Radiation Transfer Computation

for Hewlett-Packard Scalable Visualization Array

Dec, 2008

Budapest University of Technology and Economics

Department of Control Engineering and Information Technology

Contents

Chapter 1

Introduction

The multiple-scattering simulation in participating media is one of the most challenging problems in computer graphics, radiotherapy, PET/SPECT reconstruction, etc. Such simulation should solve the *radiative transport equation* that expresses the change of radiance $L(\vec{x}, \vec{\omega})$ at point $\vec{x}$ and in direction $\vec{\omega}$:

$$\vec{\omega} \cdot \vec{\nabla} L = \left. \frac{dL(\vec{x} + \vec{\omega} s, \vec{\omega})}{ds} \right|_{s=0} = -\sigma_t L(\vec{x}, \vec{\omega}) + \sigma_s \int\limits_{\Omega'} L(\vec{x}, \vec{\omega}') P(\vec{\omega}', \vec{\omega}) \, d\omega', \qquad (1.1)$$

where σ_t is the *extinction coefficient* describing the probability of collision in a unit distance. When collision happens, the photon is either scattered or absorbed, so the extinction coefficient is broken down to *scattering coefficient* σ_s and *absorption coefficient* σ_a:

$$\sigma_t = \sigma_a + \sigma_s.$$

The probability of reflection given that collision happened is called the *albedo* of the material:

$$a = \frac{\sigma_s}{\sigma_t}.$$

If reflection happens, the probability density of the reflected direction is defined by *phase function* $P(\vec{\omega}', \vec{\omega})$. The extent of anisotropy is usually expressed by the mean cosine of the phase function:

$$g = \int\limits_{\Omega'} (\vec{\omega}' \cdot \vec{\omega}) P(\vec{\omega}' \cdot \vec{\omega}) \, d\omega'.$$

In homogeneous media volume properties σ_t, σ_s, and $P(\vec{\omega}', \vec{\omega})$ do not depend on position $\vec{x}$. In inhomogeneous media these properties depend on the actual position.

In case of measured data, material properties are usually stored in a 3D voxel grid, and are assumed to be constant or linear between voxel centers. If the diameter of the region represented by a voxel is Δ, then the total extinction is worth representing by a new parameter that is called the *opacity* and is denoted by α:

$$\alpha = 1 - e^{-\sigma_t \Delta} \approx \sigma_t \Delta. \qquad (1.2)$$

Radiance $L(\vec{x}, \vec{\omega})$ is often expressed as a sum of two terms, the *direct term* L_d that represents unscattered light, and the *media term* L_m that stands for the light component that scattered at least once:

$$L(\vec{x}, \vec{\omega}) = L_d(\vec{x}, \vec{\omega}) + L_m(\vec{x}, \vec{\omega}).$$

The direct term is reduced by out-scattering:

$$\frac{dL_d}{ds} = -\sigma_t L_d.$$

The media term is not only reduced by out-scattering, but also increased by in-scattering:

$$\frac{dL_m}{ds} = -\sigma_t L_m + \sigma_s \int\limits_{\Omega'} (L_d + L_m) P(\vec{\omega}', \vec{\omega})\, d\omega'.$$

Note that this equation can be re-written by considering the reflection of the direct term as a *volumetric source*:

$$\frac{dL_m}{ds} = -\sigma_t L_m + \sigma_s \int\limits_{\Omega'} L_m P(\vec{\omega}', \vec{\omega})\, d\omega' + Q(\vec{x}, \vec{\omega}), \tag{1.3}$$

where the source intensity is

$$Q(\vec{x}, \vec{\omega}) = \sigma_s \int\limits_{\Omega'} L_d(\vec{x}, \vec{\omega}') P(\vec{\omega}', \vec{\omega})\, d\omega'.$$

The volumetric source represents the coupling between the direct and media terms.

Although the direct term can be expressed as an integral even in inhomogeneous medium, the evaluation of this integral requires ray marching and numerical quadrature. Having obtained the direct term, and transforming it to the volumetric source, the media term needs to be computed.

Cerezo et al. [3] classified algorithms solving the transfer equation as analytic, stochastic, and iterative.

Analytic techniques rely on simplifying assumptions, such as that the volume is homogeneous, and usually consider only the single scattering case [2, 17]. Jensen et al. [7] attacked the subsurface light transport by assuming that the space is partitioned into two half spaces with homogeneous material and developed the dipole model. Narasimhan and Nayar [11] proposed a multiple scattering model for optically thick homogeneous media and isotropic light source.

Stochastic methods apply Monte Carlo integration to solve the transport problem [8, 6]. These methods are the most accurate but are far too slow in interactive applications.

Iterative techniques need to represent the current radiance estimate that is refined in each step [4]. The radiance function is specified either by *finite-elements*, using, for example, the zonal method [14], spherical harmonics [8], radial basis functions [19], metaballs, etc. or exploiting the particle system representation [18].

Stam [15] introduced *diffusion theory* to compute energy transport. Here the angular dependence of the radiance is approximated by a two-term expansion:

$$L(\vec{x}, \vec{\omega}) \approx \tilde{L}(\vec{x}, \vec{\omega}) = \frac{1}{4\pi}\phi(\vec{x}) + \frac{3}{4\pi}\vec{E}(\vec{x}) \cdot \vec{\omega}.$$

By enforcing the equality of the directional averages of L and $\tilde{L}$, we get the following equation for *fluence* $\phi(\vec{x})$:

$$\phi(\vec{x}) = \int\limits_{\Omega} L(\vec{x}, \vec{\omega})\, d\omega.$$

Requiring $\int\limits_{\Omega} L(\vec{x},\vec{\omega})\vec{\omega}\, d\omega = \int\limits_{\Omega} \tilde{L}(\vec{x},\vec{\omega})\vec{\omega}\, d\omega$, we obtain *vector irradiance* $\vec{E}(\vec{x})$ as

$$\vec{E}(\vec{x}) = \int\limits_{\Omega} L(\vec{x},\vec{\omega})\vec{\omega}\, d\omega.$$

Substituting this two-term expansion into the radiative transfer equation and averaging it for all directions, we obtain the following diffusion equations:

$$\vec{\nabla}\phi(\vec{x}) = -3\sigma_t'\vec{E}(\vec{x}), \quad \vec{\nabla}\cdot\vec{E}(\vec{x}) = -\sigma_a\phi(\vec{x}). \tag{1.4}$$

where $\sigma_t' = \sigma_t - \sigma_s g$ is the *reduced extinction coefficient*.

In [15] the diffusion equation is solved by either a multi-grid scheme or a finite-element blob method. Geist et al. [5] computed multiple scattering as a diffusion process, using a lattice-Boltzmann solution method.

In order to speed up the solutions to interactive rates, the transport problem is often simplified and the solution is implemented on the GPU. The *translucent rendering* approach [9] involves multiple scattering simulation, but considers only multiple approximate forward scattering and single backward scattering. This method aims at nice images instead of physical accuracy. Physically based global illumination methods, like the photon map, have also been used to solve the multiple scattering problem [13]. To improve speed, light paths were sampled on a finite grid.

The high computational burden of multiple scattering simulation has been attacked by parallel methods both in surface [1] and volume rendering [16]. Parallel volume rendering methods considered the visualization of very large datasets, while interactive multiple scattering simulation has not been in the focus yet. Stochastic methods scale well on parallel systems, so they would be primary candidates for parallel machines, but their convergence rate is still too slow. Iterative techniques, on the other hand, converge faster but require data exchanges between the nodes, which makes scalability sub-linear.

This paper presents a fast parallel solution for the radiative transfer equation (Figure 1.1). We have taken the iteration approach because of its better convergence. This posed challenges for the parallel implementation because we should attack the sub-linear scalability and the communication bottleneck. Our approach is based on two recognitions. Iteration is slow because it requires many "warming up" steps to distribute the power of sources to far regions. Thus, if we can find an easy way to approximate the solution, then iteration should only refine the initial approximation, which could be done in significantly fewer steps. On the other hand, iteration requires the exchange of data from all computing nodes in each step, which leads to a communication bottleneck. We propose an iteration scheme when the data are exchanged less frequently. This slows down the convergence of the iteration, so computing nodes should work longer, but reduces the communication load.

We shall assume that the primary source of illumination is a single point light source in the origin of our coordinate system. More complex light sources can be modeled by translation and superposition. We use a simple and fast technique to initially distribute the light in the medium. The distribution is governed by the diffusion theory, where the single pass approximate solution is made possible by assumptions that the medium is locally homogeneous and spherically symmetric. The solution is approximate but can be obtained at the same cost as the direct term. Having obtained the initial approximation, the final solution is computed by iteration on a GPU cluster.

