8-2009

Fast Rendering of Forest Ecosystems with Dynamic Global Illumination

Jay Steele
Clemson University, jesteel@cs.clemson.edu

Fast Rendering of Forest Ecosystems with Dynamic Global Illumination

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Jay Edward Steele
August 2009

Accepted by:
Robert M. Geist III, Committee Chair
James M. Westall, Jr.
Andrew T. Duchowski
Damon L. Woodard

Abstract

Real-time rendering of large-scale, forest ecosystems remains a challenging problem, in that important global illumination effects, such as leaf transparency and inter-object light scattering, are difficult to capture, given tight timing constraints and scenes that typically contain hundreds of millions of primitives. We propose a new lighting model, adapted from a model previously used to light convective clouds and other participating media, together with GPU ray tracing, in order to achieve these global illumination effects while maintaining near real-time performance. The lighting model is based on a lattice-Boltzmann method in which reflectance, transmittance, and absorption parameters are taken from measurements of real plants. The lighting model is solved as a preprocessing step, requires only seconds on a single GPU, and allows dynamic lighting changes at run-time. The ray tracing engine, which runs on one or multiple GPUs, combines multiple acceleration structures to achieve near real-time performance for large, complex scenes. Both the preprocessing step and the ray tracing engine make extensive use of NVIDIA's Compute Unified Device Architecture (CUDA).

Acknowledgments

I would like to thank all of those who have supported me, with friendship, knowledge, and guidance, during my time at the School of Computing at Clemson University. Most importantly, I would like to thank my supervisor, Robert M. Geist III. Over the past five years, he has greatly influenced my research, from theory to implementation to writing, while helping me secure funding and hardware. The initial lattice-Boltzmann lighting technique for participating medium [14], including proofs and implementation, are due, largely, to Dr. Geist. The work presented here, which consists of multiple extensions and improvements to the original technique, would not have been possible without his initial research (including his co-authors). Dr. Geist's advice and knowledge were invaluable to the work presented in this dissertation. My time spent working with Dr. Geist will influence my future for many, many years. I also thank my committee members, James M. Westall, Jr., Andrew T. Duchowski, and Damon L. Woodard, for taking time to be part of my dissertation and providing useful feedback. I would like thank my fellow students Zachary Jones, William Pressly, Thomas Grindinger, and Brandon Pelfrey, for helping me parse ideas, enduring countless images of trees, giving insight into many problems, and providing useful distractions outside of the research lab and classroom. Finally, I would like to thank my family and friends for supporting and positively influencing my academic endeavors. As of this writing, parts of this dissertation have been published at two conferences: the Proceedings of the 46^{th} Annual ACM SE Conference [11] and the IEEE/EG Symposium on Interactive Ray Tracing 2008 [13]. This work was supported by three NVIDIA Fellowships and NVIDIA hardware donations.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

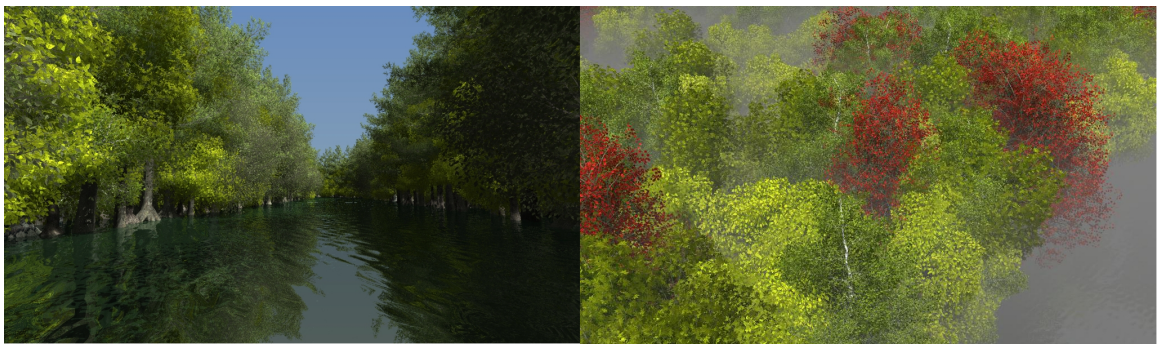


Figure 1.1: Virtual forest ecosystems examples, with water reflections.

Real-time rendering of large scale, high-density, plant ecosystems, such as those shown in Figure 1.1 and in Figure 1.2, is a topic of growing interest and wide application. There are two standard approaches to this task that continue to receive the attention of the research community. One is image-based and relies on conventional rasterization using billboard clouds [3], and the other is geometry-based and relies on ray tracing [6]. Ray tracing generally gives superior visual results, but until recently it has been too slow to provide high quality images at interactive frame rates for scenes that potentially require billions of polygons. Hardware improvements, in particular, improvements in CPU speed, have only partially ameliorated the problem because reduction in memory latency has not kept pace with the reduction in CPU clock cycle time. Instead, the application of large clusters of computing cores to an inherently parallel problem, together with careful management of the database that comprises the targeted plant ecosystem, has led to the emergence of ray tracing

as a competing technique. Rendering performance is closely tied to careful integration of the plant database with the spatial partitioning of the scene into acceleration structures that are used for hierarchical ray-surface intersection testing.

The goal of this effort is to introduce a new lighting model for forest ecosystems that includes important effects that are absent from other treatments, in particular, diffuse leaf transparency and inter-object light scattering, while maintaining at least near real-time rendering for scenes that comprise hundreds of millions of primitives. The fundamental approach is to adapt and apply a lattice-Boltzmann lighting model [12, 14], originally designed for lighting participating media, to large-scale forest ecosystems and then ray trace using CUDA [29] across multiple NVIDIA GPUs. The overall technique is similar, in spirit, to both precomputed radiance transfer [35] and photon mapping [21], in that a preprocessing step is used to compute and store lighting information within the scene itself. This preprocessing step is, comparatively, very fast. At run-time, dynamic lighting with both local and global illumination, based on the current sun position and intensity, are supported in real-time.

Subsequent sections will cover background, including the basic illumination model [12] and CUDA, related work, modifications to the model required to capture leaf transmittance and reflectance while allowing dynamic updates at run-time, the structure of the CUDA-based ray tracer, example results, including images and performance timings, and conclusions.

Figure 1.2: Two virtual forest ecosystems, composited with a real background and virtual clouds [14]. All beech trees in the top image are replaced with pine trees in the bottom image.

Chapter 2

Background

2.1 Lattice-Boltzmann Methods

Lattice-Boltzmann (LB) methods are computational alternatives to finite-element methods for solving coupled systems of partial differential equations. The LB methods have provided significant successes in modeling fluid flows and associated transport phenomena [4, 12, 34, 41]. The methods simulate transport by tracing the evolution of a single particle distribution through synchronous updates on a discrete grid. They provide stability, accuracy, and computational efficiency comparable to finite-element methods, but they realize significant advantages in ease of implementation, parallelization, and an ability to handle inter-facial dynamics and complex boundaries.

The principal drawback to the methods is the counter-intuitive direction of the derivation they require. Differential equations describing the macroscopic system behavior are derived from a postulated computational update, rather than the reverse. Thus a relatively simple computational method must first be justified by an intricate derivation.

Below, we demonstrate how LB methods provide a solution to a 1-dimensional diffusion equation. Diffusion modeling is of interest since others [20, 36] have shown that multiple photon scattering events lead to a diffusion process. It turns out that the fundamental structure of the LB derivation of a general diffusion equation is the same for any dimension. Nevertheless, the notation required for a full derivation in three dimensions is tedious to the point of obscuring that fundamental structure. Hence, we consider here a 1-dimensional derivation, which might be appropriate for, say, heat flow in a rod. Extension to three dimensions, though tedious, is then relatively straightforward

and described in the following section.

Assume we have a 1-dimensional lattice with spacing λ and that we will conduct synchronous updates with time step τ. Let $f_\pm(x,t)$ denote density of energy or matter at site x, time t, flowing in direction ± 1, and assume, perhaps due to site collisions, that at each time step a fraction σ of $f_\pm(x,t)$ continues in the current direction and the remainder reverses direction. The postulated fundamental update is thus

$$\begin{pmatrix} f_+(x+\lambda, t+\tau) \\ f_-(x-\lambda, t+\tau) \end{pmatrix} = \begin{pmatrix} \sigma & 1-\sigma \\ 1-\sigma & \sigma \end{pmatrix} \begin{pmatrix} f_+(x,t) \\ f_-(x,t) \end{pmatrix}$$

It will be most convenient to write this in incremental form,

$$\begin{pmatrix} f_+(x+\lambda, t+\tau) - f_+(x,t) \\ f_-(x-\lambda, t+\tau) - f_-(x,t) \end{pmatrix} = \Omega \begin{pmatrix} f_+(x,t) \\ f_-(x,t) \end{pmatrix} \tag{2.1}$$

where $\Omega = \begin{pmatrix} \sigma - 1 & 1-\sigma \\ 1-\sigma & \sigma - 1 \end{pmatrix}$.

Our real interest is the macroscopic behavior of total site density, $\rho(x,t) = f_+(x,t) + f_-(x,t)$, as lattice spacing and time step approach zero. We assume that τ approaches 0 faster than λ. Specifically, we write $\lambda = \epsilon\lambda_0$ and $\tau = \epsilon^2\tau_0$ for any small $\epsilon > 0$.

Our final assumption is that flow can be written as a small perturbation about a local equilibrium: $f_\pm = f_\pm^{(0)} + \epsilon f_\pm^{(1)} + \epsilon^2 f_\pm^{(2)} + ...$ where $f_+^{(0)} + f_-^{(0)} = \rho$, $f_+^{(i)} + f_-^{(i)} = 0, i > 0$, and ϵ is the mean free path between collisions (Knudsen number). This is the so-called Chapman-Enskog expansion from statistical mechanics [4].

If we now apply a (two variable) Taylor expansion to the left side of (2.1) we obtain:

$$\begin{pmatrix} \frac{\lambda\partial f_+}{\partial x} + \frac{\tau\partial f_+}{\partial t} + \frac{\lambda^2\partial^2 f_+}{2\partial x^2} + \frac{\lambda\tau\partial^2 f_+}{\partial x\partial t}... \\ \frac{-\lambda\partial f_-}{\partial x} + \frac{\tau\partial f_-}{\partial t} + \frac{\lambda^2\partial^2 f_-}{2\partial x^2} - \frac{\lambda\tau\partial^2 f_-}{\partial x\partial t}... \end{pmatrix} = \Omega \begin{pmatrix} f_+ \\ f_- \end{pmatrix} \tag{2.2}$$

The principal step in the derivation of the diffusion equation is now at hand: we substitute ϵ-based expressions for λ, τ, and $f_\pm$ into (2.2), and equate coefficients of like powers of ϵ. For

coefficients of the constant term (ϵ^0) we obtain:

$$0 = \Omega \begin{pmatrix} f_+^{(0)} \\ f_-^{(0)} \end{pmatrix} \tag{2.3}$$

Since Ω has eigenvalues 0 and $2\sigma - 2$ with eigenvectors $(1,1)$ and $(1,-1)$, we can conclude that $(f_+^{(0)}, f_-^{(0)}) = K(1,1)$, for some constant K. Further, since components sum to ρ, we must have

$$(f_+^{(0)}, f_-^{(0)}) = (\rho/2, \rho/2) \tag{2.4}$$

For coefficients of ϵ^1, we obtain:

$$\begin{pmatrix} \lambda_0 \frac{\partial f_+^{(0)}}{\partial x} \\ -\lambda_0 \frac{\partial f_-^{(0)}}{\partial x} \end{pmatrix} = \Omega \begin{pmatrix} f_+^{(1)} \\ f_-^{(1)} \end{pmatrix} \tag{2.5}$$

From (2.4)

$$\begin{pmatrix} (\lambda_0/2) \frac{\partial \rho}{\partial x} \\ -(\lambda_0/2) \frac{\partial \rho}{\partial x} \end{pmatrix} = \Omega \begin{pmatrix} f_+^{(1)} \\ f_-^{(1)} \end{pmatrix} \tag{2.6}$$

Although Ω cannot be inverted, the left side of (2.6) is a multiple of (1,-1), an eigenvector whose eigenvalue is $2\sigma - 2$. We conclude

$$(f_+^{(1)}, f_-^{(1)}) = (\frac{\lambda_0}{4\sigma - 4} \frac{\partial \rho}{\partial x}, \frac{-\lambda_0}{4\sigma - 4} \frac{\partial \rho}{\partial x}) \tag{2.7}$$

Finally, for the coefficients of ϵ^2 in (2.2) we obtain:

$$\begin{pmatrix} \tau_0 \frac{\partial f_+^{(0)}}{\partial t} + \lambda_0 \frac{\partial f_+^{(1)}}{\partial x} + (\lambda_0^2/2) \frac{\partial^2 f_+^{(0)}}{\partial x^2} \\ \tau_0 \frac{\partial f_-^{(0)}}{\partial t} - \lambda_0 \frac{\partial f_-^{(1)}}{\partial x} + (\lambda_0^2/2) \frac{\partial^2 f_-^{(0)}}{\partial x^2} \end{pmatrix} = \Omega \begin{pmatrix} f_+^{(2)} \\ f_-^{(2)} \end{pmatrix} \tag{2.8}$$

Here we can rely on the fact that the column sums of Ω are zero. If we substitute expressions for $f_\pm^{(0)}$ (2.4) and $f_\pm^{(1)}$ (2.7) into (2.8) and sum, we arrive at

$$\frac{\partial \rho}{\partial t} - \left(\frac{\lambda_0^2}{\tau_0}\right) \left(\frac{\sigma}{2 - 2\sigma}\right) \frac{\partial^2 \rho}{\partial x^2} = 0 \tag{2.9}$$

or, equivalently,

$$\frac{\partial \rho}{\partial t} = \left(\frac{\lambda^2}{\tau}\right)\left(\frac{\sigma}{2-2\sigma}\right)\frac{\partial^2 \rho}{\partial x^2} \tag{2.10}$$

the 1-D diffusion equation, with diffusion coefficient

$$D = \left(\frac{\lambda^2}{\tau}\right)\left(\frac{\sigma}{2-2\sigma}\right)$$

Thus the desired flow behavior (2.10) can be effected by implementing a nearly trivial computational update (2.1).

2.2 Lattice-Boltzmann Lighting

Our target application requires illuminating 3-dimensional forest ecosystems in real-time. The technique of [12] describes a lighting technique that we will adapt and enhance. This technique proposed a new solution to the standard volume radiative transfer equation for modeling light in a participating medium:

$$(\vec{\omega} \cdot \nabla + \sigma_t)\, L(\vec{x}, \vec{\omega}) = \sigma_s \int p(\vec{\omega}, \vec{\omega'}) L(\vec{x}, \vec{\omega'}) d\vec{\omega'} + Q(\vec{x}, \vec{\omega}) \tag{2.11}$$

where L denotes radiance, $\vec{\omega}$ is spherical direction, $p(\vec{\omega}, \vec{\omega'})$ is the phase function, σ_s is the scattering coefficient of the medium, σ_a is the absorption coefficient of the medium, $\sigma_t = \sigma_s + \sigma_a$, and $Q(\vec{x}, \vec{\omega})$ is the emissive field in the volume [1]. The solution, which is applicable to simulating photon transport through participating media such as clouds, smoke, or haze, was based on an LB method. A complication of LB methods in three dimensions is that isotropic flow requires that all neighboring lattice points of any site be equidistant. A standard approach, due to d'Humières, Lallemand, and Frisch [5], is to use 24 points equidistant from the origin in 4D space and project onto 3D. The points are:

$$\begin{array}{lll} (\pm 1, 0, 0, \pm 1) & (0, \pm 1, \pm 1, 0) & (0, \pm 1, 0, \pm 1) \\ (\pm 1, 0, \pm 1, 0) & (0, 0, \pm 1, \pm 1) & (\pm 1, \pm 1, 0, 0) \end{array}$$

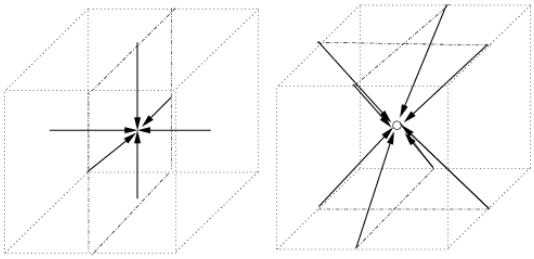


Figure 2.1: The 18 lattice directions: 6 axial directions (left) and 8 of the 12 non-axial directions (right) [12].

and projection is truncation of the fourth component, which yields 18 directions, $\vec{c_m}, m \in 1, 2, ..., 18$ (Figure 2.1). Axial directions ($\vec{c_m}, m \in 1, 2, ..., 6$) then receive double weights. Representation of several phenomena, including energy absorption and energy transmission, is facilitated by adding a direction, $\vec{c_0}$, from each lattice point back to itself, which thus yields 19 directions, the non-corner lattice points of a cube of unit radius. The key quantity of interest is the per-site photon density, $f_m(\vec{r}, t)$, which is the density arriving at lattice site $\vec{r} \in \Re^3$ at time t in cube direction $\vec{c}_m$, $m \in \{0, 1, ..., 18\}$.

The update in 3D is simply the expected analog of (2.1):

$$f_m(\vec{r} + \lambda\vec{c}_m, t + \tau) - f_m(\vec{r}, t) = \Omega_m \cdot f(\vec{r}, t) \tag{2.12}$$

where Ω_m denotes row m of a 19×19 matrix, Ω, that describes scattering, absorption, and (potentially) wavelength shift at each site. If $\rho(\vec{r}, t) = \sum_m f_m(\vec{r}, t)$ denotes total site density, then a derivation in [12] shows that the limiting case of (2.12) as $\lambda, \tau \to 0$ is the diffusion equation

$$\frac{\partial \rho}{\partial t} = D\nabla^2_{\vec{r}}\rho \tag{2.13}$$

where the diffusion coefficient

$$D = \left(\frac{\lambda^2}{\tau}\right)\left[\frac{(2/\sigma_t) - 1}{4(1 + \sigma_a)}\right] \tag{2.14}$$

This is consistent with previous approaches to modeling multiple photon scattering events [20, 36], which invariably lead to diffusion processes.

For any LB method, the choice of Ω is not unique. Standard constraints are conservation of mass, $\sum_m(\Omega_m \cdot f) = 0$, and conservation of momentum, $\sum_m(\Omega_m \cdot f)\vec{v_m} = \tau\vec{F}$, where $\vec{v_m} = (\lambda/\tau)\vec{c_m}$

and $\vec{F}$ represents any site external force. In [12, 14], for the case of isotropic scattering, Ω was chosen as follows:

For row 0:

$$\Omega_{0j} = \begin{cases} -1 & j = 0 \\ \sigma_a & j > 0 \end{cases} \tag{2.15}$$

For the axial rows, $i = 1, ..., 6$:

$$\Omega_{ij} = \begin{cases} 1/12 & j = 0 \\ \sigma_s/12 & j > 0, \quad j \neq i \\ -\sigma_t + \sigma_s/12, & j = i \end{cases} \tag{2.16}$$

For the non-axial rows, $i = 7, ..., 18$:

$$\Omega_{ij} = \begin{cases} 1/24 & j = 0 \\ \sigma_s/24 & j > 0, \quad j \neq i \\ -\sigma_t + \sigma_s/24, & j = i \end{cases} \tag{2.17}$$

Entry i, j controls scattering from direction $\vec{c_j}$ into direction $\vec{c_i}$, and directional density f_0 holds the absorption/emission component. On update, i.e., $\Omega \cdot f$, fraction σ_a from each directional density will be moved into f_0. The entries of Ω are then multiplied by the density of the medium at each lattice site, so that a zero density yields a pass-through in (2.12), and a density of 1 yields a full scattering.

