Efficient sampling of products of functions using wavelets
M.S. Thesis

Petrik Clarberg
Department of Computer Science
Lund University

Advisor:

Tomas Akenine-Möller
Department of Computer Science
Lund University

March 2005

Abstract

This thesis presents a novel method for importance sampling the product of two-dimensional functions. The functions are represented as wavelets, which enables rapid computation of the product as well as efficient compression. The sampling is done by computing wavelet coefficients for the product on-the-fly, and hierarchically sampling the wavelet tree. The wavelet multiplication is guided by the sampling distribution. Hence, the proposed method is very efficient as it avoids a full computation of the product. The generated sampling distribution is superior to previous methods, as it exactly matches the probability density of the product of the two functions. As an application, the method is applied to the problem of rendering a virtual scene with realistic measured materials under complex direct illumination provided by a high-dynamic range environment map.

Contents

1 Introduction

Many applications in science involve complicated integrals that need to be evaluated efficiently. Often, the integrand is a multi-dimensional function and no analytical solution exists. To evaluate such integrals, a common approach is to use *Monte Carlo* integration, which relies on random sampling. Essentially, the value of the integral is approximated as the average of the function values sampled at a number of different locations, chosen at random in the integration domain. Monte Carlo sampling is a powerful tool, as it gives an unbiased estimate of the true value, together with a statistical error bound for the estimate. However, the convergence rate is slow, making the method computationally expensive.

The performance can be improved if we incorporate knowledge about the function being integrated into the sampling process. The idea is to concentrate samples to parts of the function where it is likely to be large. This technique is called *importance sampling*, and can vastly reduce the variance in Monte Carlo techniques.

One way of importance sampling a high-dimensional function is to express the function as *Haar wavelets*, and use a probabilistic sampling algorithm. Haar wavelets are basically a set of basis functions, constructed so that they make up an orthonormal basis with certain useful properties. Wavelet theory is a relatively new tool in mathematics, which enables both efficient (lossy) compression and hierarchical reconstruction of the function. By exploiting the hierarchical properties of wavelets, samples can be distributed very efficiently according to the wavelet representation.

The first contribution of this work is a new technique for hierarchically sampling a wavelet tree. The algorithm starts at the coarsest resolution and recursively moves down to finer resolutions. At each step, the number of samples in different areas of the function domain is deterministically chosen proportional to the average value of the function in each area. This way, we can efficiently distribute any number of samples with a single traversal of the wavelet tree. The end result is a sampling distribution that accurately matches the function, and gives a lower variance than pure random sampling of the wavelet representation.

It was recently demonstrated that two-dimensional Haar wavelet representations can be directly multiplied and the compressed result obtained, without first decompressing the functions. Due to the narrow support of the basis functions and the sparsity of the representation, it is possible to compute the product of two wavelet representations very rapidly. Using this new theory, we apply wavelet importance sampling to *products* of functions, something that has never been done before.

The proposed method is a new general tool for evaluating the integral of a product with several two-dimensional terms. To show its strengths in a real application, we use wavelet importance sampling of products to render a virtual scene with realistic measured materials under complex direct illumination from a high-dynamic range environment map [8].

1.1 Previous Work

In this section, we give a brief overview of the previous work on Monte Carlo rendering and importance sampling. We focus on techniques used in computer graphics, although the proposed importance sampling scheme is useful in other applications as well.

The use of Monte Carlo techniques for rendering photo-realistic images started with the seminal work by Cook et al. [6] and Kajiya [14]. Today, a variety of methods exists — see the book by Dutré et al. [9] for an overview. Common to all methods is that they try to evaluate the rendering equation [14] as efficiently as possible. The rendering equation involves an integral over the product of lighting and surface properties:

$$L(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_\Omega f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_i, \tag{1}$$

where $f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o)$ is the *bidirectional reflectance distribution function* (BRDF) of the surface, $L_i(\mathbf{x}, \vec{\omega}_i)$ describes the incident radiance, and $\cos\theta_i$ is a simple cosine-term. Most previous work uses importance sampling of either the BRDF or the incident lighting, while we address the problem of sampling the product of the two.

A wide range of methods for importance sampling the BRDF exists. Some of the common analytical BRDF models such as the Phong model, the Ward model, and the Lafortune model, can be directly importance sampled [35, 39, 18]. See Pharr and Humphreys [32] for more examples. Other BRDF models, such as the Torrance-Sparrow model, cannot be analytically inverted, and hence require numerical approximations.

It is becoming increasingly important to be able to use measured BRDF data instead of analytical models. These data sets are usually very large, and efficient methods for representing the BRDFs are needed. Lalonde [19] used a 4D wavelet representation of the BRDF, and presented an importance sampling scheme based on random sampling of the wavelet tree. Similar methods were used by Claustres et al. [4, 5] and by Matusik [26]. Another approach is to factorize the BRDF into lower-dimensional terms. For example, Lawrence et al. [21] presented a technique based on non-negative matrix factorization (NMF) [22]. They represent the BRDF in Rusinkiewicz's parameterization [34], which is compact and compresses well, and decompose the BRDF into a set of one and two-dimensional terms. These lower-dimensional terms can then be efficiently importance sampled.

Although BRDF importance sampling significantly improves the result, the technique is best suited for relatively specular BRDFs. For more diffuse BRDFs, a very large number of samples are needed to capture a complex lighting environment.

In the case where a scene is illuminated by a high-dynamic range environment map [8], we can evaluate the rendering equation by importance sampling the environment map instead of the BRDF. Such techniques have been presented by, among others, Agarwal et al. [1], Kollig and Keller [17], and Ostromoukhov et al. [31]. These methods resample the environment map by placing directional lights at the brightest locations, which can be directly used for rendering the scene. The intensity of each light source is the pre-integrated value of the environment map over the area the light represents. This is an efficient method for rendering non-specular materials under environment map lighting as it significantly reduces the complexity of the environment map, at the expense of precomputing a set of pre-integrated lights. However, with increasingly specular BRDFs, these methods become inefficient as a very large number of lights are needed to truthfully represent the environment map.

In summary, BRDF importance sampling is better suited for specular materials, while environment map importance sampling is better for diffuser BRDFs. To address this problem, Veach and Guibas [38] presented a novel technique for combining estimators in Monte Carlo methods using multiple importance sampling. Multiple importance sampling is a powerful method for the case where either the lighting or the BRDF is complex, as it will pick the best of the available sampling techniques. However, when both the lighting and the BRDF are complicated, their technique provides a smaller advantage.

Recently, Burke et al. [2] introduced a technique for rendering objects with complex materials illuminated by an environment map. Their technique uses importance sampling of either the environment map or the BRDF, and then applies rejection sampling to discard samples for which the product of the BRDF and the lighting is not large enough to motivate sampling. This helps reduce the number of samples, but at the expense of densely evaluating the functions in order to generate samples that are possibly rejected. If both the BRDF and the lighting are complex, Burke et al. report that more than 90% of the samples are rejected.

1.2 Outline

In contrast to the previous work, our method is capable of efficiently importance sampling the product of the lighting *and* the BRDF. In the following sections, we will describe how this is done by using a compact wavelet representation together with fast wavelet multiplication. As wavelets play an essential role in the proposed importance sampling scheme, a good understanding of the basic concepts of Haar wavelets is needed to fully appreciate this thesis. In Section 2, the Haar basis is reviewed and the theory for wavelet products is explained.

Section 3 first gives a short overview of Monte Carlo integration and importance sampling in mathematical terms. Then, our method for distributing samples is presented. As a practical example, we use wavelet importance sampling for rendering a scene under complex direct illumination. This application is described in Section 4. Implementation details are given in Section 5. Here, focus is given to the wavelet-specific parts of the system, and the implementation of the sampling scheme. Important parts of the algorithm are given as pseudo-code. Finally, results and conclusions are presented and discussed in Section 6 and 7 respectively.

2 Wavelets

Wavelet theory has its roots in Fourier analysis dating back to 1805. The Fourier transform decomposes a signal into a sum of sinusoids of different frequency, making it possible to determine all frequencies present in a signal. It does not, however, give any information about when they occur. Wavelets address this problem, providing means to analyze a function in both time and frequency.

Although the term "wavelets" is relatively new (early 1980s), the concepts have been around for a long time and were developed independently in the fields of quantum physics, electrical engineering, and seismic geology. In the 1980s, Mallat and Meyer unified the previous work and developed the theory of *multiresolution analysis* [23]. In multiresolution analysis, wavelets are used to analyze a signal at many different resolutions. During the last two decades, many new applications of wavelets have been found. In the field of computer science, wavelets are used in, for example, data compression, image analysis and computer graphics.

Wavelets are a family of hierarchical and usually orthogonal basis functions with compact support. Fast wavelet transforms exist, making wavelets practically useful. The simplest wavelet basis, the *Haar basis*, was first described already in 1910. The following sections give a brief introduction to Haar wavelets in one and two dimensions. For further reading, a good textbook on wavelet theory is recommended [24, 36].

2.1 The Haar Basis

Consider all piecewise constant functions defined on the interval $[0, 1)$ with 2^l subintervals of equal length. The functions can be said to be vectors in the *vector space* $\mathcal{V}^l$. That is, the space $\mathcal{V}^l$ includes all piecewise constant functions defined on the unit interval with 2^l equal subintervals. For example, $\mathcal{V}^0$ is the space of all constant functions on the unit interval, $\mathcal{V}^1$ is the space of all functions with two constant pieces over the intervals $[0, \frac{1}{2})$ and $[\frac{1}{2}, 1)$, and so on. Note that every such function with 2^l intervals can also be described with 2^{l+1} intervals. Thus, the spaces $\mathcal{V}^l$ are nested:

$$\mathcal{V}^0 \subset \mathcal{V}^1 \subset \mathcal{V}^2 \subset \cdots . \tag{2}$$

In the Haar basis, a simple set of basis functions are defined for each space $\mathcal{V}^l$. The basis functions are called *scaling functions* and consist of dilations and translations of a step function. In the normalized form, the scaling functions are given by:

$$\phi_{l,t}(x) := 2^{l/2}\phi(2^l x - t), \qquad t = 0, \ldots, 2^l - 1, \tag{3}$$

where

$$\phi(x) := \begin{cases} 1, & \text{for } 0 \leq x < 1 \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

A one-dimensional image, F, with 2^l pixels can be represented in the basis defined by the scaling functions as follows:

$$F = \sum_t F^0_{l,t}\phi_{l,t}, \tag{5}$$

where the $F^0_{l,t}$ are called *scaling coefficients*.

The scaling functions have a set of corresponding basis functions called *wavelet functions*, $\psi_{l,t}$, that are orthogonal to the scaling functions. This means that the inner product of each scaling function with each wavelet function at the same level is zero. The wavelet functions are given by:

$$\psi_{l,t}(x) := 2^{l/2}\psi(2^l x - t), \qquad t = 0, \ldots, 2^l - 1, \tag{6}$$

where

$$\psi(x) := \begin{cases} 1, & \text{for } 0 \le x < 1/2 \\ -1, & \text{for } 1/2 \le x < 1 \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

Note that the wavelet functions $\psi_{l,t}$, together with the scaling functions $\phi_{l,t}$ for a space $\mathcal{V}^l$, form a basis for $\mathcal{V}^{l+1}$. By induction, this implies that our image F, can be expressed using only the first scaling function plus a number of wavelet functions:

$$F = F^0_{0,0}\phi_{0,0} + \sum_{k=0}^{l-1}\sum_{t} F^1_{k,t}\psi_{k,t}, \tag{8}$$

where the $F^1_{l,t}$ are called *detail coefficients*.

2.2 The 2D Haar Basis

The Haar basis can be generalized into two dimensions in two different ways. In the *standard decomposition*, the one-dimensional wavelet transform is applied independently on the two dimensions. The other method, the *non-standard decomposition*, alternates between operations on the two dimensions. We are only considering the non-standard decomposition as it is more suitable for our application, and also slightly more efficient to compute.

In the nonstandard decomposition, the normalized basis functions are given as dilations and translations of a two-dimensional mother scaling function and three two-dimensional mother wavelet functions (illustrated in Figure 1):

$$\phi_{l,\mathbf{t}}(x,y) = 2^l\phi(2^l x - t_1, 2^l y - t_2), \tag{9}$$

$$\psi^m_{l,\mathbf{t}}(x,y) = 2^l\psi^m(2^l x - t_1, 2^l y - t_2), \quad m = 1,2,3, \tag{10}$$

where we have used vector notation $\mathbf{t} = (t_1, t_2)$ for the translation in two dimensions. The mother scaling function is given by:

$$\phi(x,y) = \phi(x)\phi(y), \tag{11}$$

and the two-dimensional mother wavelet functions are defined as:

$$\psi^m(x,y) = \begin{cases} \psi(x)\phi(y), & \text{if } m = 1, \\ \phi(x)\psi(y), & \text{if } m = 2, \\ \psi(x)\psi(y), & \text{if } m = 3. \end{cases} \tag{12}$$

As in the one-dimensional case, a two-dimensional image can be expressed as a sum of the first scaling function plus the wavelet functions. Here, F is a two-dimensional image with $2^l \times 2^l$ pixels:

$$F = F^0_{0,0}\phi_{0,0} + \sum_{k=0}^{l-1}\sum_{\mathbf{t}}\sum_{m} F^m_{k,\mathbf{t}}\psi^m_{k,\mathbf{t}} = \sum_i F_i\Psi_i, \tag{13}$$

Figure 1: The mother scaling function and the three mother wavelet functions. The functions are $+1$ where white and -1 where black, and implicitly zero outside the unit square.

In the last step, we have introduced a shorthand notation that combines the first scaling function and all the wavelet functions into a single set of basis functions Ψ_i, with corresponding basis coefficients F_i. The subscript i is a sequential index, with $i = 0$ for the scaling function and $i > 0$ for the wavelet functions. In the next section, this notation will be convenient for discussing wavelet products.

2.3 Lossy Compression

The wavelet transform effectively analyzes a function in time and space, concentrating the information to a small number of wavelet coefficients. The rest of the coefficients will be close to zero. Hence, efficient compression is possible by setting small coefficients to zero. This compression method is lossy, but in practice a compression factor of 10–100 is realistic with very little loss of precision.

It is a well-known fact that, for Haar wavelets, the approximation error introduced by discarding some coefficients is the sum of the squares of the discarded coefficients [36]. If F is the original image, and F_A is an image reconstructed from a subset A of the wavelet coefficients, the L^2-error is given by:

$$||F - F_A|| = \left(\sum_{i \notin A} |\psi_i|^2 \right)^{1/2}, \tag{14}$$

where ψ_i are the detail coefficients. It is obvious that the optimal method for minimizing the error, is to remove the smallest coefficients in absolute value. In practice, we perform the compression by *thresholding* all coefficients against a threshold α, removing all coefficients with an absolute value below that threshold:

$$\widetilde{\psi}_i := \begin{cases} 0, & \text{if } |\psi_i| < \alpha \\ \psi_i, & \text{otherwise,} \end{cases} \tag{15}$$

where $\widetilde{\psi}_i$ are the detail coefficients for the compressed function. The benefit of thresholding, as compared to removing a fixed number of coefficients, is that we avoid the time-consuming process of first sorting all the coefficients.