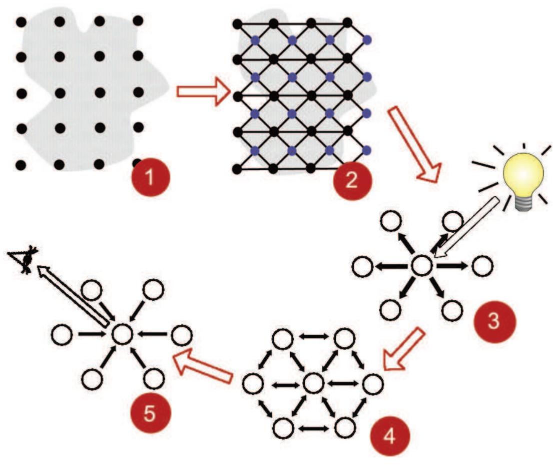


Figure 1.1: The outline of the algorithm. 1: The volume is defined on a grid. 2: An illumination network is established between the grid points. 3: Single scattering and estimated multiple scattering are distributed from each light source. 4: The final results are obtained by iteration which corrects the errors of the estimation. 5: The image is rendered by standard alpha blending.

This paper is organized as follows. Section 2 explains the iteration solution, the importance of having a good initial approximation, and the challenges of parallel execution. Section 3 discusses the computation of the direct term. Section 4 presents the initial estimation of the radiance. Section 5 deals with the iterative refinement. Section 6 presents our distributed implementation and Section 7 summarizes the results.

Chapter 2

Iteration solution of the radiative transfer equation

The transport equation is an integro-differential equation. Integrating both sides, the equation can be turned to an integral equation of the following form:

$$L = \mathcal{T} L + Q^e$$

where $\mathcal{T}$ is a linear integral operator and Q^e is the source term provided by the boundary conditions. Applying finite-element techniques, the continuous radiance function is represented by finite data, which turns the integrated transfer equation to a system of linear equations:

$$\mathbf{L} = \mathbf{T} \cdot \mathbf{L} + \mathbf{Q}^e,$$

where vector $\mathbf{L}$ is the radiance of the sample locations and directions, $\mathbf{Q}^e$ is the vector of source terms and boundary conditions, and $\mathbf{T}$ is the transport matrix.

Iteration obtains the solution as the limiting value of the following iteration sequence

$$\mathbf{L}_n = \mathbf{T} \cdot \mathbf{L}_{n-1} + \mathbf{Q}^e$$

so if this scheme is convergent, then the solution can be obtained by starting with an arbitrary radiance distribution $\mathbf{L}_0$ and iteratively repeating operator $\mathbf{T}$. The convergence is guaranteed if $\mathbf{T}$ is a contraction, i.e. for the norm of this matrix, we have

$$||\mathbf{T} \cdot \mathbf{L}|| < \lambda ||\mathbf{L}||, \quad \lambda < 1,$$

which is the case if the albedo is less than 1.

The error at a particular step n can be found by subtracting the solution $\mathbf{L}$ from the iteration scheme, and applying the definition of the contraction iteratively:

$$||\mathbf{L}_n - \mathbf{L}|| = ||\mathbf{T} \cdot (\mathbf{L}_{n-1} - \mathbf{L})|| < \lambda ||\mathbf{L}_{n-1} - \mathbf{L}|| < \lambda^n ||\mathbf{L}_0 - \mathbf{L}||.$$

Note that the error is proportional to the norm of the difference of the initial guess $\mathbf{L}_0$ and the final solution $\mathbf{L}$. Thus, having a good initial guess that is not far from the solution, the error after n iteration steps can be significantly reduced.

2.1 Iteration on parallel machines

In order to execute the iteration on a parallel machine, the radiance vector $\mathbf{L}_n$ is broken to parts and each computing node is responsible for the update of its own part. However, the new value of a part also depends on other parts, which would necessitate state exchanges between the nodes in every iteration. This would quickly make the communication the bottleneck of the parallel computation.

This problem can be attacked by not exchanging the current state in every iteration cycle. Suppose, for example, that we exchange data just in every second iteration cycle. When the data is exchanged before executing the matrix-vector multiplication, the iteration looks like the original formula:

$$\mathbf{L}_n = \mathbf{T} \cdot \mathbf{L}_{n-1} + \mathbf{Q}^e.$$

However, when the data is not exchanged, a part of the matrix is multiplied by the radiance estimate of the older iteration. Let us denote the matrix by $\mathbf{T}^*$ that is similar to $\mathbf{T}$ where the own part is multiplied and zero elsewhere. With this notation, the cycle without previous data exchange is:

$$\mathbf{L}_n = \mathbf{T}^* \cdot \mathbf{L}_{n-1} + (\mathbf{T} - \mathbf{T}^*) \cdot \mathbf{L}_{n-2} + \mathbf{Q}^e.$$

Putting the two equations together, the execution of an iteration without state changes and then an iteration with state changes would result in:

$$\mathbf{L}_n = \mathbf{T}^2 \cdot \mathbf{L}_{n-2} + \mathbf{T} \cdot \mathbf{Q}^e + \mathbf{Q}^e + \mathbf{T} \cdot (\mathbf{T} - \mathbf{T}^*) \cdot (\mathbf{L}_{n-3} - \mathbf{L}_{n-2}).$$

Note that if this scheme is convergent, then $\mathbf{L}_n$, $\mathbf{L}_{n-2}$, and $\mathbf{L}_{n-3}$ should converge to the same vector $\mathbf{L}$, thus the limiting value satisfies the following equation:

$$\mathbf{L} = \mathbf{T}^2 \cdot \mathbf{L} + \mathbf{T} \cdot \mathbf{Q}^e + \mathbf{Q}^e.$$

This equation is equivalent to the original equation, which can be proven if the right side's $\mathbf{L}$ is substituted by the complete right side:

$$\mathbf{L} = \mathbf{T} \cdot \mathbf{L} + \mathbf{Q}^e = \mathbf{T} \cdot (\mathbf{T} \cdot \mathbf{L} + \mathbf{Q}^e) + \mathbf{Q}^e.$$

The price of not exchanging the data in every iteration step is the additional error term $\mathbf{T} \cdot (\mathbf{T} - \mathbf{T}^*) \cdot (\mathbf{L}_{n-3} - \mathbf{L}_{n-2})$. This error term also converges to zero, but slows down the iteration process especially at the beginning of the iteration.

Using the same argument, we can prove a similar statement for cases when the data is exchanged just in every third, fourth, etc. cycles. The number of iterations done by the nodes between data exchanges should be specified by finding an optimal compromise, which depends on the relative computation and communications speeds.

Chapter 3

Initial radiance distribution

3.1 Direct term

The direct term is reduced by out-scattering. As the source is in the origin, the direct term is non-zero only for the direction from the origin to the considered point. Let us consider a point at distance r on a beam started at the source and having solid angle $\Delta\Omega$, and step on this beam by dr. As a photon collides with the medium with probability $\sigma_t(r)dr$ during the step, the *radiant intensity* (i.e. the power per solid angle) $\Phi(r)$ at distance r satisfies the following equation

$$\Phi(r+dr) = \Phi(r) - \sigma_t(r)I(r) \implies \frac{d\Phi(r)}{dr} = -\sigma_t(r)I(r). \tag{3.1}$$

If the radiant intensity is Φ_0 at the source, then the solution of this equation is

$$\Phi(r) = \Phi_0 e^{-\int_0^r \sigma_t(s)ds}.$$

The *radiance* is the power per differential solid angle and differential area. In our beam the power is the product of radiant intensity $\Phi(r)$ and solid angle $\Delta\Omega$. On the other hand, the solid angle in which the source is visible equals to zero, which introduces a Dirac delta in the radiance formula. The area at distance r grows as $\Delta A = \Delta\Omega r^2$. Thus, the radiance of the direct term is

$$L_d(\vec{x},\vec{\omega}) = \frac{\Phi(r)\Delta\Omega}{\Delta\Omega r^2}\delta(\vec{\omega}-\vec{\omega}_{\vec{x}}) = \frac{\Phi(r)}{r^2}\delta(\vec{\omega}-\vec{\omega}_{\vec{x}}), \tag{3.2}$$

where $r = |\vec{x}|$ is the distance and $\vec{\omega}_{\vec{x}} = \vec{x}/|\vec{x}|$ is the direction of the point from the source.

3.2 Initial distribution of the estimated radiance

Let us consider just a single beam starting at the origin where the point source of radiant intensity Φ is. When a beam is processed, we shall assume that other beams face to the same material characteristics, i.e. we assume that the scene is *spherically symmetric*. Note that the assumption on spherical symmetry does not mean that only one beam is processed. We take many beams originating from the source, and each of them are traced independently assuming that other rays face the same material properties as the current beam.

In case of spherical symmetry, the radiance of the inspected beam at point $\vec{x}$ and in direction $\vec{\omega}$ may depend just on distance $r = |\vec{x}|$ from the origin and on angle θ between direction $\vec{\omega}$ and the direction of point $\vec{x}$. The unit direction vector of point $\vec{x}$ will be denoted by $\vec{\omega}_{\vec{x}} = \vec{x}/|\vec{x}|$. This allows parametrization $L(r, \theta)$ instead of $L(\vec{x}, \vec{\omega})$. The fluence depends just on distance r and vector irradiance $\vec{E}(\vec{x})$ has the direction of the given point, that is $\vec{E}(\vec{x}) = E(r)\vec{\omega}_{\vec{x}}$.