Isotropic scattering is technically incorrect for photon transport through clouds, but modifications to Ω can easily generate a more accurate, anisotropic photon transport, or, clearly, models of other types of flows. As noted in [12], anisotropic scattering is incorporated by multiplying σ_s that appears in entry $\Omega_{i,j}$ by a normalized phase function:

$$pn_{i,j}(g) = \frac{p_{i,j}(g)}{\left(\sum_{i=1}^{6} 2p_{i,j}(g) + \sum_{i=7}^{18} p_{i,j}(g)\right)/24} \tag{2.18}$$

where $p_{i,j}(g)$ is a discrete version of the Henyey-Greenstein phase function [16] (Figure 2.2),

$$p_{i,j}(g) = \frac{1 - g^2}{(1 - 2g\vec{n_i} \cdot \vec{n_j} + g^2)^{3/2}} \tag{2.19}$$

Here $\vec{n_i}$ is the normalized direction, $\vec{c_i}$. Parameter $g \in [-1, 1]$ controls scattering direction. Value

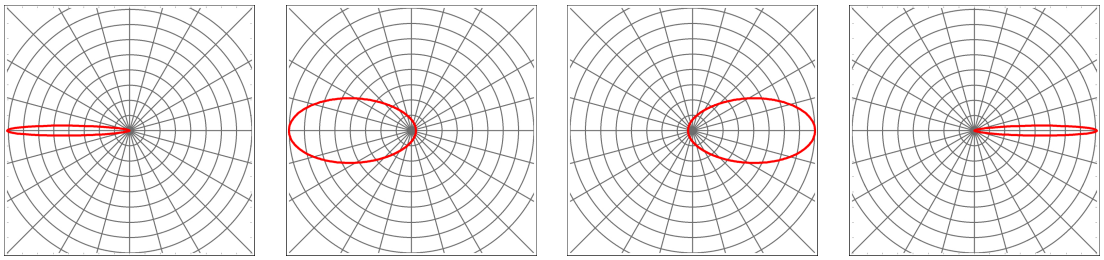


Figure 2.2: Henyey-Greenstein phase function (2.19) for back-scattering (two left images with $g = -0.9$ and $g = -0.5$) and forward-scattering (two right images with $g = 0.5$, and $g = 0.9$). Note that the plots are not to scale.

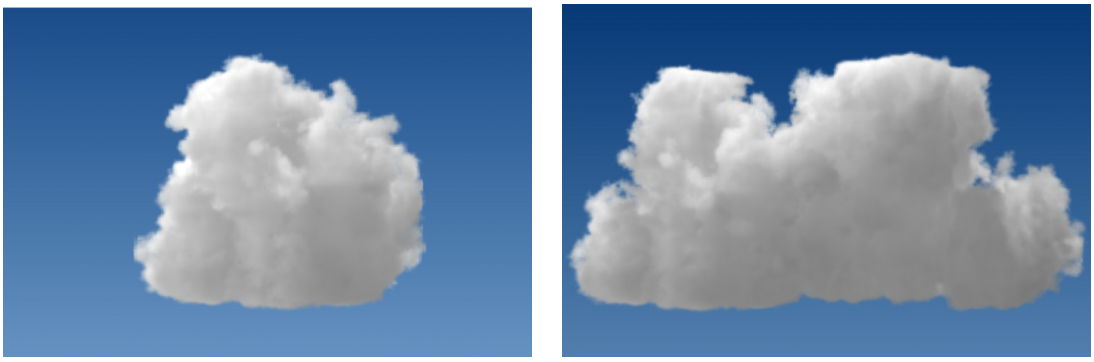


Figure 2.3: Clouds lighted with LB lighting.

$g > 0$ provides forward scattering, $g < 0$ provides backward scattering, and $g = 0$ yields isotropic. Mie scattering [9] is generally considered preferable, but the significant approximations induced here by a relatively coarse grid render the additional complexity unwarranted. Note that (2.18) differs from the original treatment in [12]. Setting $\sigma_a = 0$ and $g = 1$ now yields an effect that is identical to a pass-through.

This model, and variations thereon, were used to represent photon transport through clouds, as seen in Figure 2.3. In this case one can start with a model of vapor density per lattice site, scale the σ entries of Ω by the vapor density, and then apply the basic update (2.12). With some significant modifications, this lighting model can be used to capture leaf transparency and inter-object light scattering for forest ecosystems, which will be demonstrated in Chapter 4.

2.3 Haar Wavelets

The general introduction to Haar wavelets presented in this section follows the excellent introduction found in Chapter 2 of *Wavelets for Computer Graphics* [37]. Wavelets allow the hierarchical decomposition of functions, thus expressing functions in multiresolution form. We will consider the simplest wavelet, known as the Haar wavelet. As detailed in [37], multiresolution analysis considers functions defined over a nested set of vector spaces

$$V^0 \subset V^1 \subset V^2 \subset ...$$

where the resolution of functions in V^j doubles for each increment of j. We define *wavelet spaces*, W^j, such that W^j is the orthogonal complement of V^j in V^{j+1}. Thus, W^j and V^j form a basis for V^{j+1}. The basis functions for W^j are called *wavelets*. Given the set of scaled and translated box functions as the basis functions for V^j,

$$\phi_i^j(x) = \phi(2^j x - i), i = 0, ..., 2^j - 1$$

where

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

yields the corresponding wavelets, known as the Haar wavelets, defined as

$$\psi_i^j(x) = \psi(2^j x - i), i = 0, ..., 2^j - 1$$

where

$$\psi(x) = \begin{cases} 1 & for\ 0 \ \leq \ 0 \ < \ 1/2 \\ -1 & for\ 1/2 \ \leq \ 0 \ < \ 1 \\ 0 & otherwise. \end{cases}$$

In practice, the Haar wavelet transform for a given a sequence of 2^n samples,

$$x_0^n, x_1^n, ..., x_{2^n-1}^n$$

proceeds as follows. First, an average filter (pairwise average) is applied to the the original sequence

of samples to produce a new sequence of 2^{n-1} average coefficients

$$x_i^{n-1} = \frac{x_{2i}^n + x_{2i+1}^n}{2} \tag{2.20}$$

This new sequence is a lower resolution approximation of the original sequence. In order to rebuild the original sequence from the lower resolution approximation, a second filter, known as the detail filter, is applied to the original sequence to produce 2^{n-1} detail coefficients

$$y_i^{n-1} = \frac{x_{2i}^n - x_{2i+1}^n}{2} \tag{2.21}$$

Both filters are applied recursively to the average coefficients of the previous step until the resulting sequence consists of one average coefficient along with $2^n - 1$ detail coefficients. Applying a wavelet transform to a sequence is known as *decomposition*. Note that the decomposition of the original sequence has resulted in neither information loss nor compression. The original sequence has been transformed into a new basis. Computing the original sequence, known as *reconstruction*, is performed by inverting the decomposition steps.

As an example, consider the sequence

$$[4 \quad 19 \quad 44 \quad 0]$$

The first step in decomposition results in two average coefficients and two detail coefficients. The two average coefficients are computed by applying the average filter (2.20), and the two detail coefficients are computed by applying the detail filter (2.21). Applying both filters to the original sequence results in

$$[11.5 \quad 22 \quad -7.5 \quad 22]$$

Note that the original sequence can be easily restored (exactly) by combining the average coefficients with the detail coefficients. Both filters are applied once more, but only to the first two samples (the average coefficients), producing the following sequence:

$$[16.75 \quad -5.25 \quad -7.5 \quad 22]$$

Decomposition of the original sequence into the its wavelet transform is now complete as there is

one average coefficient ([16.75]) and three detail coefficients ([−5.25 −7.5 22]).

Wavelet compression becomes a lossy compression scheme when insignificant (small) detail coefficients in the wavelet stream are replaced by zeroes. The resulting errors in the reconstruction of the original data stream, for many applications, such as image compression, are small and entirely acceptable. Deconstructing smooth sequences, such as images, often results in many small, detail coefficients. Compression requires marking the location of these small, detail coefficients and removing (*decimating*) them from the data stream, thus reducing the size of the data. Before reconstruction, the decompression step replaces decimated detail coefficients with zeroes, and reconstruction is performed as normal. As shown in [37], decimating detail coefficients whose absolute value is less than a user defined threshold τ results in the smallest error when reconstructing the original sequence. Varying τ allows a user to compromise between compression rates and error.

In summary, wavelet compression is a two step process. First, a data stream is transformed into its wavelet representation. Second, small, detail coefficients are decimated from the stream, thus reducing the amount of data to be stored. Decompression is the inverse of compression. First, zeroes are used to represent the decimated detail coefficients in the data stream. Then, reconstruction is performed, producing an approximation to the original data stream.

2.4 CUDA

Graphics Processing Unit (GPU) was a term introduced by NVIDIA in the late 1990s when "VGA controller" was deemed an inadequate description of the graphics hardware found in the typical desktop PC [7]. Since that time, GPU designs have undergone a series of dramatic transformations that have collectively produced the spectacular graphics capability that is now available for the PC at very low cost. In 2001, user programmability of part of the graphics pipeline first appeared in NVIDIA's GeForce3 and ATI's Radeon 8500. This programmability was restricted to a particular sub-engine of the GPU called the vertex processor, but in 2002, programmability was extended to the sub-engine called the fragment processor, which gave programmers per-pixel control. Researchers soon realized that these highly-parallel processors could be effectively employed in solving non-graphics problems, and a community of developers interested in general-purpose computation on graphics processing units (GPGPU) quickly emerged (`http://www.gpgpu.org`). An excellent survey of the GPGPU state of the art as of 2007 can be found in [31].

The principal motivation for this rapid development of the GPGPU community is simple: floating point performance. NVIDIA's latest GPU series, the GeForce 200, runs internally at a maximum clock rate of 1.476 GHz and thus has a theoretical peak of 1.062 TFLOPS (1.476 GHz $\times$ 3 inst/cycle $\times$ 240 cores), which dwarfs the 48.0 GFLOPS peak of a high-end Intel CPU, the 3.00 GHz quad core Xeon. The GPU targets data-parallel computations of high arithmetic intensity and thus, compared to the CPU, has more transistors devoted to data processing and fewer to caching and flow control. The goal of the design was to hide memory access latency with computation rather than large data caches [29].

Starting with the GeForce 8 series, NVIDIA took a significant departure from previous GPUs in that vector-based, task-specific vertex processors and fragment processors were replaced by multipurpose, *scalar processors* (SP). The number of SPs per GPU ranges from 8 in the low-power, mobile series to 240 in the high-performance consumer and professional series. Eight SPs are combined to form one *multiprocessor* (MP). Each MP shares 16 KB of fast, on-chip memory, known as *shared memory*, and up to 16,384 registers among its SPs. Shared memory can be accessed at register speed.

With the release of the GeForce 8 Series, NVIDIA realized that GPGPU developers needed better access to the hardware than that available through the graphics API (OpenGL or DirectX). Mastery of the graphics API was most often an annoying, time-consuming obstacle to those interested in non-graphics computation. Further, since it focused on fragment (pixel) updates, the graphics API provided extremely limited memory write capability. Each thread could gather (read) from arbitrary card memory locations, but each could write, essentially, to just one. Scatter-writes were unavailable, which constrained algorithm design.

The Compute Unified Device Architecture (CUDA) was NVIDIA's response. CUDA combines specific hardware, including the GeForce 8 series and the recently introduced Tesla S1070, together with drivers, libraries, and C language extensions, in order to provide access to the GPU hardware without the constraints imposed by traditional GPU programming techniques, which typically rely on a graphics API. Unlike previous GPU languages, CUDA is not geared towards graphics. For example, CUDA capable Tesla series devices do not contain display interfaces. They are designed for server room environments and considered to be co-processors for highly parallel, compute intensive operations. (Nevertheless, throughout this document, we still refer to CUDA-capable hardware as GPUs.) Code is organized around *kernels*, which are functions that are invoked from the

CPU (the host) but execute on the GPU (the device). The MPs of the GPU execute these kernels most efficiently in SIMD mode, but standard C control flow is available. In addition to invoking the device kernels, the host is responsible for managing device memory, which is segmented into multiple memory spaces of varying capabilities. The largest memory space on the device, global memory, can be up to 4GB per GPU, and it has an impressive bandwidth of up to 141.7 GB/sec. But, global memory is off-chip, and global memory transactions are hindered by relatively large latencies, typically 500 cycles. Management of the memory hierarchy strongly impacts performance.

Kernels are invoked simultaneously on many (typically thousands) threads. Abstractly, the programmer must assume that all threads for a kernel are executed in parallel. Internally, CUDA schedules the execution of threads to maximize utilization of the SPs. This per kernel scheduling cannot be controlled by the programmer and is not deterministic. But, the programmer does maintain control over the organization of threads, known as a kernel's *execution configuration*. Threads are organized into *blocks*. Blocks are further organized as 2-dimensional *grids*. Threads within a block share device resources (shared memory and registers), can communicate, and can synchronize; conversely, threads can not deterministically communicate with threads in other blocks. All threads in a block are loaded onto the same MP at the same time. Once scheduled on an MP, a block is never removed from that MP until all of its threads have completed. Threads within a block can communicate through the MP's shared memory, which, as the name implies, is shared by all threads in a block. Within a block, threads are further arranged into groups of size 32 called *warps*. Threads within a warp execute in SIMD fashion on an MP. Threads within a warp can diverge (that is, take different execution paths), but, divergence within a warp may extract a significant performance penalty. Issuing one instruction for a warp takes at least 4 cycles, as 32 threads execute in parallel on the 8 SPs of an MP.

As NVIDIA introduces new hardware, CUDA gains new features. A device's *compute capability* designates the feature set supported by the hardware. Currently, there are 4 compute capabilities: 1.0, 1.1, 1.2, and 1.3. Our implementation targets devices with compute capability 1.2 or greater, such as the GeForce GTX 280 and the Tesla S1070, though our algorithms are applicable to cards with compute capabilities 1.0 and 1.1 with slight modification. When applicable, we note any alterations for an algorithm required to perform on lower compute capability hardware.

CUDA devices contain either one, two, or four GPUs. For example, the Tesla S1070 is composed of four GPUs contained in an external 1U rack. Each GPU is independent, and the

programmer must explicitly access each GPU. To work in parallel, each GPU requires its own thread of execution residing on its own CPU core. Syncing data among GPUs, which requires moving data from one device to the host and then to the other device(s)m is tasked to the programmer.

2.5 Optimization Strategies for CUDA

The general optimization strategies described in this section summarize those found in Chapter 5 of the *NVIDIA CUDA Programming Guide* [29]. We ignore algorithmic improvements such as maximizing parallelism in computation, and instead focus on optimizing CUDA kernels by maximizing instruction throughput and maximizing memory throughput. Kernels are either *compute-bound* or *memory-bound*. A kernel is compute-bound if the time it takes to complete is determined by the speed at which non-memory (compute) instructions execute. By contrast, a kernel is memory-bound if the time it takes to complete is determined by the speed at which memory instructions execute. Memory bound kernels leave the device idle waiting for memory transactions to complete. Optimizing instruction throughput is important for compute-bound kernels, while, optimizing memory throughput is important for memory-bound kernels.

Improving the performance of compute-bound kernels requires reducing the number of instructions executed per warp. For each warp, an MP requires 4 cycles for common mathematical operations, such as floating point add, multiply, and multiply-add. Other operations, such as 32-bit integer multiplication, require 16 clock cycles. Certain operations, such as integer division, are even more costly, though CUDA contains many intrinsic functions, such as $\_\_mul24$ for 24-bit integer multiplication in 4 cycles, that compute many slow operations faster while reducing either range or accuracy.

When threads in a warp diverge, i.e. follow separate execution paths, significant performance penalties can ensue, as all resulting execution paths must be serialized. The serialization of execution paths increases the total number of instructions executed for the warp. Many algorithms can intelligently avoid thread divergence, as thread ids within a warp are deterministic. The first thread's id is a multiple of 32, and all thread ids increase sequentially. Kernels can avoid divergence within warps by defining control flow conditions based on thread ids. Overall, compute-bound kernels should attempt to avoid extraneous instructions, as the effects of extra instructions are amplified by the sheer number of threads executing.

Optimizing memory-bound kernels is a complex topic, and a vast number of strategies are available. Devices contain six separate memory spaces, each possessing unique performance characteristics. Input and output arrays typically reside in the global memory space, as it is the largest memory space. Global memory supports random access reads and writes from the host and the device. The bandwidth to global memory is very high, but the latency is also very high, up to 500 clock cycles. This high latency often results in MPs sitting idle, waiting for memory transactions to complete. Unlike other memory spaces, global memory space has no cache.

Minimizing the effect of latency is key when optimizing memory-bound kernels. First, the device attempts to hide memory latency through computation. Internally, the device will schedule execution of warps to minimize idle time while waiting for global memory transactions to complete. While warps do stay attached to a single MP until completion, an MP is free to switch from one warp to another in order to minimize idle time. Though the programmer has no direct control over this scheduling, the programmer should attempt to maximize the *occupancy* of a kernel. Increasing occupancy increases the amount of computational work available per MP, allowing the device to schedule work in an attempt to hide global memory latencies. Occupancy is defined as the ratio of the number of threads concurrently residing on an MP to the maximum number of threads allowed to reside on an MP. For devices of compute capability 1.2 and greater, the maximum number of concurrent threads per MP is 1024, and the maximum number of concurrent blocks per MP is 8. The actual number of concurrent threads per MP is a function of a kernel's resource usage (registers and shared memory) and execution configuration. For example, if each thread requires 64 registers, then the maximum number of threads that can reside concurrently on an MP is 256 ($\frac{16348}{64}$). The occupancy is 0.25 ($\frac{256}{1024}$). Increasing occupancy does not guarantee improved performance, but, as a general guideline, efforts should be made to maximize occupancy.

A simple strategy for reducing latency involves reducing the number of global memory transactions. Shared memory is much faster to access than global memory. When properly utilized, accessing shared memory is as fast as accessing a register (4 cycles per warp). Shared memory is limited in size (16KB per MP), and it cannot be directly accessed from the host. A general strategy, useful when threads within a block access the same data in global memory, is to instruct each thread to load part of the data from global memory into an array in shared memory. Threads then synchronize, ensuring data is loaded into the shared memory array before computing. After synchronization, all threads within the block may access the shared memory array to perform computations. Once

complete, each thread writes its results to shared memory and then synchronizes. Finally, after all threads have synchronized, each thread writes part of the results to global memory.

Global memory bandwidth is quite large, but care must be taken to ensure that the simultaneous memory transactions by threads in each *half-warp* (16 threads) can be coalesced into a small number of memory transactions, preferably one. Minimizing the number of memory transactions reduces memory latency and maximizes memory bandwidth. Devices are capable of reading and writing 32, 64, or 128 bytes with one memory transaction, assuming alignment requirements are satisfied. Coalescing requirements differ based on the compute capability of the device. Devices with compute capabilities 1.0 and 1.1 have stricter coalescing requirements than device with compute capabilities 1.2 and 1.3.

