cases, a different form of the rendering equation, specifically area integration, is used. We consider the area integration of the rendering equation in the following section.

13.2 *Area Formulation*

To derive the hemispherical integration of the rendering equation, we used differential solid angles to consider all the possible incoming light direction to the location x. We now derive the area integration of the rendering equation by considering a differential area unit, in a similar manner using the differential solid angle unit.

Let us introduce a visible point, y, given the negated direction, $-\Psi$, of an incoming ray direction, Ψ, from the location x (Fig. 13.2). We can then have the following equation thanks to the invariance of radiance:

$$L(x \leftarrow \Psi) = L(y \rightarrow -\Psi). \tag{13.2}$$

Our intention is to integrate any incoming light directions based on y. To do this, we need to substitute the differential solid angle by the differential area. By the definition of the solid angle, we have the following equation:

$$dw_\Psi = \frac{dA\cos\theta_y}{r_{xy}^2}, \tag{13.3}$$

where θ_y is the angle between the differential area dA and the orthogonal area from the incoming ray direction, and r_{xy} is the distance between x and y.

When we plug the above two equations, we have the following equation:

$$L_r(x \rightarrow \Theta) = \int_y L(y \rightarrow -\Psi) f_r(x, \Psi \rightarrow \Theta) \frac{\cos\theta_x \cos\theta_y}{r_{xy}^2} dA, \tag{13.4}$$

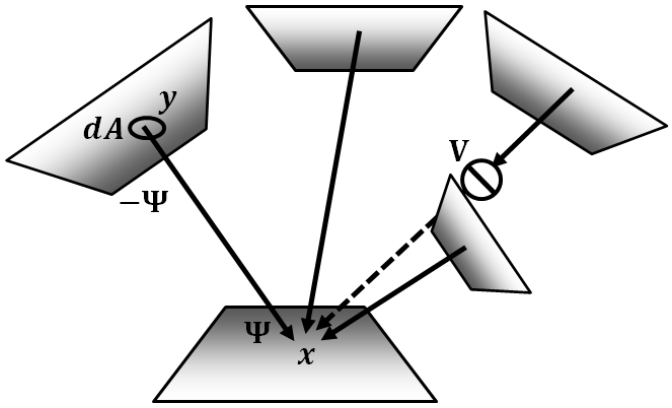


Figure 13.2: This figure shows a configuration for deriving the area formulation of the rendering equation.

where y is any visible area on triangles from x. In the above equation, we need to first compute visible areas from x on triangles. Instead, we would like to integrate the equation on any possible area while considering visibility, $V(x, y)$, which is 1 when y is visible from x, and 0 otherwise. We then have the following area integration of the rendering equation:

$$L_r(x \to \Theta) = \int_A L(y \to -\Psi) f_r(x, \Psi \to \Theta) \frac{\cos\theta_x \cos\theta_y}{r_{xy}^2} V(x, y) dA, \tag{13.5}$$

where A indicates any area on triangles.

Form factor. The radiosity algorithm requires to compute form factors that measure how much light from a patch is transferred to another patch (Sec. 11.2). The area integration of the rendering equation (Eq. 13.5) is equivalent to a form factor between a point on a surface and any points on another surface, while a diffuse BRDF is used in the equation. For the form factor between two surfaces, we simply perform one more integration over the surface.

14
Monte Carlo Integration

In this chapter, we study Monte Carlo integration to evaluate complex integral functions such as our rendering equation. In the next chapter, we will discuss Monte Carlo based ray tracing techniques that are specialized techniques for evaluating the rendering equations.

The rendering equation (Eq. 13.1) is a complex integration function. First of all, to compute a radiance for a ray starting from a surface point x, we need to integrate all the incoming radiances that arrive at x. Moreover, evaluating those incoming radiances requires us to evaluate the same procedure in a recursive way. Since there could be an infinite number of light paths starting from a light source to the eye, it is almost impossible to find an analytic solution for the rendering equation, except simple cases.

Rendering equations can be high dimensional, since we need to consider motion blur and many other effects with time and complex camera lens.

Second, the rendering equation can be high dimensional. The rendering equation shown in Eq. 13.1 is two dimensional. In practice, we need to support the motion blur for dynamic models and moving cameras. Considering such motion blur, we need to integrate radiance over time in each pixel, resulting in three dimensional rendering equation. Furthermore, supporting realistic cameras requires two or more additional dimensions on the equation. As a result, the equation for generating realistic images and video could be five or more dimensional.

Due to these issues, high dimensionality and infinite number of possible light paths, deriving analytic solutions and using deterministic approaches such as quadrature rules are impossible for virtually all of rendering environments that we encounter. Monte Carlo integration was proposed to integrate such high-dimensional functions based on random samples.

Overall, Monte Carlo (MC) integration is a numerical solution to integrate high complex and high-dimensional function. Since it uses sampling, it has stochastic errors, commonly quantified as Mean Squared Error (MSE). Fortunately, MC integration is unbiased,

indicating that it gives us a correct solution with an infinite number of samples on average.

14.1 MC Estimator

Suppose that we have the following integration, whose solution is I:

$$I = \int_a^b f(x)dx. \tag{14.1}$$

The goal of MC integration is to take N different random samples, x_i, that follow the same probability density function, $p(x_i)$. We then use the following estimator:

$$\hat{I} = \frac{1}{N}\sum_i \frac{f(x_i)}{p(x_i)}. \tag{14.2}$$

We now discuss how the MC estimator is good. One of measures for this goal is Mean Squared Error (MSE), measuring the difference between the estimated values, $\hat{Y}_i$, and observed, real values, Y_i:

$$MSE(\hat{Y}) = E[(\hat{Y} - Y)^2] = \frac{1}{N}\sum_i (\hat{Y}_i - Y_i)^2. \tag{14.3}$$

MSE can be decomposed into bias and variances terms as the following:

$$\begin{aligned} MSE(\hat{Y}) &= E\left[\left(\hat{Y} - E[\hat{Y}]\right)^2\right] + \left(E(\hat{Y}) - Y\right)^2 && (14.4)\\ &= Var(\hat{Y}) + Bias(\hat{Y}, Y)^2. && (14.5)\end{aligned}$$

The bias term $Bias(\hat{Y}, Y)$ measures how much the average value of the estimator $\hat{Y}$ is away from its ground-truth value Y. On other hand, the variance term $Var(\hat{Y})$ measures how the estimator values are away from its average values. We would like to discuss bias and variance of the MC estimator (Eq. 14.2).