2.4 2D Wavelet Product

Given two functions expressed in an orthonormal basis, it is possible to multiply them together and get the product expanded in the same basis. Ng et al. [28] showed how this can be done for the case where the functions are two-dimensional images represented in the Haar basis. Let $G = \sum G_j \Psi_j$ and $H = \sum H_k \Psi_k$ be the two images. The *wavelet product*, $F = \sum F_i \Psi_i$, of G and H is then given by:

$$F = G \cdot H \Leftrightarrow \sum F_i \Psi_i = \left(\sum G_j \Psi_j \right) \cdot \left(\sum H_k \Psi_k \right). \tag{16}$$

By integrating against the i^{th} basis function, we can directly obtain the i^{th} coefficient for the wavelet representation of the product F as follows:

$$\begin{aligned} F_i &= \iint \Psi_i(\mathbf{x})F(\mathbf{x})d\mathbf{x} = \iint \Psi_i(\mathbf{x})G(\mathbf{x})H(\mathbf{x})d\mathbf{x} \\ &= \iint \Psi_i(\mathbf{x}) \left(\sum_j G_j\Psi_j(\mathbf{x})\right)\left(\sum_k H_k\Psi_k(\mathbf{x})\right) d\mathbf{x} \\ &= \sum_j\sum_k G_jH_k \iint \Psi_i(\mathbf{x})\Psi_j(\mathbf{x})\Psi_k(\mathbf{x})d\mathbf{x} \\ &= \sum_j\sum_k C_{ijk}G_jH_k. \end{aligned} \tag{17}$$

The terms C_{ijk} are called *tripling coefficients*, and are given by:

$$C_{ijk} = \iint \Psi_i(\mathbf{x})\Psi_j(\mathbf{x})\Psi_k(\mathbf{x})d\mathbf{x}. \tag{18}$$

Note that these equations are valid for any domain and suitable orthonormal basis, only the tripling coefficients will differ. Due to the compact support of the Haar basis functions, most of the tripling coefficients will be zero. The non-zero coefficients are given by the *Haar tripling coefficient theorem* [28]. The integral of three 2D Haar basis functions is non-zero if and only if one of the following three cases holds:

1. All three are the scaling function. In this case, $C_{ijk} = 1$.
2. All three functions occupy the same wavelet square, and all are of different wavelet types. $C_{ijk} = 2^l$, where the square is at level l.
3. Two are identical wavelets, and the third is either the scaling function or a wavelet that overlaps at a strictly coarser level. $C_{ijk} = \pm 2^l$, where the third function exists at level l.

The tripling coefficient theorem is written in general terms, and describes the cases where the tripling coefficients are non-zero. In this application, where we are looking at a specific basis function, Ψ_i, the theorem can be rewritten to make the different cases more clear:

1. Ψ_i is the mother scaling function:
   (a) Ψ_j and Ψ_k are also the mother scaling function. $C_{ijk} = 1$.
   (b) Ψ_j and Ψ_k are identical wavelets (at any level). $C_{ijk} = 1$.
2. Ψ_i is a wavelet function at level l:
   (a) All three functions occupy the same wavelet square and all are of different wavelet types. $C_{ijk} = 2^l$.
   (b) Ψ_j and Ψ_k are identical wavelets under the support of Ψ_i and exist at a strictly finer level. $C_{ijk} = \pm 2^l$.
   (c) One of the wavelets is identical to Ψ_i, and the other is either the mother scaling function or a wavelet that overlaps at a strictly coarser level. $C_{ijk} = \pm 2^{l'}$, where the coarser function exists at level l'.

2.5 Optimized Product

As we will show later, in wavelet importance sampling it is unnecessary to compute detail coefficients for the product, as only the scaling coefficients at each level are needed for sampling. Therefore, we have developed a simplified wavelet product that directly gives the scaling coefficients for the product. In equation 17, we replace Ψ_i with the specific scaling function $\phi_{l,\mathbf{t}}$, for which we want to compute the scaling coefficient. The scaling coefficient for the product is then given by:

$$\begin{aligned} F^0_{l,\mathbf{t}} &= \iint \phi_{l,\mathbf{t}}(\mathbf{x})F(\mathbf{x})d\mathbf{x} = \iint \phi_{l,\mathbf{t}}(\mathbf{x})G(\mathbf{x})H(\mathbf{x})d\mathbf{x} \\ &= \iint \phi_{l,\mathbf{t}}(\mathbf{x}) \left(\sum_j G_j\Psi_j(\mathbf{x})\right)\left(\sum_k H_k\Psi_k(\mathbf{x})\right)d\mathbf{x} \\ &= \sum_j\sum_k G_jH_k \iint \phi_{l,\mathbf{t}}(\mathbf{x})\Psi_j(\mathbf{x})\Psi_k(\mathbf{x})d\mathbf{x} \\ &= \sum_j\sum_k C'_{ijk}G_jH_k, \end{aligned} \tag{19}$$

where C'_{ijk} are modified tripling coefficients, defined as:

$$C'_{ijk} = \iint \phi_{l,\mathbf{t}}(\mathbf{x})\Psi_j(\mathbf{x})\Psi_k(\mathbf{x})d\mathbf{x}. \tag{20}$$

As in the general case, a theorem for the computation of these modified tripling coefficients can be developed. It turns out that the C'_{ijk} for a scaling function at level l are non-zero if and only if one of the following two cases holds (see proof in Appendix B.1):

1. Ψ_j and Ψ_k are either the mother scaling function or wavelets at strictly coarser levels, l_j and l_k. $C_{ijk} = \pm 2^{l_j+l_k-l}$.
2. Ψ_j and Ψ_k are identical wavelets under the support of $\phi_{l,\mathbf{t}}$, and exist at the same or finer levels. $C_{ijk} = 2^l$.

It is not obvious why this new theorem provides any advantage over the previous general theorem. A key observation is that the first case corresponds to a multiplication of the scaling coefficients for G and H at level l that overlap $\phi_{l,\mathbf{t}}$, scaled by 2^l, i.e., a multiplication of the scaling coefficients $G^0_{l,\mathbf{t}}$ and $H^0_{l,\mathbf{t}}$. The derivation is presented in Appendix B.2. Hence, we can compute scaling coefficients for the product as:

$$F^0_{l,\mathbf{t}} = 2^l G^0_{l,\mathbf{t}}H^0_{l,\mathbf{t}} + 2^l \sum_{l'\geq l,\mathbf{t}'\in\mathbf{t},m} G^m_{l',\mathbf{t}'}H^m_{l',\mathbf{t}'}, \tag{21}$$

where the summation is over all wavelet coefficients that are under the support of $\phi_{l,\mathbf{t}}$. To use this formula in practice, we must know the values of the scaling coefficients $G^0_{l,\mathbf{t}}$ and $H^0_{l,\mathbf{t}}$. These can easily be computed separately for the two functions, using standard wavelet reconstruction from their respective wavelet coefficients.

2.6 Implementation

In Appendix C.1, we present pseudo-code for the general two-dimensional wavelet product, as well as for the optimized product for computing only scaling coefficients. First, we introduce

a helper function called *parent sum* (PSUM). This function was used by Ng et al. [28] in their computation of wavelet triple products. The parent sum of a square $\mathbf{s} = (l, x, y)$ in a wavelet representation F, is the reconstructed function value over the square, which we call $F(\mathbf{s})$. The reconstructed value is found by taking the sum of the coefficients of basis functions overlapping the square, scaled by the value of the corresponding basis functions over the square we are considering. We implemented the parent sum as a recursive function that caches the computed values in a hash table, see Appendix C.1.1.

Second, we also define another helper function, which we call *children sum* (CSUM). The children sum of two wavelet representations, G and H, at a square $\mathbf{s}$, is the sum of the product of the coefficients of identical basis functions that overlap the square at the same or finer levels. We start by computing all the non-zero children sums and storing them in a hash table. The children sums are very sparse, since both coefficients for an identical basis function need to be non-zero for the result to be non-zero. Pseudo-code for computing the children sums is given in Appendix C.1.2.

Now, with the necessary support functions, PSUM and CSUM, it is straightforward to implement the wavelet product. We implement the wavelet multiplication as a function that takes two wavelets as input, and returns a specific wavelet coefficient for the product, see Appendix C.1.3. This function can be used in the general case for computing coefficients of the product of two wavelet representations. For importance sampling, we are only interested in computing scaling coefficients for the product. Using the theory in Section 2.5, we arrive at the optimized function described in Appendix C.1.4.

3 Sampling

In this chapter, we will describe how wavelets can be used for efficient evaluation of integrals, where we have some prior knowledge of the integrand. Specifically, we are considering the case when the integrand is a product with several terms, of which some are known functions and other unknown. We represent the known functions as Haar wavelets and use a novel method for distributing sample points according to the intensity in different areas. The samples are used for evaluating the value of the integral through *Monte Carlo* integration. The basics of Monte Carlo integration is reviewed in the following section.

3.1 Monte Carlo Integration

We want to estimate the integral of a complicated multi-dimensional function, $f(\mathbf{x})$:

$$I = \int f(\mathbf{x}) d\mathbf{x}, \tag{22}$$

for which no analytic solution exists. For one-dimensional integrals, classical numerical quadrature rules such as the Newton-Cotes formulas and the Gaussian quadratures can be used. For an overview of these methods, see [7]. In multi-dimensional problems these methods quickly become inefficient. An alternative is Monte Carlo integration [11, 15], which has an error bound that decreases by $1/\sqrt{N}$ independent of the dimension of the integral. Monte Carlo integration relies on random sampling of the integral, and an estimate for the integral can be written

$$\langle I \rangle = \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x}_n), \tag{23}$$

where $\mathbf{x}_n$ are random points in the integration domain. This estimator is unbiased, which means it converges to the true value of the integral as N goes towards infinity,

$$\lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x}_n) = I. \tag{24}$$

It can be shown that the error in the estimate is on average $\sigma(f)/\sqrt{N}$, where $\sigma(f)$ is the standard deviation of f. The variance $\sigma^2(f)$ of f is given by:

$$\sigma^2(f) = \int \left(f(\mathbf{x}) - I\right)^2 d\mathbf{x}. \tag{25}$$

In practice, the variance cannot be computed easily, but an estimate of the variance can be written:

$$\langle \sigma^2(f) \rangle = \frac{1}{N-1} \sum_{n=1}^{N} \left(f(\mathbf{x}_n) - \langle I \rangle\right)^2 \approx \frac{1}{N} \sum_{n=1}^{N} \left(f(\mathbf{x}_n)\right)^2 - \langle I \rangle^2. \tag{26}$$

For higher dimensional problems, Monte Carlo integration is often the only practically possible option. However, the convergence rate of $1/\sqrt{N}$ is still relatively slow. To decrease the error of the estimate, a number of variance reduction techniques can be applied. One such technique is *importance sampling*.

3.2 Importance Sampling

Importance sampling is a powerful technique for reducing the error in Monte Carlo integration. The idea is to pick sample points according to a distribution that resembles the function being integrated, instead of using uniformly distributed random points. Mathematically speaking, this corresponds to a change of integration variables:

$$I = \int f(\mathbf{x})d\mathbf{x} = \int \frac{f(\mathbf{x})}{p(\mathbf{x})}p(\mathbf{x})d\mathbf{x} = \int \frac{f(\mathbf{x})}{p(\mathbf{x})}dP(\mathbf{x}). \tag{27}$$

We restrict $p(\mathbf{x}) = dP(\mathbf{x})/d\mathbf{x}$ to be a probability density function, that is, a positive $p(\mathbf{x}) \geq 0$ function normalized to unity:

$$\int p(\mathbf{x})d\mathbf{x} = 1. \tag{28}$$

By drawing random numbers from the density function $p(\mathbf{x})$, we can estimate the integral as:

$$\langle I \rangle = \frac{1}{N}\sum_{n=1}^{N}\frac{f(\mathbf{x}_n)}{p(\mathbf{x}_n)}. \tag{29}$$

The average error of the estimate is now given by $\sigma(f/p)/\sqrt{N}$, where an estimator for the variance $\sigma^2(f/p)$ can be written as:

$$\langle \sigma^2(f/p) \rangle \approx \frac{1}{N}\sum_{n=1}^{N}\left(\frac{f(\mathbf{x}_n)}{p(\mathbf{x}_n)}\right)^2 - \langle I \rangle^2. \tag{30}$$

It is easy to show that the best possible solution is $p(\mathbf{x}) = |f(\mathbf{x})|/I$, which reduces the variance to zero. But, if we already knew the value of I, there would be no reason to estimate the integral. In practice, $p(\mathbf{x})$ is chosen so that it approximates $|f(\mathbf{x})|$ as closely as possible, and so that we can efficiently draw random numbers from $p(\mathbf{x})$.

3.3 Distribution of Samples

For importance sampling, we need random numbers with a distribution that follows some density function $p(\mathbf{x})$. One way of generating such numbers is to find a transformation that transforms a sequence of uniformly distributed random numbers in $[0, 1]$ into a sequence with the desired distribution. If the analytical inverse of the cumulative distribution function $P(\mathbf{x})$ is known, the inverse transform can be directly applied. Given a sequence of uniform random numbers $\mathbf{u}$, the transformed sequence is then given by:

$$\mathbf{x} = P^{-1}(\mathbf{u}). \tag{31}$$

However, the inverse $P^{-1}(\mathbf{u})$ is often not known, so we have to resort to other methods. In, for example, the acceptance-rejection method, random numbers are generated according to some other density function $h(\mathbf{x})$ and then compared to the target function $p(\mathbf{x})$. Points that are inside $p(\mathbf{x})$ are accepted and the rest rejected. The function $h(\mathbf{x})$ is often chosen to be the uniform density function, or some other function whose cumulative distribution function can be easily inverted.

As a side note, it is worth mentioning that the error in Monte Carlo integration can be further reduced by using quasi-random numbers instead of ordinary random numbers. Quasi-random

numbers are deterministic and designed to sample the d-dimensional space as uniformly as possible. As long as the integrand is smooth, the error bound will decrease faster than $\sigma/\sqrt{N}$. For an overview of quasi-random sequences, see the book by Niederreiter [30].

3.4 Wavelet Importance Sampling

In this section, we will describe how wavelets can be used for generating samples distributed according to an arbitrary function. The function is assumed to be two-dimensional, although the same techniques can be used in the general case with only minor modifications.

3.4.1 Preliminaries

Our goal is to distribute samples according to the probability density function $p(\mathbf{x}) = |f(\mathbf{x})|/I$, where I is the integral over $f(\mathbf{x})$. In this work, we consider the case where $f(\mathbf{x})$ is a positive two-dimensional function.