For compute capabilities 1.0 and 1.1, threads within a half-warp must access 32-bit, 64-bit, or 128-bit words. Accessing 32-bit and 64-bit words results in one 64-byte (32-bits×16) or one 128-byte (64-bits×16) memory transaction. Accessing 128-bit words results in two 128-byte memory transactions. All words must lie in the same sequential memory segment. This memory segment must be aligned to either 4, 8, or 16 bytes (equal to the size of each individual memory transaction). The size of the memory segment must be equal to the size of the total memory transaction, unless the threads are accessing 128-bit words, in which case the size of the memory segment must be double the memory transaction size. Finally, all threads must access the words sequentially. (Thread 0 accesses word 0, thread 1 accesses word 1, and so on.) If not all of the above conditions are met, the device issues a separate memory transaction for each thread, which can be a huge performance penalty.

For compute capabilities 1.2 and 1.3, the coalescing requirements are not as strict. Coalescing is achieved if threads in a half-warp access words that lie in the same memory segment, following the same alignment requirements as above. For 8-bit words, the memory segment must be 32 bytes in size. For 16-bit words, the memory segment must be 64 bytes in size. For 32-bit or 64-bit words, the memory segment must be 128 bytes in size. Threads need not access words in sequence. If threads access words in different memory segments, then n memory transactions will be issued for the n memory segments.

Many algorithms cannot meet the above coalescing requirements when accessing arrays in global memory. It may be beneficial to move this data into another memory space of the device. *Constant memory space*, which is limited to 64KB, is writable by the host and read-only on the

device. Each MP contains a small constant cache (8KB) that is optimized for all threads in a warp accessing the same address in constant memory. If this requirement is met, then accessing constant cache is as fast as accessing a register. Large arrays, or those with varying access patterns, may benefit from *texture memory space*. Texture memory space, which is limited by the size of the global memory space, is writable from the host and read-only on the device. Each MP contains a small texture cache (8KB) that is optimized for spatial locality. Textures can be 1-dimensional, 2-dimensional, or 3-dimensional. Textures have other advantages, such as hardware filtering.

Finally, accessing shared memory is equivalent to accessing a register provided there are no *bank conflicts*. Shared memory is arranged as 16 banks of size 32-bits where 32-bit words are assigned sequentially to banks. Once two or more threads within a half-warp simultaneously access the same shared memory bank, a bank conflict occurs and those memory transactions must be serialized. Avoiding bank conflicts, by having each thread in a half-warp access a unique bank per instruction, is essential to maximizing shared memory accesses.

2.6 OpenCL

Open Computing Language (OpenCL) is a new framework for developing highly parallel applications while remaining, as much as possible, platform agnostic. OpenCL initially targets CPUs (specifically multi-core variants) and GPUs. OpenCL is very similar in spirit to CUDA for data-parallel applications. Unlike CUDA, OpenCL is managed by the Kronos Group, not NVIDIA, and, in theory, is a more open standard; though OpenCL implementations, such as NVIDIA's, are proprietary. As of this writing, OpenCL is still under considerable development and was not used.

Chapter 3

Related Work

3.1 Lighting Plants

The overall technique presented in this study is similar, in spirit, to both precomputed radiance transfer of Sloan et al [35] and photon mapping of Jensen [21], in that a preprocessing step is used to compute and store lighting information within the scene itself. Comparatively, the LB lighting preprocessing step is relatively very fast.

Wang et al [40] achieved beautiful results in rendering small collections of plant leaves using carefully constructed bidirectional reflectance and transmittance functions that were based on measurements of real plants. Their method is computationally intensive, with large memory requirements, and so as yet unsuitable for real-time rendering of large-scale, high-density ecosystems.

Reeves et al [33] represented trees as a particle system. A probabilistic shading model shaded each particle based on the particle's position and orientation. Hegeman et al [15] ignored physical accuracy in a technique that attempted to achieve visual plausibility and fast computation through approximating ambient occlusion. Trees are approximated by bounding spheres containing randomly distributed small occluders. Fragments are shaded based on the average probability of the fragment being occluded, which is based on its position within the bounding sphere. Though simple to compute, this method, as mentioned by the authors, only considers local information and results can differ widely from more physically accurate approaches. Luft et al [25] were able to capture ambient occlusion in rendering foliage through the addition of a preprocessing step in which overall tree geometry was simplified to an implicit surface, i.e., a density field, using Marching Cubes [24].

The ambient coefficient in a standard, local illumination model was then modified by a transfer function that was exponentially decreasing in the field density. They also realigned leaf normal vectors to match the implicit surface in order to reduce lighting noise.

3.2 Ray Tracing of Large Ecosystems

Dietrich et al [6] combined the OpenRT real-time ray tracing engine [38], Xfrog plant models [30], geometry instancing, and adaptive transparency control to achieve interactive rendering of large, high-density plant ecosystems. The adaptive transparency control was in response to the structure of the Xfrog models, wherein leaves are represented by coarse triangles within which leaf shape is determined by an alpha channel. Ray-triangle intersections may then simply generate an additional forward ray, rather than a reflected value. Their test scene contained more than 365,000 plants, of 68 distinct species, with a total of approximately 1.5 billion triangles. With 32 CPUs they achieved 6 fps on a 640×480 scene. They did not attempt to account for global illumination effects such as leaf transparency and inter-object light scattering.

3.3 Ray Tracing on GPUs

Ray tracing is a computationally intensive task, as each ray must be intersected with a scene that may be composed of hundreds of millions of triangles. Over the past three decades, multiple acceleration structures have been developed to reduce the computational cost of each ray-scene intersection test. Generally, these data structures sort objects and are accompanied by a traversal algorithm that, for each ray, intelligently tests for ray-object intersections. Adaptive data structures, such as kd-tress, adapt to the underlying objects in the scene. Uniform structures, such as grids, maintain a uniform structure regardless of the underlying geometry.

In general, kd-trees have achieved the best results for CPU-based ray tracing. Naturally, this has led to attempts to port kd-tree algorithms to GPUs. Foley et al [8] introduced an important, stackless kd-tree traversal algorithm called *kd-restart* which allowed GPU-based ray tracing to use kd-tree acceleration structures. This early work was significantly hampered by tight GPU instruction limits imposed by early architectures. Horn et al [17] took advantage of improved hardware, in the form of an ATI X1900 XTX, and introduced a new algorithm, *short-stack*, to achieve better results.

Short-stack uses a stack of bounded size during traversal and falls back to the stackless algorithm on underflow. They used 4-wide ray packets per fragment, and they were able to achieve 15 - 18 million primary rays per second on test scenes. More recently, Zhou et al [43] implemented a traditional stack-based kd-tree traversal algorithm in CUDA, using a per-thread allocated array in local memory for the stack. Their results show that GPU ray tracers are competitive with multi-core, CPU ray tracers. A sample scene with 47K triangles, 800×600 pixels, shadows and reflections, rendered at 22 fps on an NVIDIA 8 series card.

Initial research into uniform grids was conducted by Fujimoto et al [10] in their ARTS system. This paper introduced the 3DDDA (3D digital differential analyzer) algorithm for traversing a uniform grid. Because of the adaptive nature of kd-trees and new kd-tree traversal techniques, such as traversing packets of neighboring rays in parallel on SIMD hardware, traversing a kd-tree has been traditionally faster than traversing a uniform grid. But, as discussed by Wald et al [39], constructing uniform grids is generally faster than constructing kd-trees. Construction time is important for real-time rendering of animated scenes, since, at each frame, acceleration structures may need to be recomputed. The total rendering time per frame is the sum of construction time and traversal time. Using ray packets to traverse uniform grids, as detailed in [39], results in traversal times that are equivalent to the best traversal times for kd-trees.

The work of Hunt and Mark [18] enhanced the effectiveness of uniform grids through introducing perspective grids. Today's best traversal algorithms rely on coherency among neighboring rays. As neighboring rays traverse adaptive acceleration structures in packets, such as kd-trees, the rays are very likely to remain coherent at the top nodes because of the adaptive nature of the structure. All nodes in a uniform grid are the same size, which implies that neighboring rays will diverge quickly. This is apparent, since the splitting planes of a uniform grid will not be parallel to the majority of ray directions. Taking into consideration the ray directions when constructing an acceleration structure allows the construction of a much more efficient data structure. A perspective grid is built in perspective space. Thus, the splitting planes can be made parallel to the majority of ray directions, assuming rays emit from or travel to one point, such as primary rays or hard shadow rays. A perspective grid is built for each camera and light in the scene. Before traversal, rays are projected into the grid's perspective space. Coherency is excellent, as neighboring ray directions are very likely to be parallel to the splitting planes. Also, building an acceleration structure per camera and per light allows intelligent culling of triangles. All back-facing triangles are culled for

the camera perspective grids. All front-facing triangles are culled for the light perspective grids. Although the algorithms above target the CPU, porting uniform grids and perspective grids to the GPU is straightforward. Lagea et al [23] describe a well known method, which they call the compact grid, for intelligently representing a grid in GPU memory by compacting a grid into two arrays. The use of uniform grids on GPUs was first introduced by the work of Purcell et al [32].

3.4 Volumetric Compression

Muraki [26] introduced the idea of using wavelets to provide compressed approximations for volumetric data. Varying levels of approximations were computed through ignoring the most insignificant detail coefficients. Westermann [42] details an approach for approximating the volume rendering integral using wavelets. Volumetric data is first projected into a wavelet basis, insignificant coefficients are truncated, significant coefficients are quantized, and a bit-mask is created that marks the location of truncated and quantized coefficients. The main insight of this paper relative to our current work is the comparison between the Haar wavelet basis and the Daubechies wavelet basis. The author concluded that the Daubechies wavelet basis provides higher compression rates and less noise, but the Haar wavelet, because of its simplicity, results in faster reconstruction times. Neither [26] nor [42] deal with fast, random access to the compressed data.

Ihm and Park [19] tackle the challenge of interactively visualizing large volumetric data sets. Volumetric data sets can easily surpass the available memory, and users typically view pieces of the data at a time. With a goal of compromising between high compression rates and fast, random access, the authors suggest compressing sub-blocks of volumetric data individually. Specifically, the Haar wavelet basis is used to compress sub-blocks of size 4^3, which are then truncated and quantized. Encoding is based on a bit-mask to signify null coefficients (coefficients set to zero in the truncation stage) and non-null coefficients. The Haar wavelet basis was chosen for fast reconstruction. Bajaj et al [2] extend the work of [19] by adding support for volumetric, RGB data and improving on the encoding scheme. The improved encoding scheme relies on spatial coherency. Null coefficients tend to group in space. The 64-bit bit-mask is replaced by a two-level bit-mask. Many of the 2^3 sub-blocks will contain only null coefficients, which is encoded in the top-level bit-mask, thus avoiding a lower-level bit-mask and saving space.

Nguyen and Suape [28] also propose a wavelet-based technique for compressing volumetric

data. Once again, the goals of this technique are to find a proper balance between high compression rates and fast, random access. Similar to other methods, this technique works on sub-blocks of size 16^3. This technique differs from previous techniques in that the Daubechies wavelet basis is used instead of the Haar wavelet, and a new encoding scheme is introduced. The authors show that the compression ratio using the Daubechies wavelet basis is superior to the compression ratio using the Haar wavelet basis. But, no verdict is given on decompression and reconstruction time.

Chapter 4

Lighting Forest Ecosystems

Our lighting model captures global illumination effects that occur in complex forest scenes. These effects include scattering, transmission, and absorption due to the optical properties of leaves. The lighting in our target scenes consists of one infinite point light, the sun, and ambient light due to the reflection and scattering of the sky and ground. Individual plants are rotated and translated instances of shared plant models. Thus, a typical scene may contain hundreds of individual plants, but only ten or so unique plant models. Instancing is necessary to cope with the memory requirements of such large scenes. For example, the American Beech model from the Xfrog plant models [30] contains 496,719 triangles, each of which requires at least an 18-float specification. As such, a small forest of individually represented plants would easily consume all memory.

First, we describe our lighting model as applied to a single plant instance with a static sun. We then extend this fine-grain illumination model to allow for dynamic updates to the sun position and sun intensity at run-time. Next, we introduce a coarse-grain illumination model that accounts for global, scene effects, such as plant-plant occlusion, that the fine-grain illumination model does not account for by itself. We then discuss acceleration structures useful for tracing both primary rays and shadow rays through large forest scenes. Finally, we detail a compression technique that allows our lighting model to be viable for real-time simulations involving hundreds of instanced plants on today's hardware.

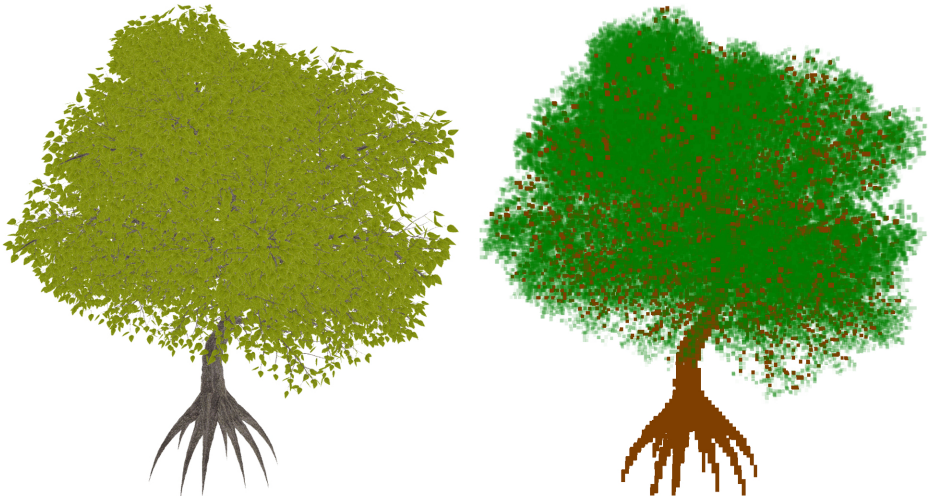


Figure 4.1: Polygonal tree (left) and corresponding volumetric representation (128^3), both unlit.

4.1 Lighting Model

The overall illumination model begins with a standard, local illumination model that captures direct lighting effects in the usual way. Local diffuse lighting is of the form $k_d(\vec{n}\cdot\vec{l})$, where $\vec{l}$ is the direction of the sun, $\vec{n}$ is leaf/wood normal, and k_d is the combined sun color and sampled leaf/wood texture color. (We assume that the sun, and thus $\vec{l}$, has been transformed into the instance's model space.) Local specular lighting is of the form $k_s(\vec{v}\cdot\vec{r})^s$, where $\vec{v}$ is the viewer position vector, $\vec{r}$ is the sun reflection vector, s is the specular exponent, and k_s is the combined sun color and leaf/wood specular color. Indirect global illumination effects are captured by an LB lighting solution, as described below.

First, each plant model is converted to a volumetric representation of size equal to the chosen lighting resolution. Each such plant model grid, here of size 64^3 or 128^3 nodes, has a per-node density (biomass) factor estimated from local leaf count and leaf area within its associated cell. If a significant portion of the biomass is wood, rather than leaf, the density is classified as "brown" rather than "green", so that scattering may be restricted to backward only. The scattering properties of "green" nodes, discussed below, are wavelength dependent and anisotropic. Figure 4.1 provides a non-photorealistic volumetric representation of a typical tree, unlit.

Unlike the case of lighting atmospheric clouds [12], where absorption is extremely small ($\sigma_a < 0.01$), plants absorb a significant fraction of the visible light reaching them, and this energy

Figure 4.2: A tree rendered using only 1 component (intensity) LB light with the sun straight overhead. The left image was rendered with backward scattering ($g = -0.75$), the center image rendered with isotropic scattering ($g = 0.0$), and the right image was rendered with forward scattering (0.75).

is not re-radiated within the visible spectrum. Further, absorption, reflection, and transmission are heavily wavelength-dependent. It is natural to conjecture that these components are also heavily species-dependent, but surprisingly, this is not the case. Knapp and Carter [22] measured leaf optical properties, in particular, reflectance, transmittance, and absorptance of 26 species of plants from widely varying habitats. They concluded that the lack of variability across species was remarkable, given the broad habitat range and unusual anatomical characteristics of several of the species included in the study. Thus a single set of wavelength-dependent model parameters suffice in determining σ_s and σ_a.

Scattering is, of course, anisotropic and wavelength-dependent. As noted earlier, anisotropic scattering can be incorporated by multiplying σ_s that appears in entry $\Omega_{i,j}$ by the normalized phase function:

$$pn_{i,j}(g) = \frac{p_{i,j}(g)}{\left(\sum_{i=1}^{6} 2p_{i,j}(g) + \sum_{i=7}^{18} p_{i,j}(g)\right)/24} \tag{4.1}$$

where $p_{i,j}(g)$ is a discrete version of the Henyey-Greenstein phase function [16],

$$p_{i,j}(g) = \frac{1 - g^2}{(1 - 2g\vec{n_i} \cdot \vec{n_j} + g^2)^{3/2}} \tag{4.2}$$

Again, $\vec{n_i}$ is the normalized direction, $\vec{c_i}$. Parameter $g \in [-1, 1]$ controls scattering direction. Value $g > 0$ provides forward scattering, $g < 0$ provides backward scattering, and $g = 0$ yields isotropic, as illustrated in Figure 4.2

Wavelength-dependence is limited here to three color components. The model does not attempt to account for total leaf absorption as expressed in [22], since this represents almost all incident light energy. (The minimum is 72%, which occurs at 550 nm.) Instead, the absorptance values from [22] are scaled by a single, experimentally determined factor (here 0.125) to yield per-component model absorption coefficients, σ_a^X, for $X = R, G, B$. The per-component model scattering coefficients are then given by $\sigma_s^X = 1 - \sigma_a^X$, again for $X = R, G, B$. Per-component transmittance and reflectance ratios from [22] are used to determine forward and backward scattering components, fs^X and bs^X, by the constraint $fs^X + bs^X = \sigma_S^X$. Finally, values of the phase function parameter, g, are chosen as:

$$g^X = \frac{fs^X - bs^X}{fs^X + bs^X} \quad \text{for } X = R, G, B \tag{4.3}$$

Thus identical transmittance and reflectance values for color component X would yield $fs^X = bs^X$, and scattering would be isotropic ($g^X = 0$). If a node is classified as "brown," rather than "green," $g^X = -1$ for $X = R, G, B$.

The directional photon densities of the boundary nodes of the plant model lattice must be set to constant values that are based on the sun direction ($\vec{l}$) and sun intensity. The initial treatment of LB lighting [12] computed these boundary conditions through a procedure similar to Gram-Schmidt orthogonalization. A lattice direction was selected from the yet selected lattice directions as that direction having the largest dot product with the remaining light direction. This contribution was subtracted from the remaining light direction, and this process was repeated until no light energy was left. Experimentation has shown that this method is not suitable for animated scenes; as the sun traverses the sky, the directional photon densities on the boundaries vary significantly from frame to frame. This results in undesirable popping of illumination from frame to frame.