Expressing the divergence operator in spherical coordinates, we get:

$$\vec{\nabla} \cdot \vec{E}(\vec{x}) = \vec{\nabla} \cdot (E(r)\vec{\omega}_{\vec{x}}) = \frac{1}{r^2}\frac{\partial(r^2 E(r))}{\partial r}.$$

Thus, the scalar versions of the diffusion equations are:

$$\frac{d\phi(r)}{dr} = -3\sigma_t' E(r), \quad \frac{1}{r^2}\frac{d(r^2 E(r))}{dr} = -\sigma_a \phi(r). \tag{3.3}$$

If we have a point light source, then this equation has a singularity at $r = 0$ where the radiance gets infinite. To solve this problem, we rewrite the equations to use power ψ instead of the fluence. In case of spherical symmetry, at distance r the power is computed on area $4r^2\pi$, thus the correspondences between the fluence and the vector irradiance with the power measures are $\psi_0 = r^2\phi$ and $\psi_1 = r^2 E$ (note that the correspondence between the radiance and the fluence already includes the 4π factor). Substituting these into the differential equation we obtain:

$$\frac{d\psi_0(r)}{dr} = \frac{2}{r}\psi_0 - 3\sigma_t'\psi_1(r), \quad \frac{d\psi_1(r)}{dr} = -\sigma_a\psi_0(r). \tag{3.4}$$

For homogeneous infinite material, the differential equation can be solved analytically:

$$\psi_0^h(r) = Ae^{-\sigma_e r}r,$$

$$\psi_1^h(r) = \frac{2}{3r\sigma_t'}\psi_0(r) - \frac{1}{3\sigma_t'}\frac{d\psi_0(r)}{dr} = \frac{A}{3\sigma_t'}e^{-\sigma_e r}(\sigma_e r + 1). \tag{3.5}$$

where $\sigma_e = \sqrt{3\sigma_a\sigma_t'}$ is the *effective transport coefficient*, and A is an arbitrary constant that should be determined from the boundary conditions. According to the first equation $\psi_0(0) = 0$, thus only the second equation is free at the boundary. The radiant intensity of the source is Φ, which is made equal to the vector irradiance:

$$\psi_1(0) = \frac{A}{3\sigma_t'} = \Phi \quad \Longrightarrow \quad A = 3\sigma_t'\Phi.$$

3.2.1 Initial approximation with wavefront tracing

With equation 3.4 we established two differential equations that describe the power evolving as we move along a ray started at the origin. These equations can be solved by numerical integration while marching on the ray and taking samples from the material properties $\sigma_t(r)$ and $\sigma_s(r)$.

In order to obtain the initial values, we take the solution for homogeneous material:

$$\psi_0(0) = \psi_0^h(0) = 0, \quad \psi_1(0) = \psi_1^h(0) = 3\sigma_t'\Phi.$$

Care should be practiced when starting the iteration. At the beginning $\psi_0 = 0$ and $r = 0$, so when evaluating $\frac{2}{r}\psi_0$, we have a $0/0$ type undefined value. Using again the solution of the homogeneous case

$$\lim_{r \to 0} \frac{2}{r}\psi_0 = \lim_{r \to 0} \frac{2}{r} 3\sigma_t' \Phi e^{-\sigma_e r} r = 6\sigma_t' \Phi.$$

As ray marching proceeds taking steps Δ increasing distance r, material properties σ_t, σ_s, and g are fetched at the sample location, and state variables $\psi_0[n]$, and $\psi_1[n]$ are updated according to the numerical quadrature, resulting in the following iteration formula for step n:

$$\begin{aligned} \psi_0[n] &= \psi_0[n-1]\left(1 + \frac{2\Delta}{r}\right) - 3\sigma_t' \psi_1[n-1]\Delta, \\ \psi_1[n] &= \psi_1[n-1] - \sigma_a \psi_0[n-1]\Delta. \end{aligned} \tag{3.6}$$

Chapter 4

Refinement of the initial solution by iteration

At the end of the approximate radiance distribution we have good estimates for the direct term L_d and volumetric source

$$Q(\vec{x},\vec{\omega}) = \sigma_s \int\limits_{\Omega'} L_d(\vec{x},\vec{\omega}')P(\vec{\omega}',\vec{\omega})\,d\omega' = \frac{\Phi(r)}{r^2}\sigma_s P(\vec{\omega}_{\vec{x}},\vec{\omega}),$$

and probably less accurate estimates for the total radiance

$$L(\vec{x},\vec{\omega}) \approx \frac{1}{4\pi}\phi(\vec{x}) + \frac{3}{4\pi}E(\vec{x})(\vec{\omega}_{\vec{x}}\cdot\vec{\omega}).$$

Thus, we can accept direct term L_d, but the media term $L_m = L - L_d$ needs further refinement. We use an iteration scheme to make the media term more accurate, which is based on equation 1.3, but considers only the voxel centers. The *incoming medium radiance* arriving at voxel p from direction $\vec{\omega}$ is denoted by $I_m^{(p)}(\vec{\omega})$. Similarly, the *outgoing medium radiance* is denoted by $L_m^{(p)}(\vec{\omega})$. Using these notations, the discretized version of equation 1.3 at voxel p is:

$$L_m^p(\vec{\omega}) = (1-\alpha^p)I_m^p(\vec{\omega}) + \alpha^p a^p \int\limits_{\Omega'} I_m^p(\vec{\omega}')P^p(\vec{\omega}',\vec{\omega})\,d\omega' + Q^p(\vec{\omega}) \tag{4.1}$$

since $\sigma_t\Delta \approx \alpha$ and $\sigma_s\Delta \approx \alpha a$.

The incoming radiance of a voxel is equal to the outgoing radiance of another voxel that is the neighbor in the given direction, or it is set explicitly by the boundary conditions. Since in the discretized model a voxel has just finite number of neighbors, the in-scattering integral can also be replaced by a finite sum:

$$\int\limits_{\Omega'} I^p(\vec{\omega}')P^p(\vec{\omega}',\vec{\omega})\,d\omega' \approx \frac{4\pi}{D}\sum_{d=1}^{D} I^p(\vec{\omega}'_d)P^p(\vec{\omega}'_d,\vec{\omega}).$$

where D is the number of neighbors, which are in directions $\vec{\omega}'_1,\ldots,\vec{\omega}'_D$ with respect to the given voxel. The number of neighbors depends on the structure of the grid. In a conventional, so called *Body Centered Cubic grid* a voxel has only 6 neighboring

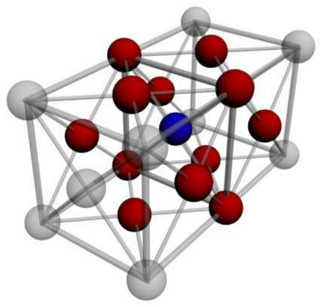


Figure 4.1: Structure of the Face Centered Cubic Grid. Grid points are the voxel corners, voxel centers, and the centers of the voxel faces. Here every grid point has 12 neighbors, all at the same distance.

voxels that share a face, which seems to be too small to approximate a directional integral. Thus, it is better to use a *Face Centered Cubic grid* (FCC grid) [13], where each voxel has $D = 12$ neighbors (Figure 4.1).

Note that unknown radiance values appear both on the left and the right sides of the discretized transfer equation. If we have an estimate of radiance values (and consequently, of incoming radiance values), then these values can be inserted into the formula of the right side and a new estimate of the radiance can be provided. Iteration keeps repeating this step. If the process is convergent, then in the limiting case the formula would not alter the radiance values, which are therefore the roots of the equation.

Chapter 5

Implementation

The system has been implemented on a 5 node HP Scalable Visualization Array (SVA), where each node is equipped with an NVIDIA GeForce 8800 GTX GPU, programmed under CUDA. The nodes are interconnected by Infiniband.

During iterational refinement separate kernels are executed on the GPU for each computational step.

The following section provides a brief overview of the programming platform, then we discuss the implementation details.

5.1 The CUDA programming environment

A CUDA program consists of two parts: the conventional program on the CPU is the host code, the one running on the GPU (or device) is the kernel code. Both the host and the kernel code has its own DRAM memory space in the system memory and the GPU memory. By transferring data between these two storages the communication between the two components is possible. The graphical hardware executes each kernel in large number of parallel threads which run independently from each other and are able to synchronize in a limited way. These threads, fitted to the SIMD (Single Instruction Multiple Data) architecture of the hardware, are organized in a special scheduling structure.

The individual threads are grouped into thread blocks. A given block is executed on the same SIMD unit of he hardware, so its threads can communicate with each other through the internal memory of the processor. This is the so-called shared memory. The number of threads in a block is constrained by the number of registers which is divided among them. The GPU is capable of running thousands of threads by organizing the same sized blocks in a grid. The blocks of the grid are evenly distributed among the SIMD processors on the device. Since having no common memory space synchronization is not possible between the threads of different blocks. Scheduling adapts to the number of the SIMD units on the device, thus provides excellent scalability.

In the kernel code one can index the current thread with two built-in variables. Each block has its own *block ID*, which identifies it in the grid. Each thread running in the block has a *thread ID*. The execution order of the different blocks, and the thread warps in the blocks is undefined and should be considered random.

Finally let us get familiar shortly with the memory model of CUDA. The GPU has own DRAM memory, in CUDA terminology this storage space is the *device memory*.