In wavelet importance sampling, we approximate the function $f(\mathbf{x})$ with an image $F(\mathbf{x})$, expressed in the Haar wavelet basis. For simplicity, the image is defined to cover the unit square. Consider a wavelet square $\mathbf{s} = (l, \mathbf{t})$ at level l and translation $\mathbf{t}$. The square has an area of $A(\mathbf{s}) = 2^{-l} \times 2^{-l} = 2^{-2l}$. The average function value over the square $\mathbf{s}$, can be found by integrating the function over $\mathbf{s}$. However, due to the constant and disjoint scaling functions, the average function value is given by the scaling coefficient for the square as follows:

$$F(\mathbf{s}) = \iint_{\mathbf{s}} F(\mathbf{x}) d\mathbf{x} = 2^l \iint \phi^0_{l,\mathbf{t}}(\mathbf{x}) F(\mathbf{x}) d\mathbf{x} = 2^l F^0_{l,\mathbf{t}}, \tag{32}$$

where the 2^l factor is derived from the area of the square and the value of the scaling function over the same. Hence, at the top level, the first scaling coefficient holds the average value over the whole image:

$$I = \iint F(\mathbf{x}) d\mathbf{x} = F^0_{0,0}. \tag{33}$$

Thus, the probability density of the square $\mathbf{s}$, is given by:

$$p(\mathbf{s}) = \frac{F(\mathbf{s})}{I} = 2^l \frac{F^0_{l,\mathbf{t}}}{F^0_{0,0}}, \tag{34}$$

which leads to the conclusion that the probability of placing a sample at a coordinate $\mathbf{x}$ within the square $\mathbf{s}$, should be equal to:

$$P(\mathbf{x} \in \mathbf{s}) = p(\mathbf{s}) A(\mathbf{s}) = 2^{-2l} \frac{F(\mathbf{s})}{I} = 2^{-l} \frac{F^0_{l,\mathbf{t}}}{F^0_{0,0}}. \tag{35}$$

For recursive algorithms, it is useful to know the conditional probabilities for each child square, given that the parent square is sampled. Let $\mathbf{s}$ be the parent square at level l, and let $\mathbf{s}_i,\ i = 1 \ldots 4$, be the four child squares at level $l+1$. The conditional probability for each of the four children can be expressed in the function values for the parent and child squares as:

$$P(\mathbf{x} \in \mathbf{s}_i | \mathbf{x} \in \mathbf{s}) = \frac{P(\mathbf{x} \in \mathbf{s}_i)}{P(\mathbf{x} \in \mathbf{s})} = \frac{2^{-2(l+1)} F(\mathbf{s}_i)/I}{2^{-2l} F(\mathbf{s})/I} = \frac{1}{4} \frac{F(\mathbf{s}_i)}{F(\mathbf{s})}, \tag{36}$$

and similarly expressed in scaling coefficients as:

$$P(\mathbf{x} \in \mathbf{s}_i | \mathbf{x} \in \mathbf{s}) = \frac{P(\mathbf{x} \in \mathbf{s}_i)}{P(\mathbf{x} \in \mathbf{s})} = \frac{2^{-(l+1)} F^0_{l+1,\mathbf{t}_i} / F^0_{0,0}}{2^{-l} F^0_{l,\mathbf{t}} / F^0_{0,0}} = \frac{1}{2} \frac{F^0_{l+1,\mathbf{t}_i}}{F^0_{l,\mathbf{t}}}. \tag{37}$$

Now, with the probabilities defined, there are different ways of sampling the wavelet tree. First, a method for generating a single sample through random sampling is presented. Then, our methods for generating multiple samples are described.

3.4.2 Random Sampling – Single Sample

A naïve solution for sampling the wavelet representation, would be to build a histogram of the probabilities for all wavelet squares at the finest resolution, and choose one of them based on a random number. However, this method requires a full reconstruction of the wavelet image, and does not exploit the hierarchical properties of the wavelet basis.

Instead, the sampling can be done by treating the probabilities as a decision tree. Starting with a uniform random number at the top level, a recursive search is performed. At each level, the random number decides which of the child quadrants to sample, based on the probability associated with each square. Practically, this can be done by building a simple histogram of the four child probabilities at each level. During sampling, the function is reconstructed on-the-fly from the wavelet representation, and the function values are used for computing the conditional probabilities using Equation 36. At the finest resolution, a sample point is placed at random within that pixel, and the recursion terminates. This algorithm is illustrated in Figure 2, and given as pseudo-code in Appendix C.2.1.

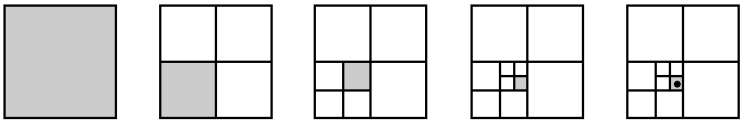


Figure 2: Random sampling of the wavelet tree through a recursive search based on the probabilities at each level. In the final square, a single sample point is placed at random.

This method of random sampling a wavelet representation was introduced by Lalonde [19], and was also used by Claustres et al. [4, 5] and Matusik [25]. In some applications such as path-tracing, only one sample at the time is needed. However, in our application, where we want to generate many samples from the same distribution, this method of random sampling becomes inefficient. To generate N samples, the above recursive search has to be repeated N times.

3.4.3 Random Sampling – Multiple Samples

The above described algorithm for generating a single sample according to the wavelet representation, can be easily extended to generate many samples in a single recursive tree traversal. Let N_{total} be the total number of samples to generate. The modified algorithm proceeds in the same manner. Starting at the top level, the probabilities are treated as a decision tree, but instead of just picking a single child quadrant to sample, we place N_{total} samples according to the conditional probabilities of the four children. In practice, we do this by building

a histogram over the four child probabilities as before. Let $N(\mathbf{s}_i)$ be the number of samples placed in each of the child squares $\mathbf{s}_i$, $i = 1 \ldots 4$.

The process is repeated recursively for the child quadrants that have at least one sample allocated, i.e., where $N(\mathbf{s}_i) > 0$. For each of those squares, $N(\mathbf{s}_i)$ samples are distributed over its children, and so on. The recursion terminates at the finest resolution, and sample points are placed at random within the sampled pixels as before. Pseudo-code for our implementation of this algorithm is given in Appendix C.2.2.

3.4.4 Deterministic Sampling – Multiple Samples

The main disadvantage of the random sampling algorithms is that there is no guarantee that the generated sampling distributions are well spread out over the function domain. Due to the randomness, clumping of samples often occur, and we arrive at situations where large portions of the integration domain are not being sampled. Note, however, that the sample points still have the correct distribution.

In this section, we introduce a deterministic variant of the above random sampling scheme that is capable of directly generating multiple samples according to the wavelet representation of an importance function. The new algorithm provides variance reduction by producing a sampling distribution that more uniformly covers the function domain. Input to the algorithm is the number of samples to generate, and output is a list of sample points with a distribution that follows our wavelet approximated importance function F.

Let N_{total} be the total number of samples to generate over the whole function, and $N(\mathbf{s})$ be the number of samples to place in a given wavelet square $\mathbf{s}$. The expected value of $N(\mathbf{s})$ is given by:

$$E[N(\mathbf{s})] = N_{total} P(\mathbf{x} \in \mathbf{s}), \tag{38}$$

where we have multiplied the total number of samples by the probability of sampling the square (Equation 35). Similarly, we can express the expected number of samples for each of the four child squares $\mathbf{s}_i$, using the conditional probabilities defined earlier:

$$E[N(\mathbf{s}_i)] = N(\mathbf{s}) P(\mathbf{x} \in \mathbf{s}_i | \mathbf{x} \in \mathbf{s}). \tag{39}$$

At each step in our algorithm, we use this equation to determine the number of samples allocated to each child square. Instead of randomly placing each sample according to the probabilities, we directly allocate $N(\mathbf{s}_i)$ samples for each square. In practice, we need an integer number of samples, so the floor operator is used to determine the minimum number of samples to place in each child square:

$$N(\mathbf{s}_i) = \lfloor N(\mathbf{s}) P(\mathbf{x} \in \mathbf{s}_i | \mathbf{x} \in \mathbf{s}) \rfloor. \tag{40}$$

If there are samples leftover, i.e., if $n = N(\mathbf{s}) - \sum N(\mathbf{s}_i) > 0$, those extra samples need to be placed randomly in the four child quadrants. For this, we must compute new child probabilities under the condition that we have already placed $N(\mathbf{s}_i)$ samples in each square, as the conditional probabilities defined earlier (Equation 36) are no longer applicable. The probability of sampling a square $\mathbf{s}_i$, which already has $N(\mathbf{s}_i)$ samples, is given by:

$$P(\mathbf{x} \in \mathbf{s}_i | \mathbf{x} \in \mathbf{s}, N(\mathbf{s}_i)) = \frac{\mathrm{frac}(N(\mathbf{s}) P(\mathbf{x} \in \mathbf{s}_i | \mathbf{x} \in \mathbf{s}))}{n}. \tag{41}$$

In the last equation, the probability is computed as the fractional part of the expected number of samples given by Equation 40, divided by the number of samples that are left. No normalization is needed as the fractions for the four quadrants sum up to n. We build a simple histogram over these new probabilities, and use that for distributing any leftover samples.

As before, squares that receive at least one sample are visited recursively down to the finest resolution. The final sample points are then chosen at random within those squares. Pseudo-code for this algorithm is given in Appendix C.2.3. The importance function is reconstructed from the wavelet representation on-the-fly, and the algorithm generates multiple samples in a single recursive traversal of the wavelet tree. The sampling is very efficient, as sample points are allocated in chunks instead of one at the time.

In all three sampling methods presented here, only the parts of the function that are being sampled need to be reconstructed from the wavelet representation. This is the key to fast sampling as large portions of the wavelet coefficients are never accessed. In the next section, we will show how these algorithms are directly applicable to sampling of products of functions.

3.5 Sampling of Wavelet Products

In many applications, the importance function $f(\mathbf{x})$ is a product of two functions, $f(\mathbf{x}) = g(\mathbf{x})h(\mathbf{x})$. Intuitively, to distribute samples according to the product, the two functions must first be multiplied together. With discrete representations of the functions, the multiplication can be done value by value over the entire function domain. However, this is time consuming and limits the flexibility. If one of the functions changes, the full product needs to be recomputed.

We store approximations of $g(\mathbf{x})$ and $h(\mathbf{x})$ as images, G and H respectively, expressed as Haar wavelets. Using the theory in Section 2.4, coefficients for the product $F = G \cdot H$ of the two wavelets can be computed very efficiently. These wavelet coefficients can then be used for reconstructing the function, and the above sampling algorithms used for distributing samples according to the product. However, since we only need scaling coefficients for sampling a wavelet representation, we use the optimized product presented in Section 2.5 instead of the general wavelet product.

The time complexity is further reduced by computing the product on-the-fly during the sampling process. This method has many advantages. First, only the parts of the function that are sampled need to be evaluated. Second, the product does not need to be stored, since the wavelet product coefficients are used only once during the sampling and then discarded. The flexibility of having separate representations of the two functions is retained. Changing one of the functions, does not cause any pre-processed data to become invalidated.

Algorithmically, sampling of wavelet products requires no changes to the pseudo-code given in Appendix C.2.1—C.2.3. The only difference is that all accesses to the wavelet coefficients $F^m_{l,\mathbf{t}}$, should be replaced with calls to a function that evaluates the wavelet product for the specified coefficient and returns the result. This, of course, includes the initial scaling coefficient, $F^0_{0,0}$, which should be replaced with the scaling coefficient for the product.

4 Application

Our method for importance sampling products of functions is a new general technique, which is not limited to a specific application. However, motivation for our research came from the field of computer graphics, where many problems involve complicated integrals. Most rendering techniques rely on computing an approximation of the rendering equation [14]:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_\Omega f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_i, \tag{42}$$

which describes the outgoing radiance in the outgoing direction $\vec{\omega}_o$ at a point $\mathbf{x}$ in the scene. The rendering equation is fundamental for photorealistic rendering, and involves an integral over the product of the BRDF, f_r, the incident radiance, L_i, and a simple cosine-term. As we saw in Section 1.1, there are numerous techniques for evaluating this integral through importance sampling. We use wavelet importance sampling for distributing samples according to the product of BRDF and lighting, for the case when the scene is under *direct illumination* by an environment map.

4.1 Direct Illumination

The rendering equation can be split into several components:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + L_{dir}(\mathbf{x}, \vec{\omega}_o) + L_{ind}(\mathbf{x}, \vec{\omega}_o), \tag{43}$$

and expressed as a sum of the contributions from the direct illumination, L_{dir}, the indirect illumination, L_{ind}, and the self-emitted radiance, L_e. Here, the direct illumination is given by the integral:

$$L_{dir}(\mathbf{x}, \vec{\omega}_o) = \int_\Omega f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L(\mathbf{x}, \vec{\omega}_i) v(\mathbf{x}, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_i, \tag{44}$$

where the incident radiance, $L(\mathbf{x}, \vec{\omega}_i)$, is provided by light sources in the scene, and $v(\mathbf{x}, \vec{\omega}_i)$ is the visibility of a light source in direction $\vec{\omega}_i$. In order to apply realistic lighting to a virtual scene, it is common to capture real lighting in a high-dynamic range environment map [8], and use that for L during rendering.

With both f_r and L defined, the only unknown quantity in Equation 44 is the visibility term $v(\mathbf{x}, \vec{\omega}_i)$. For a fixed outgoing direction, the BRDF and the environment map can be expressed as two-dimensional functions. We store approximations of both as Haar wavelets and use wavelet importance sampling for distributing samples according to the product of the two. The generated samples are then used for evaluating the integral by sampling the visibility.

In the following section, we describe our way of representing BRDFs and environment maps in a format that is suitable for wavelet importance sampling.

4.2 Wavelet Representation

4.2.1 BRDF

The *bidirectional reflectance distribution function* (BRDF) characterizes the reflection of light on a surface. The BRDF was first formally defined by Nicodemus et al. [29]. In radiometric terms, the BRDF is the surface radiance divided by the surface irradiance, i.e., the incident light flux per unit illuminated surface area:

$$f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) = \frac{\text{differential radiance}}{\text{differential irradiance}} = \frac{dL_o(\vec{\omega}_o)}{dL_i(\vec{\omega}_i)} = \frac{dL_o(\vec{\omega}_o)}{L_i(\vec{\omega}_i) cos\theta_i d\vec{\omega}_i}, \tag{45}$$

where $\mathbf{x}$ is the surface position, $\vec{\omega}_i$ is the incident direction, and $\vec{\omega}_o$ is the outgoing (viewing) direction. See Figure 3 for an illustration. Theoretically, the BRDF also depends on other variables such as wavelength and polarization of the light, but we are usually considering only unpolarized light of one specific wavelength at the time. Therefore, the BRDF can be written as a four-dimensional function. For a detailed discussion of the BRDF and its use in computer graphics, see the books by Glassner [10].

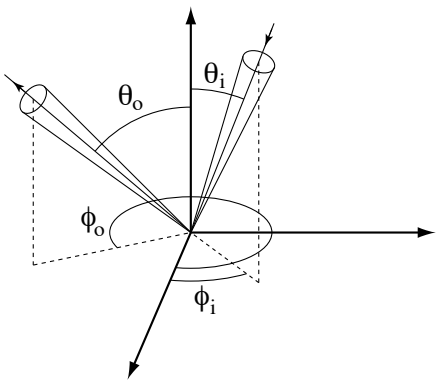


Figure 3: The bidirectional reflectance distribution function (BRDF) describes the ratio of outgoing radiance to incident radiance (irradiance). The function is typically parameterized over the spherical coordinates for the incident direction $\vec{\omega}_i = (\theta_i, \phi_i)$, and outgoing direction $\vec{\omega}_o = (\theta_o, \phi_o)$.

In our application, we combine the BRDF with the cosine term in Equation 44, and work with the reflectivity function $\rho(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o)$, instead of the BRDF:

$$\rho(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) = f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) \cos\theta_i. \tag{46}$$

The incident and the outgoing directions are typically given as spherical coordinates, $\vec{\omega}_i = (\theta_i, \phi_i)$ and $\vec{\omega}_o = (\theta_o, \phi_o)$, relative to a local surface frame at the point $\mathbf{x}$ on the surface, see Appendix A.1–A.3 for definitions. In this parameterization, most BRDFs are not particularly simple and do not compress well. For example, a shiny surface will have a high-intensity specular peak that lies in the direction of ideal specular reflection. In terms of outgoing direction, the position of this peak varies rapidly as the incident direction is changed.

By a change of variables, the BRDF can be transformed into a function that is more compact. There are many options for such *reparameterizations*. For example, in Rusinkiewicz's parameterization [34], the BRDF is expressed in terms of the half vector, i.e., the vector

halfway between the incident and the outgoing vectors. In our application, we need a parameterization that is suitable for both the BRDF and for the environment map. Therefore, as noted previously [33], Rusinkiewicz's parameterization cannot be used as it is unsuitable for the environment map. The half vector depends on outgoing direction, while the environment map depends only on the incident direction.