Our new technique provides smooth transitions as the sun moves through the sky. First, as in [12], all directional photon densities on boundaries are set proportional to the ambient light in the scene. We then determine the lattice directions (and corresponding weights) that best align with $\vec{l}$. As there is no *a priori* method for finding these lattice directions for arbitrary values of $\vec{l}$, we now describe a geometric approach that provides quality results.

Consider the convex polyhedron of Figure 4.3, which has 32 faces (triangles) formed by adjacent triples of unit length directions, $\vec{c}_i/\|\vec{c}_i\|, i = 1, ..., 18$. A ray from the origin in the sun direction will intersect one triangle. If the vertices of that triangle have associated directions $\vec{c}_{i_0}$, $\vec{c}_{i_1}$,

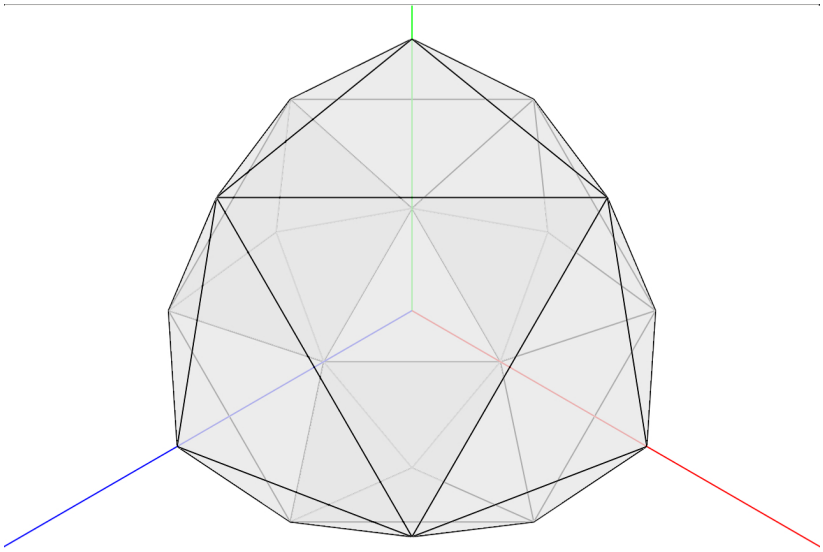


Figure 4.3: Polyhedron formed from unit lattice directions.

and $\vec{c}_{i_2}$, then, on all boundaries, we increase the directional photon densities corresponding to the directions $\vec{c}_{i_0}$, $\vec{c}_{i_1}$, and $\vec{c}_{i_2}$ with weights determined by the barycentric coordinates of the intersection point. The weights sum to one, thus conserving total energy.

Visually, this method provides smooth transitions with no visible popping as the sun moves through the sky. Another benefit, which will be described further Section 4.2, is that light from the sun is decomposed into at most three lattice directions.

With these constant boundary conditions, computing a local LB lighting solution for a plant model amounts to iterating (2.12) to steady-state, which yields a total photon density per RGB component per lattice direction at each grid node. The significant loss of light energy from the visible spectrum due to plant absorption is modeled by zeroing out component f_0 at each "green" node on each iteration. The directional photon densities are summed to create a final photon density per component at each grid node. Figure 4.4 provides a visualization of the final LB lighting data for a plant through 128 slices along the z-dimension.

For each visible fragment (produced through ray tracing or rasterization), illumination due to LB lighting is computed by sampling the volumetric output of the lighting model as a 3D texture. We use linear interpolation to sample from the LB lighting solution. As shown in Figure 4.5,

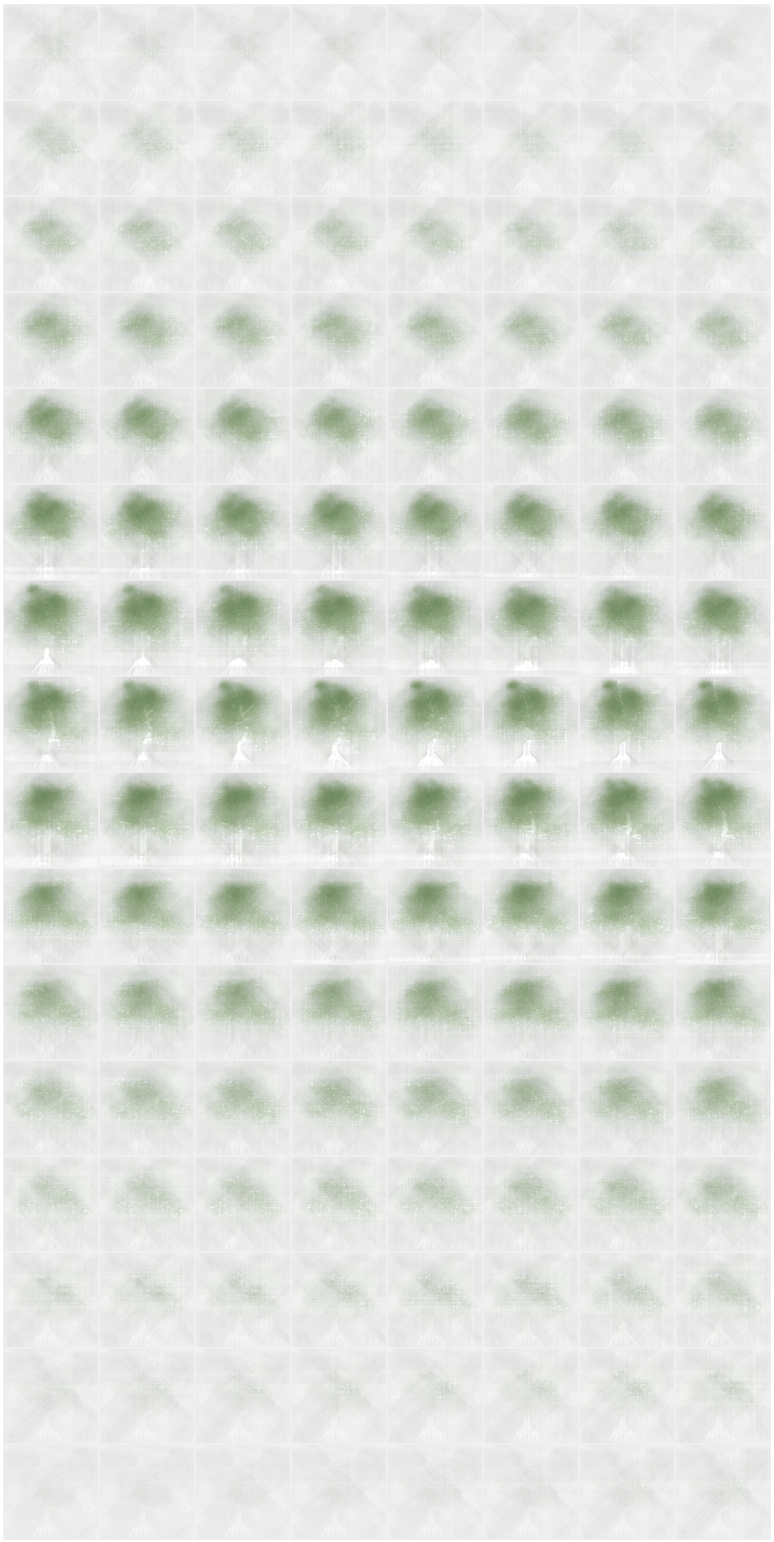


Figure 4.4: Visualization of the final LB lighting data (128^3) for a plant through 128 slices along the z-dimension.

sampling the volume with linear interpolation results in higher quality images than nearest-neighbor sampling. (Both images were rendered without shadows to highlight the discontinuities present in nearest-neighbor sampling.) With nearest-neighbor sampling (top image), discontinuities in the indirect illumination are visible across surfaces due to the discrete volumetric representation. Linear interpolation removes these discontinuities while preserving the fidelity of the LB lighting.

To clarify implementation details (Chapter 5) it is important at this stage to specify the interpolation technique. First, we project world space coordinates of the fragment p into lattice space. As detailed in [29], linear interpolating from a volume texture, T, can be computed by

$$\begin{aligned}
&(1-\alpha)(1-\beta)(1-\gamma)T[i,j,k] + \alpha(1-\beta)(1-\gamma)T[i+1,j,k] \\
&+(1-\alpha)\beta(1-\gamma)T[i,j+1,k] + \alpha\beta(1-\gamma)T[i+1,j+1,k] \\
&+(1-\alpha)(1-\beta)\gamma T[i,j,k+1] + \alpha(1-\beta)\gamma T[i+1,j,k+1] \\
&+(1-\alpha)\beta\gamma T[i,j+1,k+1] + \alpha\beta\gamma T[i+1,j+1,k+1]
\end{aligned} \tag{4.4}$$

where

$$\begin{aligned}
i &= floor(x_B),\ \alpha = frac(x_B),\ x_B = max(0, p_x - 0.5) \\
j &= floor(y_B),\ \beta = frac(y_B),\ y_B = max(0, p_y - 0.5) \\
k &= floor(z_B),\ \gamma = frac(z_B),\ z_B = max(0, p_z - 0.5).
\end{aligned} \tag{4.5}$$

From the above specifications, it is clear that sampling from the LB lighting data requires the texel located at i, j, k (the texel containing p) and the seven forward neighbors of that texel. The interpolated LB solution is modulated by texture color and added to the local, direct illumination to compute the final color of the fragment.

Complex scenes contain multiple plant instances that reference a much smaller set of base models. Instances that share the same base model may differ in orientation to the sun. Thus, the boundary conditions for each lattice instance may differ. As described above, our lighting model would need to compute a new solution per instance. Also, new solutions per instance would be required as the sun traverses the sky. Though fast, our lighting algorithm is not fast enough to compute multiple solution instances for each frame while maintaining real-time speed. Further, our lighting model currently only considers local, intra-plant effects; but, plants occlude other plants in complex scenes. We continue with extensions to our lighting model to handle these shortcomings.

Figure 4.5: Nearest neighbor sampling (top image) and linear interpolation sampling (bottom image) with no shadows.

4.2 Dynamic Lighting

An LB lighting solution computed with the technique in Section 4.1 is only valid for one orientation of the plant instance relative to the sun. As the sun traverses the sky, new solutions must be computed. Even with lower resolution grids, such as 64^3, computing a new LB solution per instance per frame is computationally too expensive for real-time rendering. Further, as noted earlier, forest scenes typically comprise hundreds of plant instances, wherein each instance carries a unique translation and rotation of some base model. Plant instances that reference the same plant model, but have different orientations to the sun direction, require individual solutions. Computing and storing a solution per plant instance quickly exhausts the available memory of today's hardware. We now describe an extension to the lighting model that allows both translated and rotated instances as well as dynamic lighting changes to complex scenes in real-time. The approach is conceptually simple. A set of base lighting solutions is precomputed per plant model. These base lighting solutions are then combined, at run-time, based on the instance's orientation and the current sun direction. This technique allows dynamic, lighting updates to complex scenes containing hundreds of plants in real-time.

First, for each plant model (not instance) we precompute and store 19 base lighting solutions $\{B_j | j = 0, 1, ..., 18\}$ by using (2.12) with boundary conditions based on direction index j. For solution B_j with $j > 0$, all boundary nodes have fixed densities $f_i(\vec{r}, t) = \delta_{ij}$ (Kronecker delta), all i. For solution B_0, the ambient solution, all boundary nodes have $f_i(\vec{r}, t) = 1$, all i. Computing each base solution follows the static LB lighting algorithm in Section 4.1.

At run-time, a shader combines multiple base solutions to compute the LB lighting for each fragment. Before invoking our lighting shader on the visible fragments, we determine which base solutions (and corresponding weights) to use, based on the current sun direction. First, for each instance, the sun is transformed into the instance's model space. Next, using the technique described in Section 4.1 for setting boundary conditions when computing a static LB lighting solution, we intersect a ray, placed at the instance's model space origin in the direction of the transformed sun, with the convex polyhedron of Figure 4.3. As before, the ray will intersect only 1 triangle. If the vertices of that triangle have associated directions $\vec{c}_{i_0}$, $\vec{c}_{i_1}$, and $\vec{c}_{i_2}$, then we use base solutions B_{i_0}, B_{i_1}, and B_{i_2} and the barycentric coordinates of the intersection as weights. For each plant instance, we store the three base solutions, along with the weights.

For each fragment, we sample the three base solutions for that fragment's instance and linearly combine them using the associated weights. We then add ambient light by sampling from the ambient base solution B_0, which has scaling controllable by the user. Thus, at run-time, our dynamic technique requires at most 32 ray-triangle intersections per instance and four 3D texture lookups per fragment.

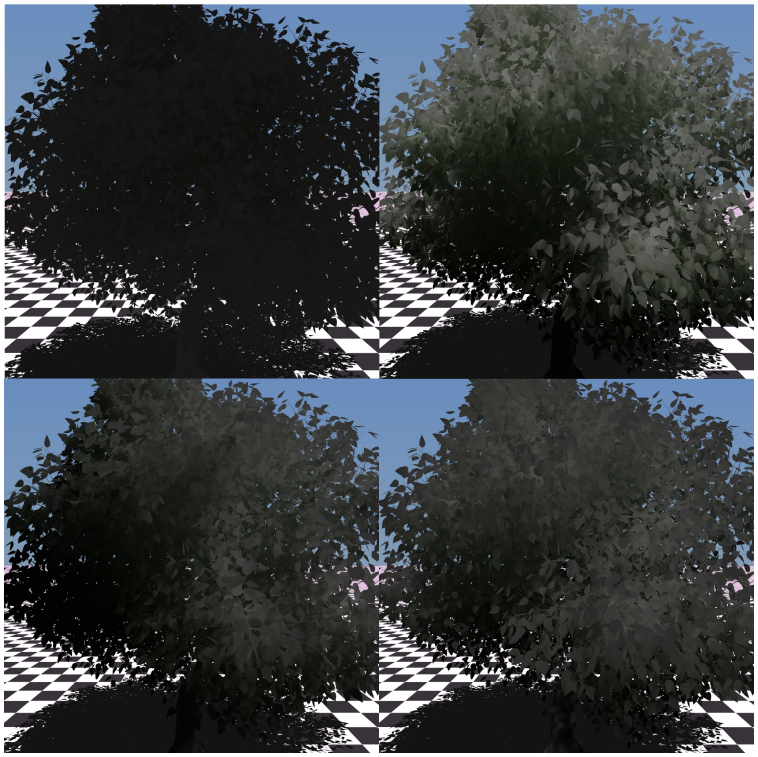
Figure 4.6 illustrates how four base solutions are combined to compute the final LB lighting solution. The top left tree in the top image shows the ambient base LB lighting solution, scaled based on a user defined value. The three other trees in the top image show the three base LB lighting solutions that were chosen based on the current sun position. These four base solutions, which are precomputed, are combined at run-time to produce the bottom image, which is a visualization of the final LB lighting solution for the tree, based on the current sun position.

4.3 Hierarchical Lighting Model

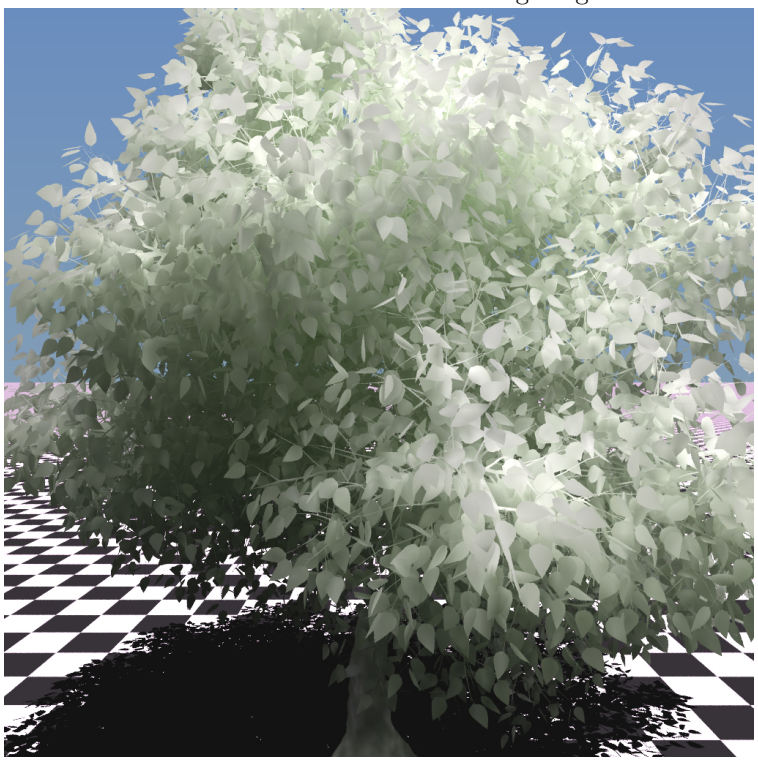
Our LB lighting method, as described to this point, computes indirect illumination effects under the assumption that global illumination of the plant is not occluded by other plants. (Occlusion of direct illumination can be handled by any shadowing algorithm, such as shadow rays or shadow maps.) Obviously, complex scenes contain multiple plants that occlude each other (inter-object occlusion), thus invalidating our lighting model's assumption that only self-occlusion exists. Without proper indirect occlusion, too much light energy may be set at the boundary nodes, resulting in too much indirect illumination for occluded plants. We have developed a simple extension that accounts for such inter-object (plant-plant) occlusions.

We combine our fine-grain, local LB lighting solution with a coarse-grain, global LB lighting solution. A coarse-grain, global LB lighting grid is imposed upon an instanced forest system. Each node in the coarse-grain grid has a density factor estimated from the tree instances that intersect the node. Solution of this coarse-grain grid over the entire forest, using iterations of (2.12), simply provides scale factors that weight the indirect illumination of a fragment computed from the fine-grain, local LB lighting solution.

The effect of applying this hierarchical method is shown in Figure 4.7. In this scene, the sun is located towards the upper right of the image. The two smaller tree instances both reference the same tree model and, thus, the same precomputed LB lighting solution, although each accounts



Ambient and three directional base LB lighting solutions.



Combined LB lighting solution.

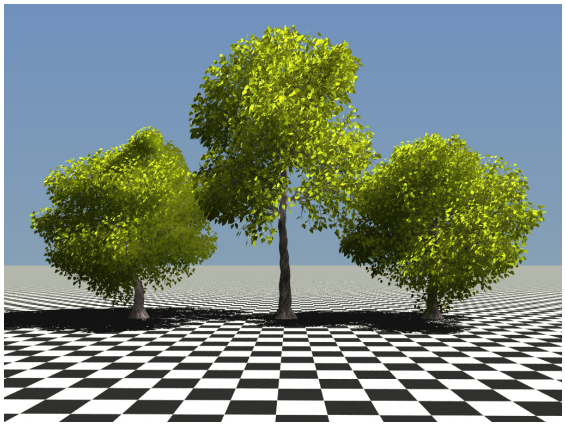
Figure 4.6: Combining base LB lighting solutions to produce the final LB lighting solution.

for instance translation and rotation. Image (a) combines direct illumination with local LB lighting, while image (b) visualizes the local LB lighting only. By contrast, image (c) combines direct illumination, local LB lighting, and scene level LB lighting, and image (d) visualizes the local LB lighting scaled by the scene level LB lighting. Notice that in comparing (a) and (c), the tree on the left is darker in (c) than the same tree in (a). Image (c) is more accurate, as the tree on the left is being occluded by the middle tree. Image (e) visualizes the scene level LB lighting only, while image (f) was rendered with only direct lighting for comparison. As with the fine-grain, local LB solution, a coarse-grain, global LB solution can be precomputed for a scene. The same technique described in Section 4.2 can be applied to the coarse-grain, global LB lighting solution to allow dynamic updates in real-time.