Bias of the MC estimator. The MC estimator is unbiased, i.e., on average, it returns the correct solution, as shown in below:

$$\begin{aligned} E[\hat{I}] &= E\left[\frac{1}{N}\sum_i \frac{f(x_i)}{p(x_i)}\right] \\ &= \frac{1}{N}\int \sum_i \frac{f(x_i)}{p(x_i)} p(x)dx \\ &= \frac{1}{N}\sum_i \int \frac{f(x)}{p(x)} p(x)dx, \because x_i \text{ samples have the same } p(x) \\ &= \frac{N}{N}\int f(x)dx = I. \end{aligned} \tag{14.6}$$

Variance of the MC estimator. To derive the variance of the MC estimator, we utilize a few properties of variance. Based on those properties, and Independent and Identically Distributed samples (IID) of random samples, the variance of the MC estimator can be derived as the following:

$$\begin{aligned}
Var(\hat{I}) &= Var(\frac{1}{N}\sum_i \frac{f(x_i)}{p(x_i)}) \\
&= \frac{1}{N^2} Var(\sum_i \frac{f(x_i)}{p(x_i)}) \\
&= \frac{1}{N^2}\sum_i Var(\frac{f(x_i)}{p(x_i)}), \because x_i \text{ samples are independent from each other.} \\
&= \frac{1}{N^2} N Var(\frac{f(x)}{p(x)}), \because x_i \text{ samples are from the same distribution.} \\
&= \frac{1}{N} Var(\frac{f(x)}{p(x)}) = \frac{1}{N}\int \left(\frac{f(x)}{p(x)} - E\left[\frac{f(x)}{p(x)}\right]\right)^2 p(x)dx. \quad (14.7)
\end{aligned}$$

As can be in the above equations, the variance of the MC estimator decreases as a function of $\frac{1}{N}$, where N is the number of samples.

Simple experiments with MC estimators. Suppose that we would like to compute the following, simple integration:

$$I = \int_0^1 4x^3 dx = 1. \quad (14.8)$$

We know its ground truth value, 1, for the integration. We can now study various properties of the MC estimator by comparing its result against the ground truth. When we use the uniform sampling on the integration domain, the MC estimator is defined as the following:

$$\hat{I} = \frac{1}{N}\sum_{i=1}^{N} 4x_i^3, \quad (14.9)$$

where $p(x_i) = p_x = 1$, since the sampling domain is $[0, 1]$, and the integration of uniform sampling on the domain has to be one, $\int_0^1 p_x = 1$.

Fig. 14.1 shows how the MC estimator behaves as we have more samples, N. As can be seen, MC estimators approach to its ground truth value, as we have more samples. Furthermore, when we measure the mean and variance of different MC estimators that have different random numbers given the same MC estimator equation (Eq. 14.9), their mean and variance shows the expected behaviors; its mean is same to the ground truth and the variance decreases as a function of $\frac{1}{N}$, respectively.

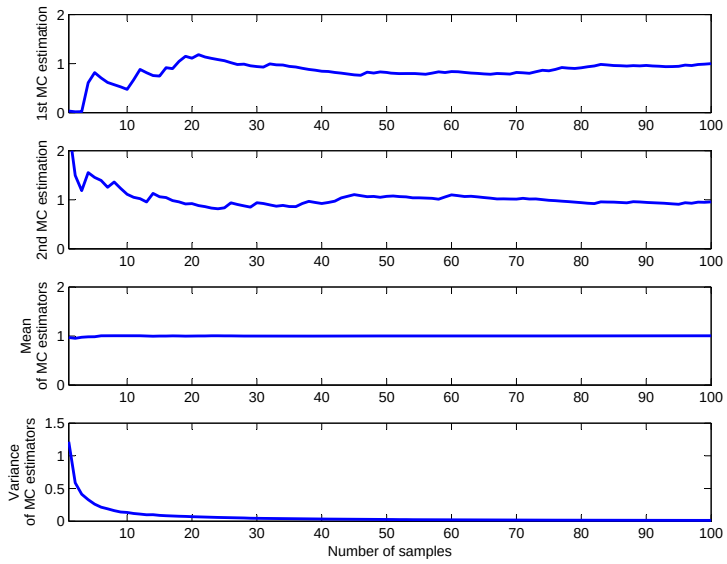


Figure 14.1: The top two subfigures show the first and second MC estimators of $\int_0^1 4x^3 dx$, whose ground truth value is 1. These MC estimators approach to their ground-truth, as we have more number of samples. While these individual MC estimators have up and down depending on their randomly generated values, their mean and variance measured with 600 estimators show the expected behavior, as theoretically predicted in Sec. 14.1. Its source code, mc_int_ex.m, is available.

14.2 *High Dimensions*

Suppose that we have an integration with higher dimensions than one:

$$I = \int \int f(x,y) dx dy. \tag{14.10}$$

Even in this case, our MC estimator is extended straightforwardly to handle such an two-dimensional integration (and other higher ones):

$$\hat{I} = \frac{1}{N} \sum \frac{f(x_i, y_i)}{p(x_i, y_i)}, \tag{14.11}$$

where we generate N random samples following a two dimensional probability density function, $p(x,y)$. We see how to generate samples according to pdf in Sec. 14.4. This demonstrates that MC integration supports well high dimensional integrations including the rendering equation with many integration domains, e.g., image positions, time, and lens parameters.

In addition, MC integration has the following characteristics:

- **Simplicity.** We can compute MC estimators based only on point sampling. This results in very convenient and simple computation.
- **Generality.** As long as we can compute values at particular points of functions under the integration, we can use MC estimations. As a result, we can compute integrations of discontinuous functions, high dimensional functions, etc.

Example. Suppose that we would like to compute the following integration defined over a hemisphere:

$$I = \int_{\Theta} f(\Theta) dw_{\Theta}, \tag{14.12}$$

$$= \int_0^{2\pi} \int_0^{\frac{\pi}{2}} f(\theta, \phi) \sin\theta d\theta d\phi. \tag{14.13}$$

where Θ is the hemispherical coordinates, (θ, ϕ).

The MC estimator for the above integration can be defined as follows:

$$\hat{I} = \frac{1}{N} \sum \frac{f(\theta_i, \phi_i) \sin\theta}{p(\theta_i, \phi_i)}, \tag{14.14}$$

where we generate (θ_i, ϕ_i) following $p(\theta_i, \phi_i)$.

Now let's get back to the irradiance example mentioned in Sec. 12.2. The irradiance equation we discussed in the irradiance example is to use $L_s \cos\theta$ for $f(\theta, \phi)$. In this case, the MC estimator of Eq. 14.14 is transformed to:

$$\hat{I} = \frac{1}{N} \sum \frac{L_s \cos\theta \sin\theta}{p(\theta_i, \phi_i)}. \tag{14.15}$$

One can use different pdf $p(\theta, \phi)$ for the MC estimator, but we can use the following one:

$$p(\theta_i, \phi_i) = \frac{\cos\theta \sin\theta}{\pi}, \tag{14.16}$$

where the integration of the pdf in the domain is one: i.e., $\int_0^{2\pi} \int_0^{\frac{\pi}{2}} \cos\theta \sin\theta = 1$. Plugging the pdf into the estimator of Eq. 14.14, we get the following:

$$\hat{I} = \frac{\pi}{N} \sum L_s. \tag{14.17}$$

14.3 *Importance Sampling*

In this section, we see how different pdfs affect variance of our MC estimators. As we see in Sec. 14.1, our MC estimator is unbiased regardless of pdf employed, i.e., its mean value becomes the ground truth of the integration. Variances, however, vary depending on chosen pdf.