We use a parameterization about the reflection vector as in [33]. In this representation, the BRDF is centered about the reflection vector $\vec{\omega}_r = (\theta_r, \phi_r)$, instead of around the surface normal $\vec{n}$. The reflection vector is defined as the ideal reflection of the incident direction in the surface normal:

$$\vec{\omega}_r = 2\vec{n}(\vec{n} \cdot \vec{\omega}_i) - \vec{\omega}_i. \tag{47}$$

The directions $\vec{\omega}_i$ and $\vec{\omega}_o$ are now given with respect to $\vec{\omega}_r$, and the dependence on the normal is implicit. The advantage of this parameterization is that specular lobes are usually aligned with $\vec{\omega}_r$, which makes the function much more compressible. As noted by Cabral et al. [3], the variation over $\vec{\omega}_o$ is usually slow. Thus, we can store the BRDF with a relatively sparse sampling over outgoing directions.

In practice, we store the BRDF tabulated as a sparse 2D set of 2D wavelet compressed reflection maps. The tabulation is done over outgoing direction $\vec{\omega}_o$, expressed in spherical coordinates. Each reflection map represents the reflectance over incident direction, centered about the reflection vector for a specific tabulated outgoing direction. The maps are stored at the resolution 64×64 or 128×128. For the tabulation, we use a lower resolution of 16×16 or 32×32 different outgoing directions, although an even lower resolution is motivated for some materials.

4.2.2 Environment Map

An environment map is typically defined as a two-dimensional function $L(\vec{\omega})$, representing the incident radiance as a function of world space direction $\vec{\omega}$. The lighting is assumed to be distant, meaning that $L(\vec{\omega})$ depends only on the direction and does not vary across the scene.

We want to apply wavelet importance sampling to the product of an environment map and a BRDF. A fundamental limitation of using wavelet multiplication is that the two functions must be expressed in the same coordinate system. As noted previously by Ng et al. [28], there is no straightforward way of rotating wavelets. In our application, the BRDF is given in local coordinates with respect to the reflection vector, while an environment map is commonly expressed in global coordinates.

We solve this problem by rewriting the environment map as a four-dimensional function $L(\vec{\omega}, \vec{\omega}_r)$, that explicitly depends on the global coordinates of the reflection vector $\vec{\omega}_r$. In our representation, the direction $\vec{\omega}$ is given with respect to $\vec{\omega}_r$. Essentially, for a fixed reflection vector, $L(\vec{\omega}, \vec{\omega}_r)$ represents a pre-rotated two-dimensional environment map centered about that vector. Hence, the environment map is in the same local space as the BRDF.

Similarly to the BRDF, we store the environment map as a 2D tabulation of 2D wavelet compressed images. The tabulation is done over the reflected direction $\vec{\omega}_r$, expressed in global spherical coordinates, $\vec{\omega}_r = (\theta_r, \phi_r)$, with $0 \leq \theta_r \leq \pi$ and $0 \leq \phi_r < 2\pi$. For each tabulated direction, the corresponding wavelet image is the full environment map centered about that direction. Unfortunately, we cannot use a sparse tabulation as the environment map usually has high-frequency content, i.e., it has rapidly varying light intensity. Therefore, to get pixel

accuracy, we use a tabulation of the same resolution as the wavelet image. Specifically, we store a 128×128 tabulation of 128×128 environment maps, or alternatively a 64×64 tabulation of 64×64 maps. During rendering, the current reflection vector is computed and the environment maps corresponding to the nearest tabulated directions are bi-linearly interpolated and used for importance sampling.

The integral in the equation for direct illumination (Equation 44) is over solid angle, $d\vec{\omega}_i$. In order to get the correct result, we must take the solid angle of each pixel in the discrete representations of the BRDF and the environment map into account. We have chosen to normalize the environment for solid angle, and leave the values in the BRDF unchanged. Each pixel in the environment map is multiplied by a normalization factor, whose value is derived in Appendix A.4.

It is worth noting that the representation presented here is not the only solution. Ng et al. [28] rewrite the BRDF as a 6D function expressed in global coordinates, which depends on the surface normal. Thus, they can directly multiply the BRDF with a 2D environment map. However, even with efficient compression they report memory usage in the order of hundreds of megabytes for each BRDF. In a scene with many materials, their representation quickly becomes unusable.

4.3 Rendering

For rendering an image, we trace rays from the camera through each pixel. When a surface is hit at a point $\mathbf{x}$, we compute the relevant vectors such as the surface normal $\vec{n}$, the outgoing direction $\vec{\omega}_o$, and the reflection vector $\vec{\omega}_r$. These are then used to find the correct "slices" of the tabulated environment map and the BRDF. With wavelet representations of the two functions at hand, we can apply our importance sampling scheme to the wavelet product of the two functions, generating a set of sample points expressed in spherical coordinates.

Each sample point corresponds to an incident direction $\vec{\omega}_i$, given in a local frame relative to $\vec{\omega}_r$. The sampling directions are transformed back to world coordinates, and used for evaluating the visibility through ray tracing. In practice, we cast a shadow ray from the surface point $\mathbf{x}$ in direction $\vec{\omega}_i$. Each visibility sample returns a binary value:

$$v(\mathbf{x}, \vec{\omega}_i) := \begin{cases} 1, & \text{if background is visible in direction } \vec{\omega}_i \\ 0, & \text{otherwise.} \end{cases} \tag{48}$$

Now, with all the terms in Equation 44 known for every sample, we can evaluate the value of the integral using Monte Carlo integration. We first show how an unbiased result can be obtained, and then show how the variance can be drastically reduced, at the expense of introducing a slight bias.

4.3.1 Unbiased Rendering

An unbiased estimator of the integral for direct illumination can be written as:

$$\bar{L}_{dir}(\mathbf{x}, \vec{\omega}_o) = \frac{1}{N} \sum_{i=1}^{N} \frac{\rho(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L(\mathbf{x}, \vec{\omega}_i) v(\mathbf{x}, \vec{\omega}_i)}{p(\vec{\omega}_i)}, \tag{49}$$

where we have replaced the BRDF and the cosine term with the reflectivity function, $\rho(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o)$. To obtain the unbiased estimate of the lighting, $\bar{L}_{dir}(\mathbf{x}, \vec{\omega}_o)$, we evaluate the true value of

the reflectivity and the environment map for each sampling direction, and divide by the probability associated with each sample.

Using our deterministic wavelet sampling scheme, we distribute samples according to the product of the reflectivity (BRDF) and the environment map. Let $F = G \cdot H$ be the wavelet product, where G is the reflectivity and H is the environment map. The probability associated with a sample $\vec{\omega}_i$, is given by:

$$p(\vec{\omega}_i) = \frac{F(\mathbf{s})}{F^0_{0,0}}, \tag{50}$$

where $\mathbf{s}$ is the wavelet square at the finest resolution, in which the sample point corresponding to $\vec{\omega}_i$ is located. These probabilities are directly given by the wavelet reconstruction during the sampling. Hence, we can compute an unbiased estimation of the direct illumination by evaluating Equation 49.

4.3.2 Biased Rendering

Unbiased techniques generally produce a fair amount of noise, unless a very large number of samples is used. In many cases, it is better to introduce some bias to significantly reduce the noise level. This is particularly true in our application.

Our wavelet representations of the reflectivity and the environment map approximate the functions fairly well, i.e., $G(\mathbf{s}) \approx \rho(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o)$ and $H(\mathbf{s}) \approx L(\mathbf{x}, \vec{\omega}_i)$. In the sampling process, the wavelet product gives the approximated value of the product of reflectivity and lighting for each sample, $F(\mathbf{s}) \approx \rho(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L(\mathbf{x}, \vec{\omega}_i)$. Combining this approximation with Equation 49 and 50, yields:

$$\begin{aligned} \bar{L}_{dir}(\mathbf{x}, \vec{\omega}_o) &\approx \frac{1}{N} \sum_{i=1}^{N} \frac{F(\mathbf{s}) v(\mathbf{x}, \vec{\omega}_i)}{F(\mathbf{s})/F^0_{0,0}} \\ &= \frac{F^0_{0,0}}{N} \sum_{i=1}^{N} v(\mathbf{x}, \vec{\omega}_i). \end{aligned} \tag{51}$$

By accepting the small bias introduced with the wavelet approximation, we drastically reduce the noise in the rendered image. Effectively, two of the terms in the Monte Carlo estimator, which are a major source of variance, are eliminated. Using Equation 51, rendering is reduced to a summation of the visibility terms, multiplied by the scaling coefficient for the product. Rendering is also very fast, since we do not have to evaluate the true values of the BRDF and the environment map for each sampling direction.

The first scaling coefficient of the product, $F^0_{0,0}$, is available for free as it is needed in the sampling process. A key observation is that this scaling coefficient represents the pre-integrated value of the reflectivity multiplied by the environment map, assuming full visibility. In areas with no occlusion, the rendered image will be completely noise-free. In shadow areas, noise is inevitable, but since the samples are distributed according to the combination of the BRDF and the lighting, our method quickly converges towards a noise-free result.

5 Implementation

In this chapter, we briefly describe some of the implementation specific details of our example application. This chapter is not necessary for the understanding of our method, but should provide useful information for the reader who wants to use wavelet importance sampling of products in practice.

5.1 Wavelet Framework

5.1.1 Creation

As noted earlier, we use normalized Haar wavelets and the non-standard decomposition. Starting with an image with $n \times n$ pixels, where n is a power of two, the one-dimensional wavelet transform is applied first on rows and then on columns. The wavelet transform is done using the *lifting scheme* [37], which is a fast linear-time operation that does not require any additional temporary storage. Each pass splits the data into a set of scaling coefficients and a set of detail coefficients. The transform is applied recursively on the scaling coefficients, until only a single scaling coefficient remains. This is the normal approach and many good textbooks describe the algorithm in more detail [24, 36].

After our data has been wavelet transformed, we apply lossy compression by thresholdning the wavelet coefficients, as described in Section 2.3. In our implementation, we manually set the threshold to give the desired amount of compression. One could alternatively use an automatic method for determining a reasonable threshold.

5.1.2 Storage

After compression, the sparse set of wavelet coefficients needs to be represented in a compact way. The options are to use a hash map or a wavelet coefficient tree [19]. We have chosen to use hash maps, as they allow for quick constant-time $O(1)$ random access and easy insertion of elements.

We use Knuth's multiplicative hash function [16], and key on the sequential index of the wavelet coefficient. The key is multiplied by the golden ratio of 2^{32}, which is 2654435761, to produce a hash result. Since these two numbers have no common factors, this method produces a complete mapping of keys to hash values with no overlap. The method is fast and works well if the keys have small values.

The hash map is implemented with open addressing and linear probing to handle collisions. For a load factor of α, the expected number of lookups in an unsuccessful search is at most $1/(1-\alpha)$. The size of the hash map is chosen as a power of two, such that the size of the map is approximately twice the number of elements. In our case, this means an average of two lookups per unsuccessful search, which is an acceptable tradeoff between speed and memory. See Knuth's book [16] for more details.

5.1.3 Color

We handle color information by storing wavelet coefficients as RGB-triplets instead of single values, i.e., $\mathbf{\Psi}_i = (\Psi_i^R, \Psi_i^G, \Psi_i^B)$. All mathematical operations on the coefficients are done

component-wise. However, for computing the probabilities needed for sampling, we are only interested in the intensity of the function. Thus, we distribute samples according to average value of the three color components, $I = (R + G + B)/3$.

The drawback of storing color information in a single wavelet representation, as compared to having a separate wavelet image for each color channel, is less efficient compression. We believe, however, that this method is faster as the coefficients are kept together in the memory instead of being scattered in three different places.

5.1.4 Interpolation

In our application, both the BRDF and the environment map are represented as 2D tabulations of 2D wavelets. In order to get a smooth result, we use bilinear interpolation between the four nearest wavelets in the tabulation of each function. The interpolation is done on-the-fly, directly on the wavelet coefficients, as coefficients are looked up in the wavelet representations.

In practice, we first pre-compute the bilinear interpolation weights (α, β), $0 \leq \alpha, \beta < 1$, and then compute the value of each wavelet coefficient as:

$$\Psi_i = (1-\alpha)(1-\beta)\Psi_i^1 + \alpha(1-\beta)\Psi_i^2 + (1-\alpha)\beta\Psi_i^3 + \alpha\beta\Psi_i^4. \tag{52}$$

5.1.5 Wavelet Product

We have implemented the functionality required for computing the wavelet product of two images as a separate class in our object-oriented system. This class provides the same interface as the classes representing a single wavelet-expressed function, and hence allows us to directly sample the product without any modifications to the implementation of the sampling scheme. Internally, the wavelet product class uses the optimized wavelet product described in Section 2.5, and computes scaling coefficients on-the-fly as they are needed during the sampling. Our implementation closely follows the given pseudo-code, and could probably be further optimized.

5.2 Wavelet Data

5.2.1 BRDF

In our renderings, we use isotropic BRDFs acquired from real materials. The BRDF data sets we use were created by Matusik [25, 26], and consist of dense reflectance measurements in color for over 100 different materials. The measurements were done for $90 \times 90 \times 180$ discrete directions represented in Rusinkiewicz's parameterization [34], with a denser sampling around the specular highlight. A few of the BRDFs and source-code for loading them can be downloaded from Matusik's website[1].

We resample the measured reflectance data into our BRDF representation described in Section 4.2.1. Each two-dimensional reflectance map is first created at a high resolution (256×256), and then subsampled to the desired resolution to avoid aliasing.

[1] `http://graphics.csail.mit.edu/~wojciech/BRDF/`

For testing purposes, we also use the traditional Lambertian model [20], and the Phong model [35]. We center the Lambertian (a perfectly diffuse BRDF) around the surface normal, and the Phong lobe around the reflection vector. In this representation, the functions are independent of the outgoing direction, so we only need to store a single two-dimensional reflectance map for each.

5.2.2 Environment Map

We use high-dynamic range environment maps [8], so called *light probes*, for rendering scenes under realistic illumination. Paul Debevec's Light Probe Gallery[2] has a selection of environment maps available for download. The light probes are stored in Ward's RGBE format, described in [40]. The maps represent the full sphere, 4π steradians, and use a simple angular parameterization. For viewing and manipulating high-dynamic range images, we recommend the program HDRShop[3].

The light probes are resampled into our environment map representation described in Section 4.2.2. For each tabulated two-dimensional environment map, we loop over its pixels and sample the light probe in the corresponding direction. To avoid aliasing, the resampling is done at a higher resolution than the final result, usually 512×512 or 1024×1024, and the image is downsampled to the desired resolution using a box filter. Each two-dimensional environment map is then wavelet compressed, and only the non-zero coefficients are stored.

5.3 Ray Tracer

A simple Monte Carlo ray tracer was implemented in order to evaluate how well wavelet importance sampling of products performs compared to other methods. The system was written in C++ and uses Windows MFC for providing a graphical user interface.

5.3.1 Scene Description

We use XML as a scene description language, describing all properties of the scene with custom nodes. The use of XML makes the scene description human readable, and simplifies the implementation tremendously as good parser libraries such as TinyXml[4] are available.

The renderer supports triangle-based geometry in the PLY[5] format and in the Wavefront OBJ format. Plugins for importing and exporting objects in for example Alias Maya, exist for both formats, making it relatively easy to create or modify geometry.