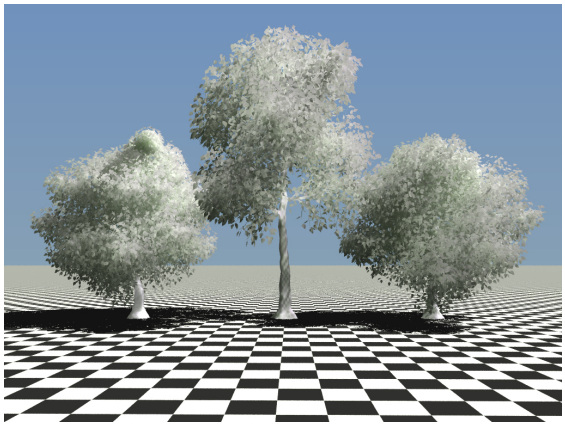
4.4 Ray tracing

Our LB lighting model is not tied to ray tracing or rasterization. Visible fragments, generated through ray tracing or rasterization, are shaded accordingly. Individual plant models contain hundreds of thousands of primitives (the models we use require two triangles per leaf), and forest scenes can contain hundreds of tree instances. Thus, we prefer ray tracing for visible surface computation. It provides superior image quality, and, as scene complexity increases, ray tracing, which is $O(log\ n)$ in scene complexity, will provide better performance than rasterization.

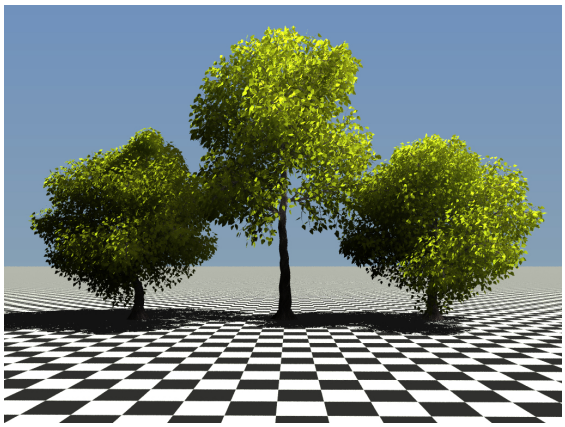
The structure of the ray tracing engine follows the general techniques described by Dietrich et al [6], though we make use of different acceleration structures. Tree models are instanced multiple times to compose larger scenes. Similar to [18], a high level, perspective grid is constructed per frame that contains each instance's world space, axis aligned bounding box (AABB). The purpose of this high level grid is to accelerate the intersection of primary rays with instances. Unlike the work of [18], the perspective grid is only 2-dimensional, since only primary rays will traverse this grid, and is built from AABBs, not geometry. First, the image plane is divided into a rectangular, 2-dimensional grid, in which blocks of pixels map to grid cells. Each grid cell contains a hit list, which contains the instance id's sorted by depth. For each instance, the instance's AABB is projected to image space, and for every grid cell intersected by this projection, the instance is added to the grid cell's hit list. Each grid cell's hit list is kept in sorted order, based on distance from the image plane. Each plant model references a lower level uniform grid constructed from the model's triangles. Each instance



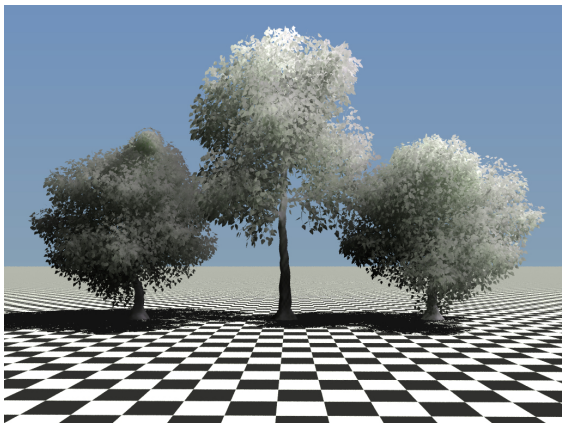
(a) Local LB only



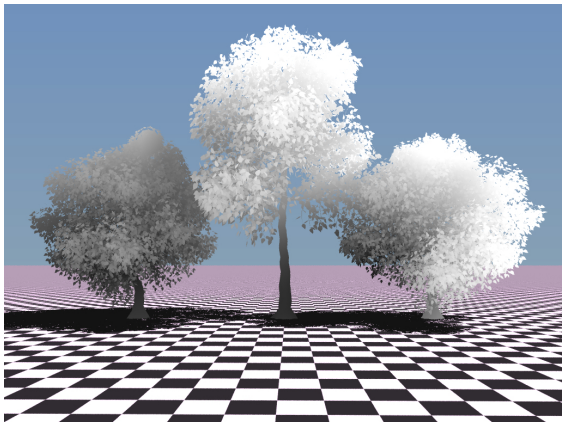
(b) Visualization of local LB only



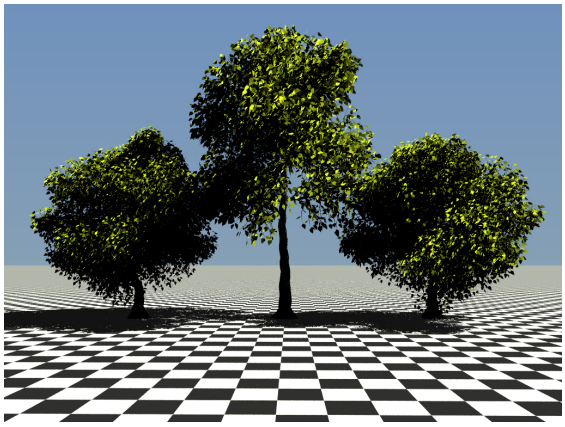
(c) Local and global LB



(d) Visualization of local and global LB



(e) Global LB only



(f) Direct lighting only

Figure 4.7: Rendering comparison: Local LB vs. local and global LB.

```
for each ray
        initialize bestHit
        for each instance in ray.perspectiveGridCell
                if bestHit < instance.minimumDepth
                        break
                else
                        set t to intersectRayWithAABB(ray,instance.AABB)
                        if t > 0 and t < bestHit
                                set hit to
                                        intersectRayWithUniformGrid(ray,instance.grid,t)
                                if hit < bestHit
                                        set bestHit to hit
                                end if
                        end if
                end if
        end for
        set ray.hit to bestHit
end for
```

Listing 4.1: Pseudo-code for primary ray traversal of multi-level hierarchy.

in the scene stores its world-space transformation and a reference to its model's grid.

Ray traversal is a hierarchical process, as outlined in Listing 4.1. Due to perspective projection, each primary ray maps to one and only one perspective grid cell. Each primary ray visits the instances in its corresponding grid cell's hit list in order, checking for intersection with each instance's AABB before intersecting the instance's uniform grid. Since instances are sorted by depth in each perspective grid cell's hit list, primary ray traversal is terminated once an instance is further than than the best intersection found so far. No 3-dimensional grid traversal is required for the top level of the hierarchy.

Once a primary ray intersects an instance's AABB, the primary ray traverses the uniform grid of the instance's model, checking for triangle intersections. Uniform grid traversal is accomplished by an algorithm similar to the 3DDDA (3D digital differential analyzer) approach as described originally by Fujimoto et al[10]. The grid traversal algorithm steps through the uniform grid one cell at a time until an intersection is found or the ray exits the uniform grid, as detailed in Listing 4.2. First, the ray is transformed into model space. Next, the first cell intersected by the ray is computed based on the previously computed ray/AABB intersection. At each cell intersected by the ray, the algorithm checks for intersection between the ray and the cell contents (triangles). (As triangles may span multiple cells in a grid, intersections are only valid if the intersection occurs within the current cell.) The ray-triangle intersection algorithm is the fast, minimal storage technique described by Möller and Trumbore [27]. Since leaf shape is determined by the alpha channel of a leaf texture, each

```
function intersectRayWithUniformGrid(ray, grid, t)
        transform ray into grid space
        set cell to cellAt(ray.origin + ray.direction * t)
        set step.x to ray.direction.x > 0 ? 1 : -1
        set step.y to ray.direction.y > 0 ? 1 : -1
        set step.z to ray.direction.z > 0 ? 1 : -1
        initialize hit
        while hit is none and cell is valid
                set hit to intersectRayWithCellContents(ray,cell)
                if hit is not none
                        break
                end if
                set cellHit to intersectRayWithCellPlanes(ray,cell)
                if cellHit.x < cellHit.y and cellHit.x < cellhit.z
                        cell.x += step.x
                else if cellHit.y < cellHit.z
                        cell.y += step.y
                else
                        cell.z += step.z
                end if
        end while
        return hit
end function
```

Listing 4.2: Pseudo-code for traversing a uniform grid with 3DDDA.

primary ray may intersect multiple triangles before terminating. Performance-enhancing adaptive transparency control, suggested in [6], was tested, but it has not been incorporated here, since it was found to noticeably degrade image quality.

Uniform grid traversal visits only the cells intersected by the ray, and these cells must be visited in the proper order. To compute the next cell, the ray is intersected with three of the six planes implied by the current cell, based on the ray direction. Finally, the next cell is chosen based on the closest plane and ray direction.

Occlusion of direct illumination (shadows) is computed by shadow rays. As an acceleration structure, we employ a sun-aligned grid, which, once again, is similar in spirit to the perspective grids of [18]. For each plant instance, the model's AABB is projected into world-space, followed by a projection into sun-space, such that the sun is located at infinity on the z-axis and the direction of sunlight is the negative z-axis. A sun-aligned grid is a 2-dimensional data structure, consisting of multiple tiles arranged as a grid on a plane whose normal is parallel to the sun direction. Each tile contains a list of those instances whose AABBs project to that tile. Using an orthographic projection (based on the minimum and maximum bounds of the scene in sun-space), we update each tile with the instances whose AABBs overlap that tile.

At run-time, we compute a shadow ray for each fragment. As shown in Listing 4.3, we first

```
for each shadowRay
        set t to intersectRayWithAABB(shadowRay, fragment.instance.AABB)
        if t > 0 and intersectRayWithUniformGrid(shadowRay, fragment.instance.grid, t)
                        is not none
                set fragment in shadow
        else
                for each instance in shadowRay.shadowGridCell
                                not equal to fragment.instance
                        set t to intersectRayWithAABB(shadowRay, instance.AABB)
                        if t > 0 and
                                intersectRayWithUniformGrid(shadowRay, instance.grid, t)
                                is not none
                                        set fragment in shadow
                                        break
                        end if
                end for
        end if
end for
```

Listing 4.3: Pseudo-code for shadow ray traversal of multi-level hierarchy.

intersect the shadow ray with the instance model containing the fragment. If there is no occlusion in the instance's model, we intersect instances in the sun-aligned grid. Because of the orthographic projection, each shadow ray intersects only one tile in the grid, and thus we need only consider those plant instances whose AABBs intersect that tile for subsequent occlusion testing. When a shadow ray intersects an instance's AABB, the shadow ray is transformed to model space and the check for occlusion with the model by is carried out by traversing the model's uniform grid, as previously discussed.

A second, high level, uniform grid is constructed once over the scene, again containing each instance's AABB. This grid is useful for other types of secondary rays, such as reflections rays.

4.5 Compression

Our LB lighting model produces volumetric data, which quickly requires large amounts of memory per plant model. Each node in the lattice consists of three floats (1 for each wavelength), and a 128^3 lattice contains 2,097,152 nodes. Our dynamic lighting technique from Section 4.2 requires precomputing 19 base solutions per plant model. Thus, for a 128^3 lattice, each plant model requires $(2{,}097{,}152 * 3 * 19 * 4)$ 456 MB of LB lighting data. Limiting ourselves to static scenes (one LB lighting solution per plant model) results in 24 MB of data per plant per model. For large scenes, the memory requirements for static and dynamic LB lighting quickly become a limiting factor. This has motivated the development of a compression technique for LB lighting.

Our target application, fast rendering of forest ecosystems, enforces two constraints on any compression technique. The first is rather obvious. Images rendered using compressed LB lighting solutions must differ as little as possible from images rendered using uncompressed LB lighting solutions. Lossy compression is acceptable, provided any artifacts introduced are visually unnoticeable. The second constraint results from the required access to the LB lighting at run-time. A shader applies LB lighting to each visible fragment by sampling from the LB lighting data, which can be considered a 3D texture. Sampling with linear interpolation (equations (4.4) and (4.5)) requires accessing eight texels (in a block of size 2^3) from the 3D texture, with each fragment accessing the volumetric LB lighting independently of any other fragment. Therefore, any compression technique must support fast, random access to the LB lighting data in blocks of size 2^3, while maintaining a high compression ratio.

Our research has shown that Haar wavelets are capable of significantly compressing LB lighting data while allowing fast, random access and still maintaining final image quality. There is much literature regarding using wavelets to compress volumetric data. Bajaj et al [2] tackle compressing 3D, RGB data, such as LB lighting data, for interactive applications. The general technique introduced involves compressing sub-blocks of the data (referred to as *cells* in [2]), quantizing groups of sub-blocks together, and finally using a novel encoding scheme to compress the quantized data while allowing fast, random access. As noted by [42], other wavelet transforms may provide better compression rates. But, as also noted by [42], the simplicity of Haar wavelets results in faster reconstruction times, which is vital to our application.

Our technique extends the 2-dimensional Haar wavelet transform [37] to 3-dimensions. In 2-dimensions, the first step involves applying the 1-dimensional Haar wavelet decomposition to each row. Next, the 1-dimensional Haar wavelet decomposition is applied to each column of the data output from the first step. For 3-dimensions, we follow this strategy and simply apply the 1-dimensional wavelet decomposition first to the z-dimension, then the y-dimension, and finally the x-dimension.

As discussed in Section 4.1, proper sampling from the LB lighting data requires linear interpolation. Thus, we not only need fast, random access to an individual texel, we need fast, random access to each 2^3 block of texels. To capture forward neighbors, our approach reduces the compression rate in order to maintain fast access to the forward neighbors of a texel. We compress sub-blocks of size 3^3; but, for each sub-block, we include the forward neighbors of all nodes of the

sub-block. This requirement introduces a forward border of size one. Our final sub-block size is 4^3, with 37 of 64 values stored being redundant. Although this approach increases the total size of the data to compress and therefore reduces our compression rate, our results show that this redundancy significantly improves rendering times by reducing the amount of work required to properly sample the LB lighting data.

After applying the Haar wavelet transform to each sub-block in the volumetric lighting data, we then decimate coefficients below a user supplied threshold. No vector quantization, such as that of [2], is implemented. We mark zero and non-zero detail coefficients in a 64-bit mask (1 bit per node of a sub-block). The non-zero values of all sub-blocks are compacted into a single stream, and, for each sub-block, we also store the 64-bit mask and the index of its first wavelet.

When a fragment accesses the LB lighting data, our algorithm loads the compressed data for the sub-block requested. Compressing sub-blocks of size 3^3 and including a forward border of size one guarantees that only one sub-block will be decompressed per fragment. Decompression is straightforward; the compressed data is expanded according to the bit-mask. Elements in the decompressed array corresponding to a zero bit value in the bit-mask are set to zero. Elements in the decompressed array corresponding to a one bit value in the bit-mask are given the appropriate value from the compressed array. Reconstruction is simply the inverse of decomposition.

Chapter 5

Implementation

In this section, we provide implementation details for our algorithms, specifically focusing on optimization for CUDA. First, we introduce optimizations that allow for fast computation of LB lighting with CUDA. We then discuss how to decompress our LB lighting data, which has been compressed with Haar wavelets, in real-time with CUDA. Finally, we discuss the details of ray tracing large scenes with instance geometry, such as a forest, with CUDA.

5.1 Lattice-Boltzmann Lighting

Before the LB lighting is computed, as detailed below, each tree model is converted to a volume density by intersecting the model's triangles with a high resolution grid, usually of size a 1024^3. If any triangle intersects a node of the high resolution grid, the density of that node is set to 1. The high resolution grid is then downsampled, iteratively, to a low resolution grid, usually 128^3. At each iteration, eight neighboring cells from the old grid are averaged to produce the cell in the new grid.

The LB lighting computation is implemented as a CUDA kernel. The first step in porting any application to CUDA involves isolating the data-parallel computations into kernels for execution on the device. Our lighting model maps well to CUDA, as each iteration of (2.12) requires computing new values for each grid node that are independent of the new values of every other grid node. One device thread is invoked per grid node. The indirect illumination due to each wavelength (color component) is computed separately.

A simple kernel for our photon transport model is shown in Listing 5.1. We instantiate one thread per lattice node, and each thread invokes the kernel. For an LB lighting lattice of size 128^3, we invoke 128 threads per block and set our grid size to 128^2. Function templates are supported by the CUDA compiler. The kernel `lightKernel` has a template parameter `SIZE`, allowing the compiler to generate functions for each of our supported lattice sizes of 64^3 and 128^3. Setting `SIZE` at compile time improves performance. All pointer arguments supplied to the kernel are arrays located in global memory space on the device. The code in Listing 5.1 attempts to minimize loads from global memory space by use of a local array, `in`, and stores to global memory space by scattering the current node's data to each neighbor in order. Due to the latency of global memory space, reducing reads to and writes from global memory space is very important when optimizing CUDA kernels. With this simple kernel, computing the LB lighting solution for a lattice of size 64^3 on a GTX 280 (compute capability 1.3) requires 2.38 seconds per wavelength to converge. (All times reported in this section will be for a lattice of size 64^3 computed on a GTX 280.) To converge, we iterate twice the largest dimension, which results in 128 iterations. Modifications to this kernel will enable much faster processing times.

Exploiting the memory hierarchy of CUDA devices is key to maximizing kernel performance. The kernel shown places all arrays in global memory space. Loads from this memory hinder performance, as it is neither cached nor on-chip, and memory transactions to and from global memory space can require 500 or more clock cycles. Small, constant arrays, such as `omega`, `omegaBark`, and `directions`, should instead be stored in the constant memory space, which is cached. These arrays fit in constant memory space, and, as an added benefit, all fit in constant cache (8KB) on an MP. Loading from constant cache experiences the same latency as loading from registers, provided all threads in a half-warp access the same address. The code in `lightKernel` follows this access pattern for these three arrays, guaranteeing optimal performance.