Data here can be accessed from every thread, but the memory latency is a serious bottleneck. Therefore the programmer is provided with other memory types which have a limited size. As it was mentioned before, each SIMD processor is equipped wich a very fast integrated *shared memory*. This space is divided among the executing blocks and the communication of the threads can be realized here. CUDA can exploit the special texturing units of the GPU as well. *Textures* are in fact stored in the device memory, but accessing is speeded up with a cache memory. A special part of the storage space is the *constant memory* which gets copied into the constant cache, so reading from here is also very fast.

The performance of the application is therefore highly determined by the decisions between different memory spaces for the data used by the kernel part. The host has full access to every memory types except the shared memory, but because of caching textures and constant memory is read-only from the kernel.

5.2 Multiple scattering with iteration

5.2.1 Modeling the FCC lattice on the GPU

The iterative simulation models the radiance distribution in the media by sampling the volume according to an FCC lattice and transferring photons between the neighboring lattice sites along the edges in the iteration steps. For that a compound data structure had to be created which implements the FCC lattice and fulfil the following requirements:

- it can be indexed in 3D in order to store the data in a 3D array,
- a neighborhood function must be implemented: for a given element with index i, j, k we must determine i', j', k' which is the neighbor of the element in a given direction in the lattice,
- The Cartesian coordinates of the element must be easily calculated from its index.

Figure 5.1 illustrates the concept of the chosen indexing method. The FCC lattice from a CC grid is prepared by first doubling the resolution of each slice of the grid along the z axis. Then the elements of which the sum of the x an y index is odd (a chessboard pattern) is selected and shifted 0.5 units along the z axis.

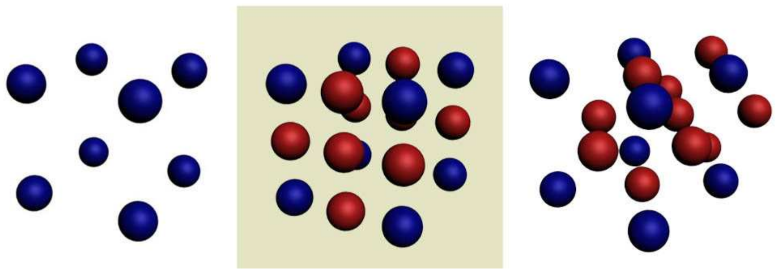


Figure 5.1: The construction of the FCC grid.

This method creates a $2N \times 2M \times L$ resolution FCC grid from an $N \times M \times L$ CC grid, and leave the original elements in place. In this data structure:

- the Cartesian coordinates of an element index $[i,j,k]$ (if the CC grid had a unit length voxel size):

$$x = i, \quad y = j, \quad z = k + \frac{(i+j) \mod 2}{2}.$$

- determining the nth neighbor of an element $[i,j,k]$:
  A *relIndices* look-up table was created in the constant memory of the GPU. The nth element of that will be the relative index of the element visible in the nth discrete direction. The indices of the directions are chosen so that 0 is the first direction, and the opposite of the nth direction is the one with index $11-n$.

5.2.2 Simulation kernel: iteration

In the FCC grid the grid points are interconnected with 12 incoming and 12 outgoing directed edges which encode the radiance transfer between the neighbors in the discrete directions. To store the current state of the grid it is enough to store only the incoming or the outgoing radiances for each grid point because the radiances in the opposite direction can be obtained from the neighbors. In this implementation the array of outgoing radiance values is maintained because it was more memory access efficient.

The radiance distribution for one wavelength in the FCC grid is represented with 12 floating point arrays — one for each discrete direction in the grid. The FCC sites can be mapped into a standard 3D array by using proper indexing, where each value means the outgoing radiance from a given grid site in one direction. The volumetric source values remain constant during the iteration, so we store them in separate 3D textures. The iteration kernel updates the state of the grid by reading the emissions and the incoming radiances from the neighboring grid sites. Therefore it is more efficient to store the previous state of the grid and the emissions in textures to utilize caching. The output of an iteration step is the input of the following one, so we copy the results back to the input textures after each kernel execution. In order to improve performance, we introduced a sensitivity constant which is a lower bound to the sum of the incoming radiances for each point. We evaluate the iteration formula only where the radiance values are greater than this constant. This method is efficient if there are larger parts of the volume without significant irradiance.

```
//alpha
sigma_t = tex3D(alphaTex, spos); //read sigma_t from the alpha texture
alpha = 1.0f - expf(-EDGE_LENGTH*sigma_t);
//get and store the incoming illumination form each discrete direction
for(int t= 0; t<NUM_DIR; t++) {
    DGetNeighbourIndex(i, j, blockIdx.x, t, dx, dy, dz);
    emissions[t] = getEmission(t, normPos.x, normPos.y, normPos.z);
    if(dz >= 0 && dz < depth) {
        inscatterings [t]= getIllumination (NUM_DIR-1-t, dx, dy, dz);
    } else {
        inscatterings [t] = 0;
    }
}
//update the outgoing illuminations in the discrete directions
for(int k=0; k<NUM_DIR; k++) {
    //pointer to the array of illuminations for one direction
    //the outgoing illumination is at least the emission of the voxel
    illuminations [index] = emissions[k];
    //now, we add the contribution of the incoming illumination of the neighbouring voxels
    illuminations [index]+= (1.0f−alpha)∗ inscatterings [NUM_DIR−1−k];
    // in− scattering term
    float inscatter = 0;
    for(int t= 0; t<NUM_DIR; t++) {
        inscatter += inscatterings [t] ∗ phasef(NUM_DIR−1−t, k);
    }
    illuminations [index]+=alpha∗albedo∗(4∗PI / NUM_DIR) ∗ inscatter;
}
```

Listing 5.1: The pseudo code of the iteration kernel for one thread

5.3 Inital radiance distribution

The application is currently capable of simulating the radiance of a single point light source. The effect of other light source types can be approximated by the superposition of multiple point light sources, which functionality is currently unavailable.

5.3.1 Wavefront tracing

In order to execute ray marching parallely for all rays during initial radiance distribution, the volume is resampled to a new grid that is parameterized with spherical coordinates. A voxel of the new grid with (u,v,w) coordinates represents fluence ϕ and vector irradiance E of point

$$\vec{x} = R(w\cos\alpha\sin\theta, w\sin\alpha\sin\theta, w\cos\theta),$$

where

$$\alpha = 2\pi u, \qquad \theta = \arccos(1-2v),$$

and R is the radius of the volume. Note that this parametrization provides uniform sampling in the directional domain. A (u,v) pair encodes a ray, while w encodes the distance from the origin. This texture is processed w-layer by w-layer, i.e. stepping the radius r simultaneously for all rays. In a single step the GPU updates the fluence and the vector irradiance according to equation 3.6.

After the wavefront tracing the direct illumination from the light source is known in a spherical coordinate system. In the next kernel call the program samples the primary scattering of the direct illumination according to the original CC grid, in Cartesian space. During the iteration, these values are regarded as emitted radiance on the lattice edges, therefore they will be added to the scattered radiance in each iteration step. Examining the source code, one can note that the emissions are sampled only on a CC grid, and 12 CUDA arrays are necessary to store them for the whole simulation.

If the user enabled the first estimation, then a third kernel is executed which place initial radiances to the edges of the FCC lattice as well. These inital radiances are calculated by sampling the intensity, `phi0` and `phi1` 3D textures.

```
/*
primData: the pointer to the 3D arrays of size [ resolution x resolution x depth] where the
        attenuated light values will be stored
 resolution : the number of the sample directions will be resolution x resolution
depth: the number of the ray marching steps
stepSize : the length of one ray step in WORLD COORDINATES, so not in normalized space
```

```
*/
__global__ void d_IlluminationSphere (PrimarySimData primdata, float3 omniPos, float
        intensity ,
                                        int resolution , int depth, float stepSize )
{
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    float u = x / (float) resolution ;
    float v = y / (float) resolution ;

    [...] // variable declarations ...

    //we also update the L0 and L1 values for the initial estimation
    //see the documentation for details
    float* data = (float*)primdata.I.ptr;
    float* L0s = (float*)primdata.L0.ptr;
    float* L1s = (float*)primdata.L1.ptr;
    int rowsize = primdata.I.pitch / sizeof(float);

    //ray marching
    //every step updates a value in the 3D texture
    if ((x < resolution) && (y < resolution)) {
        index = rowsize * y + x;

        lightRay.o = omniPos;
        samplePos = omniPos;
        I = intensity ;

        //get the unit step size
        // spherical angles ...
        phi = 2* PI * u;
        theta = acosf(1.0f- 2.0f*v);
        lightRay.d.x = __sinf(theta) * __cosf(phi);
        lightRay.d.y = __cosf(theta);
        lightRay.d.z = __sinf(theta) * __sinf(phi);
        lightRay.d = normalize(lightRay.d);

        // find intersection with box
        if (intersectBox(lightRay, c_volumeBox.minCorner, c_volumeBox.maxCorner, &tnear, &
            tfar)) {
            [...]
            //march along the ray
            for(int i=0; i< depth; i++) {
                samplePos = lightRay.o + distance*lightRay.d;
                sigma_t = tex3D(alphaTex, samplePos.x, samplePos.y, samplePos.z);
                sigma_s = albedo * sigma_t;
                sigma_a = sigma_t - sigma_s;
                sigma_r = sigma_t - g*sigma_s;
            [...]

                if(i == 0) { // initial values
                    phi_0 = 0;
                    phi_1 = intensity ;
                    phi_0_per_r = 3*sigma_r * intensity ; // 3 * sigma_r * I_0
                }

                // store the current intensities
                [...]

                //update phi0, phi1
                if(distance > stepSize) {
                    phi_0_per_r = phi_0 / (distance + 0.025*sigma_t*stepSize);
```

```
                        }
                        d_phi_0 = 2 * phi_0_per_r - 3*sigma_r * phi_1;
                        d_phi_1 =-sigma_a * phi_0;
                        phi_0 += d_phi_0 * stepSize;
                        phi_1 += d_phi_1 * stepSize;
                        if(phi_0 < 0)   phi_0 = 0;
                        if(phi_1 < 0)   phi_1 = 0;

                        // attenuate the light
                        I = I * expf(-sigma_t*stepSize);

                        //advance along the ray
                        distance = distance + stepSize;
                        if( distance > tfar) break;

                        //get the next index in the 3D texture
                        index += rowsize * resolution ;
                    }
                }
            }
        }
```