Let's see the example integration, $I = \int_0^1 4x^3 dx = 1$, again. In the following, we test three different pdfs and see their variance:

- $p(x) = 1$. As the simplest choice, we can use the uniform distribution on the domain. The variance of our MC estimator, $\hat{I} = \frac{1}{N} \sum_i 4x_i^3$ is $\frac{36}{28N} \approx \frac{1.285}{N}$, according to the variance equation (Eq. 14.7).

- $p(x) = x$**.** The variance of this MC estimator, $\frac{1}{N}\sum_i 4x^2$, is $\frac{14}{12N} \approx \frac{1.666}{N}$. Its variance is reduced from the above, uniform pdf!
- $p(x) = 4x^3$**.** The shape of this pdf is same to the underlying function under the integration. In this case, its variance turns out to be zero.

The variance of an MC estimator goes to zero, when the shape of its pdf is same to the underlying function under the integration. We, however, do not know such a shape of the rendering equation!

As demonstrated in the above examples, the variance of a pdf decreases, as the distribution of a pdf gets closer to the underlying function $f(x)$. Actually, when the pdf $p(x)$ is set to be $\frac{f(x)}{\int f(x)dx} = \frac{f(x)}{I}$, the ideal distribution, we get the lowest variance, zero. This can be shown as the following:

$$\begin{aligned} Var(\hat{I}) &= \frac{1}{N}\int (\frac{f(x)}{p(x)} - I)^2 p(x)dx \\ &= \frac{1}{N}\int (I - I)^2 p(x)dx \\ &= 0. \end{aligned} \tag{14.18}$$

Unfortunately, in some cases, we do not know the shape of the function under the integration. Especially, this is the case for the rendering equation. Nonetheless, the general idea is to generate more samples on high values on the function, since this can reduce the variance of our MC estimator, as demonstrated in aforementioned examples. In the same reason, when the pdf is chosen badly, the variance of our MC estimator can even go higher.

The main idea of importance sampling is to generate more samples on high values on the function.

This is the main idea of importance sampling, i.e., generate more samples on high values on the underlying function, resulting in a lower variance.

Fortunately, we can intuitively know which regions we can get high values on the rendering equation. For example, for the light sources, we can get high radiance values, and we need to generate rays toward such light sources to reduce the variance in our MC estimators. Technical details on importance sampling are available in Ch. 14.3.

14.4 Generating Samples

We can use any pdf for the MC estimator. In the case of the uniform distribution, we can use a random number generator, which generates random numbers uniformly given a range.

The question that we would like to ask in this section is how we can generate samples according to the pdf $p(x)$ different from the uniform pdf.

Fig. 14.2 shows a pdf and its cdf (cumulative distribution function) in a discrete setting. Suppose that we would like to generate samples

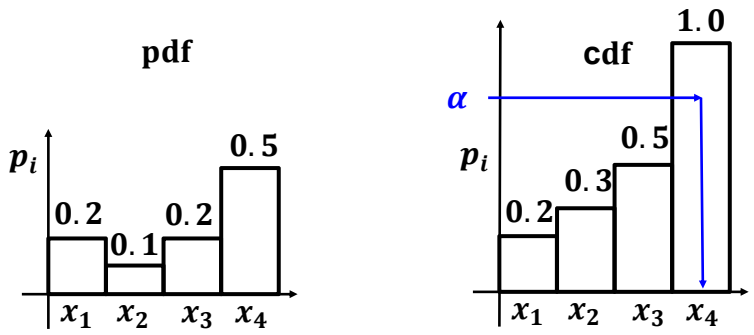


Figure 14.2: This figure shows a pdf and its cdf. Using the inverse cumulative distribution function generates samples according to the pdf by utilizing its cdf.

according to the pdf. In this case, x_1, x_2, x_3, x_4 are four events, whose probabilities are $0.2, 0.1, 0.2, 0.5$, respectively. In other words, we would like to generate those events with the pre-defined pdf.

We can use an inverse cumulative distribution function to generate samples according to a pdf.

A simple method of generating samples according to the pdf is to utilize its cdf (Fig. 14.2). This is known to be inverse cumulative distribution function. In this method, we first generate a random number α uniformly in the rnage of [0, 1). When the random number α is in the range $[\sum_0^{i-1} p_i, \sum_0^i p_i)$, we return a sample of x_i.

Let's see the probability of generating a sample x_i in this way to be p_i, as the following:

$$\begin{aligned} p(x_i) &= p(\alpha \in [\sum_0^{i-1} p_i, \sum_0^i p_i]) \\ &= p(\sum_0^i p_i) - p(\sum_0^{i-1} p_i) \\ &= p_i, \end{aligned} \tag{14.19}$$

where p_0 is set to be zero. So far, we see the discrete case, and we now extend it to the continuous case.

Continuous case. Suppose that we have a pdf, $p(x)$. Its cdf function, $F_X(x)$, is defined as $F_X(x) = p(X < x) = \int_{-\infty}^{x} p(x)dx$. We then generate a random number α uniformly in a range $[0, 1]$. A sample, y, is generated as $y = F_X^{-1}(\alpha)$.

Example for the diffuse emitter. Let's consider the following integration of measuring the irradiance with the diffuse emitter and our

sampling pdf:

$$I = \frac{1}{\pi}\int_{\Theta} dw_{\Theta},$$
$$= \frac{1}{\pi}\int_0^{2\pi}\int_0^{\frac{\pi}{2}} \sin\theta\cos\theta d\theta d\phi. \tag{14.20}$$
$$p(\theta,\phi) = \frac{\sin\theta\cos\theta}{\pi}, \tag{14.21}$$

where $\int\int p(\theta,\phi)d\theta d\phi = 1$.

Our goal is to generate samples according to the chosen pdf. We first compute its cdf, $CDF(\theta,\phi)$, as the following:

$$CDF(\theta,\phi) = \int_0^{\phi}\int_0^{\theta} \frac{\sin\theta\cos\theta}{\pi} d\theta d\phi$$
$$= (1-\cos^2\theta)\frac{\phi}{2\pi} = F(\theta)F(\pi), \tag{14.22}$$

where $F(\theta)$ and $F(\pi)$ are $(1-\cos^2\theta)$ and $\frac{\phi}{2\pi}$, respectively. Since the pdf is two dimensional, we generate two random numbers, α and β. We then utilize inverse function of those two separated functions of $F(\theta)$ and $F(\phi)$:

$$\theta = F^{-1}(\alpha) = \cos^{-1}\sqrt{1-\alpha},$$
$$\phi = F^{-1}(\beta) = 2\pi\beta. \tag{14.23}$$

The aforementioned, the inverse CDF method assumes that we can compute the inverse of the CDF. In some cases, we cannot compute the inverse of CDFs, and thus cannot use the inverse CDF method. In this case, we can use the rejection method.

In the rejection method, we first generate two random numbers, α and β. We accept β, only when $\alpha \leq p(\beta)$ (Fig. 14.3). In the example of Fig. 14.3, the ranges of α and β are $[0,1]$ and $[a,b]$. In this approach, we can generate random numbers β according to the pdf $p(x)$ without using its cdf. Nonetheless, this approach can be inefficient, especially when we do not accept and thus reject samples. This inefficiency occurs when the value of $p(x)$ is smaller than the upper bound, which we generate such random numbers up to. The upper bound of α in our example is 1.