Large arrays, such as `input`, `output`, and `density`, will not fit in constant memory space, and so they must be stored in the global memory space. The array `density`, which stores the voxelized tree's biomass densities, is a three-dimensional array, with the minor dimension corresponding to the depth. All threads in a block share the same value of the index variables `i` and `j`. The index variable `k` ranges from 0 to `SIZE`-1; all threads in a half-warp will have sequential values of `k`, with $k \ mod \ 16 = 0$ for the first thread in a half-warp. Thus, when threads in a half-warp

```
__device__ int dDirections[19*3];// Lattice directions c_0 through c_18
__device__ float dOmega[19*19]; // Collision matrix for leaf nodes
__device__ float dOmegaBark[19*19]; // Collision matrix for bark nodes

#define INPUT(m,i,j,k) (input[(((i)*SIZE*SIZE*19 + (j)*SIZE*19 + (k)*19 + (m)))])
#define OUTPUT(m,i,j,k) (output[(((i)*SIZE*SIZE*19 + (j)*SIZE*19 + (k)*19 + (m)))])
#define DENSITY(i,j,k) (density[((i)*SIZE*SIZE + (j)*SIZE + (k))])
#define OMEGA(i,j) (dOmega[(i)*19+(j)])
#define OMEGA_BARK(i,j) (dOmegaBark[(i)*19+(j)])

template<int SIZE> __global__ void lightKernel(float *output,
                float *input, float *density) {
  const unsigned int i = blockIdx.x;
  const unsigned int j = blockIdx.y;
  const unsigned int k = threadIdx.x;
  float newDensity,result,biomass,in[19];
  bool bark = false;
  int m,n,outi,outj,outk;
  biomass = DENSITY(i,j,k);
  if (biomass < 0.f) { // Negative biomass designates bark.
    bark = true;
    biomass *= -1.f;
  }
  for (m=0;m<19;++m) in[m] = INPUT(m,i,j,k);
  if (!bark) in[0] = 0.f; // dump absorbed light
  for (m=0;m<19;++m) {
    outi = i+dDirections[m*3+0];
    outj = j+dDirections[m*3+1];
    outk = k+dDirections[m*3+2];
    if (!bark) {
      newDensity = OMEGA(m,0)*in[0];
      for (n=1;n<19;++n) newDensity += OMEGA(m,n)*in[n];
    } else {
      newDensity = OMEGA_BARK(m,0)*in[0];
      for (n=1;n<19;++n) newDensity += OMEGA_BARK(m,n)*in[n];
    }
    result = max(0.f, newDensity*biomass + in[m]);
    if (outi>0 && outi<SIZE-1 && outj>0 && outj<SIZE-1 && outk>0 && outk<SIZE-1)
    OUTPUT(m,outi,outj,outk) = result;
  }
}
```

Listing 5.1: Basic LB Lighting CUDA Kernel.

access density, the addresses are contained in the same memory segment and are sequential. The device can coalesce these reads to maximize memory bandwidth. Accessing array input is different though, as it is a four-dimensional array. In the code of Listing 5.1, the device cannot coalesce memory reads from input, since the minor dimension is not related to k (the depth). Instead, the minor dimension is the number of lattice directions. Examining the INPUT macro shows that each thread in a half-warp accesses addresses separated by 19*4 bytes. But, coalescing is possible by simply rearranging the array. Switching the depth and lattice direction dimensions allows threads in a half-warp to access sequential addresses. The output array, which is also stored in global memory space, is rearranged similarly to ensure coalescing when writing results. Similar to the values of i and j, the values of outi and outj are the same for all threads in a block. The value of outk only differs from k by at most one since directional values are -1, 0, and 1. Once output is rearranged, threads in a half-warp then access at most two memory segments when writing to output, thereby allowing devices of compute capability 1.2 or greater to coalesce the writes into at most two memory transactions. Utilizing the constant memory space for smaller, constant arrays and rearranging the order of the arrays input, output, and density improves the memory bandwidth of the kernel. With these improvement, the kernel requires on average 0.661 seconds per wavelength.

Though local in scope, the array in is not guaranteed to be stored on-chip. The compiler may place local arrays in local memory space, which has the same latency as global memory space. Analyzing compiler output shows that in is indeed stored in local memory space. We exploit the number of registers available on each multiprocessor to remove in. Each multiprocessor on hardware of compute capability 1.2 or greater contains 16,384 registers (8,192 registers for compute capability 1.1 or lesser), which are divided among all threads running on that multiprocessor. We replace the array in with 19 local variables, all of which are stored in registers, and we unroll the corresponding loops. We also combine omega and omegaBark into one three-dimensional array, thus removing one if statement and some divergence from the kernel. These two steps reduce our execution time to 0.242 seconds. The final kernel is shown in Listing 5.2. Device functions are inlined by the compiler, and thus invoking handleDirection as a function call introduces no overhead.

```
__constant__ int dDirections[19*3];
__constant__ float dOmega[2*19*19];

#define INPUT(m,i,j,k) (input[(i)*SIZE*SIZE*19 + (j)*SIZE*19 + (m)*SIZE + (k)])
#define OUTPUT(m,i,j,k) (output[(i)*SIZE*SIZE*19 + (j)*SIZE*19 + (m)*SIZE + (k)])
#define OMEGA(bark,i,j) (dOmega[bark*19*19+i*19+j])

template<int SIZE> __device__ void handleDirection(
        const int &m, const float &in, const float &biomass, const bool &brown,
        const int &i, const int &j, const int &k,
        float *output,
        const float &i0, const float &i1, const float &i2, const float &i3,
        ... 12 arguments omitted ...
        const float &i16, const float &i17, const float &i18) {
        int outi, outj, outk;
        float newDensity, result;
        outi = i+dDirections[m*3+0];
        outj = j+dDirections[m*3+1];
        outk = k+dDirections[m*3+2];
        newDensity = OMEGA(brown,m,0)*i0;
        newDensity += OMEGA(brown,m,1)*i1;
        ... 16 lines omitted ...
        newDensity += OMEGA(brown,m,18)*i18;
        result = max(0.f, newDensity*biomass + in);
        if (outi>0 && outi<SIZE-1 && outj>0 && outj<SIZE-1 && outk>0 && outk<SIZE-1)
        OUTPUT(m,outi,outj,outk) = result;
}

template<int SIZE> __global__ void lightKernel(float *output,float *input,
                float *density) {
        const unsigned int i = blockIdx.x;
        const unsigned int j = blockIdx.y;
        const unsigned int k = threadIdx.x;
        float biomass;
        bool brown = false;
        float i0,i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,i15,i16,i17,i18;
        biomass = density[i*SIZE*SIZE+j*SIZE+k];
        /** Negative biomass designates brown biomass. */
        if (biomass < 0.f) {
                brown = true;
                biomass *= -1.f;
        }
        i0 = INPUT(0,i,j,k);
        ... 17 lines omitted ...
        i18 = INPUT(18,i,j,k);
        if (!brown) i0 = 0.f; // dump absorbed light
        handleDirection<SIZE>(0,i0,biomass,brown,i,j,k,output,
                i0,i1,i2,i3,i4,i5,i6,i7,
                i8,i9,i10,i11,i12,i13,i14,i15,i16,i17,i18);
        ... 17 lines omitted ...
        handleDirection<SIZE>(18,i18,biomass,brown,i,j,k,output,
                i0,i1,i2,i3,i4,i5,i6,i7,
                i8,i9,i10,i11,i12,i13,i14,i15,i16,i17,i18);
}
```

Listing 5.2: Final LB Lighting CUDA Kernel.

5.2 Ray tracing

All ray tracing computations (ray-triangle intersections and generation of secondary rays) are implemented with CUDA. Before invoking the ray tracing kernels, all relevant scene data is transferred from the host to the device(s), with care taken to store data in the appropriate memory space. Due to device limitations (all textures must be declared at compile time), all leaf and bark textures are stored in one texture on the device. A preprocessing steps resizes all textures to the same width and adjusts texture coordinates for all triangles. All other data, such as perspective grids, coherent grids, and triangle geometries, are stored in the global memory space of the device. Also, to reduce memory overhead, each set of data is stored as structures of arrays, not arrays of structures, and is compacted into a single array (i.e., all triangles for all models are stored in one array).

Maximizing read performance from a device's global memory space, which is not cached, requires that threads in a half-warp follow certain memory access patterns, as previously discussed. Unfortunately, threads in a warp quickly diverge when traversing a scene's multiple acceleration structures. Maximizing memory bandwidth becomes practically impossible. It is tempting to store non-texture data in the texture memory space, as it provides a data cache. But, experimentation with the latest GPUs have found no benefit to moving data from global memory space to texture memory space. Newer GPUs (compute capability 1.2 or greater) have looser coalescing requirements, and coherency of neighboring threads often results in improved memory bandwidth without the need for a cache.

Ray tracing is performed by executing multiple kernels on the device. One device thread is created per ray. Since all threads within a thread block share resources, the resource constraints of traversing a two-level hierarchy (high-level, perspective grid over the scene and low-level, uniform grid for each model) restrict the primary ray kernel and shadow ray kernel to 64 threads per block. To the extent possible, care is taken to prevent threads within the same warp from diverging by arranging each thread block to trace rays in an 8×8 tile, and, within this tile, each warp (32 threads) in a thread block traces rays in an 8×4 tile. Rays that trace pixels that are coherent in image space are less likely to experience divergence. A variety of block sizes and tile sizes were tested, and this layout was found to provide the best performance.

Listing 5.3 provides the core CUDA code for intersecting a ray with a uniform grid cell. (The

```
struct TriHit
{
        __device__ TriHit() : tIdx(-1),inst(-1),t(FLT_MAX) {}
        __device__ bool noHit() { return tIdx==-1; }
        __device__ void update(int cellIdx, float maxTVal, int self, Ray &tRay)
        {
                // self is not referenced in this code,
                // but is in shadow ray code
                // x is number of triangles in cell, y is index of first triangle
                vec2i cellData = dGridData.gridCells[cellIdx];
                cellData.x += cellData.y;
                t = maxTVal;
                for (int hitIdx=cellData.y; hitIdx<cellData.x; ++hitIdx) {
                        int firstVertIdx = dGridData.gridHits[hitIdx];
                        vec3f hit = kRayTriIntersect(tRay,
                                dRayTraceData.triVerts[firstVertIdx],
                                dRayTraceData.triVerts[firstVertIdx+1],
                                dRayTraceData.triVerts[firstVertIdx+2]);
                        // hit.x is t value of ray/triangle intersection
                        // hit.y and hit.z are barycentric
                        // coordinates of ray/triangle intersection
                        if (hit.x < t) {
                                // Check for hit with the actual leaf texture
                                vec4f tex = kTexture2d(firstVertIdx, hit.y, hit.z);
                                // tex.w is alpha component
                                if (tex.w > 0.95f) {
                                        t = hit.x;
                                        u = hit.y;
                                        v = hit.z;
                                        tIdx = firstVertIdx;
                                }
                        }
                }
        }
        int tIdx, inst;
        float t,u,v;
};
```

Listing 5.3: CUDA code for intersecting a ray with contents of a uniform grid cell.

code shown is for primary rays. Shadow rays differ in that any intersection will suffice; it need not be the closest intersection.) For each uniform grid cell, the array `dGridData.gridCells` contains two values: the number of triangles that intersect the cell and an offset into `dGridData.gridHits`, which contains indices, compacted, into the triangle array, `dRayTraceData.triVerts`, of the triangles that intersect a cell. (This grid representation is described in [23].) The code loops over these triangles testing for intersection. A ray/triangle intersection is only valid if it is within the current cell (shadow rays do not have this restriction), and the corresponding texel's alpha value is above a threshold, since the alpha value serves as the "cutout" for the leaf texture within the triangle.

Listing 5.4 details the core CUDA code for traversing an instance's uniform grid. Due mainly

to ray divergence that results as threads in a thread block take different paths through the uniform grid (and scene), the code is memory bound. Attempts to optimize the instruction throughput did not result in increased performance.

Rendering across all four GPUs in a Tesla S1070 requires creating one CPU thread per GPU. Load balancing is achieved by dividing the image into a set of equally sized image tiles. For most scenes, performance is best when the number of image tiles is more than twice the number of GPUs. While unrendered tiles exist for a frame, each CPU thread picks a tile to render. While the GPUs render tiles, a fifth CPU thread (not associated with a GPU) computes any data that needs to be updated for the next frame. For example, if the camera is changing position, the perspective grid must be recomputed. If the sun is moving, the sun-aligned grid and, for each instance, the choice of base LB lighting solutions and their associated weights must be recomputed. Finally, results are gathered and displayed to the user.

5.3 Compression and Decompression

Compression of LB lighting data is a preprocessing step that is implemented on the CPU. As detailed in Section 4.5, we compress overlapping sub-blocks of size 4^3 to include all forward neighbors for linear interpolation. Thus, each sub-block consists of 64 floats per wavelength. Wavelet decomposition is performed on each component individually. Compression is achieved by decimating detail coefficients less than an error threshold. For improved reconstruction performance, all detail coefficients for a node (all three wavelengths) must be less than the supplied threshold. A 64-bit mask marks zero and non-zero coefficients. For each sub-block, the compressed stream contains the 64-bit mask and non-zero detail coefficients.

Decompression is a run-time process implemented in CUDA. A lighting shader, written in CUDA, is responsible for computing and combining the direct illumination and LB lighting of each fragment. The execution configuration for our lighting shader invokes 1 thread per fragment (we support multiple fragments per pixel), divided into 64 threads per thread block. Without compression, computing the LB lighting for a fragment is accomplished through sampling a 3D texture using linear interpolation. (As of this writing, our implementation cannot exploit the built-in 3D texturing features of CUDA due to two constraints. First, all CUDA textures must be declared at compile time, and CUDA does not support arrays of textures. Thus, all LB lighting data must be

```
TriHit bestHit;
vec3f ird(1.f/tRay.d.x, 1.f/tRay.d.y, 1.f/tRay.d.z);
// tRay previously transformed (not shown) to grid space
// t is from ray/grid AABB intersection
vec3f hitPt = tRay.o + tRay.d*t;
vec3f cellNorm;
cellNorm.x = (hitPt.x - instance.aabb.min.x)/instance.aabb.size.x;
cellNorm.y = (hitPt.y - instance.aabb.min.y)/instance.aabb.size.y;
cellNorm.z = (hitPt.z - instance.aabb.min.z)/instance.aabb.size.z;

vec3i cell;
cell.x = (int)(cellNorm.x*gridDim);
cell.y = (int)(cellNorm.y*gridDim);
cell.z = (int)(cellNorm.z*gridDim);
cell.x = fminf(cell.x, gridDim-1);
cell.y = fminf(cell.y, gridDim-1);
cell.z = fminf(cell.z, gridDim-1);

vec3i cellStep;
cellStep.x = tRay.d.x > 0 ? 1 : -1;
cellStep.y = tRay.d.y > 0 ? 1 : -1;
cellStep.z = tRay.d.z > 0 ? 1 : -1;

instance.aabb.size = instance.aabb.size*(1.f/gridDim);
// clamp to best hit so far
tScene.y = min(tScene.y, minT);
while (bestHit.noHit()) {
        // Compute implicit planes for current cell
        vec3f cellPlanes;
        cellPlanes.x = cell.x+(cellStep.x+1.f)*0.5f;
        cellPlanes.y = cell.y+(cellStep.y+1.f)*0.5f;
        cellPlanes.z = cell.z+(cellStep.z+1.f)*0.5f;
        cellPlanes.x = cellPlanes.x*instance.aabb.size.x+instance.aabb.min.x;
        cellPlanes.y = cellPlanes.y*instance.aabb.size.y+instance.aabb.min.y;
        cellPlanes.z = cellPlanes.z*instance.aabb.size.z+instance.aabb.min.z;

        vec3f tVals;
        // Intersect with implicit planes
        tVals.x = (cellPlanes.x - tRay.o.x)*ird.x;
        tVals.y = (cellPlanes.y - tRay.o.y)*ird.y;
        tVals.z = (cellPlanes.z - tRay.o.z)*ird.z;
        float minTVal = fminf(tVals.x, fminf(tVals.y, tVals.z));
        // Bounds check
        if (minTVal > tScene.y) break;
        bestHit.update(cellOffset+cell.x*gridDim*gridDim+cell.y*gridDim+cell.z,
                minTVal,self,tRay);
        t=minTVal;
        // Step to the next Cell
        if (tVals.x < tVals.y && tVals.x < tVals.z) {
                cell.x += cellStep.x;
        } else if (tVals.y < tVals.z) {
                cell.y += cellStep.y;
        } else {
                cell.z += cellStep.z;
        }
}
```

Listing 5.4: CUDA code for traversing a uniform grid.

stored in one texture. Second, the maximum size of any edge dimension of a 3D texture is limited to 2^{11}. Our data quickly outgrows this restriction. Instead, we implement linear interpolation in software on the device.) With compression, the kernel computes each thread's sub-block's location in memory, loads the sub-block's compressed data stream, decompresses the compressed data stream, reconstructs the original data from the wavelet data, and samples the reconstructed LB lighting data according to (4.4) and (4.5). Because our compression technique includes all forward neighbors, each thread needs to only access one sub-block for proper sampling. Without including forward neighbors, each thread, in the worst case, would need access to eight sub-blocks for proper sampling.

To decompress and reconstruct a sub-block of LB lighting data, each thread needs access to multiple temporary, local arrays, each of size 64. Placing these arrays in the local memory space of the device introduces large, undesirable latencies. Instead, we store these local arrays in the shared memory space of the device. But, shared memory is limited to 16 KB per thread block; therefore, it is not large enough for every thread in a thread block to allocate temporary arrays in parallel. Instead, all 64 threads in a thread block cooperate to decompress and reconstruct individual data sub-blocks one at a time, storing temporary arrays in shared memory. All threads in a thread block have access to share memory, and threads in a thread block can synchronize. In parallel, each thread in a thread block stores the index of its required data sub-block in a work queue of size 64 located in shared memory. (If a thread does not require a data sub-block, it stores a value of -1 in the work queue.) After synchronizing, in parallel, all threads in a thread block loop over the work queue in sequence, loading, decompressing, and reconstructing one data sub-block in parallel.

When the compressed stream for a data sub-block contains only a single non-zero entry, decompression and reconstruction are fast and straightforward as all values in the data sub-block are assigned that non-zero value. The code contains a fast path to handle this situation. Experimentation has shown this to be quite beneficial to performance.

Otherwise, decompressing a single sub-block involves expanding the sub-block's compressed stream into an array of size 64 stored in shared memory. Listing 5.5, which is executed by each thread in parallel, contains the core code of the CUDA kernel for parallel decompression of a single a sub-block in parallel. Each thread stores its thread block index in `myIdx`, and the base address of the data sub-block in global memory is stored in `compressedStreamBase`, which is the same value for all threads in a thread block. First, in parallel, threads with `myIdx` less than the size of the compressed data load from the compressed stream to an array stored in shared memory

```
1 __shared__ float3 compressed[64];
2 __shared__ float3 decompressed[64];
3 ...
4 if (myIdx < sizeOfCompressedStream) {
5         compressed[myIdx] = compressedStreamBase[metaIdx+myIdx]
6 }
7 __syncthreads();
8 decompressed[myIdx] = make_float3(0.f, 0.f, 0.f);
9 if (myIdx < 32 && (metaMask.x & (1<<myIdx))) {
10        uint compressedIdx = (myIdx == 0 ? 0 : __popc(metaMask.x & ((1<<myIdx)-1)));
11        decompressed[myIdx] = compressed[compressedIdx];
12 } else if (myIdx >= 32 && (metaMask.y & (1<<(myIdx-32)))) {
13        uint tIdx = myIdx-32;
14        uint compressedIdx = __popc(metaMask.x)+
15                        (tIdx == 0 ? 0 : __popc(metaMask.y & ((1<<tIdx)-1)));
16        decompressed[myIdx] = compressed[compressedIdx];
17 }
18 __syncthreads();
```

Listing 5.5: Decompressing with CUDA.