Listing 5.2: Source code of the wavefront tracing kernel

5.4 Visualization

The task of the visualization kernel is to display the simulation results interactively for the user. The image synthesis is performed by ray-marching. We use transfer functions and alpha blending for rendering, sampling must be done in reverse order, marching back to front on the rays.

The final image is composited of two different information: the first one is the density, the second is the radiance channel. The appearance of those can be fine tuned by two different transfer functions on the user interface (Figure 5.2).

Before ray-marching the FCC grid must be converted back to a CC grid, because we would like to enable 3D texture filtering, and FCC data would give incorrect results. After that we calculate the radiance scattering in the direction of the camera for each CC voxel (final gathering) and we store the result in a 3D texture. During ray-marching both the sampling of this texture and the density texture gives a scalar value, which the transfer functions convert into RGBA values. The transfer functions are stored in GPU memory as 1D textures.

```
__global__ void
d_RayMarch(uint *d_output, uint imageW, uint imageH, int maxSteps, float tstep , float
      displaydensity , float displayillumination )
{
    [...] // declarations

    // calculate eye ray in world space
    eyeRay.o = make_float3(mul(c_invViewMatrix, make_float4(0.0f, 0.0f, 0.0f, 1.0f)));
    eyeRay.d = normalize(make_float3(u, v, -2.0f));
    eyeRay.d = mul(c_invViewMatrix, eyeRay.d);

    // find intersection with box
    if ( intersectBox(eyeRay, c_displayBox.minCorner, c_displayBox.maxCorner, &tnear, &tfar))
        {
```

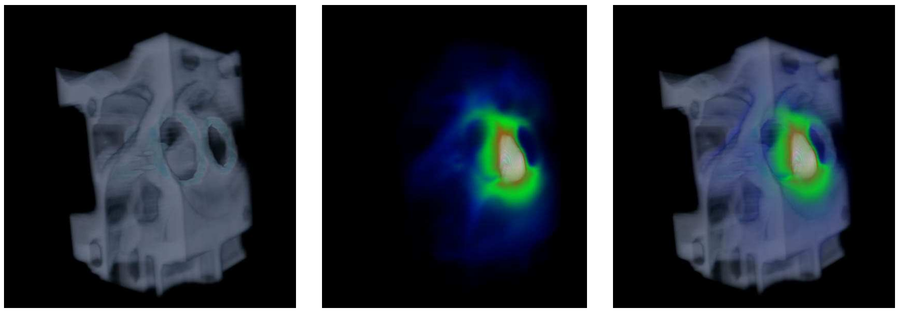


Figure 5.2: The visualization of the result is composited of two channels.

```
        // march along ray from back to front, accumulating color
        float t = tfar;
        for(int i=0; i<maxSteps; i++) {
            float3 pos = eyeRay.o + eyeRay.d*t;
            spos = (pos - c_volumeBox.minCorner) / (c_volumeBox.maxCorner-c_volumeBox.
                minCorner);

            //sample the density value
            illumination = tex3D(alphaTex, spos.x, spos.y, spos.z);
            // look up in transfer function texture
            float4 col = tex1D(transferDensityTex, illumination);

            // accumulate result with alpha blending: density channel
            float a = displaydensity*col.w*stepfactor;
            float4 sumt = lerp(sum, col, a);
            sum.w = a + (1.0f - a) * sum.w;
            sum = make_float4(sumt.x, sumt.y, sumt.z, sum.w);

            [...]   //do the same for the radiance channel ...

            t -= tstep;
            if (t < tnear) break;
        }
    }

    if ((x < imageW) && (y < imageH)) {
        uint i = __umul24(y, imageW) + x;
        d_output[i] = rgbaFloatToInt(sum);
    }
}
```

Listing 5.3: Source code of the visualization kernel for a single thread

Chapter 6

Parallel Implementation

6.1 Distribution of the simulation and visualization

We implemented our algorithm on a GPU cluster. This is a shared memory parallel rendering and compositing environment that uses the *ParaComp library* of the *HP Scalable Visualization Array* [12]. The main aspect of this library is that each host contributes to the pixels of rectangular image areas, called *framelets*. The process of merging the pixels of framelets into a single image is called *compositing*. The ParaComp library allows not only the framelet computation but also the composition to run parallely on the cluster. In our implementation one node takes the role of the master, which means that it has additional tasks beside rendering and compositing. The master sets the proper order of the nodes for composition and displays the composited image on screen. All nodes (including the master) do simulation steps on one portion of the data set, render the subvolume, and contributes this image as one framelet. The composition of framelets is done with alpha blending in a parallel way using the *parallel pipeline algorithm* [10].
The initial radiance distribution is not parallelized since the beams are distributed on spherical layers, which is not compatible with the volume decomposition along a given axis. We could, however, distribute the rays among the nodes, but that would significantly increase the communication overhead.

However, the iteration, visualization, and image compositing are executed in a distributed way.

The tasks are distributed by subdividing the volume along one axis and each node is responsible for both the radiative transfer simulation and the rendering of its associated subvolume.

In addition to solving the iteration formula on the individual subvolumes, we need to implement the radiance transport between the neighboring volume parts. The simulation areas overlap so that the radiance values can be seamlessly passed from one subvolume to the other. MPI communication between the nodes is used to exchange the solutions at the boundary layers. It is important to notice that each node needs to pass only 4 arrays to its appropriate neighbor: the FCC grid has 4 outgoing and 4 incoming directions for each axis-aligned boundaries.

6.2 HP Parallel Compositing Library

The *HP Parallel Compositing Library (ParaComp)* is a *sort-last parallel compositing* API suitable for *hybrid object-space screen-space decomposition*. The API was originally developed by Computational Engineering International (CEI) to make its products run efficiently in a distributed environments. The latest version is based on the abstract Parallel Image Compositing API (PICA) designed by Lawrence Livermore National Lab, HP, and Chromium team.

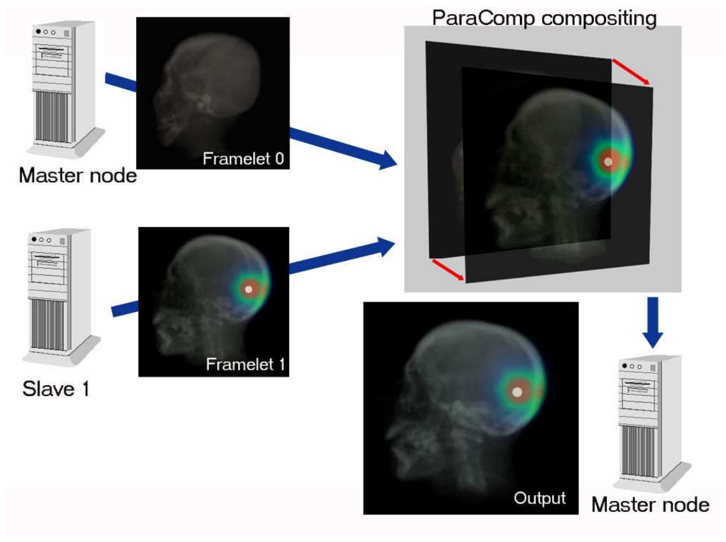


Figure 6.1: The workflow of distributed rendering with ParaComp.

ParaComp is a *message passing* library for graphics clusters enabling users to take advantage of the performance scalability of clusters with network-based pixel compositing without understanding its inner structure and operation. The library makes it possible for multiple graphics nodes in a cluster to collectively produce images, thus significantly larger data sets can be processed and larger images can be created than on any individual graphics hardware by distributing the load over multiple nodes.