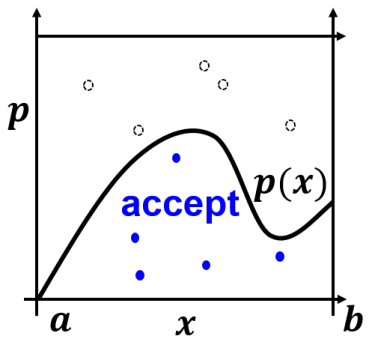


Figure 14.3: In the rejection method, we generate random numbers and accept numbers only when those numbers are within the pdf $p(x)$.

15
Monte Carlo Ray Tracing

In the prior chapters, we have discussed the rendering equation, which is represented in a high dimensional integral equation (Ch. 13.1), followed by the Monte Carlo integration method, a numerical approach to solve such equations (Ch. 14). In this chapter, we discuss how to use the Monte Carlo integration method to solve the rendering equation. This algorithm is known as a Monte Carlo ray tracing method. Specifically, we discuss the path tracing method that connects the eye and the light with a light path.

15.1 Path Tracing

The rendering equation shown below is a high dimensional integration equation defined over a hemisphere. The radiance that we observe from a location x to a direction Θ, $L(x \rightarrow \Theta)$, is defined as the following:

$$\begin{aligned} L(x \rightarrow \Theta) &= L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta), \\ L_r(x \rightarrow \Theta) &= \int_{\Psi} L(x \leftarrow \Psi) f_r(x, \Psi \rightarrow \Theta) \cos\theta_x dw_{\Psi}, \end{aligned} \tag{15.1}$$

where $L_e(\cdot)$ is a self-emitted energy at the location x, $L_r(x \rightarrow \Theta)$ is a reflected energy, $L(x \leftarrow \Psi)$ is a radiance arriving at x from the incoming direction, Ψ, $\cos\theta_x$ is used to consider the angle between the incoming direction and the surface normal, and the BRDF $f_r(\cdot)$ returns the outgoing radiance given its input. Fig. 15.1 shows examples of the reflected term and its incoming radices.

$L(x \rightarrow \Theta)$ of Eq. 15.1 consists of two parts, emitted and reflected energy. To compute the emitted energy, we check whether the hit point x is a part of a light source. Depending whether it is in a light source or not, we compute its self-emitted energy.

The main problem of computing the radiance is on computing the reflected energy. It has several computational issues:

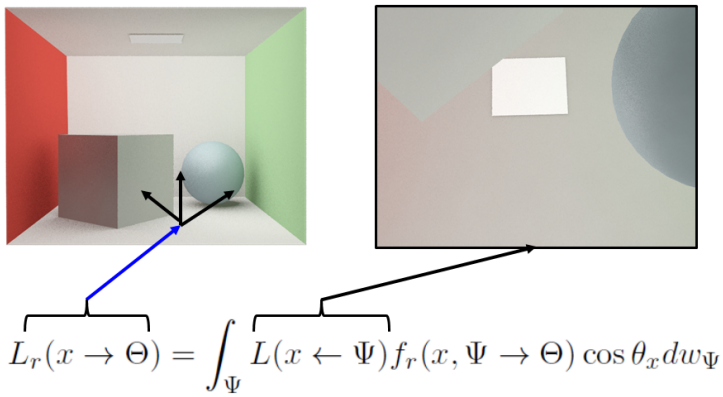


Figure 15.1: This figure shows graphical mapping between terms of the rendering equation and images. The right image represents the incoming radiance passing through the hemisphere.

1. Since the rendering equation is complex, its analytic solution is not available.
2. Computing the reflected energy requires us to compute the incoming energy $L(x \leftarrow \Psi)$, which also recursively requires us to computer another incoming energy. Furthermore, there are an infinite number of light paths from the light sources and to the eye. It is virtually impossible to consider all of them.

Since an analytic approach to the rendering equation is not an option, we consider different approaches, especially numerical approaches. In this section, we discuss the Monte Carlo approach (Ch. 14) to solve the rendering equation. Especially, we introduce path tracing, which generates a single path from the eye to the light based on the Monte Carlo method.

15.2 MC Estimator to Rendering Equation

Given the rendering equation shown in Eq. 15.1, we omit the self-emitting term $L_e(\cdot)$ for simplicity; computing this term can be done easily by accessing the material property of the intersecting object with a ray.

To solve the rendering equation, we apply the Monte Carlo (MC) approach, and the MC estimator of the rendering equation is defined as the following:

$$\hat{L}_r(x \to \Theta) = \frac{1}{N} \sum_{i=1}^{N} \frac{L(x \leftarrow \Psi_i) f_r(x, \Psi_i \to \Theta) \cos\theta_x}{p(\Psi_i)}, \tag{15.2}$$

where Ψ_i is a randomly generated direction over the hemisphere and N is the number of random samples generated.

To evaluate the MC estimator, we generate a random incoming direction Ψ_i, which is uniformly generated over the hemisphere. We

Figure 15.2: Top: computing the outgoing radiance from x requires us to compute the radiance from y to x, which is also recursively computed by simulating additional bounce to y. Bottom: this sequence visualizes rendering results by considering the direct emission and single, double, and triple bounces, adding more energy to the image. Images are excerpted from slides of Prof. Bala.

then evaluate BRDF $f_r(\cdot)$ and the cosine term. The question is how to compute the radiance we can observe from the incoming direction $L(x \leftarrow \Psi_i)$. To compute the radiance, we compute a visible point, y, from x toward Ψ_i direction and then recursively use another MC estimator. This recursion process effectively simulates an additional bounce of photon (Fig. 15.2), and repeatedly performing this process can handle most light transports that we observe in our daily lives.

The aforementioned process uses the recursion process and can simulate various light transport. The recursion process terminates when a ray intersects with a light source, establishing a light path from the light source to the eye. Unfortunately, hitting the light source can have a low probability and it may require an excessive amount of recursion and thus computational time.

Many heuristics are available to break the recursion. Some of them uses a maximum recursion depth (say, 5 bounces) and uses some thresholds on radiance difference to check whether we go into a more recursion depth. These are easy to implement, but using these heuristics and simply ignoring radiances that we can compute with additional bounces results in bias in our MC estimator. To terminate the recursion process without introducing a bias, Russian roulette is introduced.

Russian roulette. Its main idea is that we artificially introduce a case where we have zero radiance, which effectively terminate recursion process. The Russian roulette method realizes this idea without introducing a bias, but with an increased variance. Suppose that we aim to keep the recursion P percentage (e.g., 95%), i.e., cancel the

recursion $1 - P$ percentage. Since we lose some energy by terminating the recursion, we increase the energy when we accept the recursion, in particular, $\frac{1}{P}$, to compensate the lost energy.

