SUNG-EUI YOON, KAIST

RENDERING

FREELY AVAILABLE ON THE INTERNET

First printing, July 2018

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:

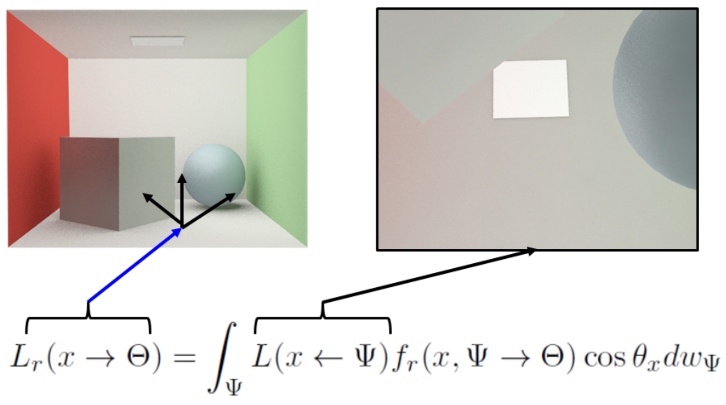

$L_r(x \to \Theta) = \int_\Psi L(x \leftarrow \Psi) f_r(x, \Psi \to \Theta) \cos\theta_x dw_\Psi$

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:

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

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

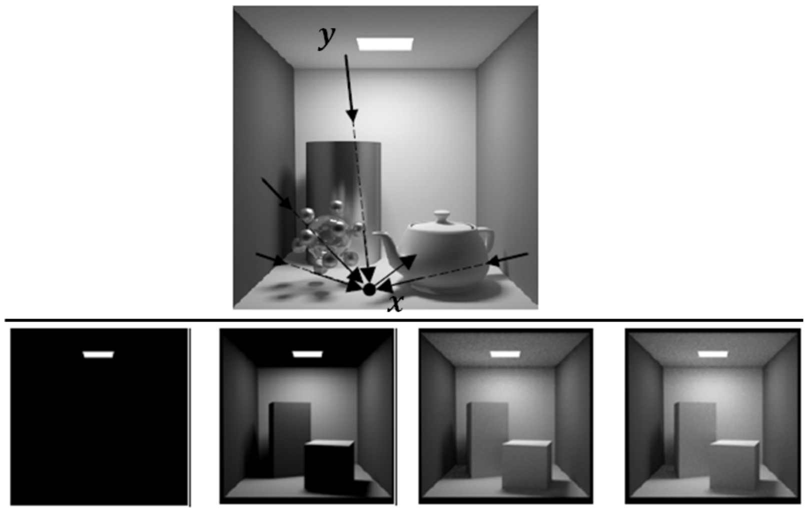


Figure 15.2: Top: computing the outgoing radiance from x requires us to compute the radiance from y to x, which is also recursively computed by simulating additional bounce to y. Bottom: this sequence visualizes rendering results by considering the direct emission and single, double, and triple bounces, 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.

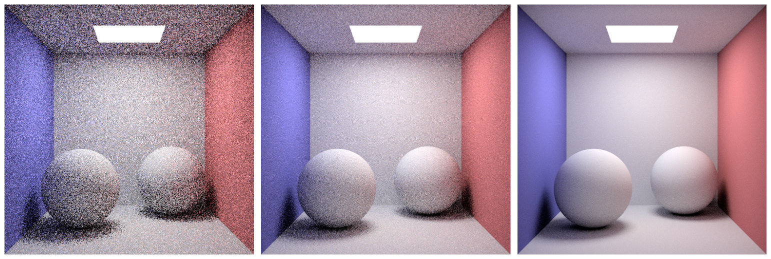


Figure 15.3: This figure shows images that are generated with varying numbers of samples per each pixel. Note that direct illumination sampling, generate a ray toward the light (Sec. 16.1), is also used. From the left, 1 spp (sample per pixel), 4 spp, and 16 spp are used.

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

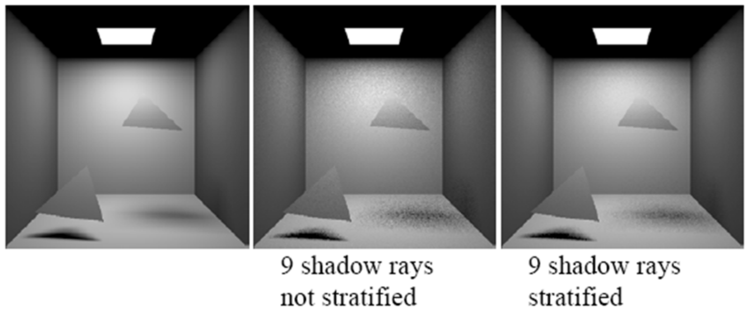


Figure 15.4: The reference image is shown on the leftmost, while images with and without stratified sampling are shown on the right. Images are excerpted from slides of Prof. Bala.

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.