However, there is no explicit data distribution so no load balancing is done by the API. The philosophy of the designers is keeping the API as thin as possible. Therefore, only a *global frame* is defined and one or more nodes can contribute pixels to this frame and one or more nodes can receive a specified subset of the frame. ParaComp controls the operation of the nodes based on their request; it takes the results of their renderings and generates the needed composited images. According to the nomenclature of the API a sub-image contribution is called *framelet* and the received image area is called the *output*. These framelets and the outputs can overlap each other without any restriction to their origin or destination nodes. The attributes of a framelet are the following:

- **horizontal and vertical position** in the global frame;
- **width and height** of the framelet in pixels;
- the **data source** which can be both the system memory and the frame buffer; and

- the **depth order** of the framelets which is needed by non-commutative compositing operators like alpha blending.

The size of the output does not necessarily equal the size of the global frame. For example, each tile can be connected to a separate node in a multi-tile display. The attributes of an output are:

- **horizontal and vertical position** in the global frame;
- **width and height** of the output in pixels; and
- the **pixel data** to be returned (RGB, RGBA, RGBA+depth).

For details see the official documentation of the HP Parallel Compositing Library [12].

Chapter 7

Installation and usage

The 3D texture volume rendering application was implemented based on a very thin graphics library called Minimalist OpenGL Environment. This library was designed to handle the common issues of the development of a visualization application with the possibly maximal code reusability. This library has a parallel extension that eases the implementation of a parallel visualization application.

7.1 Installation

Both source and prebuilt versions of the application and the library can be found on the web site of the project[1].

7.1.1 Library dependencies

The following libraries are required by the application:

- **mingle**: Minimalist OpenGL Environment library (version 0.11)
- **mingle-parallel**: the Parallel Rendering extension of MinGLE (version 0.11)
- **paracomp**: Hewlett Packard implementation of the Parallel Compositing API (version 1.0-beta1 or later)
- **devil**: Developer's Image Library (version 1.6.7)
- **glew**: OpenGL Extension Wrangler library (version 1.3.4 or later)
- **gl**: library implementing OpenGL API
- **glu**: OpenGL Utility Library
- **glut**: OpenGL Utility Toolkit
- **glui**: The free OpenGL user interface library

[1]`http://amon.ik.bme.hu/radtransf/`

There are prebuilt packages for HP XC V3.2 RC1 platform for AMD64 architecture on the web site of the project for Developer's Image Library, OpenGL Extension Wrangler, glui, MinGLE, and MinGLE-parallel libraries. If one of them is missing from the target system, it can be installed in the usual way using the `rpm` package manager program:

```
# rpm -i devil-1.6.7-1.x86_64.rpm
# rpm -i devil-devel-1.6.7-1.x86_64.rpm
# rpm -i glew-1.3.4-1.x86_64.rpm
# rpm -i glew-devel-1.3.4-1.x86_64.rpm
# rpm -i mingle-0.11-1.x86_64.rpm
# rpm -i mingle-devel-0.11-1.x86_64.rpm
# rpm -i mingle-parallel-0.11-1.x86_64.rpm
# rpm -i mingle-parallel-devel-0.11-1.x86_64.rpm
```

The `XXX-devel-YYY.rpm` packages are only needed when the volume renderer application is built from sources. Otherwise, only the shared libraries are to be installed.

The other libraries like the Parallel Compositing library, the standard C/C++ libraries, and the OpenGL libraries are platform specific and have to be installed based on the actual software stack.

7.1.2 RPM Package

The application (`radtransf`) can be also installed from a prebuilt RPM[2] package in the same way:

```
# rpm -i radtransf-0.1-1.x86_64.rpm
```

7.1.3 Building from Sources

The build system of the program is based on GNU Autotools. So, it can be built with the usual procedure:

```
$ ./configure --with-inc-dir=<additional include directory> \
 --with-lib-dir=<additional library directory>
$ make
$ sudo make install
```

Since the only implemented parallel rendering support is the HP Parallel Compositing Library, it must be enabled. On a 64-bit HP XC platform the additional path values are the following:

- `<additional include directory>` = `/opt/paracomp/include`
- `<additional library directory>` = `/opt/paracomp/lib64`

MinGLE and MinGLE-parallel libraries can be also built from sources as follows.

[2]Red Hat Package Manager

Building MinGLE from Sources

The build system of Minimalist OpenGL Environment library is also based on GNU Autotools:

```
$ ./configure
$ make
$ sudo make install
```

Currently MinGLE supports only the GLUT windowing system. Hence, OpenGL headers and GLUT headers are needed. MinGLE is customizable, each feature can be disabled in the following way in the configuration step:

```
$ ./configure --disable-glew \
            --disable-devil \
            --disable-freetype
```

However, please note that the application uses OpenGL extensions, therefore OpenGL Extension Wrangler support should not be disabled. Please also note that the application has a graphical user interface that requires font rendering, so Developer's Image Library is also needed. Nevertheless, FreeType support can be disabled if necessary, since the fonts are read from precalculated image files.

Building MinGLE-parallel from Sources

The parallel extension can be built and installed with the following configuration options:

```
$ ./configure \
 --with-inc-dir=<additional include directory> \
 --with-lib-dir=<additional library directory>
$ make
$ sudo make install
```

The meaning of the path options is the same as the volume rendering application.

7.2 Usage

The application can be executed in parallel mode using the SVA subsystem of the visualization XC clusters. The data to be simulated and visualized have to be a raw data file (containing only the data without extra format headers in the file). The data attributes can be described in an additional text file (see Section 7.2.3).

7.2.1 SVA Startup Script

A SLURM[3] startup script is provided to use `radtransf` for parallel rendering. It can be invoked with the following command:

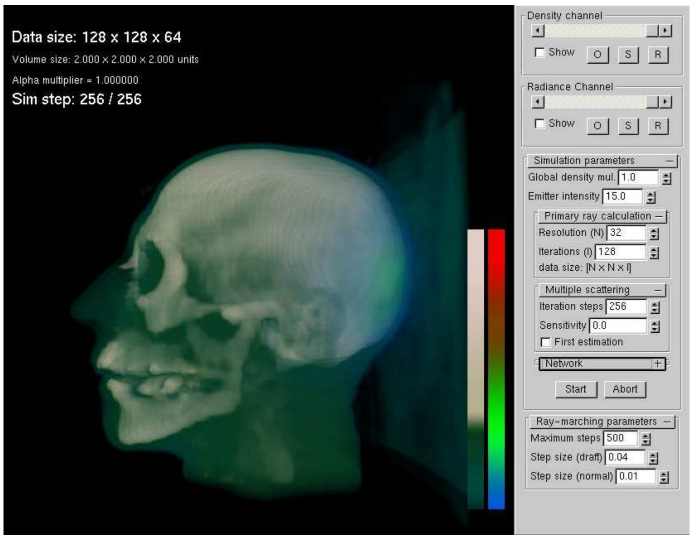
[3]SLURM is an abbreviation for Simple Linux Utility for Resource Management. It is an open-source resource manager designed for Linux clusters of all sizes. This software solution is used for HP-XC clusters.

```
$ radtransf-hpxc.sh -r|--render <renderers> --volume <descriptor-file>
```

The startup script has two parameters that should be set. The first one (`--render`) tells SLURM the number of *additional render nodes* to be allocated. The later one (`--volume`) sets the volume descriptor file.

7.2.2 User Interface

Figure 7.1 shows the graphical user interface of the program. The controls of the GUI are built up with GLUI, the free OpenGL user interface library. Only the master instance has a user interface, and only it can handle user interaction. The instances running on the slave nodes run in full screen mode but do not contain any GLUT callback functions to handle user events. The slaves are notified through MPI by the master when, for instance, the user rotates the camera, or updates a transfer function.


Data size: 128 x 128 x 64
Volume size: 2.000 × 2.000 × 2.000 units
Alpha multiplier = 1.000000
Sim step: 256 / 256

Density channel
[ ] Show

Radiance Channel
[ ] Show

Simulation parameters
Global density mul. 1.0
Emitter intensity 15.0

Primary ray calculation
Resolution (N) 32
Iterations (I) 128
data size: [N × N × I]

Multiple scattering
Iteration steps 256
Sensitivity 0.0
[ ] First estimation

Ray-marching parameters
Maximum steps 500
Step size (draft) 0.04
Step size (normal) 0.01

Figure 7.1: Screenshot of the application.

The main element of the user interface is the viewport, which displays the composited visualization to the user. The navigation in the viewport can be performed using the mouse in the following way:

- the *left button* can be used for *rotating* the scene,
- the *right button* is for *zooming*, and
- the *left* + *right button* can be used for *translating* the scene.

On the right side lay the customization panel where the transfer functions can be adjusted, and the simulation and visualization parameters fine-tuned.

Transfer functions

Probably the most spectacular feature in the user interface are the transfer function controls. The user can customize the color and blending properties of the visualization with them, as illustrated in Figure 7.2.

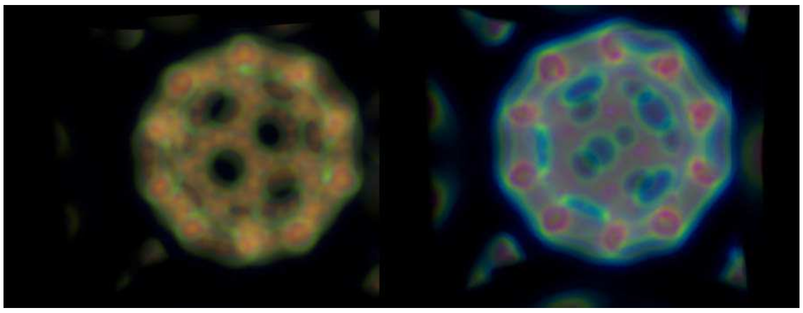


Figure 7.2: The same data set visualized with different transfer functions.