5.3.2 Ray Shooting

Ray tracing of large scenes is inherently slow unless an efficient acceleration technique is used. Most methods rely on the use of a spatial data structure to limit the number of intersection tests required. Naturally, the choice of best acceleration technique largely depends on the application. However, it was shown by Havran [13] that for a set of 30 example scenes,

[2] `http://www.debevec.org/Probes/`
[3] `http://www.debevec.org/HDRShop/`
[4] `http://sourceforge.net/projects/tinyxml/`
[5] `http://graphics.stanford.edu/data/3Dscanrep/`

algorithms based on the *kd*-tree performed on average better than methods based on BSP trees, octrees, uniform grids, bounding volume hierarchies, and hierarchical grids.

A *kd*-tree is a binary space partioning with axis-aligned splitting planes. The location of the splitting planes has a big impact on the performance. By constructing the tree using a cost model for estimating the average cost of traversing a ray through the tree, the traversal time can be minimized. In our ray tracer, the *ordinary surface area heuristic* (OSAH) cost model [13] is used for constructing the tree, and a recursive ray traversal algorithm introduced by Havran et al. [12] is used for tree traversal.

Ray-triangle intersection testing was done with code based on the algorithm by Möller and Trumbore [27].

6 Results

It is difficult to fully evaluate the performance of wavelet importance sampling of products, as the parameter space is very large. The result depends on the wavelet resolution, the sparsity of the BRDF and the environment map, which sampling method is used, the number of samples per pixel, and so forth. Naturally, the performance of our method, like any other method, is also very dependent on the scene. In this chapter, we try to evaluate what effect some of these parameters have on the result. Our goal is to find a set of parameters that work well for most scenes.

First, we compare the random sampling method to our deterministic sampling method described in Section 3.4.4. Second, we investigate how BRDF compression affects the rendered images. Finally, we compare our method to environment map importance sampling, by visually comparing the result on a complex scene with many different materials. In the last section, we present the computation times for creating wavelet representations of the environment map and the BRDFs. All results were generated on a PC with an AMD Athlon XP2600+ (1.91GHz) processor and 1GB memory. The *biased* rendering technique presented in Section 4.3.2, was used for rendering all of the images.

6.1 Sampling Method – Random vs Deterministic Sampling

In this section, we compare our deterministic sampling algorithm introduced in Section 3.4.4 against ordinary random thresholdning, as described in Section 3.4.3. We render a simple scene under direct illumination captured in a high-dynamic range environment map of St. Peters Basilica. The scene consists of a sphere on a plane, both using a measured *rose-quartz* BRDF. This BRDF was chosen because it contains both a sharp specular peak and a relatively large diffuse component. The reference image shown in Figure 4, was computed using brute-force ray tracing with 1 million samples per pixel.

Figure 4: Reference image showing the simple test scene, rendered using brute-force ray tracing. A measured rose-quartz BRDF was used, and the lighting is from St. Peters Basilica.

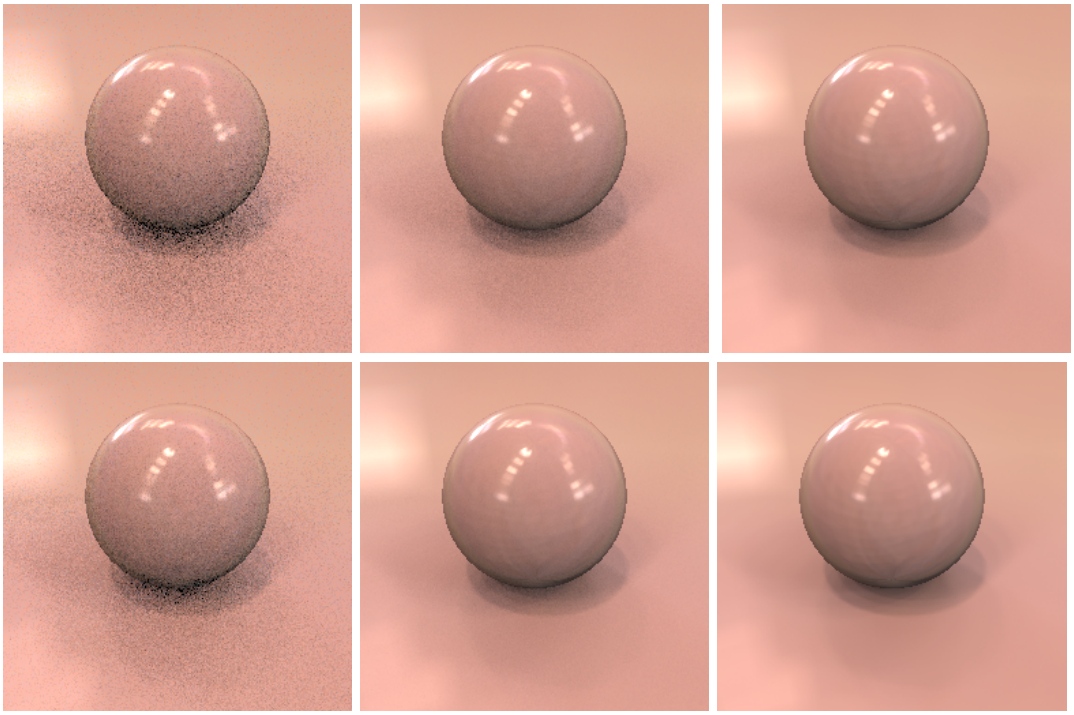


Figure 5: The top row was rendered with random sampling of the wavelet product, and the bottom row was rendered with our deterministic sampling algorithm. From left to right: 10, 100, and 1000 samples/pixel.

To evaluate the performance of the two sampling algorithms, we rendered the test scene using wavelet importance sampling with the number of samples per pixel gradually increasing from 1 to 1000. The wavelet resolution was 64×64 for both the environment map and the BRDF, and the BRDF was compressed to 2.84% sparsity (a compression of 1:35). Figure 5 shows the result for 10, 100, and 1000 samples/pixel. For the top row, random sampling was used, and for the bottom row we used the deterministic sampling method. From these images, it is obvious that the limited wavelet resolution introduces some error. The low resolution (64×64) wavelet representations are essentially low-pass filtered representations of the original data, and hence the result is blurred reflections.

Since the biased rendering technique was used, it is only in shadow areas there is any difference between the two sampling methods. To quantitatively compare them, we study the noise in a section of the shadow region. Figure 6 shows this part of the scene, rendered with 1 to 1000 samples/pixel. We measured the variance in the difference to the reference image, and the results are presented in Figure 7. The left diagram shows the variance versus the number of samples for the two methods, while the variance versus rendering time is shown on the right.

The conclusion is that our deterministic sampling technique gives better results than the random sampling method. The difference is largest when an average number of samples is used. In the range 10–100 samples/pixel, the deterministic sampling algorithm performs significantly better than random sampling. Both methods converge towards a low, but larger than zero, error due to the bias introduced by the wavelet approximation.

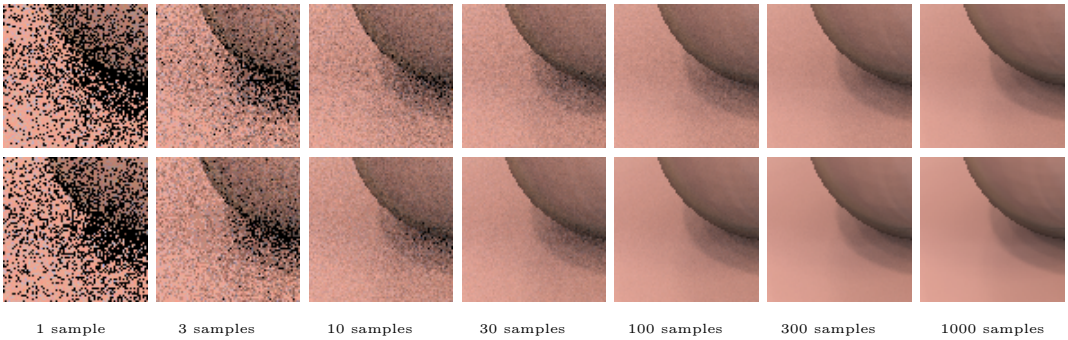


Figure 6: Part of the shadow region showing the difference between random sampling (top row) and deterministic sampling (bottom row). The scene was rendered with 1, 3, 10, 30, 100, 300, and 1000 visibility samples per pixel.

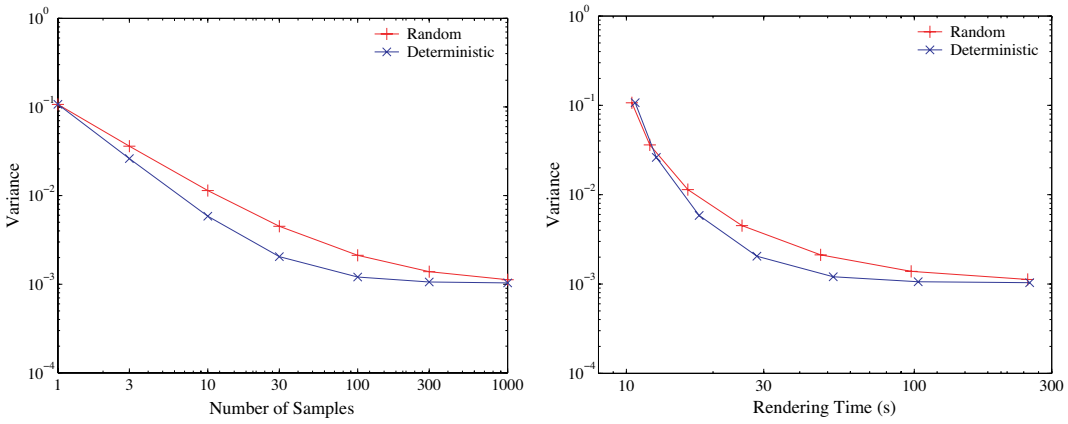


Figure 7: Variance in the rendered images compared to the reference image.

6.2 BRDF Compression

In this section, we study the effect of BRDF compression on the rendered result. In the first example, we rendered the test scene using a wavelet resolution of 64×64 and different levels of BRDF sparsity, ranging from 8.64% to 2.02%. The rendered images are presented in Figure 8, and variance diagrams are shown in Figure 9. The left plot shows the variance as a function of the number of samples, and the right image shows the variance versus rendering time.

We repeated the test using a wavelet resolution of 128×128, and BRDFs with sparsities in the range 4.72% to 0.84%. These images are shown in Figure 10, and the corresponding variance diagrams can be found in Figure 11. The higher wavelet resolution gives a result much closer to the reference image. The specular reflections of strong light sources in the environment map, appear much sharper than with lower resolution wavelet representations.

The level of BRDF compression does not seem to have a significant impact on the visual result. This is confirmed by the variance diagrams, which show that there are only minor differences between the images. The only noticeable artifact from using a too high level

of compression, is that the shading appears slightly "blocky". However, as shown by the variance-vs-time diagrams in Figure 9 and 11, the level of BRDF compression has a huge impact on the rendering speed. As always, there is a tradeoff between quality and rendering time. For short rendering times, the best results are obtained by using highly compressed BRDFs, but for longer rendering times a lower degree of compression gives better results.

For a wavelet resolution of 64×64, a BRDF sparsity in the range 2%–3% generally gives good results. For 128×128 wavelets, the corresponding number is 1%–1.5%. The variance diagrams show that these levels of sparsity give close to optimal results when 30–100 samples per pixel are used for rendering. Fewer samples tend to give too much noise in the shadows, while more samples only give a slight improvement in quality, at the expense of considerably longer rendering times.

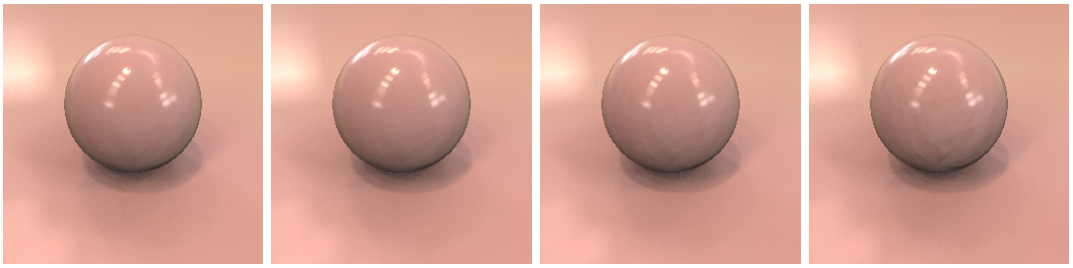


Figure 8: The test scene rendered using wavelet importance sampling with 100 samples/pixel, but for different levels of BRDF sparsity. The wavelet resolution was 64×64.

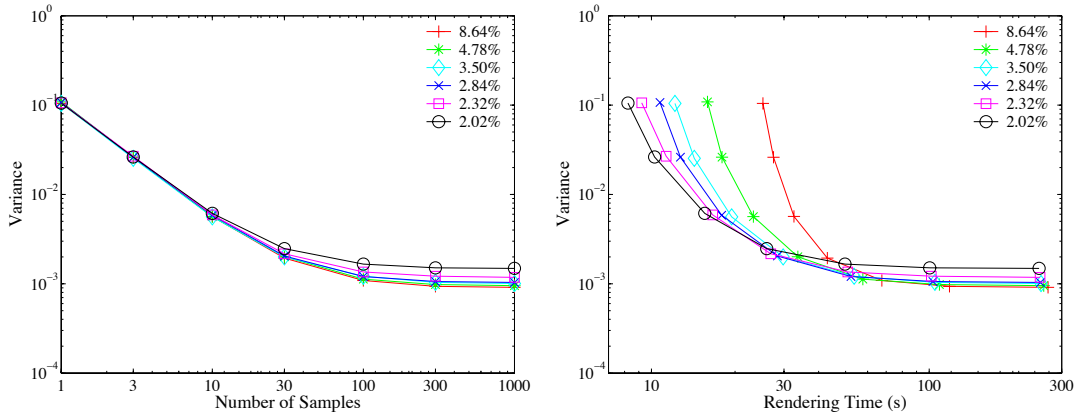


Figure 9: Variance as a function of the number of samples (left), and as a function of rendering time (right), for different levels of BRDF sparsity. The wavelet resolution was 64×64, and the variance was computed over the area shown in Figure 6.

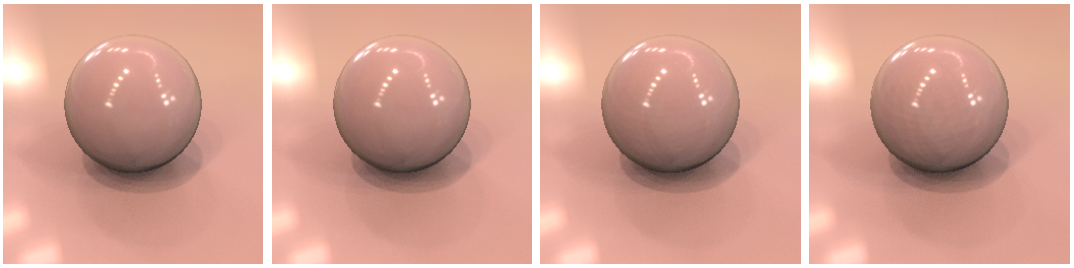


Figure 10: The test scene rendered using wavelet importance sampling with 100 samples/pixel for different levels of BRDF sparsity, but at a wavelet resolution of 128×128.