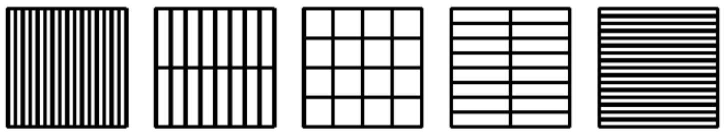
$$\begin{aligned} Var[X] &= E[X^2] - E[X]^2 \\ &= E[E[X^2|Y]] - E[E[X|Y]]^2, \because \text{Law of total expectation} \\ &= E[Var[X|Y]] + E[E[X|Y]^2] - E[E[X|Y]]^2, \\ &= E[Var[X|Y]] + Var(E[X|Y]). \end{aligned} \tag{15.5}$$

□

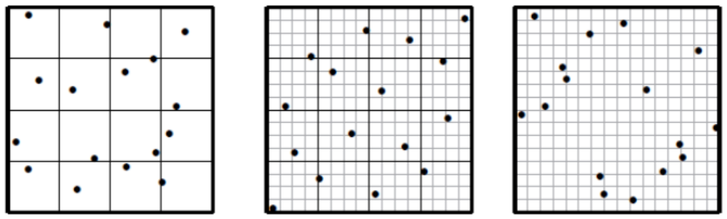
According to the law of total variance, we can show that the variance of the original random variance is equal to or less than the variance of the random variance in each sub-region.

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

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].

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

Specifically, the probabilistic error bound of the MC method

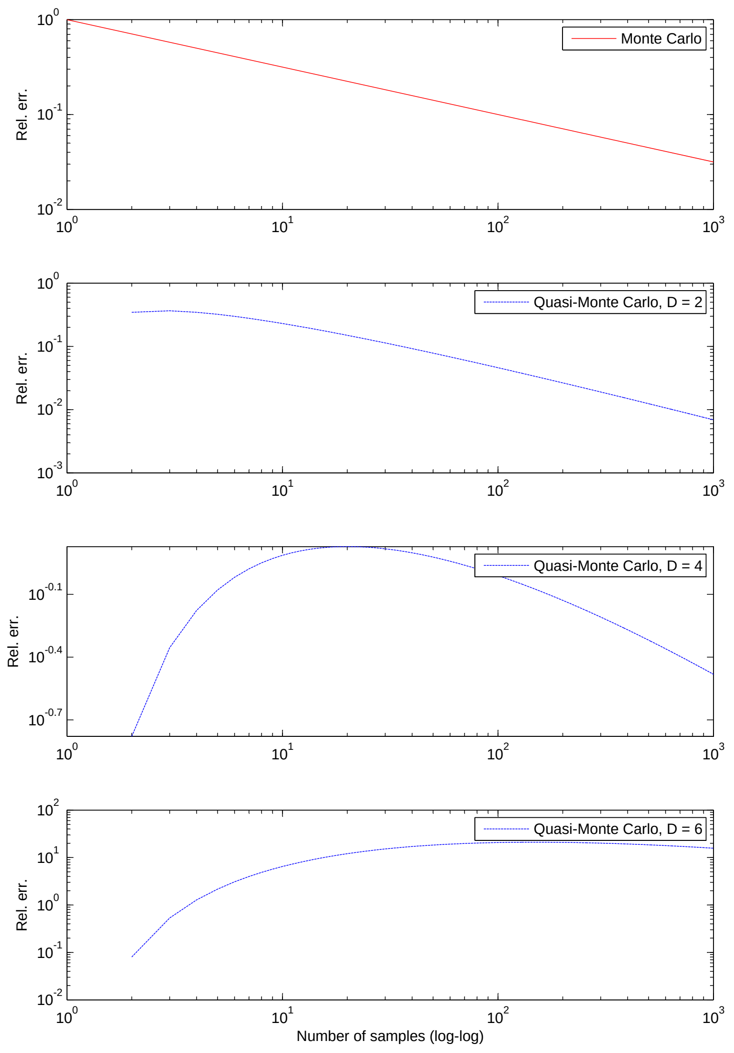


Figure 15.6: This figure shows error behavior of MC and quasi-Monte Carlo methods. They are not aligned in the same error magnitude. As a result, only shapes of these curves are meaningful. The basic quasi-Motel Carlo shows better performance than MC on low dimensional spaces (e.g, two).

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

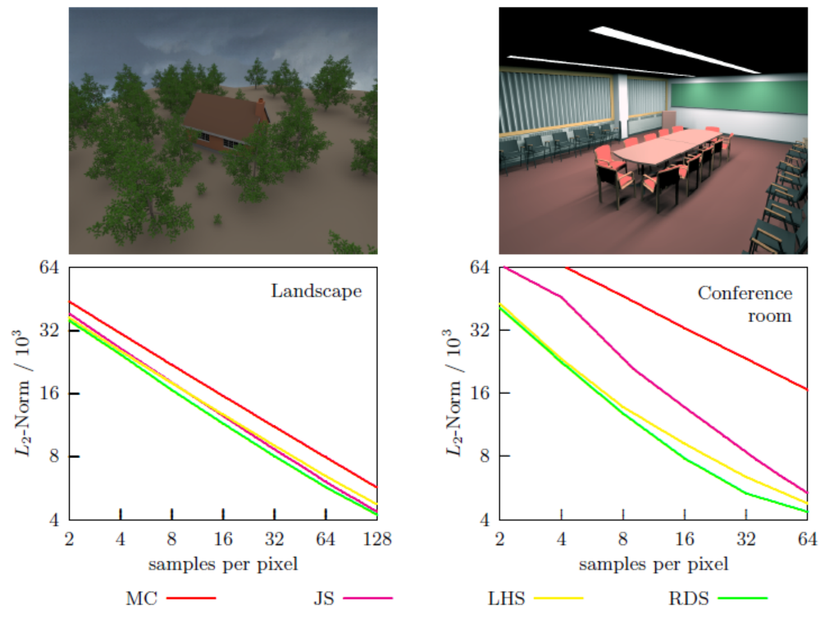


Figure 15.7: These graphs show different error reduction rates of Monte Carlo (MC), jittered (JS), Latin hypercube (LHS), and randomized Sobol sequence (RDS). These techniques are applied to four dimensional rendering problems with direct illumination.