At the top-right corner of the customization panel lay the transfer function controls (Figure 7.3).

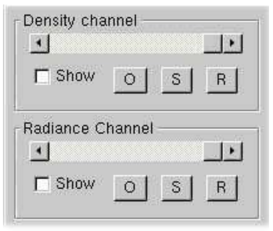


Figure 7.3: The transfer function control group.

- **Density / Radiance Channel Slider:** the user can adjust the strength of the individual display channels with these. For example drag the density slider to the leftmost position, and only the radiance channel will be visible.
- **Show (checkbox):** indicates whether the transfer function tuner controls are visible for the given channel.
- **O (open):** load the last saved transfer functions.
- **S (save):** save the transfer functions to a file.
- **R (reset):** reset the transfer function curves to the default position.

The characteristics of the transfer functions can be adjusted with the *transfer function tuner controls* (Figure 7.4). To make it easier, two vertical color bars are also diplayed in the viewport for the density and radiance channels, where the top is the largest, and the bottom is the smallest value. The user can add new knots to the curves by left-clicking to the appropriate curve, and removing knots by right-clicking. Click and drag a knot to change its position. The visualization interactively updates the transfer characteristics in the meantime.

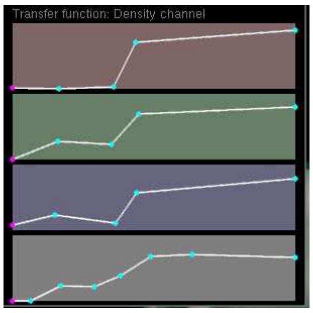
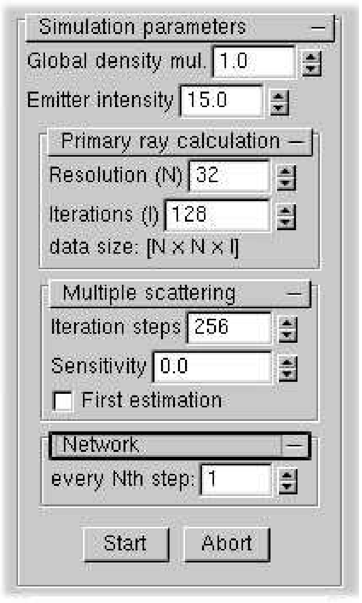


Figure 7.4: The transfer function tuner control The user can separately adjust the transfer curves for the RGBA channels separately.

Simulation parameters



Simulation parameters

Global density mul. 1.0

Emitter intensity 15.0

Primary ray calculation

Resolution (N) 32

Iterations (I) 128

data size: [N x N x I]

Multiple scattering

Iteration steps 256

Sensitivity 0.0

☐ First estimation

Network

every Nth step: 1

Figure 7.5: The Simulation parameters rollout.

- **Global density mul.:** global density multiplier. The density of the volume (to be exact, the extinction coefficient of the volume) will be globally multiplied with this constant.
- **Emitter intensity:** the intensity of the single point emitter in the volume.
- **Resolution** (*N*)**:** the number of rays in the wavefront tracing will be $N \times N$.
- **Iterations** (*I*)**:** the ray-marching of the wavefront tracing will take I steps.
- **Iteration steps:** the number of the iterations in the simulation.

- **Sensitivity:** to speed up the simulation, it is sometimes useful, to neglect the ”very small” radiance values on the grid. The iteration updates the values on the FCC grid, if the total incoming radiance at a the current point is greater than *sensitivity*.
- **First estimation:** If checked, the program will use the first estimation algorithm presented in this paper to speed up the convergence of the simulation.
- **every *N*th step:** the radiance data between the neighboring nodes will be exchange in *every Nth step*.
- **Start:** starts the simulation.
- **Abort:** aborts the simulation.

Ray-marching parameters

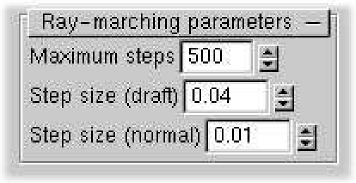

Ray-marching parameters
Maximum steps 500
Step size (draft) 0.04
Step size (normal) 0.01

Figure 7.6: The Ray-marching parameters rollout.

- **Maximum steps:** The number of maximum ray-marching steps
- **Step size (draft / normal):** Because the simulation consumes a lot of GPU time, the rendering during simulation can be performed in lower quality (draft mode). After the simulation is complete, the ray-marching should take smaller steps to provide fine details to the user.

Hotkeys

- `Esc` quits from the program.
- `O` aborts the simulation.
- `T` toggles text information at the top left corner.
- `M` toggles timers at the top right corner.
- `A`, `D` moves the light source on the X axis.
- `W`, `S` moves the light source on the Y axis.
- `Q`, `E` moves the light source on the Z axis.

```
# Resolution
width=1024
height=1024
depth=1024

# Voxel type
voxeltype=unsigned-char

# Physical size
sizex=1.0
sizey=1.0
sizez=1.0

# Data files
# values from 0 to 255
den1020.10033.01-unsigned-char
den1020.10033.02-unsigned-char
den1020.10033.03-unsigned-char
den1020.10033.04-unsigned-char
den1020.10033.05-unsigned-char
den1020.10033.06-unsigned-char
den1020.10033.07-unsigned-char
den1020.10033.08-unsigned-char
```

Listing 7.1: Sample Volume Descriptor File (McMaster University's astrophysical data set, unsigned char data type)

7.2.3 Volume Descriptor File

The volume descriptor file has two main sections. In the first one, there are name-value pairs for setting different parameters like resolution, physical size, and voxel type. In the second part the data files are listed in a sequence. The list of parameters is the following:

- `width`, `height`, and `depth` describe the dimensions of the volumetric data, i.e. the number of voxels in each dimension,
- `voxeltype` specifies the data type of the volumetric data. Currently the following values are accepted:
  - `unsigned-char` sets byte/voxel data type,
  - `unsigned-short` sets word/voxel data type,
  - `float-msb` sets IEEE 754 float/voxel data type;
- `sizex`, `sizey`, and `sizez` sets the sizes of the bounding box.

See Listing 7.1 for a sample descriptor file. The volume descriptor files for the Visible Human and the McMaster University's data sets can be also downloaded from our data server.

Chapter 8

Program Structure

The main parts of the application are the following:

8.1 Main program

Files:`main.cpp`

8.2 LatticeSim class

Files: `LatticeSim.[h|cpp]` This is a wrapper class for the CUDA functions and global variables in the .cu files.

8.3 CUDA files

Files: SimpleTexture3D.cu: The host part of the CUDA code
SimpleTexture3D_kernel.cu: The kernel part of the CUDA code

8.4 The definition of the FCC lattice

Files: `IllumLattice.h`

8.5 Transfer curve controls

Files: `TransferCurve.[h|cpp]` `RGBTransfer.[h|cpp]`

8.6 Volume loader class

Files: `voldata.[h|cpp]`

8.7 Common utilities

Files: `utilities.h`

Chapter 9

Results

For our experiments we used a *Hewlett-Packard's Scalable Visualization Array* consisting of five computing nodes. Each node has a dual-core AMD Opteron 246 processor, an nVidia GeForce 8800 GTX graphics controller, and an InfiniBand network adapter. One node is only responsible for compositing and managing the framelet generations and does not take part in the rendering processes, so we could divide our data set into maximum four parts.

First, we have examined the effect of the initial radiance approximation. Figure 9.1 shows the error curves of the iteration obtained when the radiance is initialized by the direct term only and when the media term approximation is also used. Note that the application of the media term approximation halved the number of iteration steps required to obtain a given accuracy.

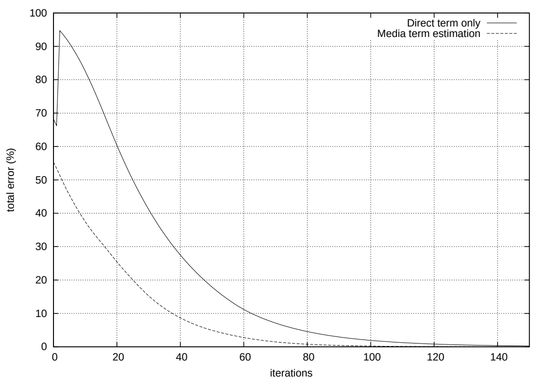


Figure 9.1: Error curves of the iteration when the radiance is initialized to the single-scattering term and when the radiance is initialized to the media term approximation. Note that in the latter case, roughly only 50% of the iteration steps are needed to obtain the same accuracy.

The evolution of the iteration can also be followed in Figures 9.2 and 9.3. Note that if we initialize the iteration with the direct term, we need about 100 iteration steps to eliminate any further visual change in the image. However, when the radiance is initialized to the approximated media term, we obtain the same result executing only 60 iterations.