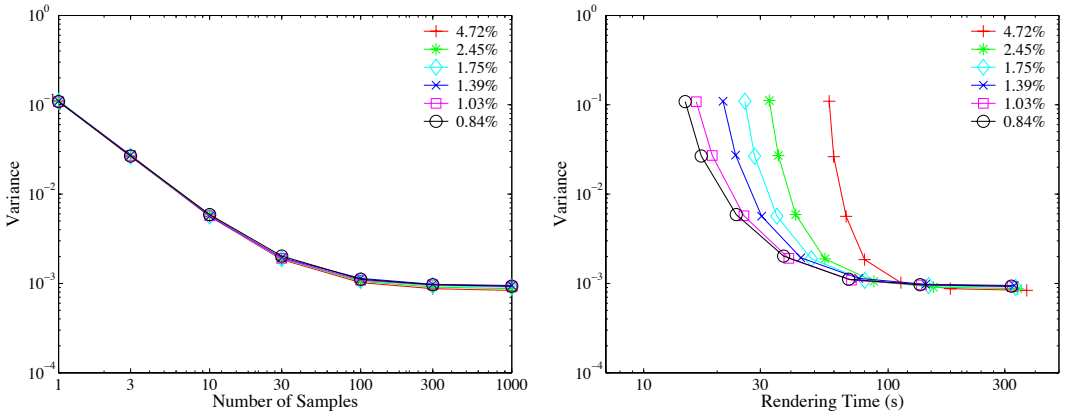


Figure 11: Variance as a function of the number of samples (left), and as a function of rendering time (right), for different levels of BRDF sparsity. The wavelet resolution was 128×128, and the variance was computed over the same area as before.

6.3 Visual Results

We rendered a more complex scene consisting of a set of objects with various measured BRDFs on a perfectly diffuse table. The lighting is provided by a high-dynamic range environment map captured in Galileo's Tomb. Figure 12 was rendered using deterministic sampling of the wavelet product of BRDF and lighting, with 30 visibility samples per pixel. The wavelet resolution was 64×64, and the BRDFs were compressed to about 2%–3% sparsity. The rendering time was *31 minutes* for 1600×1200 pixels resolution with 2×2 anti-aliasing.

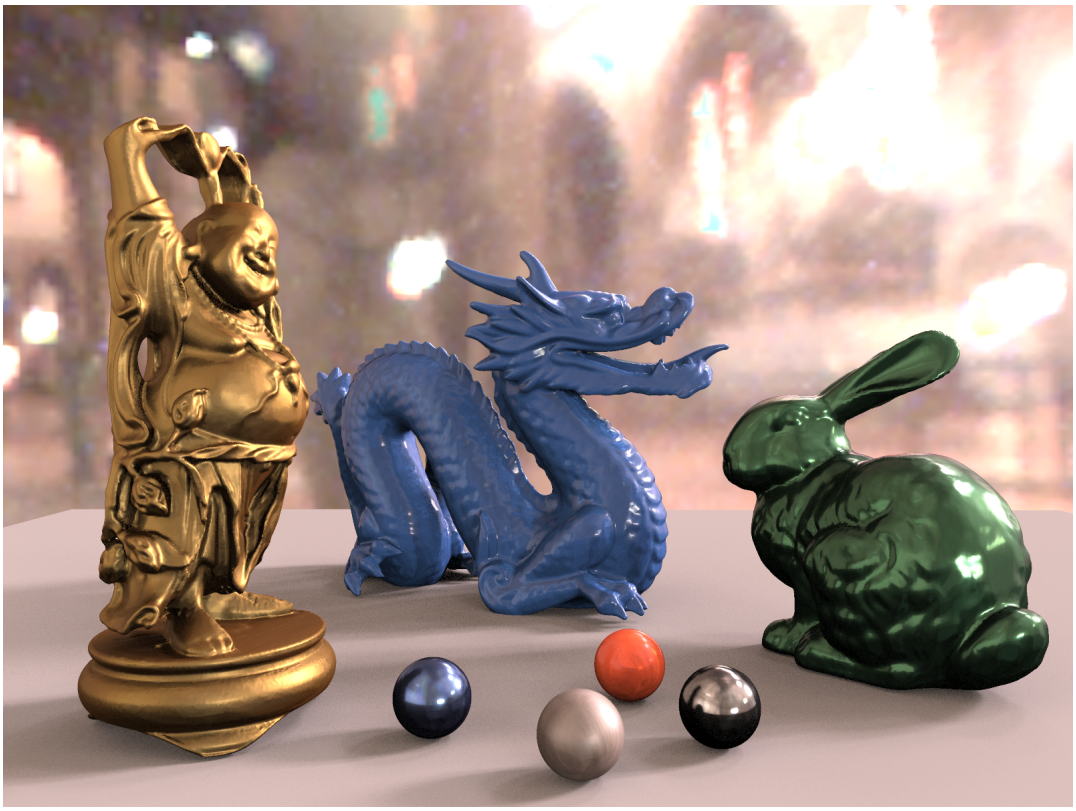


Figure 12: Objects with various measured BRDFs on a diffuse table, under direct illumination from Galileo's Tomb. The image was rendered using deterministic sampling of the wavelet product with *30 samples/pixel*. Rendering time: 31 minutes.

To compare our method with environment map importance sampling, we rendered the same scene using structured importance sampling [1]. For Figure 13, we used 100 samples per pixel, giving a rendering time of *32 minutes*. The number of samples was chosen to achieve approximately the same rendering time as for Figure 12. For equal rendering times, about 3 times as many samples can be used since structured importance sampling is a preprocess that incurs no additional run-time cost. On the other hand, for wavelet importance sampling, in addition to ray tracing we have the cost of computing and sampling the wavelet product for each pixel.

Although structured importance sampling is a popular technique for rendering scenes under environment map lighting, the technique is clearly only usable for rendering non-specular materials. For BRDFs with a sharp specular peak, the reflections of the pre-integrated lights

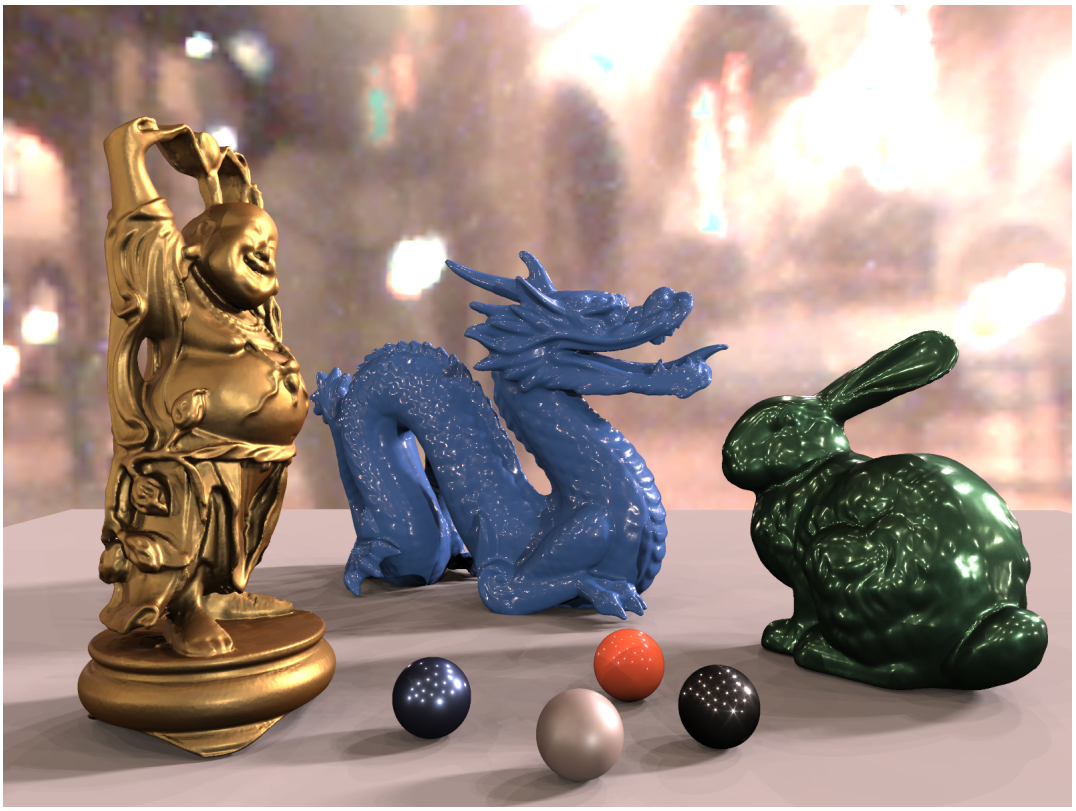


Figure 13: The same scene as in Figure 12, rendered using structured importance sampling of the environment map, with 100 samples per pixel. Rendering time: 32 minutes.

become very disturbing. The same applies to all other environment map sampling techniques that approximate the lighting by a set of pre-integrated directional lights.

Figure 14 highlights the differences between our wavelet product sampling technique and environment map importance sampling. In the top two rows, the scene was rendered using 30 samples/pixel, generated by wavelet importance sampling of the product of BRDF and lighting environment for two different wavelet resolutions: 64×64 and 128×128 respectively. Using the higher wavelet resolution has no visible effect on the shadows or on the diffuser BRDFs, but the glossy objects look better because the reflection of the environment map is less blurred. For the bottom two rows, we used structured importance sampling, and rendered the scene using 100 and 1000 samples/pixel respectively. With 100 lights, there is clearly visible banding of the shadows. Increasing the number of samples to 1000 improves the shadows, but for rendering glossy objects, even this large number of samples is not nearly enough. Rendering times for these four examples are presented in the following table:

Method	*Samples/Pixel*	*Time*
Wavelet product sampling, 64×64	30	31 min
Wavelet product sampling, 128×128	30	46 min
Structured importance sampling	100	32 min
Structured importance sampling	1000	346 min

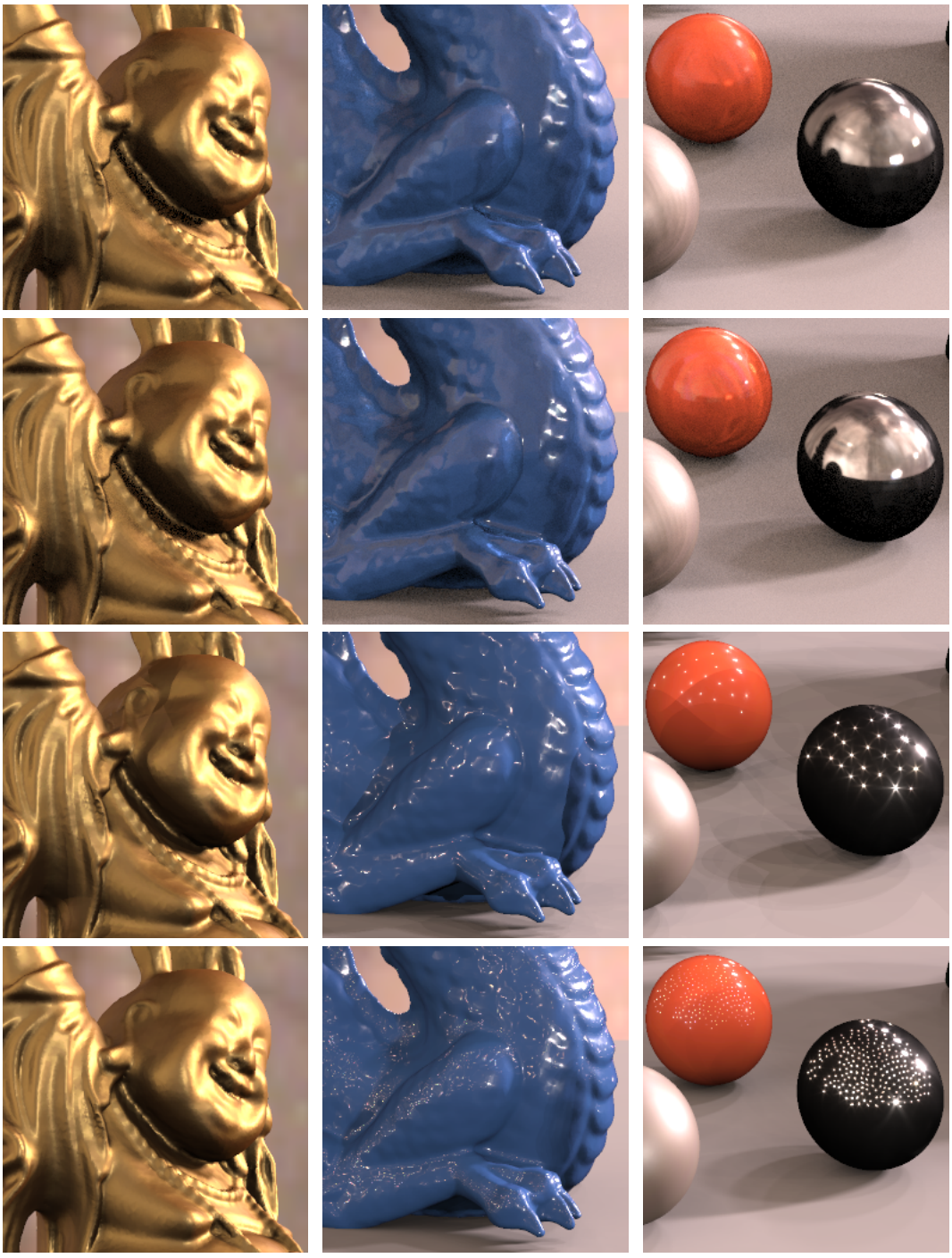


Figure 14: Top row: Wavelet importance sampling, 30 samples/pixel, 64×64 wavelet resolution, 31 min. Second row: Wavelet importance sampling, 30 samples/pixel, 128×128 wavelet resolution, 46 min. Third row: Structured importance sampling, 100 samples/pixel, 32 min. Bottom row: Structured importance sampling, 1000 samples/pixel, 346 min.

6.4 Precomputation Time

In precomputation time, we include the creation of wavelet representations of the environment map and the BRDFs. Note that this needs to be done only once, as the wavelet representations can be compactly stored to disk and reused later.

Computing a wavelet representation of a full measured 4D BRDF took *20 seconds* for a 16×16 tabulation of 64×64 reflectance maps, using 4×4 super-sampling to avoid aliasing (i.e., BRDF was sampled at 256×256 and downscaled to the desired resolution). For higher resolution BRDFs ($32 \times 32 \times 128 \times 128$), the computation finished in *79 seconds*. The level of wavelet compression did not affect the computation time noticeably.

For computing a pre-rotated wavelet environment map (a 2D tabulation of 2D wavelet compressed maps) we must use a much denser sampling, since the lighting is typically not as smooth as the BRDFs. The resolution of a light probe is also much higher than that of a typical measured BRDF. Hence, aliasing is much more of a problem. We sample the original environment map at a resolution of 1024×1024, and downscale the result to the target wavelet resolution. For a wavelet resolution of $64 \times 64 \times 64 \times 64$, the computation time was about *1 hour*, and for $128 \times 128 \times 128 \times 128$ resolution it was approximately *4 hours*.

The majority of the computation time is spent super-sampling the environment map. Note that these computation times are not the final word. By, for example, using graphics hardware to generate the pre-rotated and subsampled environment maps, we could drastically reduce the precomputation time. Another solution would be to low-pass filter the original environment map, in order to reduce the amount of super-sampling needed.

7 Conclusions and Future Work

In this thesis, a new general tool for importance sampling products of two-dimensional functions has been introduced. The proposed method uses fast wavelet multiplication together with an efficient sampling algorithm to produce sampling distributions that accurately match the energy distribution in the product of two functions. The generated samples are used for significantly reducing the variance in Monte Carlo techniques.

For the example application, i.e., rendering objects with realistic measured materials under complex distant illumination by a high-dynamic range environment map, our new sampling method proved to give high-quality renderings with a very small number of samples per pixel. Our technique compares favorably to environment map importance sampling. The extra cost of evaluating and sampling the wavelet product on-the-fly during rendering, is well motivated by the reduction in variance.