In other words, we use the following estimator:

$$\hat{I}_{roulette} = \begin{cases} \frac{f(x_i)}{P} & \text{if } x_i \leq P, \\ 0 & \text{if } x_i > P. \end{cases} \tag{15.3}$$

One can show its bias to be zero, but also show that the original integration is reformulated as the following with a substitute, $y = Px$:

$$I = \int_0^1 f(x)dx = \int_0^P \frac{f(y/P)}{P} dy. \tag{15.4}$$

While the bias of the MC estimate with the Russian roulette is zero, its variance is higher than the original one, since we have more drastic value difference, zero value in a region, while bigger values in other regions, on our sampling.

A left issue is how to choose the constant of P. Intuitively, P is related to the reflectance of the material of a surface, while $1 - P$ is considered as the absorption probability. As a result, we commonly set P as the albedo of an object. For example, albedo of water, ice, and snow is approximately about 7%, 35%, and 65%, respectively.

Branching factor. We can generate multiple ray Samples Per Pixel (SPP). For each primary ray sample in a pixel, we compute its hit point x and then need to estimate incoming radiance to x. The next question is how many secondary rays we need to generate for estimating the incoming radiance well. This is commonly known as a branching factor. Intuitively, generating more secondary rays, i.e., having a higher branching factor, may result in better estimation of incoming radiance. In practice, this approach turns out to be less effective than having a single branching factor, generating a single secondary ray. This is because while we have many branching factors, their importance can be less significant than other rays, e.g., primary ray. This is related to importance sampling (Ch. 14.3) and is discussed more there.

Path tracing is one of simple MC ray tracing for solving the rendering equation. Since it is very slow, it is commonly used for generating the reference results compared to other advanced techniques.

Path tracing. The rendering algorithm with a branching factor of one is called path tracing, since we generate a light path from the eye to the light source. To perform path tracing, we need to set the number of ray samples per pixel (SPP), while the branching factor is set to be one. Once we have N samples per each pixel, we apply the MC estimator, which is effectively the average sum of those N sample values, radiance.

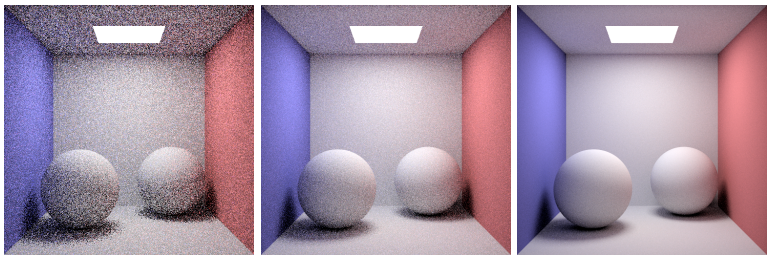


Figure 15.3: This figure shows images that are generated with varying numbers of samples per each pixel. Note that direct illumination sampling, generate a ray toward the light (Sec. 16.1), is also used. From the left, 1 spp (sample per pixel), 4 spp, and 16 spp are used.

Fig. 15.3 shows rendering results with different number of ray samples per pixel. As we use more samples, the variance, which is observed as noise, is reduced.

The theory tells us that as we generate more samples, the variance is reduced more, but it requires a high number of samples and long computational time. As a result, a lot of techniques have been developed to achieve high-quality rendering results while reducing the number of samples.

Programming assignment. It is very important to see how the rendering results vary as a function of ray samples and a different types of sampling methods. Fortunately, many ray tracing based rendering methods are available. Some of well known techniques are Embree, Optix, and pbrt (Sec. 9.6). Please download one of those softwares and test the rendering quality with different settings. In my own class, I ask my students to download pbrt and test uniform sampling and an adaptive sampling method that varies the number of samples. Also, measuring its error compared to a reference image is important to analyze different rendering algorithms in a quantitative manner. I therefore ask to compute a reference image, which is typically computed by generating an excessive number of samples (e.g., 1 k or 10 k samples per pixel), and measure the mean of squared root difference between a rendering result and its reference. Based on those computed errors, we can know which algorithm is better than the other.

15.2.1 Stratified Sampling

We commonly use a uniform distribution or other probability density function to generate a random number. For the sake of simple explanation, let assume that we use a uniform sampling distribution on a sampling domain. While those random numbers in a domain, say, $[0, 1)$, are generated in a uniform way, some random numbers can be arbitrarily close to each other, resulting in noise in the estimation.

A simple method of ameliorating this issue is to use stratified sampling, also known as jittered sampling. Its main idea is to partition

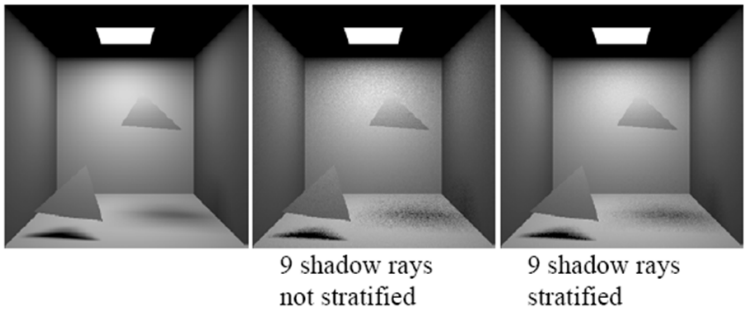


Figure 15.4: The reference image is shown on the leftmost, while images with and without stratified sampling are shown on the right. Images are excerpted from slides of Prof. Bala.

the original sampling domains into multiple regions, say, $[0, 1/2)]$ and $[1/2, 1)$, and perform sampling in those regions independently.

While this approach cannot avoid a close proximity of those random samples, it has been theoretically and experimentally demonstrated to reduce the variance of MC estimators. Fig. 15.4 shows images w/ and w/o using stratified sampling. We can observe that the image with stratified sampling shows less noise.

Theoretically, stratified sampling is shown to reduce the variance over the non-stratified approach. Suppose X to be a random variable representing values of our MC sampling. Let k to be the number of partitioning regions of the original sampling domain, and Y to be an event indicating which region is chosen among k different regions. We then have the following theorem:

Theorem 15.2.1 (Law of total variance)**.** $Var[X] = E(Var[X|Y]) + Var(E[X|Y])$.

Proof.

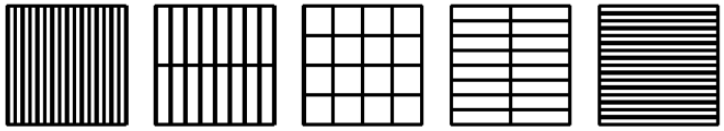
$$\begin{aligned} Var[X] &= E[X^2] - E[X]^2 \\ &= E[E[X^2|Y]] - E[E[X|Y]]^2, \because \text{Law of total expectation} \\ &= E[Var[X|Y]] + E[E[X|Y]^2] - E[E[X|Y]]^2, \\ &= E[Var[X|Y]] + Var(E[X|Y]). \end{aligned} \tag{15.5}$$

□

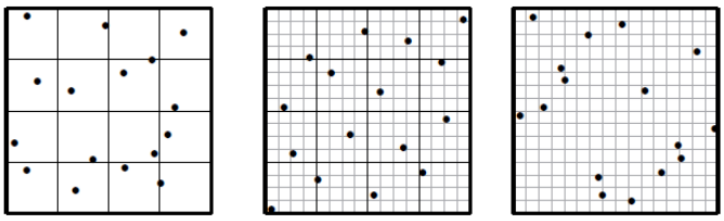
According to the law of total variance, we can show that the variance of the original random variance is equal to or less than the variance of the random variance in each sub-region.

$$Var[X] \geq E(Var[X|Y]) = \frac{1}{k} k Var[X|Y_r] = Var[X|Y_r], \tag{15.6}$$

where Y_r is an event indicating that random variances are generated given each sub-region, and we assume iid for those sub-regions.