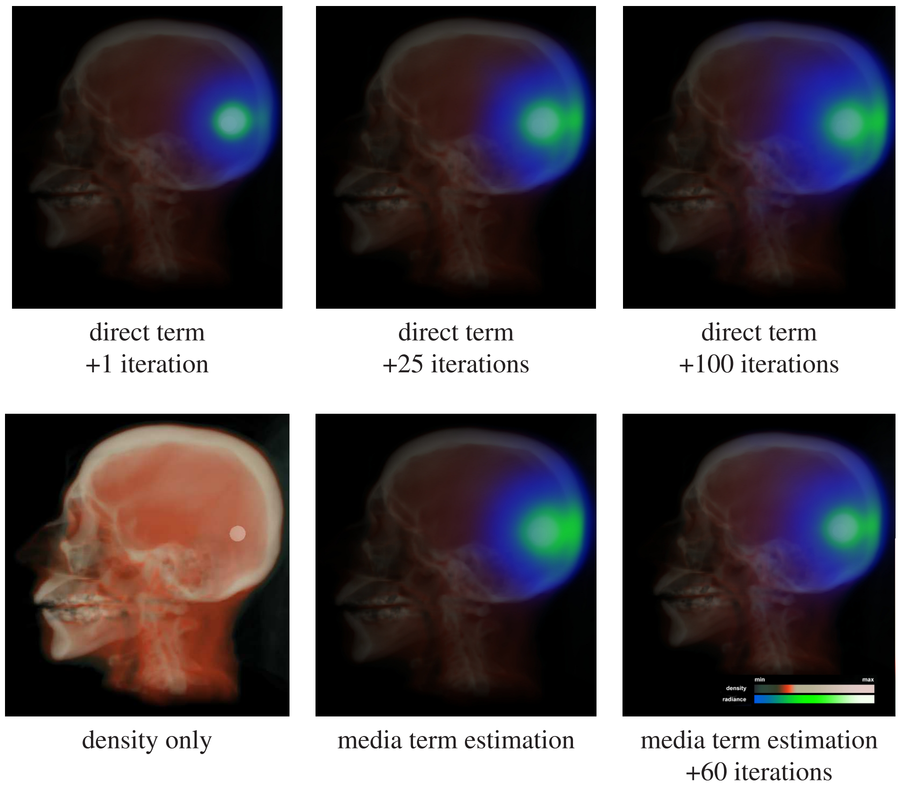


Figure 9.2: Evolution of the iteration when the radiance is initialized to the direct term and to the estimated media term, respectively. The radiance is color coded to emphasize the differences and is superimposed on the density field.

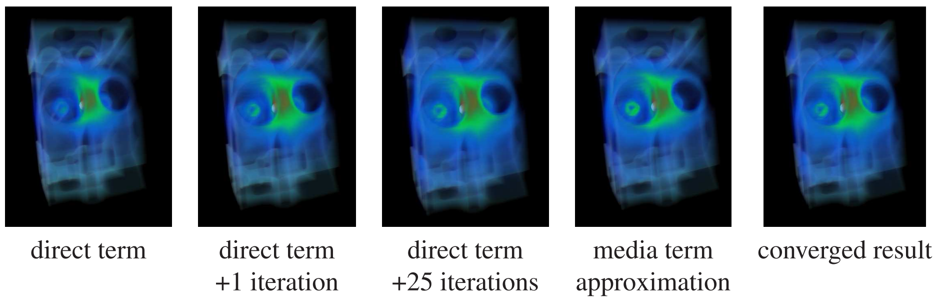


Figure 9.3: Evolution of the iteration when the radiance is initialized to the direct term and to the estimated media term, respectively. The radiance is color coded to emphasize the differences and is superimposed on the density field.

Finally, we tested the scalability of our parallel implementation. The volume is decomposed to 4 blocks along axis z, and the transfer of each block is computed on a separate node equipped with its own GPU. Table 9.1 summarizes the time data measured when a classical iteration scheme is executed, that exchanges the boundary layers of the blocks in each iteration.

Nodes	Initial	Iteration	Visualization
2	30 ms	29 ms	19 ms
3	30 ms	28 ms	15 ms
4	30 ms	26 ms	12 ms

Table 9.1: Performance figures with respect to the number of nodes when boundary conditions are exchanged after each iteration step. The volume is a $128 \times 128 \times 64$ grid. The resolution of the screen is 600×600. "Initial" time is needed by the initial radiance distribution, "Iteration" is the time of a single iteration cycle, "Visualization" is needed by the final ray casting and compositing the partial images.

We can observe that the visualization scales well with the introduction of new nodes but iteration time improves just moderately when boundary conditions are exchanged in each iteration. The explanation is that on a smaller grid the communication becomes the bottleneck. This bottleneck can be eliminated by exchanging the boundary conditions less frequently. This reduces the speed of convergence, so we trade communication overhead for GPU computation power. The performance data are shown by Table 9.2. Note that when we exchanged the boundary conditions just after every 10 iteration cycles, the iteration speed scaled very well with the introduction of newer nodes. The price for this is the slightly increased number of iterations. We observed that the error caused by exchanging the boundary conditions just after every 10 iteration cycles can be compensated by about 10% more cycles, which is a good tradeoff.

Nodes	Initial	Iteration	Visualization
2	30 ms	29 ms	19 ms
3	30 ms	24 ms	15 ms
4	30 ms	18 ms	12 ms

Table 9.2: Performance figures with respect to the number of nodes when boundary conditions are exchanged just after every 10th iteration step.

Bibliography

[1] AGGARWAL, V., CHALMERS, A., AND DEBATTISTA, K. High-Fidelity Rendering of Animations on the Grid: A Case Study. J. M. Favre and K.-L. Ma, Eds., Eurographics Association, pp. 41–48.

[2] BLINN, J. F. Light reflection functions for simulation of clouds and dusty surfaces. In *SIGGRAPH '82 Proceedings* (1982), pp. 21–29.

[3] CEREZO, E., PÉREZ, F., PUEYO, X., SERON, F. J., AND SILLION, F. X. A survey on participating media rendering techniques. *The Visual Computer 21*, 5 (2005), 303–328.

[4] DACHILLE, F., MUELLER, K., AND KAUFMAN, A. Volumetric global illumination and reconstruction via energy backprojection. In *Symposium on Volume Rendering* (2000).

[5] GEIST, R., RASCHE, K., WESTALL, J., AND SCHALKOFF, R. Lattice-boltzmann lighting. In *Eurographics Symposium on Rendering* (2004).

[6] JENSEN, H. W., AND CHRISTENSEN, P. H. Efficient simulation of light transport in scenes with participating media using photon maps. *SIGGRAPH '98 Proceedings* (1998), 311–320.

[7] JENSEN, H. W., MARSCHNER, S., LEVOY, M., AND HANRAHAN, P. A practical model for subsurface light transport. *Computer Graphics (SIGGRAPH 2001 Proceedings)* (2001).

[8] KAJIYA, J., AND HERZEN, B. V. Ray tracing volume densities. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (1984), pp. 165–174.

[9] KNISS, J., PREMOZE, S., HANSEN, C., AND EBERT, D. Interactive translucent volume rendering and procedural modeling. In *VIS '02: Proceedings of the conference on Visualization '02* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 109–116.

[10] LEE, T.-Y., RAGHAVENDRA, C., AND NICHOLAS, J. B. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2* (1996).

[11] NARASIMHAN, S. G., AND NAYAR, S. K. Shedding light on the weather. In *In CVPR 03* (2003), pp. 665–672.

[12] PARACOMP. Hp scalable visualization array version 2.1. Tech. rep., HP, 2007. http://docs.hp.com/en/A-SVAPC-2C/A-SVAPC-2C.pdf.

[13] QIU, F., XU, F., FAN, Z., AND NEOPHYTOS, N. Lattice-based volumetric global illumination. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1576–1583. Fellow-Arie Kaufman and Senior Member-Klaus Mueller.

[14] RUSHMEIER, H. E., AND TORRANCE, K. E. The zonal method for calculating light intensities in the presence of a participating medium. In *SIGGRAPH 87* (1987), pp. 293–302.

[15] STAM, J. Multiple scattering as a diffusion process. In *In Eurographics Rendering Workshop* (1995), pp. 41–50.

[16] STRENGERT, M., MAGALLN, M., WEISKOPF, D., GUTHE, S., AND ERTL, T. Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters . D. Bartz, B. Raffin, and H.-W. Shen, Eds., Eurographics Association, pp. 41–48.

[17] SUN, B., RAMAMOORTHI, R., NARASIMHAN, S. G., AND NAYAR, S. K. A practical analytic single scattering model for real time rendering. *ACM Trans. Graph. 24*, 3 (2005), 1040–1049.

[18] SZIRMAY-KALOS, L., SBERT, M., AND UMENHOFFER, T. Real-time multiple scattering in participating media with illumination networks. In *Eurographics Symposium on Rendering* (2005), pp. 277–282.

[19] ZHOU, K., REN, Z., LIN, S., BAO, H., GUO, B., AND SHUM, H.-Y. Real-time smoke rendering using compensated ray marching. *ACM Trans. Graph. 27*, 3 (2008), 36.