(`compressed`). After synchronizing to ensure all loads have completed, each thread is responsible for computing the value in the decompressed array (`decompressed`, also in shared memory) at index `myIdx`. Two 32-bit registers, `metaMask.x` and `metaMask.y`, contain the target sub-block's 64-bit mask. A thread's corresponding bit value in the bit mask is computed differently based on whether `myIdx` is less than 32 or not. If `myIdx` is less than 32, then the bit value is `metaMask.x & (1<<myIdx)`, as seen on line 9. Otherwise, the bit value is `metaMask.y & (1<<(myIdx-32))`, as seen on line 12.

If a thread's bit value is 0, then the thread stores a value of zero at `decompressed[myIdx]`, as the corresponding wavelet coefficient was decimated during compression. Otherwise, the thread loads data from `compressed` to `decompressed[myIdx]`. A thread's index into `compressed`, likely a smaller array, since it contains only non-zero terms, is computed by summing the number of 1 bits in the bit mask before this thread's bit index (`myIdx`). The intrinsic function `__popc` returns the number of bits that are 1 in a 32-bit integer. Summing, on lines 10 or 14-15 based on the value of `myIdx`, is accomplished by first setting all bits greater than or equal to `myIdx` to zero and then invoking `__popc`. Finally, all threads in the thread block synchronize to ensure `decompressed` is complete before moving to reconstruction.

Once the data has been decompressed into shared memory, reconstruction is performed for the sub-block. Only the first half-warp (16 threads) of the first warp in a block reconstructs the data in parallel, as there exist only 16 values per axis. The 16 threads in parallel perform reconstruction,

```
#define IDX(x,y,z) ((x)*16+(y)*4+(z))
#define WREC_I(x,y,z,s) { \
        myAux[2*(x)] = (compressed[IDX(x,y,z)]+compressed[IDX((x)+s,y,z)]); \
        myAux[2*(x)+1] = (compressed[IDX(x,y,z)]-compressed[IDX((x)+s,y,z)]); \
        }

__shared__ float3 aux[64];
float3 *myAux;
int myIdx = threadIdx.x + threadIdx.y*blockDim.x;
int x = myIdx>>2;    // myIdx/4;
int y = myIdx&(4-1); // myIdx%4;
myAux = aux + x*16 + y*4;
WREC_I(0,x,y,1);
compressed[IDX(0,x,y)] = myAux[0];
compressed[IDX(1,x,y)] = myAux[1];
WREC_I(0,x,y,2);
WREC_I(1,x,y,2);
compressed[IDX(0,x,y)] = myAux[0];
compressed[IDX(1,x,y)] = myAux[1];
compressed[IDX(2,x,y)] = myAux[2];
compressed[IDX(3,x,y)] = myAux[3];
```

Listing 5.6: Reconstruction with CUDA.

applying the standard Haar wavelet reconstruction algorithm first to the x-axis, then the y-axis, and finally the z-axis. Note that using only 16 of 64 threads is not a huge performance penalty; conditionals ensure all threads in the second warp (32 in total) will quickly skip the reconstruction computation. Efforts to utilize all 32 threads in the first warp, through reconstructing two sub-blocks at once, yielded worse performance due to the extra overhead required. Listing 5.6 contains the core code, which is only executed by the first half-warp, for reconstructing across the x-axis. Reconstructing across the y-axis and the z-axis follows this template.

Because of image coherency, threads within a thread block map to neighboring samples in image space; hence, these threads often require the same LB lighting sub-block. Duplicate entries in the sub-block work queue are removed to avoid computing the same sub-block multiple times within a thread block. In Figure 5.1, green pixels denote where, due to image coherency, the fragment gained access to its sub-block before its sub-block became active in the work list. Red pixels denote that the fragment gained access to its sub-block due to its sub-block being active in the work list. Note that, as expected, image coherency is more prevalent when the plant is closer to the camera.

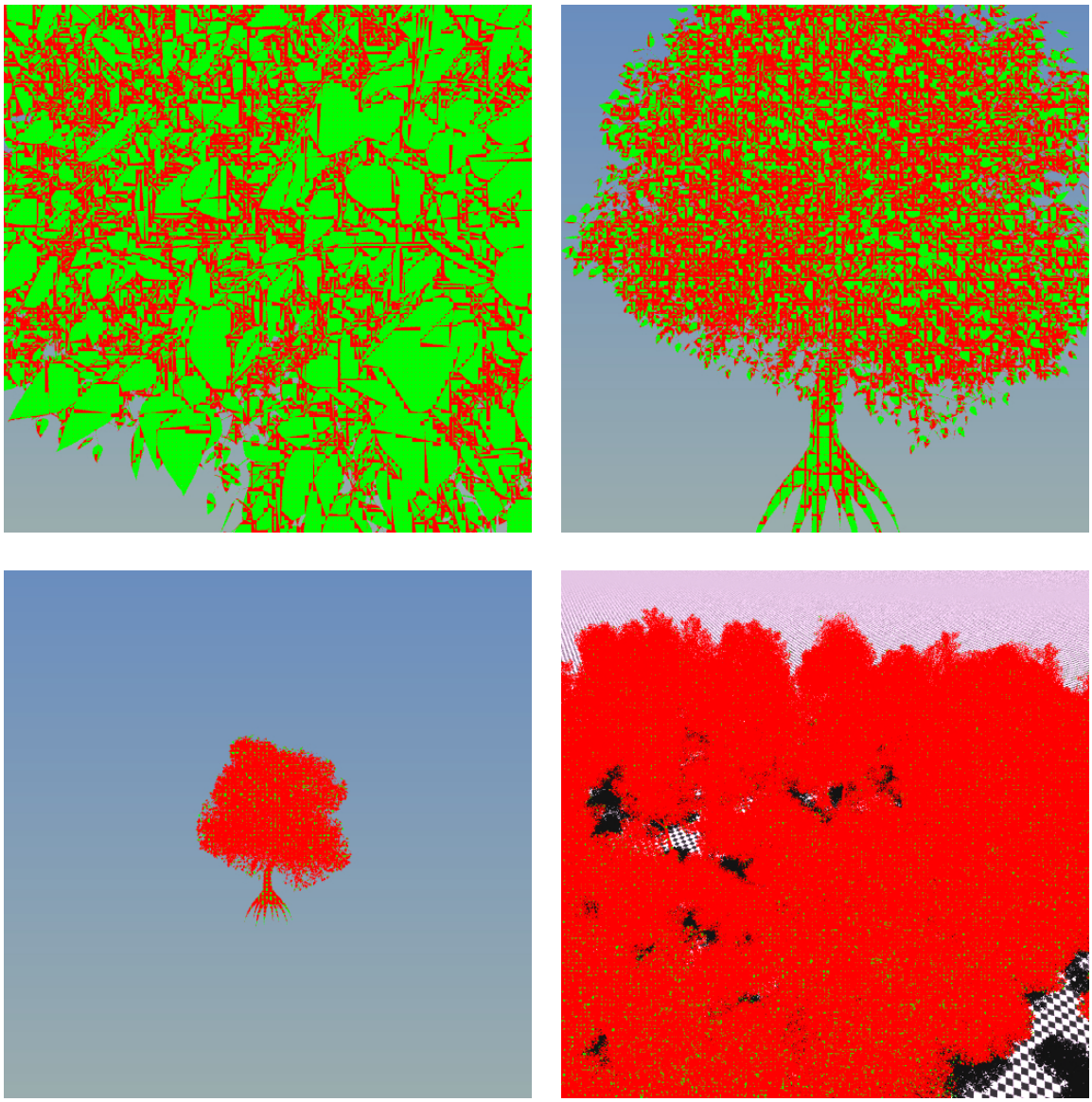


Figure 5.1: LB Lighting coherency. Green pixels denote where coherency can be exploited.

Chapter 6

Results

Though subtle at distances, the effects of forward scattering in the proposed illumination model are fairly dramatic at close range. Figures 6.1 and 6.2 demonstrated the lighting technique introduced in this paper in full scenes. Note that the reflected cloud was lighted with the original LB technique [14] and generated by the technique describe in [12]. The water surface in these scenes was generated by selecting small, random coefficients in frequency space for each node of a 2D grid and then performing a standard Fourier inversion. As is common practice, the random coefficients were constrained so that upon inversion both the surface and its gradient were real-valued. Specific lighting model parameters for plants are shown in Table 6.1, and these parameters are used for all images in this chapter. Also, the LB lighting grid was 128^3 for all images in this chapter.

Figure 6.3 shows a nearby view of a Southern Catalpa tree [30] rendered with the proposed technique and (for comparison) with only backward scattering and with only local illumination. Also shown is a volume visualization of the indirect illumination from the LB scattering grid used in this rendering.

Table 6.1: Lighting model parameter values.

$\sigma_a = (\sigma_a^R, \sigma_a^G, \sigma_a^B)$	(0.109, 0.091, 0.118)
$\sigma_s = (\sigma_s^R, \sigma_s^G, \sigma_s^B)$	(0.891, 0.909, 0.882)
$fs = (fs^R, fs^G, fs^B)$	(0.055, 0.150, 0.020)
$bs = (bs^R, bs^G, bs^B)$	(0.070, 0.125, 0.040)
$g = (g^R, g^G, g^B)$	(-0.120, 0.091, -0.333)

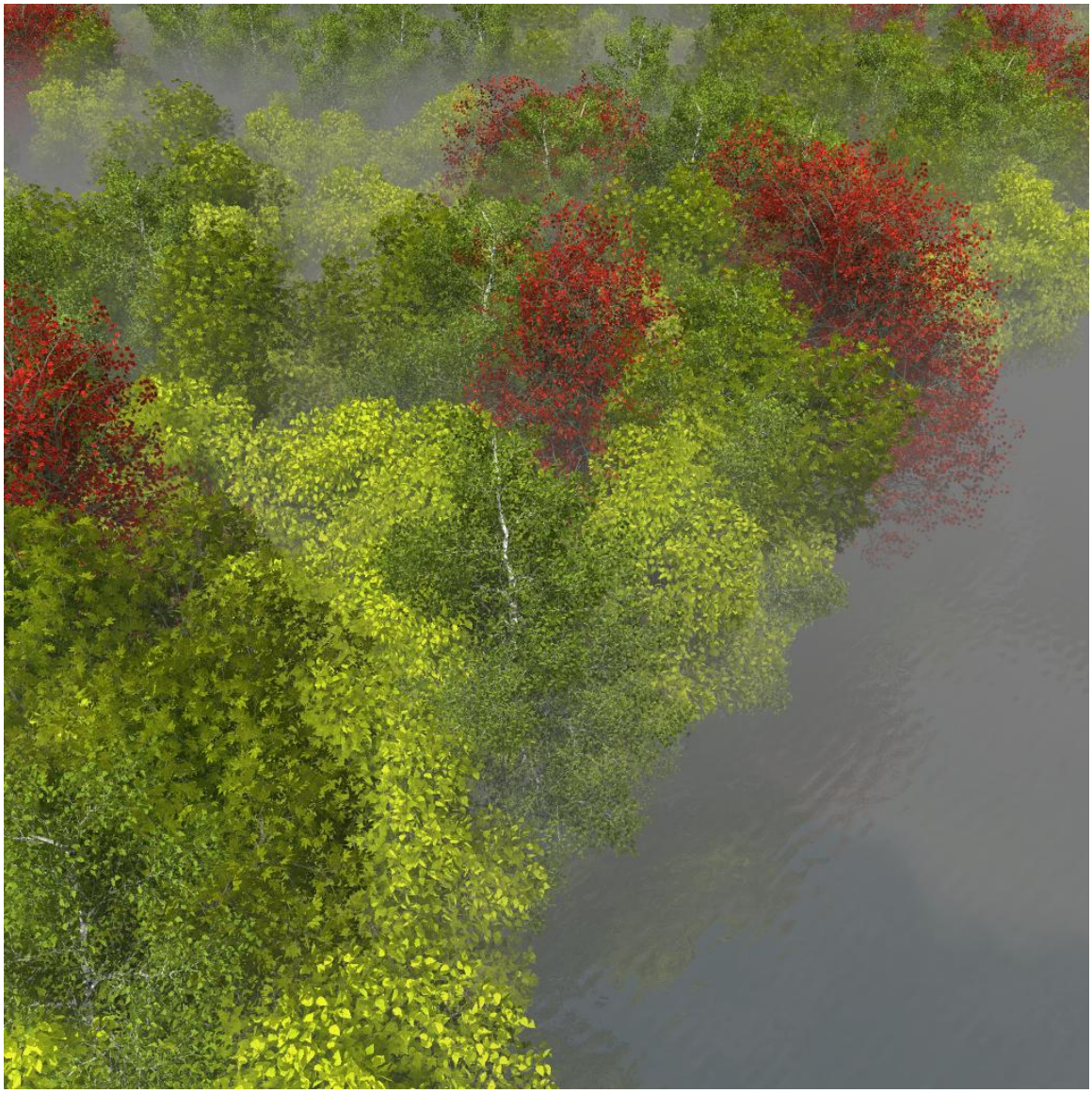


Figure 6.1: Lake scene.

Figure 6.2: River scene.

Both images in Figure 6.4 show the same scene, with the top image rendered using uncompressed LB lighting and the bottom image rendered using compressed LB lighting. For the Southern Catalpa tree shown, uncompressed LB lighting requires 456 MB while compressed LB lighting requires about 81 MB. At a resolution of 1024×1024 using 1 GPU, sampling the uncompressed LB lighting requires 23 ms, and sampling the compressed LB lighting requires 53 ms. The root mean square difference of the two images is 0.0136.

Figure 6.5 shows multiple frames captured as the sun moves across the sky, ranging 90 degrees from the zenith. Average execution times for rendering the animation at a resolution of 512×512 pixels with 1 ray per pixel and 4 rays per pixel are shown in Table 6.2. Values for one GPU, two GPUs, and all four GPUs of a Tesla S1070 are given. Note that increasing the the number of rays per pixel increases the coherency of neighboring rays, which results in improved performance per ray. When rendering with 4 rays per pixel on 1 GPU, primary ray casting averaged 217 ms, shadow ray computation averaged 162 ms, and uncompressed LB light sampling (including direct light computation) averaged 51 ms. Many applications, such as cinematic relighting, do not require camera movement. Thus, after computing fragments once, our techniques allow relighting (altering

the sun and not the camera) at multiple frames per second for large scenes, as only shadow rays and lighting need to be recomputed per frame. Average execution times for relighting the animation in Figure 6.5 at a resolution of 512×512 pixels with 1 ray per pixel and 4 rays per pixel are shown in Table 6.3. Once again, values for one GPU, two GPUs, and all four GPUs of a Tesla S1070 are given.

Table 6.2: Execution times for rendering Figure 6.5, Tesla S1070.

Number of GPUs	1 ray/pixel	4 rays/pixel
1	140 ms	477 ms
2	76 ms	244 ms
4	43 ms	154 ms

Table 6.3: Execution times for relighting Figure 6.5, Tesla S1070.

Number of GPUs	1 ray/pixel	4 rays/pixel
1	72 ms	258 ms
2	40 ms	150 ms
4	23 ms	96 ms

Figure 6.6 shows an example rendering of a high-density, forest ecosystem ray traced at 4 rays per pixel, with dynamic LB lighting. At a resolution of 1024×1024, with 4 rays per pixel, using 4 GPUs, sampling the uncompressed LB lighting requires on average 60 ms per GPU, and sampling the compressed LB lighting requires 120 ms per GPU. The root mean square difference between the two images (the top image was rendered with uncompressed LB lighting and the bottom image was rendered with compressed LB lighting) is 0.00843. Scene composition for Figure 6.6, including the compressed size of the precomputed LB lighting solution, is shown in Table 6.4. Uncompressed, each precomputed LB lighting solution (one per species) requires 456 MB, resulting in a total of almost 2.3 GB of uncompressed data. Note that on average, not including the forward neighbors of each sub-block, as discussed in Section 4.5, would result in compressed lighting that is about 42% the size of compressed lighting with forward neighbors included in each sub-block. But, including the forward neighbors results in LB sampling times that, on average, are more than 6 times faster.

Execution times for the LB lighting preprocessing step (computing 19 base solutions, three

full scattering

backscattering only

local illumination only

volume visualization of LB solution

Figure 6.3: Rendering comparison: nearby view of Southern Catalpa tree.

Rendered with uncompressed LB lighting.

Rendered with compressed LB lighting.

Figure 6.4: Southern Catalpa: Comparing rendering with uncompressed and compressed LB lighting.