(a) All the elementary intervals with the volume of $\frac{1}{16}$.



(b) This figure shows sampling patterns of jittered, Sobol, and N-Rooks samplings, respectively from the left.

Figure 15.5: These images are excerpted from the cited paper.

N-Rooks sampling. N-Rooks sampling or Latin hypercube sampling is a variant of stratified sampling with an additional requirement that has only a single sample in each row and column of sampling domains. An example of N-Rooks sampling is shown in Fig. 15.5. For stratified sampling, we generate N^d samples for a d-dimensional space, where we generate N samples for each space. On the other hand, since it generates only a single sample per each column and row, we can arbitrary generate N samples when we create N columns and rows for high dimensional cases.

Sobol sequence. Sobol sequence is designed to maintain additional constraints for achieving better uniformity. It aims to generate a single sample on each elementary interval. Instead of giving its exact definition, we show all the elementary intervals having the volume of $\frac{1}{16}$ in the 2 D sampling space in Fig. 15.5; images are excerpted from [1].

[1] Thomas Kollig and Alexander Keller. Efficient multidimensional sampling. *Comput. Graph. Forum*, 21(3):557–563, 2002

15.3 *Quasi-Monte Carlo Sampling*

Quasi-Monte Carlo sampling is another numerical tool to evaluate integral interactions such as the rendering equation. The main difference over MC sampling is to use deterministic sampling, not random sampling. While quasi-Monte Carlo sampling uses deterministic sampling, those samples are designed to look random.

The main benefit of using quasi-Monte Carlo sampling is that we can have a particular guarantee on error bounds, while MC methods do not. Moreover, we can have a better convergence to Monte Carlo sampling, especially, when we have low sampling dimensions and need to generate many samples [2].

Specifically, the probabilistic error bound of the MC method

[2] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992

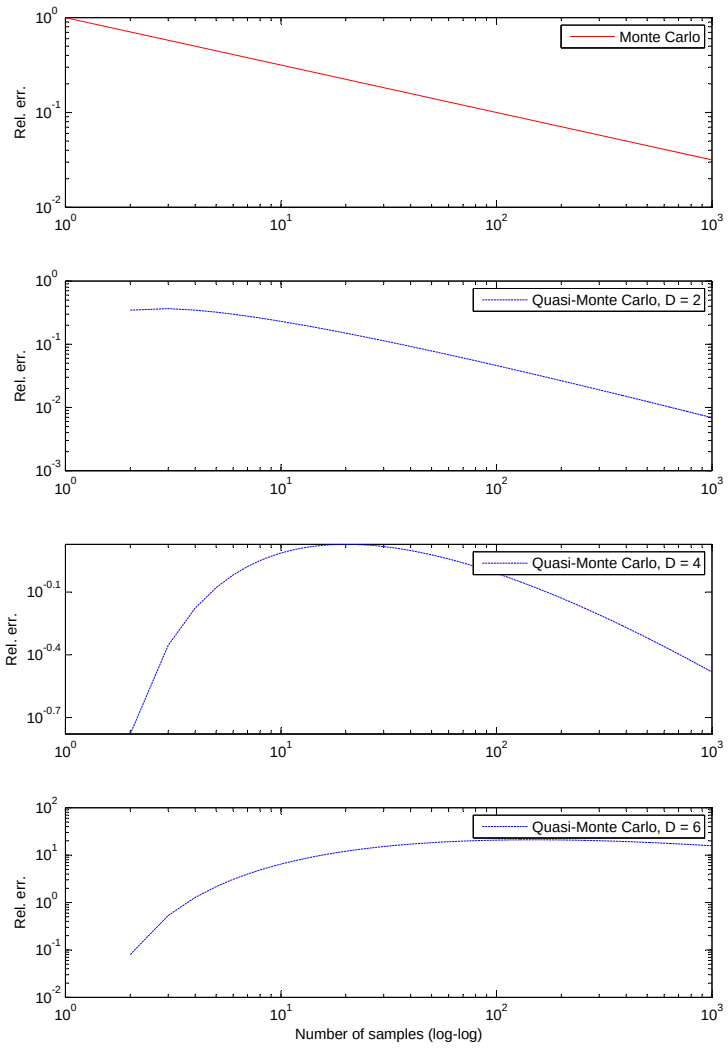


Figure 15.6: This figure shows error behavior of MC and quasi-Monte Carlo methods. They are not aligned in the same error magnitude. As a result, only shapes of these curves are meaningful. The basic quasi-Motel Carlo shows better performance than MC on low dimensional spaces (e.g, two).

reduces $O(\frac{1}{\sqrt{N}})$. On the other hand, the quasi-Monte Carlo can provide a deterministic error bound of $O(\frac{\log N^{D-1}}{N})$ for a well chosen set of samples and for integrands with a low degree of regularity, where D is the dimensionality. Better error bounds are also available for integrands with higher regularity.

Fig. 15.6 shows shapes of two different error bounds of Monte Carlo and quasi-Monte Carlo. Note that they are not aligned in the same error magnitude, and thus only their shapes are meaningful. Furthermore, the one of MC is a probabilistic bound, while that of quasi-Monte Carlo is a deterministic bound. The quasi-Monte Carlo has demonstrated to show superior performance than MC on low dimensional sample space (e.g., two). On the other hand, for a high dimensional case, say six dimensional case, the quasi-Monte Carlo is

not effectively reducing its error on a small number of samples.

The question is how to construct such a deterministic sampling pattern than looks like random and how to quantify such pattern? A common approach for this is to use a discrepancy measure that quantifies the gap, i.e. discrepancy, between the generated sampling and an ideal uniform and random sequence. Sampling methods realizing low values for the discrepancy measure is low-discrepancy sampling.

Various stratified sampling techniques such as Sobol sequence is also used as a low-discrepancy sampling even for the quasi-Monte Carlo sampling, while we use pre-computed sampling pattern and do not randomize during the rendering process. In additional to that, other deterministic techniques such as Halton and Hammersley sequences are used. In this section, we do not discuss these techniques in detail, but discuss the discrepancy measure that we try to minimize with low-discrepancy sampling.

For the sake of simplicity, suppose that we have a sequence of points $P = \{x_i\}$ in a one dimensional sampling space, say $[0, 1]$. The discrepancy measure, $D_N(P, x)$, can be defined as the following:

$$D_N(P, x) = |x - \frac{n}{N}|, \tag{15.7}$$

where $x \in [0, 1]$ and n is the number of points that are in $[0, x]$. Intuitively speaking, we can achieve uniform distribution by minimizing this discrepancy measure. Its general version is available at the book of Niederreiter [3];see pp. 14.