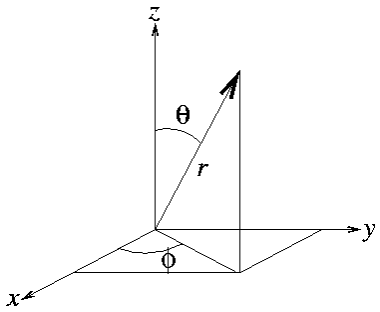
With today's computers, the maximum wavelet resolution is still somewhat limited by the memory usage. One future research direction is to find better parameterizations of the environment map and the BRDF, in order to enable better compression and reduce the memory requirements. Quantization of the wavelet coefficients could also help reduce the size. Another direction of research is to develop better sampling algorithms, both in terms of speed and in the properties of the final point distribution.

8 Acknowledgements

First of all, I would like to thank my supervisor, Tomas Akenine-Möller, for a good collaboration. I would also like to thank Henrik Wann Jensen for fruitful discussions and for hosting me at the University of California at San Diego, where I was visiting during my work. The implementation of the system was done together with Wojciech Jarosz, who deserves a thank for many useful ideas. Last, I am grateful for all the love and support received from my family, my girlfriend Kristen, and her family during the intense work of finishing this thesis.

A Mathematical Definitions

A.1 Spherical Coordinates



Spherical coordinates (r, θ, ϕ) are natural for describing positions on the sphere. We define θ to be the polar angle (the zenith) from the positive z-axis with $0 \leq \theta \leq \pi$, and ϕ as the rotation about the z-axis in the xy-plane with $0 \leq \phi < 2\pi$. The distance (radius) from the origin is r.

The conversion from Cartesian coordinates is given by:

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \theta &= \sin^{-1}\left(\frac{\sqrt{x^2+y^2}}{r}\right) = \cos^{-1}\left(\frac{z}{r}\right) \\ \phi &= \begin{cases} \tan^{-1}\left(\frac{y}{x}\right), & x \geq 0 \\ \pi + \tan^{-1}\left(\frac{y}{x}\right), & x < 0 \end{cases} . \end{aligned}$$

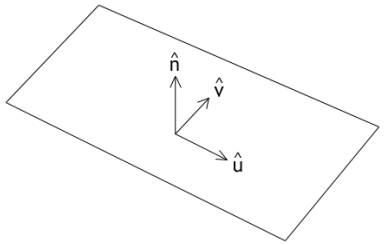
It is convenient to use the atan2(y, x) function for calculating ϕ with correct sign. The atan2-function gives the four quadrant arctangent, and is supported in most programming languages.

The opposite conversion, from spherical to Cartesian coordinates, is defined as:

$$\begin{aligned} x &= r\cos\phi\sin\theta \\ y &= r\sin\phi\sin\theta \\ z &= r\cos\theta. \end{aligned}$$

In many applications, the coordinate axes are rotated so that the y-axis points up and the zx-plane defines the equatorial plane. In this case it is convenient to define θ as the polar angle (zenith) from the positive y-axis, and ϕ as the azimuth in the zx-plane.

A.2 Surface Frame

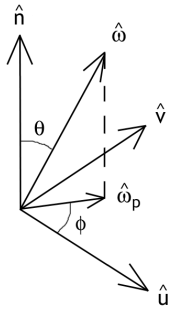


The basis vectors $(\vec{u}, \vec{v}, \vec{n})$ combined with a point x define the *surface frame* at x. The normal $\vec{n}$ is perpendicular to the surface, and the vectors $\vec{u}$ and $\vec{v}$ are surface tangents. $\vec{u}$ is called *primary tangent* or just *tangent*, and $\vec{v}$ is called *secondary tangent*, *bitangent* or sometimes *binormal*.

The basis $(\vec{u}, \vec{v}, \vec{n})$ is right-handed, so positive rotations will be counter-clockwise about the normal. We usually want the basis to be orthonormal, in which case:

$$\begin{aligned} \vec{n} &= \vec{u} \times \vec{v} \\ \vec{v} &= \vec{n} \times \vec{u}. \end{aligned}$$

A.3 Spherical Surface Coordinates



Spherical coordinates relative to a given surface frame are useful in many cases. For example, the standard parameterization of a BRDF is with respect to the incident direction $\vec{\omega}_i$ and viewing direction $\vec{\omega}_o$ relative to the local surface frame at a point x.

The spherical coordinates (θ, ϕ) of a vector $\vec{\omega}$ relative to the surface, are given by:

$$\begin{aligned} \cos\theta &= \vec{n} \cdot \vec{\omega} \\ \vec{\omega}_p &= \text{norm}(\vec{\omega} - \vec{n}(\vec{n} \cdot \vec{\omega})) \\ \cos\phi &= \vec{u} \cdot \vec{\omega}_p \\ \sin\phi &= \vec{v} \cdot \vec{\omega}_p, \end{aligned}$$

where $\vec{\omega}_p$ is the projection of $\vec{\omega}$ on the tangent plane, and $\text{norm}(\vec{v})$ is a function that normalizes $\vec{v}$, such that $||\vec{v}|| = 1$. To get the azimuth ϕ, the $\text{atan2}(y, x)$ function can be used.

A.4 Solid Angle

We use two-dimensional images to represent functions on the unit sphere, parameterized over spherical coordinates. For normalization purposes, it is important to know the solid angle each pixel in such a map represents. The value can be found by integrating over the pixel projected onto the unit sphere.

Consider a small patch on the sphere with spherical coordinates (θ, ϕ). The patch is located at a radius $r = \sin\theta$ from the center, and covers a solid angle of $dA_\Omega = \frac{d\phi}{2\pi} 2\pi r d\theta = \sin\theta \; d\phi d\theta$. We integrate over the range of spherical coordinates the pixel represents. Hence, in an image H with $n \times m$ pixels, a pixel with integer coordinate (i, j), with $0 \leq i < n$ and $0 \leq j < m$, has the following solid angle on the unit sphere:

$$\begin{aligned} A_\Omega(i,j) &= \int_{\frac{\pi j}{m}}^{\frac{\pi(j+1)}{m}} \int_{\frac{2\pi i}{n}}^{\frac{2\pi(i+1)}{n}} \sin\theta \; d\phi d\theta \\ &= \frac{2\pi}{n} \int_{\frac{\pi j}{m}}^{\frac{\pi(j+1)}{m}} \sin\theta \; d\theta \\ &= \left[-\frac{2\pi}{n} \cos\theta \right]_{\frac{\pi j}{m}}^{\frac{\pi(j+1)}{m}} \\ &= \frac{2\pi}{n} \left(\cos\left(\frac{\pi j}{m}\right) - \cos\left(\frac{\pi(j+1)}{m}\right) \right), \end{aligned} \tag{53}$$

where the unit is *steradians* [sr]. The full sphere covers 4π sr, which is easy to prove if we evaluate the above equation for $n = m = 1$. To normalize a spherical map for solid angle, we multiply each pixel with its normalized area on the sphere, $A_\Omega(i,j)/4\pi$, and divide by its area, $1/nm$, in the image:

$$\widetilde{H}(i,j) = \frac{A_\Omega(i,j)/4\pi}{1/nm} H(i,j) = \frac{m}{2} \left(\cos\left(\frac{\pi j}{m}\right) - \cos\left(\frac{\pi(j+1)}{m}\right) \right) H(i,j), \tag{54}$$

where $\widetilde{H}(i,j)$ is the normalized pixel.

B Optimized Wavelet Product

We first make a number of general observations about the properties of two-dimensional Haar basis functions and products of such. The following statements are easily verifiable:

1. The wavelet functions have vanishing integrals: $\iint \psi^m_{l,\mathbf{t}} d\mathbf{x} = 0$.
2. The integral over a scaling function at level l is equal to: $\iint \phi_{l,\mathbf{t}} d\mathbf{x} = 2^{-l}$.
3. The product of two basis functions (scaling or wavelet functions) is zero if they do not overlap.
4. The product of a scaling function and a wavelet function in the same wavelet square $(l, \mathbf{t})$, is the wavelet function scaled by 2^l.
5. The product of two different wavelet functions in $(l, \mathbf{t})$ is the third wavelet function in the same square, scaled by 2^l. For instance: $\psi^1_{l,\mathbf{t}} \psi^3_{l,\mathbf{t}} = 2^l \psi^2_{l,\mathbf{t}}$.
6. The product of two identical wavelet functions at level l is equal to the scaling function at the same level, scaled by 2^l, i.e., $\psi^m_{l,\mathbf{t}} \psi^m_{l,\mathbf{t}} = 2^l \phi_{l,\mathbf{t}}$.
7. The product of two overlapping basis functions at different levels $l < l'$, is the finer function scaled by $\pm 2^l$, with the sign determined by the sign of the coarser basis function over the support of the finer function.

B.1 Proof of Optimized Wavelet Product Theorem

In this section, we present the proof of the theorem presented in Section 2.5. We are considering the modified tripling coefficients:

$$C'_{ijk} = \iint \phi_{l,\mathbf{t}}(\mathbf{x}) \Psi_j(\mathbf{x}) \Psi_k(\mathbf{x}) d\mathbf{x}, \tag{55}$$

where $\phi_{l,\mathbf{t}}(\mathbf{x})$ is a scaling function at level l, and $\Psi_j(\mathbf{x})$ and $\Psi_k(\mathbf{x})$ are either the mother scaling function or wavelet functions at level l_j and l_k respectively. For clarity we divide the proof into three cases:

Case A: $l \leq l_j \leq l_k$

The scaling function exists at the same or a coarser level than the two wavelet functions. By applying observation 4 ($l = l_j$) or 7 ($l < l_j$), we arrive at $\phi_{l,\mathbf{t}}(\mathbf{x}) \Psi_j(\mathbf{x}) \Psi_k(\mathbf{x}) = 2^l \Psi_j(\mathbf{x}) \Psi_k(\mathbf{x})$. If the two wavelet functions are at different levels, or if they are at the same level but of different type, the integrand is further reduced to $\pm 2^l 2^{l_j} \Psi_k(\mathbf{x})$ by observation 7 and 5 respectively, and $C'_{ijk} = 0$ due to vanishing integrals. If the two wavelet functions are identical and exist in the square $(l', \mathbf{t}')$, then $2^l \Psi_j(\mathbf{x}) \Psi_k(\mathbf{x}) = 2^l 2^{l'} \phi_{l',\mathbf{t}'}$. In which case $C'_{ijk} = 2^l$.

Case B: $l_j < l \leq l_k$

One of the functions is at a strictly coarser level than the scaling function, and the other is at the same level or at a finer level. Observation 4 ($l = l_k$) or observation 7 ($l < l_k$) reduces

the integrand to $2^l \Psi_j(\mathbf{x})\Psi_k(\mathbf{x})$. Applying observation 7 results in $\pm 2^l 2^{l_j} \Psi_k(\mathbf{x})$ and $C'_{ijk} = 0$, again due to vanishing integrals.

Case C: $l_j \leq l_k < l$

Both functions are at a strictly coarser level than the scaling function. By applying observation 7 twice, we arrive at $\phi_{l,\mathbf{t}}(\mathbf{x})\Psi_j(\mathbf{x})\Psi_k(\mathbf{x}) = \pm 2^{l_j} 2^{l_k} \phi_{l,\mathbf{t}}(\mathbf{x})$. The integral over the wavelet square yields the modified tripling coefficient $C'_{ijk} = \pm 2^{l_j + l_k - l}$.

B.2 Derivation of Simplified Multiplication

We want to show that case 1 of the optimized tripling coefficient theorem presented in Section 2.5 on page 9, corresponds to a multiplication of the scaling coefficients of the two functions G and H at level l. Case 1 says that the tripling coefficients are $C'_{ijk} = \pm 2^{l_j + l_k - l}$, when both basis functions Ψ_j and Ψ_k are at strictly coarser levels and overlapping the scaling function.

We start by examining the expansion of a single scaling coefficient at level l. Here, we use $G^0_{l,\mathbf{t}}$, but the expression for $H^0_{l,\mathbf{t}}$ is of course the same. We have:

$$\begin{aligned}
G^0_{l,\mathbf{t}} &= \iint \phi_{l,\mathbf{t}}(\mathbf{x}) G(\mathbf{x}) d\mathbf{x} \\
&= \iint \phi_{l,\mathbf{t}}(\mathbf{x}) \left(\sum_j G_j \Psi_j(\mathbf{x}) \right) d\mathbf{x} \\
&= \sum_j G_j \left(\iint \phi_{l,\mathbf{t}}(\mathbf{x}) \Psi_j(\mathbf{x}) d\mathbf{x} \right) \\
&= \sum_j C_j G_j, \quad \text{with} \quad C_j = \iint \phi_{l,\mathbf{t}}(\mathbf{x}) \Psi_j(\mathbf{x}) d\mathbf{x}.
\end{aligned} \tag{56}$$

Here $C_j = \pm 2^{lj-l}$, if and only if $l_j < l$ and the two basis functions $\Psi_j(\mathbf{x})$ and $\phi_{l,\mathbf{t}}(\mathbf{x})$ are overlapping. In all other cases $C_j = 0$. We prove this by considering the two different cases: $l_j \geq l$ and $l_j < l$. In the first case, because of observation 4 and 7, C_j is reduced to the integral over a single wavelet function, which is zero due to vanishing integrals. In the second case, observation 7 gives: $C_j = \pm 2^{l_j} \iint \phi_{l,\mathbf{t}}(\mathbf{x}) d\mathbf{x} = \pm 2^{l_j - l}$. In summary, it is only the overlapping wavelets at strictly coarser levels that affect the value of the scaling coefficient.

Using this result, the product of the two scaling coefficients, scaled by 2^l, can be expanded as:

$$2^l G^0_{l,\mathbf{t}} H^0_{l,\mathbf{t}} = 2^l \left(\sum_j C_j G_j \right) \left(\sum_k C_k H_k \right) = 2^l \sum_j \sum_k C_j C_k G_j H_k. \tag{57}$$

This concludes the proof, as $2^l C_j C_k = \pm 2^{l_j + l_k - l}$ when Ψ_j and Ψ_k are overlapping $\phi_{l,\mathbf{t}}(\mathbf{x})$, and exist at strictly coarser levels. The sign is given by the product of the signs of the two basis functions in the wavelet square we are considering.

C Pseudo-code

C.1 Wavelet Product

We first give pseudo-code for our two helper functions, PSUM and CSUM, which were defined in Section 2.6. Then, we present code for the wavelet product and the optimized wavelet product for computing scaling coefficients.

In these algorithms, we often need to refer to the sign of a particular 2D Haar basis function over one of its four quadrants. Therefore, we introduce a function, $\text{sign}(m, q_x, q_y)$, that returns the sign of a wavelet function ψ^m of type $m = 1, 2, 3$ in the quadrant $(q_x, q_y) \in \{0, 1\}$. This function can either be hard-coded or implemented as a simple lookup-table.

C.1.1 Parent Sum (PSUM)

In this algorithm we use a hash map, psum_table, for caching the computed values. In the sampling algorithms, we access coefficients hierarchically, starting at the coarsest level and moving down to finer resolutions. Thus, the hash map will fill up with cached values as parent sums are requested, and each call will only access the previously cached values instead of doing a full recursion.