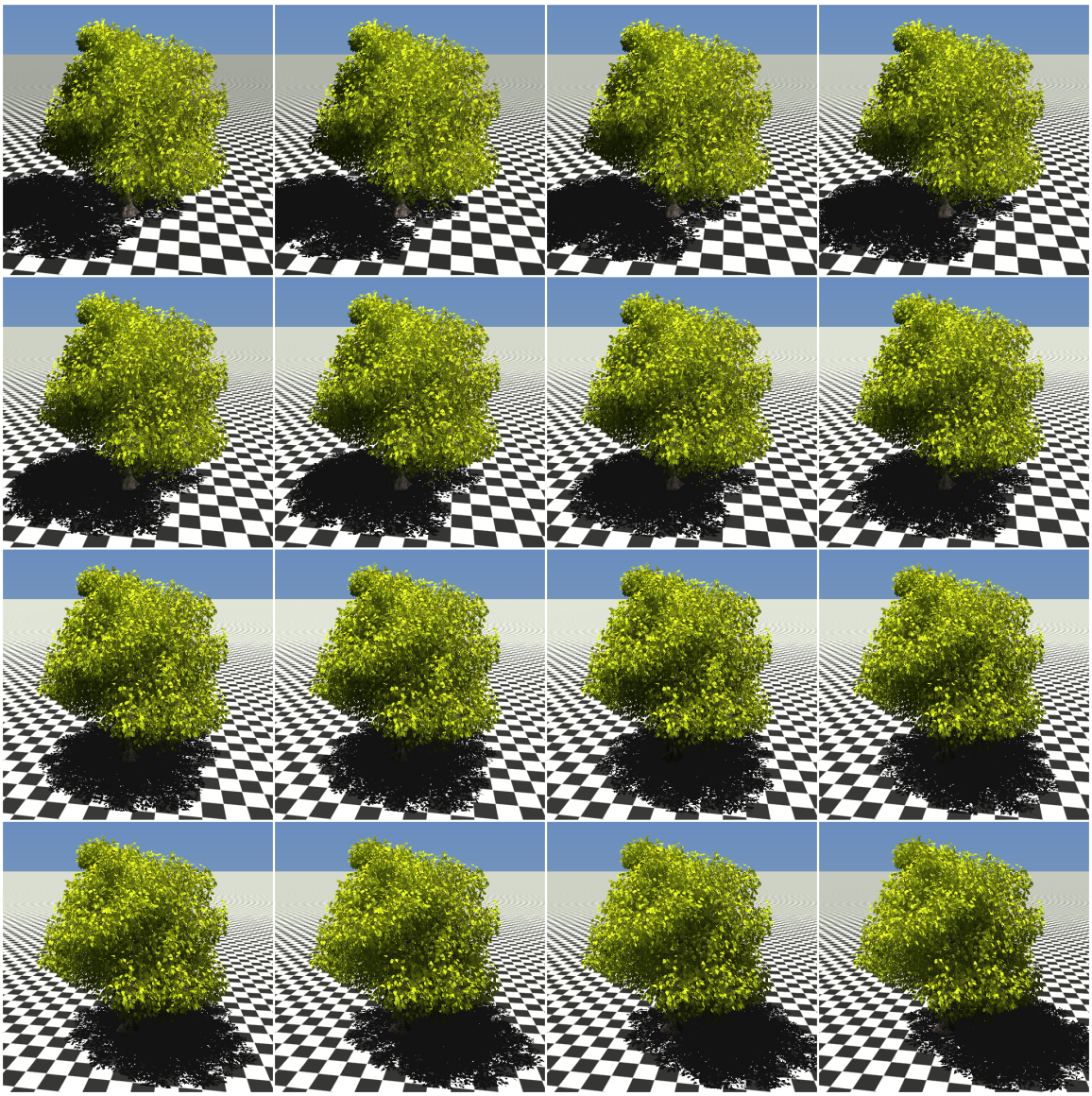
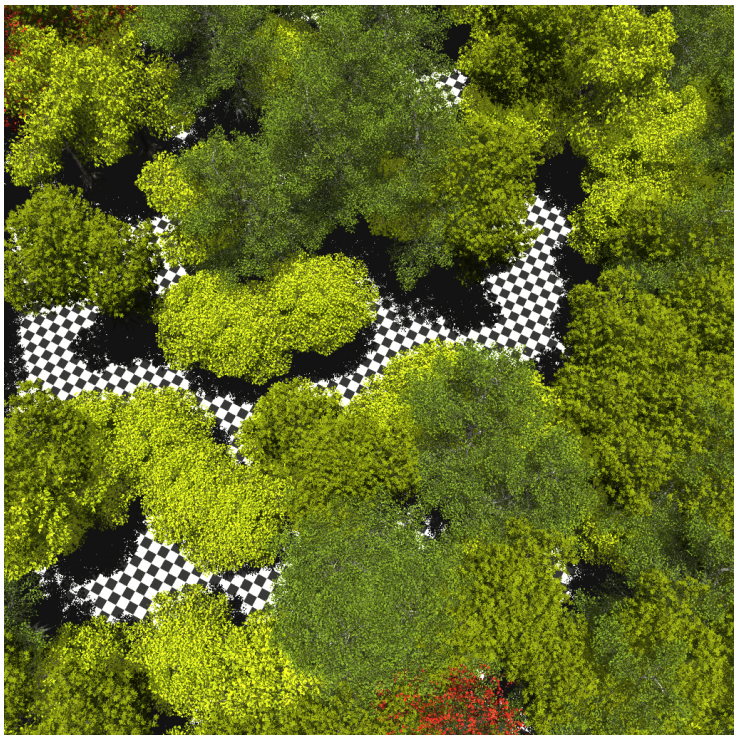
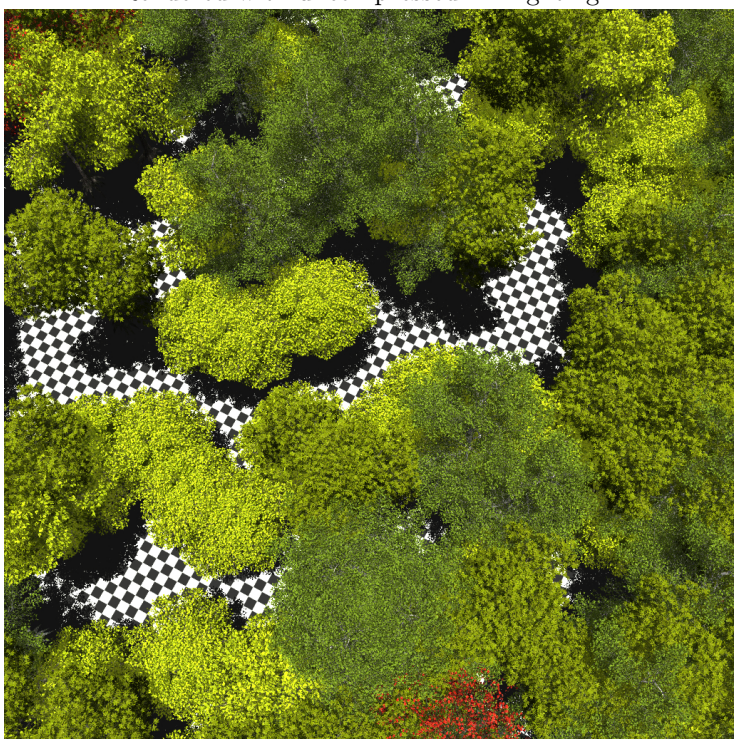


Figure 6.5: Southern Catalpa: Frames from animation of sun traversing the sky.



Rendered with uncompressed LB lighting.



Rendered with compressed LB lighting.

Figure 6.6: Forest: Comparing rendering with uncompressed and compressed LB lighting.

Table 6.4: Composition of scene in Figure 6.6.

Species	Instances	Triangles/Instance	Compressed LB Size
Red Maple	20	115,529	81 MB
Ohio Buckeye	89	168,520	64 MB
Paper Birch	81	372,896	63 MB
Southern Catalpa (adult)	74	155,342	77 MB
Southern Catalpa (medium)	160	316,767	81 MB
Total Scene	424	109,691,464	366 MB

wavelengths each, on a GTX 280) for the trees in Figure 6.6 are shown in Table 6.5. Note the times shown are the average times for five trees. For such models, the number of iterations required to achieve convergence to steady-state is approximately twice the longest edge dimension. Thus, it is expected that the time required to compute an LB lighting solution for a lattice of size 128^3 would be sixteen times as long as that for a lattice of size 64^3. The per wavelength column only reports the time required to compute the solution for one wavelength of one base LB lighting solution. Total time includes any extra processing outside the LB lighting solution, such as transferring data between the CPU and GPU.

Table 6.5: Precomputation time for dynamic LB lighting, GTX 280.

Lattice size	Per wavelength	Total time
64^3	0.196 s	18.5 s
128^3	2.926 s	3 min 20 s

Figure 6.7 shows multiple frames captured as the sun moves across the sky, ranging 90 degrees from the zenith. Average execution times for rendering this animation at a resolution of 512×512 pixels with 1 ray per pixel and 4 rays per pixel are shown in Table 6.6, and relighting only times are provided in Table 6.7. Again, values for one GPU, two GPUs, and all four GPUs of a Tesla S1070 are shown. When rendering with 4 rays per pixel on 1 GPU, primary ray casting averaged 1175 ms, shadow ray computation averaged 613 ms, and uncompressed LB light sampling (including direct light computation) averaged 191 ms.

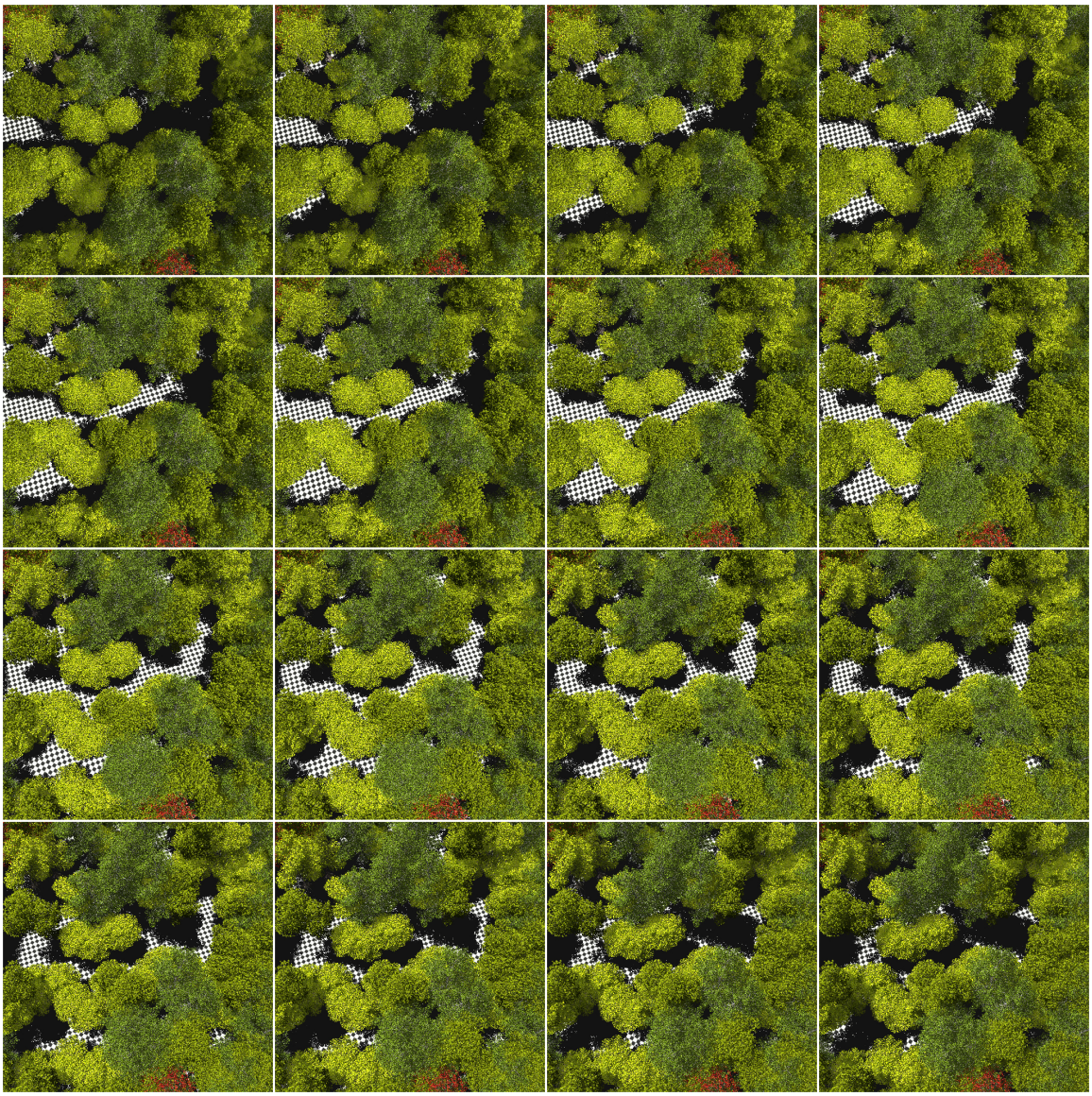


Figure 6.7: Forest: Frames from animation of sun traversing the sky.

Table 6.6: Execution times for rendering Figure 6.7, Tesla S1070.

Number of GPUs	1 ray/pixel	4 rays/pixel
1	628 ms	2000 ms
2	337 ms	1003 ms
4	176 ms	540 ms

Table 6.7: Execution times for relighting Figure 6.7, Tesla S1070.

Number of GPUs	1 ray/pixel	4 rays/pixel
1	259 ms	858 ms
2	132 ms	432 ms
4	73 ms	247 ms

Chapter 7

Conclusion

Real-time rendering of large-scale, forest ecosystems remains a challenging problem when global illumination effects, such as leaf transparency and inter-object reflection, which are important to visual realism, must be incorporated. One approach to achieving such effects, suggested herein, is through the use of a lattice-Boltzmann lighting model to approximate the indirect illumination. The LB model can be executed as a preprocessing step to generate multiple solutions that are then interpolated at run-time to allow dynamic movement of the sun and correct lighting of rotated and translated plant model instances. A ray tracing or rasterization engine can then combine local, direct illumination at any visible fragment point with an indirect value obtained by interpolating values from the preprocessed LB lighting solution. Ray tracing is generally preferable since it offers better image quality and its time is $O(log\ n)$, where n represents scene complexity; thus, using intelligent acceleration structures, ray tracing scales well as the scene complexity (number of triangles), n, increases. Near real-time performance is obtained by mapping the ray tracing engine, as well as the LB lighting model, to NVIDIA's Compute Unified Device Architecture and then distributing across multiple GPUs.

The LB lighting model uses parameters derived from measurements of real plants to approximate global illumination. It solves a diffusion-like process for light scattering and absorption. Although not described in this paper, the techniques for distributing the LB model across multiple GPUs are reasonably straightforward. For a single solution, boundary nodes in sub-grids are replicated, and only these replicated boundary nodes need communicate with one another across GPUs. When preprocessing multiple solutions, each solution is assigned to a GPU to minimize

communication overhead.

There are several drawbacks to the original LB lighting technique [12] that have been addressed in this study. First, a more thorough preprocessing step now allows for lights at an infinite distance, such as the sun, to dynamically update at run-time. This also allows plant instances that reference the same plant model to share the same LB lighting solution, regardless of each instance's orientation. Second, an LB lighting solution is a volumetric solution that produces large amounts of data. We have shown that compression using Haar wavelets is viable, in that relatively high compression rates can be achieved while maintaining final image quality and rendering performance.

Future work will continue to improve the lighting model and ray tracing engine. The lighting technique introduced allows dynamic updates to the sun at run-time, although it is not applicable for lights at a finite distance. Future research should investigate exploiting the multiresolution nature of wavelets for mipmapping LB lighting at different distances, thus removing aliasing issues that appear when sampling with one ray per pixel. A detailed comparison of kd-trees, uniform grids, and perspective grids, as applied to aggregate objects such as plants and traversed with CUDA, is in order. Also, secondary rays other than shadow rays, such as reflection rays, are still a performance bottleneck.

Overall, the proposed techniques allow high quality rendering of large forest ecosystems in near real-time. The lighting model is both physically accurate and visually pleasing, and, through careful optimizations, it is practical for production environments.

Bibliography

[1] J. Arvo. Transfer equations in global illumination. In *Global Illumination, SIGGRAPH '93 Course Notes*, August 1993.

[2] Chandrajit Bajaj, Insung Ihm, and Sanghun Park. 3d rgb image compression for interactive applications. *ACM Trans. Graph.*, 20(1):10–38, 2001.

[3] S. Behrendt, C. Colditz, O. Franzke, J. Kopf, and O. Deussen. Realistic real-time rendering of landscapes using billboard clouds. *Computer Graphics Forum*, 24(3):507–516, 2005.

[4] B. Chopard and M. Droz. *Cellular Automata Modeling of Physical Systems*. Cambridge Univ. Press, Cambridge, UK, 1998.

[5] D. d'Humières, P. Lallemand, and U. Frisch. Lattice gas models for 3d hydrodynamics. *Europhysics Letters*, 2:291–297, 1986.

[6] Andreas Dietrich, Carsten Colditz, Oliver Deussen, and Philipp Slusallek. Realistic and Interactive Visualization of High-Density Plant Ecosystems . In *Eurographics Workshop on Natural Phenomena*, pages 73–81, Dublin, Ireland, 2005.

[7] R. Fernando and M. Kilgard. *The Cg Tutorial*. Addison Wesley, Boston, MA, 2003.

[8] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, 2005.

[9] Jeppe Revall Frisvad, Niels Jorgen Christensen, and Henrik Wann Jensen. Computing the scattering properties of participating media using lorenz-mie theory. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, pages 60–1 – 60–10, 2007.

[10] A. Fujimoto, Takayuki Tanaka, and K. Iwata. Arts: accelerated ray-tracing system. pages 148–159, 1988.

[11] R. Geist, Z. Jones, and J. Steele. Parallel processing flow models on desktop hardware. In *Proc. of the* 46^{th} *Annual ACM SE Conf.*, pages 417 – 422, Auburn, Alabama, March 2008.

[12] R. Geist, K. Rasche, J. Westall, and R. Schalkoff. Lattice-boltzmann lighting. In *Rendering Techniques 2004 (Proc. Eurographics Symposium on Rendering)*, pages 355 – 362,423, Norrkőping, Sweden, June 2004.

[13] R. Geist and J. Steele. A lighting model for fast rendering of forest ecosystems. In *IEEE/EG Symposium on Interactive Ray Tracing 2008*, pages 99–106, and back cover. IEEE/EG, Aug 2008.

[14] R. Geist, J. Steele, and J. Westall. Convective clouds. In *Natural Phenomena 2007 (Proc. of the Eurographics Workshop on Natural Phenomena)*, pages 23 – 30, 83, and back cover, Prague, Czech Republic, September 2007.

[15] Kyle Hegeman, Simon Premože, Michael Ashikhmin, and George Drettakis. Approximate ambient occlusion for trees. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 87–92, New York, NY, USA, 2006. ACM.

[16] G. Henyey and J. Greenstein. Diffuse radiation in the galaxy. *Astrophysical Journal*, 88:70–73, 1940.

[17] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[18] Warren Hunt and William R. Mark. Ray-specialized acceleration structures for ray tracing. In *IEEE/EG Symposium on Interactive Ray Tracing 2008*, pages 3–10. IEEE/EG, Aug 2008.

[19] Insung Ihm and Sanghun Park. Wavelet-based 3d compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18:3–15, 1999.

[20] H. Jensen, S. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. In *Proceedings of SIGGRAPH 2001*, pages 511–518, August 2001.

[21] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A.K. Peters, Natick, MA, 2001.

[22] Alan Knapp and Gregory Carter. Variability in leaf optical properties among 26 species from a broad range of habitats. *American Journal of Botany*, 85(7):940–946, 1998.

[23] Ares Lagae and Philip Dutré. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(4):1235–1244, June 2008.

[24] W. Lorensen and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proc. SIGGRAPH '87*, pages 163–169, 1987.

[25] T. Luft, M. Balzer, and O. Deussen. Expressive illumination of foliage based on implicit surfaces. In *Natural Phenomena 2007 (Proc. of the Eurographics Workshop on Natural Phenomena)*, pages 71 – 78, Prague, Czech Republic, September 2007.

[26] Shigeru Muraki. Approximation and rendering of volume data using wavelet transforms. In *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pages 21–28, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[27] Tomas Mller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics tools*, 2(1):21–28, 1997.

[28] Ky Giang Nguyen and Dietmar Saupe. Rapid high quality compression of volume data for visualization. *Computer Graphics Forum*, 20:49–56, 2001.

[29] NVIDIA Corp. Nvidia cuda programming guide, version 2.1. http://www.nvidia.com/object/cuda_get.html, December 2008.

[30] Greenworks Organic-Software. Xfrogplants v 2.0. http://www.xfrogdownloads.com /greenwebNew/products/productStart.htm, 2008.

[31] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[32] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, pages 268–277, 2005.

[33] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 313–322, New York, NY, USA, 1985. ACM.

[34] X. Shan and G. Doolen. Multicomponent lattice-boltzmann model with interparticle interaction. *J. of Statistical Physics*, 81(1/2):379–393, 1995.

[35] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, 2002.

[36] Jos Stam. Multiple scattering as a diffusion process. In *Proc.* 6^{th} *Eurographics Workshop on Rendering*, pages 51–58, Dublin, Ireland, June 1995.

[37] Eric J. Stollnitz, Tony D. Derose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann Pulishers, Inc., San Francisco, CA, 1996.

[38] Ingo Wald. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*, 2004.

[39] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3):485–493, 2006.

[40] Lifeng Wang, Wenle Wang, Julie Dorsey, Xu Yang, Baining Guo, and Heung-Yeung Shum. Real-time rendering of plant leaves. *ACM Trans. Graph.*, 24(3):712–719, 2005.

[41] Xiaoming Wei, Student Member, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10:164–176, 2004.

[42] Ruediger Westermann. A multiresolution framework for volume rendering. In *Symposium on Volume Visualization*, pages 51–58. ACM Press, 1994.

[43] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.