[3] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992

Randomized quasi-Monte Carlo integration. While quasi-Monte Carlo methods have certain benefits over Monte Carlo approaches, it also has drawbacks. Some of them include 1) it shows better performance over MC methods when we have smaller dimensions and the number of samples are high, and 2) its deterministic bound are rather complex to compute. Also, many other techniques (e.g., reconstruction) are based on stochastic analysis and thus the deterministic nature may result in lose coupling between different rendering modules.

To address the drawbacks of quasi-Monte Carlo approaches, randomization on those deterministic samples by permutation can be applied. This is known as randomized quasi-Monte Carlo techniques. For example, one can permute cells of 2 D sample patterns of the Sobol sequence and can generate a randomized sampling pattern. We can then apply various stochastic analysis and have an unbiased estimator. Fig. 15.7 shows error reduction rates of different sampling methods; images are excepted from [4].

[4] Thomas Kollig and Alexander Keller. Efficient multidimensional sampling. *Comput. Graph. Forum*, 21(3):557–563, 2002

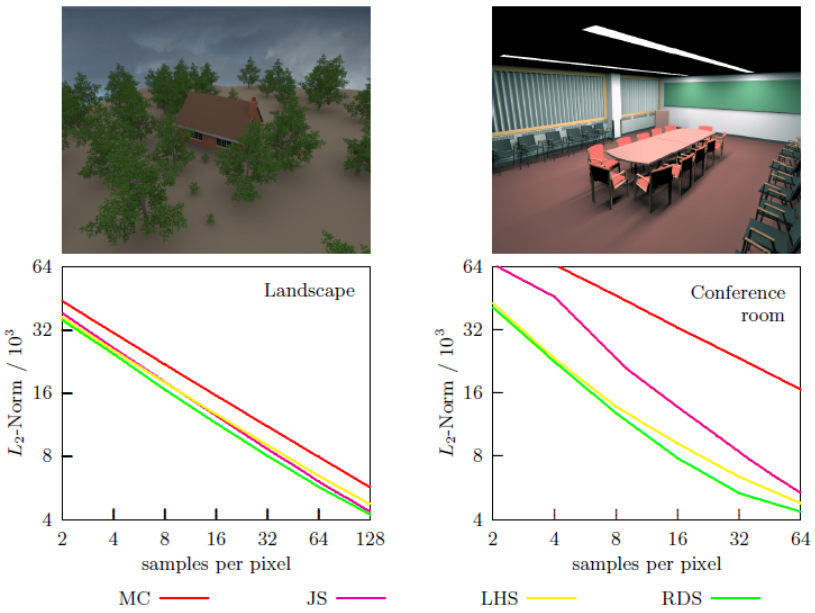
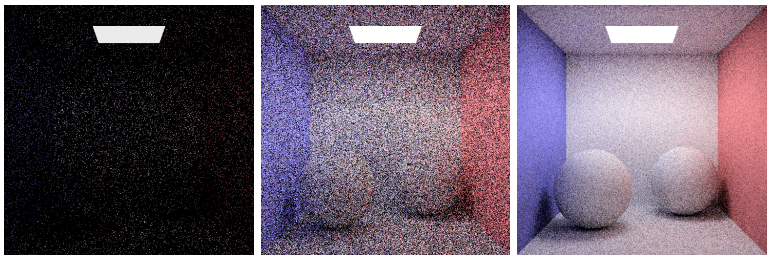


Figure 15.7: These graphs show different error reduction rates of Monte Carlo (MC), jittered (JS), Latin hypercube (LHS), and randomized Sobol sequence (RDS). These techniques are applied to four dimensional rendering problems with direct illumination.

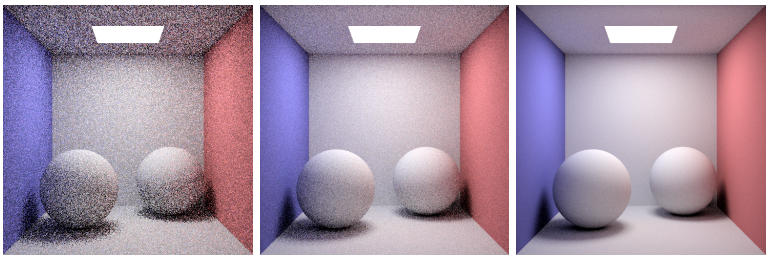
16
Importance Sampling

In the last chapter, we discussed Monte Carlo (MC) ray tracing, especially, path tracing that generates a light path from the camera to the light source. While it is an unbiased estimator, it has significant variance, i.e., noise, when we have a low ray samples per pixel. To reduce the noise of MC generated images, we studied quasi-Monte Carlo technique in Sec. 15.3.

In this chapter, as an effective way of reducing the variance, we discuss importance sampling. We first discuss an importance sampling method considering light sources, called direct illumination method. We then discuss other importance sampling methods considering various factors of the rendering equation.



(a) Results w/o direct illumination. From the left, 1 spp, 4 spp, and 16 spp are used.



(b) Results w/ direct illumination.

Figure 16.1: These images are generated by path tracer w/ and w/o direct illumination. They are created by using a path tracer created by Ritchie et al. `http://web.stanford.edu/~dritchie/path/index.html`.

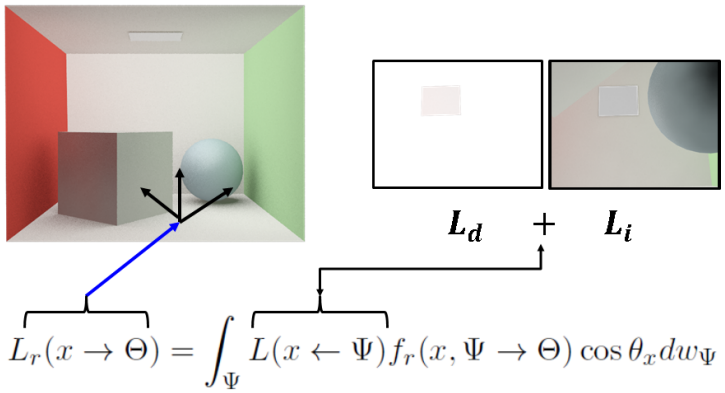


Figure 16.2: This figure illustrates the factorization of the reflected radiance into direct and indirect illumination terms.

16.1 *Direct Illumination*

Fig. 16.1 show rendering results w/ and w/o direct illumination. The first row shows rendering results w/o direct illumination under 1, 4, and 16 spp. In this scene, we adopt path tracing and observe severe noise even when we use 16 spp. This noise is mainly from the variance of the MC estimator. Note that we use random sampling on the hemisphere to generate a reflected ray direction, and it can keep bounce unless arriving at the light source located at the ceiling of the scene. Furthermore, since we are using the Russian roulette, some rays can be terminated without carrying any radiance, resulting in dark colors.