Algorithm 1 Parent Sum

function PSUM(**wavelet** F, **square** $\mathbf{s}$)
  if $\mathbf{s} \in$ psum_table **then**
    return psum_table($\mathbf{s}$)
  else
    if $\mathbf{s} = (0, 0, 0)$ **then**
      $value = F^0_{0,0}$
    else
      $\mathbf{s}' = (l', x', y') = (l - 1, \lfloor x/2 \rfloor, \lfloor y/2 \rfloor)$
      $(q_x, q_y) = (x - 2x', y - 2y')$
      $value = \text{PSUM}(F, \mathbf{s}') + 2^{l'} \sum_{m=1}^{3} F^m_{l',x',y'} \cdot \text{sign}(m, q_x, q_y)$
    psum_table->insert($\mathbf{s}, value$)
    return $value$

C.1.2 Children Sum (CSUM)

Before a new wavelet multiplication is initiated, we first call the function ComputeCsums. The function computes the sparse set of children sums for the product $G \cdot H$ of two wavelets, storing the non-zero values in a hash map. The computation is efficient, as we only need to loop over the sparse set of coefficients in one of the functions, multiplying each with the coefficient for the identical wavelet in the other function. If the result is non-zero, it is added to the children sum for the wavelet square $\mathbf{s}$ and to all squares overlapping at coarser levels.

Later, the children sums are accessed through a function csum($\mathbf{s}$) that simply returns the value stored in the hash map, or zero if there is no value associated with the square $\mathbf{s}$.

Algorithm 2 Compute Children Sums

function ComputeCsums(**wavelet** G, **wavelet** H)
  for $G^m_{l,x,y} \in G,\ G^m_{l,x,y} \neq 0$ **do**
    if $H^m_{l,x,y} \neq 0$ **then**
      $c = G^m_{l,x,y} \cdot H^m_{l,x,y}$
      $\mathbf{s} = (l, x, y)$
      while $l' \geq 0$ **do**
        csum_table($\mathbf{s}$) = csum_table($\mathbf{s}$) + c
        $\mathbf{s} = (l' - 1, \lfloor x'/2 \rfloor, \lfloor y'/2 \rfloor)$

C.1.3 Wavelet Product

The following function returns the wavelet coefficient of type m at a square $\mathbf{s}$, for the product of two wavelets, $F = G \cdot H$. We also need the first scaling coefficient for the product, which is given by the simple expression: $F^0_{0,0} = G^0_{0,0} H^0_{0,0} + \text{CSUM}(\mathbf{s})$.

Algorithm 3 Wavelet Product

function PRODUCT(**square** $\mathbf{s} = (l, x, y)$, **int** m)
  // All wavelets at the same square but of different type
  $m_1 = ((m + 1) \mod 4) + 1$
  $m_2 = ((m + 2) \mod 4) + 1$
  $c_1 = 2^l \left(G^{m_1}_{l,x,y} H^{m_2}_{l,x,y} + G^{m_2}_{l,x,y} H^{m_1}_{l,x,y} \right)$

  // The product of identical wavelets at strictly finer levels
  $c_2 = 2^l \ \text{sign}(m, 0, 0) \cdot \text{CSUM}(l{+}1,\ 2x,\ 2y)$
    $+\ 2^l \ \text{sign}(m, 1, 0) \cdot \text{CSUM}(l{+}1,\ 2x{+}1,\ 2y)$
    $+\ 2^l \ \text{sign}(m, 0, 1) \cdot \text{CSUM}(l{+}1,\ 2x,\ 2y{+}1)$
    $+\ 2^l \ \text{sign}(m, 1, 1) \cdot \text{CSUM}(l{+}1,\ 2x{+}1,\ 2y{+}1)$

  // One identical wavelet and the rest at coarser levels
  $c_3 = G^m_{l,x,y} \cdot \text{PSUM}(H, \mathbf{s}) \ + \ H^m_{l,x,y} \cdot \text{PSUM}(G, \mathbf{s})$

  // Return sum of the contributions
  return $c_1 + c_2 + c_3$

C.1.4 Optimized Product

This function directly returns the reconstructed average function value for a wavelet square $\mathbf{s}$ in the product of two wavelet represented functions, that is, the returned value is the scaling coefficient in $\mathbf{s}$ for the product, scaled by 2^l.

Algorithm 4 Optimized wavelet product

function PRODUCT(**square** $\mathbf{s} = (l, x, y)$)
// Both are scaling functions
$c_1 = \text{PSUM}(H, \mathbf{s}) \cdot \text{PSUM}(G, \mathbf{s})$

// The product of all identical wavelets at finer levels
$c_2 = 4^l \cdot \text{CSUM}(\mathbf{s})$

// Return sum of the contributions
return $c_1 + c_2$

C.2 Wavelet Sampling

In this section, we present three different algorithms for sampling a wavelet tree. During the sampling, we reconstruct the wavelet represented function hierarchically from its wavelet coefficients. To sample wavelet products, all accesses to wavelet coefficients should be replaced by calls to the function for computing wavelet product coefficients (Appendix C.1.3). To use the optimized wavelet product, we instead use the function given in Appendix C.1.4, which directly returns the reconstructed value for the given square in the wavelet product.

C.2.1 Random Sampling – Single Sample

This function generates a single 2D sample point by randomly sampling the wavelet representation.

Algorithm 5 Single Random Sample

```
//  s = (l, x, y) is the current wavelet square, initially s = (0, 0, 0).
//  F(s) is the average function value in s, initially F(s) = F^0_{0,0}.
//  r is a uniformly distributed random number, r ∈ [0, 1)
function SampleSingleRandom(square s, float F(s), float r)
    if l < maxlevel then
        // Reconstruct the four children quadrants
        F(s_1) = F(s) + 2^l(+F^1_{l,s} + F^2_{l,s} + F^3_{l,s}),   s_1 = (l + 1, 2x, 2y)
        F(s_2) = F(s) + 2^l(−F^1_{l,s} + F^2_{l,s} − F^3_{l,s}),   s_2 = (l + 1, 2x + 1, 2y)
        F(s_3) = F(s) + 2^l(+F^1_{l,s} − F^2_{l,s} − F^3_{l,s}),   s_3 = (l + 1, 2x, 2y + 1)
        F(s_4) = F(s) + 2^l(−F^1_{l,s} − F^2_{l,s} + F^3_{l,s}),   s_4 = (l + 1, 2x + 1, 2y + 1)

        // Compute conditional probabilities, p_i, and build histogram with four bins, h_i
        h_0 = 0
        for i = 1 ... 4 do
            p_i = 1/4 · F(s_i)/F(s)
            h_i = h_{i−1} + p_i

        // Pick quadrant s_i to sample
        for i = 1 ... 4 do
            if r < h_i then
                return SampleSingleRandom(s_i, F(s_i), r)
    else
        // Return random point within square
        return random(s)
```

C.2.2 Random Sampling – Multiple Samples

This algorithm generates multiple samples by randomly sampling the wavelet tree, returning an array (*samples*) of 2D sample points.

Algorithm 6 Multiple Random Samples

```
//  s = (l, x, y) is the current wavelet square, initially s = (0, 0, 0).
//  F(s) is the average function value in s, initially F(s) = F^0_{0,0}.
//  N(s) is the number of samples to place in s, initially N(s) = N.
function SampleMultipleRandom(square s, float F(s), integer N(s))
    if l < maxlevel then
        // Reconstruct the four children quadrants
        F(s_1) = F(s) + 2^l(+F^1_{l,s} + F^2_{l,s} + F^3_{l,s}),   s_1 = (l + 1, 2x, 2y)
        F(s_2) = F(s) + 2^l(−F^1_{l,s} + F^2_{l,s} − F^3_{l,s}),   s_2 = (l + 1, 2x + 1, 2y)
        F(s_3) = F(s) + 2^l(+F^1_{l,s} − F^2_{l,s} − F^3_{l,s}),   s_3 = (l + 1, 2x, 2y + 1)
        F(s_4) = F(s) + 2^l(−F^1_{l,s} − F^2_{l,s} + F^3_{l,s}),   s_4 = (l + 1, 2x + 1, 2y + 1)

        // Compute conditional probabilities, p_i, and build histogram with four bins, h_i
        h_0 = 0
        for i = 1 ... 4 do
            p_i = 1/4 · F(s_i)/F(s)
            h_i = h_{i−1} + p_i
            N(s_i) = 0

        // Place samples according to the histogram
        for i = 1 ... N(s) do
            r = random()
            for i = 1 ... 4 do
                if r < h_i then
                    N(s_i) = N(s_i) + 1; break

        // Recurse to children with one or more samples
        for i = 1 ... 4 do
            if N(s_i) ≥ 1 then
                SampleMultipleRandom(s_i, F(s_i), N(s_i))
    else
        // Place N(s) samples randomly within square
        for i = 1 ... N(s) do
            samples−> append(random(s))
```

C.2.3 Deterministic Sampling – Multiple Samples

Function for deterministically sampling a 2D wavelet representation, returning an array (*samples*) of 2D sample points.

Algorithm 7 Multiple Deterministic Samples

```
//  s = (l, x, y) is the current wavelet square, initially s = (0, 0, 0).
//  F(s) is the average function value in s, initially F(s) = F^0_{0,0}.
//  N(s) is the number of samples to place in s, initially N(s) = N.
function SampleMultipleDeterministic(square s, float F(s), integer N(s))
    if l < maxlevel then
        // Reconstruct the four children quadrants
        F(s_1) = F(s) + 2^l(+F^1_{l,s} + F^2_{l,s} + F^3_{l,s}),   s_1 = (l + 1, 2x, 2y)
        F(s_2) = F(s) + 2^l(−F^1_{l,s} + F^2_{l,s} − F^3_{l,s}),   s_2 = (l + 1, 2x + 1, 2y)
        F(s_3) = F(s) + 2^l(+F^1_{l,s} − F^2_{l,s} − F^3_{l,s}),   s_3 = (l + 1, 2x, 2y + 1)
        F(s_4) = F(s) + 2^l(−F^1_{l,s} − F^2_{l,s} + F^3_{l,s}),   s_4 = (l + 1, 2x + 1, 2y + 1)

        // Compute probabilities and allocate samples
        for i = 1 ... 4 do
            p_i = 1/4 · F(s_i)/F(s)
            N(s_i) = ⌊p_i N(s)⌋

        // Place the n remaining samples (if any)
        n = Σ N(s_i) − N(s)
        if n > 0 then
            // Update probabilities and build a histogram
            h_0 = 0
            for i = 1 ... 4 do
                p_i = frac(p_i N(s))/n
                h_i = h_{i−1} + p_i
            // Distribute samples according to the histogram
            for i = 1 ... n do
                r = random()
                for i = 1 ... 4 do
                    if r < h_i then
                        N(s_i) = N(s_i) + 1; break

        // Recurse to children with one or more samples
        for i = 1 ... 4 do
            if N(s_i) ≥ 1 then
                SampleMultipleDeterministic(s_i, F(s_i), N(s_i))
    else
        // Place N(s) samples randomly within square
        for i = 1 ... N(s) do
            samples−> append(random(s))
```

References

[1] Sameer Agarwal, Ravi Ramamoorthi, Serge Belongie, and Henrik Wann Jensen. Structured Importance Sampling of Environment Maps. *ACM Transactions on Graphics*, 22(3):605–612, 2003.

[2] David Burke, Abhijeet, and Wolfgang Heidrich. Bidirectional Importance Sampling for Illumination from Environment Maps. In *ACM SIGGRAPH Technical Sketches*, 2004.

[3] Brian Cabral, Marc Olano, and Philip Nemec. Reflection Space Image Based Rendering. In *Proceedings of ACM SIGGRAPH 99*, pages 165–170, 1999.

[4] Luc Claustres, Yannick Boucher, and Mathias Paulin. Wavelet Projection for Modelling of Acquired Spectral BRDF. In *Optical Engineering*, pages 1–41. SPIE, November 2004.

[5] Luc Claustres, Mathias Paulin, and Yannick Boucher. BRDF Measurement Modelling using Wavelets for Efficient Path Tracing. In *Computer Graphics Forum*, volume 22, pages 1–16. December 2003.

[6] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. In *Proceedings of ACM SIGGRAPH 84*, pages 137–145, 1984.

[7] Philip J. Davis and Philip Robinowitz. *Methods of Numerical Integration*. Academic Press, 2nd edition, 1984.

[8] Paul Debevec. Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-based Graphics with Global Illumination and High Dynamic Range Photography. In *Proceedings of ACM SIGGRAPH 98*, pages 189–198, 1998.

[9] Philip Dutré, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination*. AK Peters, 2003.

[10] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995.

[11] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods*. Methuen, 1964.

[12] V. Havran, T. Kopal, J. Bittner, and J. Žára. Fast Robust BSP Tree Traversal Algorithm for Ray Tracing. *Journal of Graphics Tools*, 2(4):15–23, 1997.

[13] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, November 2000.

[14] James T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, pages 143–150, 1986.

[15] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods, Volume 1: Basics*. Wiley-Interscience, New York, 1986.

[16] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Professional, 2nd edition, 1998.

[17] Thomas Kollig and Alexander Keller. Efficient Illumination by High Dynamic Range Images. In *Eurographics Rendering Symposium*, pages 45–50, 2003.

[18] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear Approximation of Reflectance Functions. In *Proceedings of ACM SIGGRAPH 97*, pages 117–126, 1997.

[19] Paul Lalonde. *Representations and Uses of Light Distribution Functions*. PhD thesis, University of British Columbia, 1997.

[20] J. H. Lambert. *Photometria sive de mensure de gratibus luminis, colorum umbrae*. Eberhard Klett, 1760.

[21] Jason Lawrence, Szymon Rusinkiewicz, and Ravi Ramamoorthi. Efficient BRDF Importance Sampling using a Factored Representation. *ACM Transactions on Graphics*, 23(3):496–505, 2004.

[22] Daniel D. Lee and H. Sebastian Seung. Algorithms for Non-negative Matrix Factorization. In *NIPS*, pages 556–562, 2000.

[23] S. G. Mallat. A Theory for Multiresolution Signal Decomposition: The Wavelet Representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

[24] Stéphane Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 1998.

[25] Wojciech Matusik. *A Data-Driven Reflectance Model*. PhD thesis, MIT, 2003.

[26] Wojciech Matusik, Hanspeter Pfister, Matt Brand, and Leonard McMillan. A Data-Driven Reflectance Model. *ACM Transactions on Graphics*, 22(3):759–769, 2003.

[27] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[28] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. Triple Product Wavelet Integrals for All-Frequency Relighting. *ACM Transactions on Graphics*, 23(3):477–487, 2004.

[29] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. *Geometric Considerations and Nomenclature for Reflectance*, volume NBS Monograph 160. National Bureau of Standards, Washington, D.C., 1977.

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

[31] Victor Ostromoukhov, Charles Donohue, and Pierre-Marc Jodoin. Fast Hierarchical Importance Sampling with Blue Noise Properties. *ACM Transactions on Graphics*, 23(3):488–495, 2004.

[32] Matt Pharr and Greg Humphreys. *Physically Based Rendering*. Morgan Kaufmann, 2004.

[33] Ravi Ramamoorthi and Pat Hanrahan. Frequency Space Environment Map Rendering. In *Proceedings of ACM SIGGRAPH 2002*, pages 517–526, 2002.

[34] Szymon M. Rusinkiewicz. A New Change of Variables for Efficient BRDF Representation. In *Eurographics Workshop on Rendering*, pages 11–22, June 1998.

[35] Peter S. Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana-Champaign, 1990.

[36] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann, 1996.

[37] W. Sweldens. The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions. In *Wavelet Applications in Signal and Image Processing III*, pages 68–79. Proceedings of SPIE, 1995.

[38] Eric Veach and Leonidas J. Guibas. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of ACM SIGGRAPH 95*, pages 419–428, 1995.

[39] Gregory J. Ward. Measuring and Modeling Anisotropic Reflection. In *Proceedings of ACM SIGGRAPH 92*, pages 265–272, 1992.

[40] Gregory J. Ward. *Graphics Gems IV*, chapter Real Pixels. Academic Press, 1994.