A better, yet intuitive approach is to generate a ray directly toward the light source, since we know that the light source is emitting energy and brightens the scene. The question is how we can accommodate this idea within the MC estimation framework! If we just generate a ray toward the light source, it will introduce a bias and we may not get a correct result, even when we generate an infinite number of samples.

Let's consider the rendering equation that computes the radiance $L(x \to \Theta)$, from a location x in the direction of Θ [1]. The radiance is composed of the self-emitted energy and reflected energy (Fig. 13.1):

[1] This notation is introduced in Sec. 13.1

$$L(x \to \Theta) = L_e(x \to \Theta) + L_r(x \to \Theta). \tag{16.1}$$

For the reflected term $L_r(\cdot)$, we decompose it into two terms: direct illumination term, $L_d(\cdot)$, and indirect illumination term, $L_i(\cdot)$:

$$L_r(x \to \Theta) = L_d(x \to \Theta) + L_i(x \to \Theta). \tag{16.2}$$

Fig. 16.2 illustrates an example of this decomposition.

Once we decomposed the radiance term into the direct and indirect illumination terms, we apply two separate MC estimators for

those two terms. For the direct illumination term, we cannot use the hemispherical integration described in Sec. 13.1, since we need to generate rays to the light source. For generating rays only to the light source, we use the area formulation, Eq. 13.5 explained in Sec. 13.2.

Rays corresponding to the direct illumination should be not duplicated considered for indirect illumination.

For estimating the indirect illumination, we use the hemispherical integration. The main difference to the regular hemispherical integration is that a ray generated from the hemispherical integration should not accumulate energy directly from the light source. In other words, when the ray intersects with the light source, we do not transfer the energy emitted from the light source, since the ray in this case is considered in the direct illumination term, and thus its energy should not be considered for the indirect illumination to avoid duplicate computation.

Many light problems. We discussed a simple importance sampling with the direct illumination sampling to reduce the variance of MC estimators. What if we have so many lights? In this case, generating rays to many lights can require a huge amount of time. In practice, simulating realistic scenes with complex light setting may require tens or hundreds of thousands of point light sources. This problem has been known as the many light problem. Some of simple approaches are to generate rays to those lights with probabilities that are proportional to their light intensity.

16.2 Multiple Importance Sampling

In the last section, we looked into direct illumination sampling as an importance sampling method. While it is useful, it cannot be a perfect solution, as hinted in our theoretical discussion (Sec. 14.3)

There are many other different terms in the rendering equation. Some of them are incoming radiance, BRDF, visibility, cosine terms, etc. The direct illumination sampling is a simple heuristic to consider the incoming radiance, while there could be many other strong indirect illuminations such as strong light reflection from a mirror. BRDF of an intersected object and cosine terms are available, and thus we can design importance sampling methods considering those factors. Nonetheless, these different importance sampling methods are designed separately and may work well in one case, but not in other cases.

Multiple importance sampling (MIS) is introduced to design a combined sampling method out of separately designed estimators. Suppose that there are n different sampling methods, and we allocate n_i samples for each sampling method. Given the total number of samples N, $n_i = c_i N$ with independent $X_{i,j}$ samples. The whole

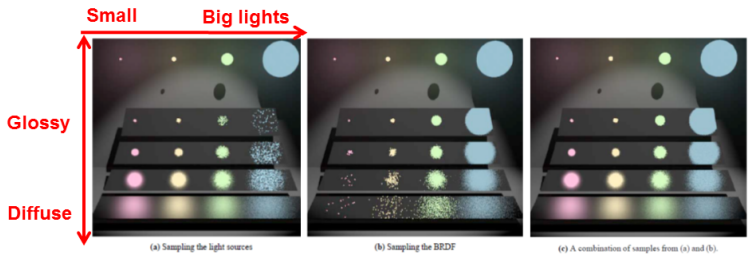


Figure 16.3: These figures show rendering results with different sampling methods. From the left, we use sampling light sources, BRDF, and both of them w/ multiple importance sampling.

distribution, $\bar{p}(x)$, combined with those n different methods, is defined as the following:

$$\bar{p}(x) = \sum_{i}^{n} c_i p_i(x), \tag{16.3}$$

where $p_i(x)$ is a i-th sampling distribution. $\bar{p}(x)$ is also called combined sample distribution [2], whose each sample $X_{i,j}$ has $1/N$ sampling probability.

By applying the standard MC estimator with the combined sampling distribution, we get the following estimator:

$$I = \frac{1}{N} \sum_{i} \sum_{n_i} \frac{f(X_{i,j})}{\bar{p}(X_{i,j})}. \tag{16.4}$$

This estimator is also derived by assigning the relative importance, i.e., probability, of a sampling method among others. In this perspective, this is also known as to be derived under balance heuristic. Surprisingly, this simple approach has been demonstrated to work quite well as shown in Fig. 16.3; these figures are excepted from the paper of Veach et al. [3]. A theoretical upper bound of the variance error of this approach is available in the original paper.

[2] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *SIGGRAPH*, pages 419–428, 1995

[3] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *SIGGRAPH*, pages 419–428, 1995

17
Conclusion

In this book, our discussions have revolved around two main topics: rasterization and ray tracing. These two techniques have their own pros and cons. For example, ray tracing is slower compared to rasterization, and is more natural to support a wide variety of rendering effects. We have mainly explains basic concepts on these topics, and there are many other advanced topics including scalable techniques and sub-surface scattering approaches. We plan to cover them in a coming edition.

Bibliography

Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., 2008.

Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

Philip Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. AK Peters, 2006.

Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 212–22, July 1984.

Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 145–154, August 1990.

Jae-Pil Heo, Joon-Kyung Seong, DukSu Kim, Miguel A. Otaduy, Jeong-Mo Hong, Min Tang, and Sung-Eui Yoon. FASTCD: Fracturing-aware stable collision detection. In *SCA '10: Proceedings of the 2010 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2010.

Thomas Kollig and Alexander Keller. Efficient multidimensional sampling. *Comput. Graph. Forum*, 21(3):557–563, 2002.

C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs. In *IEEE Symp. on Interactive Ray Tracing*, pages 39–46, 2006.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum (EG)*, 28(2):375–384, 2009.

Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 1997.

H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29:66:1–66:13, 2010.

Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation 2nd*. Morgan Kaufmann Publishers Inc., 2010a.

Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010b. ISBN 0123750792, 9780123750792.

William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 2nd edition, 1993.

G. Sellers and J.M. Kessenich. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison Wesley, 2016.

Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., 3rd edition, 2009.

Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. AK Peters, second edition, 2003.

Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, 1974.

Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *SIGGRAPH*, pages 419–428, 1995.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 2014.

E. Weisstein. From mathworld–a wolfram web resource. URL `http://mathworld.wolfram.com`.

Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. Quick-VDR: Interactive View-dependent Rendering of Massive Models. In *IEEE Visualization*, pages 131–138, 2004.

Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-Oblivious Mesh Layouts. *ACM Transactions on Graphics (SIGGRAPH)*, 24(3):886–893, 2005.

Sungeui Yoon, Sean Curtis, and Dinesh Manocha. Ray tracing dynamic scenes using selective restructuring. *Eurographics Symp. on Rendering*, pages 73–84, 2007.

Index