

# Parallel MDOM for light transport in participating media

Category:

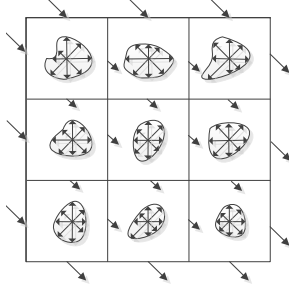


Figure 1: The figure shows a 2D DOM grid, where each grid element captures the scattered light in a finite set of discrete ordinate directions. The arrows along the grid element boundaries represent one of many light propagation directions.

## Abstract

We present a novel technique for physically based rendering of participating media like cloud, smoke, wax, marble, etc. We solve the radiative transfer equation (RTE) for participating media using the Modified Discrete Ordinate Method (MDOM), which computes the final solution as a combination of a direct and an indirect component. We propose a scalable GPU based parallel pipeline, for solving the RTE using the MDOM. This parallel RTE solver is capable of rendering intermediate results such as single scattering approximation. We overcome GPU memory size limitations by using low resolution radiance storage while doing high resolution radiance propagation. Furthermore, we achieve scalability by implementing an efficient volumetric data streaming mechanism. Our results demonstrate the ability of our method to render high quality multiple scattering effects.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Radiosity

**Keywords:** light transport, discrete ordinate method

## 1 Introduction

One of the main challenges in achieving realism while rendering participating media like cloud, fog, smoke, wax, marbles, etc. has been to accurately simulate the transport of light through them. Although most participating media exhibit only the physical processes of light absorption and scattering, some might spontaneously emit light due to high temperatures, e.g. fire. The *Radiative Transport Equation* (RTE) models all these phenomena. This integro-differential equation formulates light intensity for 3D volumes as a

five dimensional quantity.

Numerical solutions for the RTE can be carried out using either stochastic approaches or deterministic approaches. We review these approaches in section 1. The *Discrete Ordinate Method* (DOM) represents one of the most used technique among the various families of numerical solvers for the RTE. The *Modified Discrete Ordinate Method* (MDOM) is a computationally efficient version of the DOM where the radiance field is broken into *direct* and *indirect* components. This helps in overcoming the traditional DOM limitation viz, *ray effect*. Another advantage of using the MDOM is that it allows the visualization of only the *single scattering* approximation i.e the *direct* component. In section 3 we look at the mathematical formulations behind the MDOM.

The MDOM requires spatial and angular discretization of the domain. The spatial discretization is achieved by uniform voxelization of the volume. Furthermore, a quadrature scheme is used to represent the angular discretization of the radiance field at each voxel. Rendering based on the MDOM formulation requires a three stage solution, one stage for each of the 1) *direct* component computation, 2) *indirect* component computation, and 3) final view rendering. The deterministic and iterative nature of the MDOM makes it amenable for parallelization, which we exploit to achieve better performance. Our contributions include an architecture of a parallel MDOM based rendering pipeline. This pipeline is further made scalable and practical by incorporating an efficient volumetric data streaming mechanism. For the *direct* component, we implement a parallel ray marching solution. Parallel light ray marching poses a synchronization issue as multiple rays can simultaneously write radiance values to the same voxel. Therefore, to circumvent this issue we propose a unique method to serialize the writing of radiance values to every voxel viz. *sort-search-gather* technique. For the *indirect* component, we propose a parallel wavefront propagation scheme to solve the DOM iterations. These wavefronts propagate light from voxel-to-voxel along discrete directions. Finally for the rendering stage, we implement a parallel ray marching solution similar to the *direct* component stage albeit, without the synchronization method. The memory requirements for a DOM based solution are very high and often exceed the memory available on the GPU. Hence, to handle large amounts of volumetric data, we propose a novel block-based streaming mechanism for transferring data between the CPU and the GPU. This streaming mechanism is customized for each of the two light transfer stages, direct and indirect. Furthermore, to reduce the memory requirements of our pipeline we detach the propagation angular resolution from the storage angular resolution of the DOM. We discuss the details about this detachment in section 4.2.

Section 4 of this paper details our approach. In section 5, we present some results demonstrating our technique. We also present and discuss the performance evaluation of our method in the same section. Finally, in section 6 we draw conclusions and discuss future work based on our approach.

## 2 Related Work

The RTE solvers can be broadly classified into stochastic and deterministic categories. Stochastic techniques, also known as Monte Carlo techniques, trace a number of random photon paths in the medium and follow their interaction to determine a solution for any

given medium. These methods along with other deterministic approaches are surveyed by Cerezo et. al [2005]. In this paper we focus mainly on deterministic approaches specifically DOM based approaches.

The DOM method [Chandrasekhar 1950] employs both spatial and angular discretizations to simplify the problem. The volume is discretized into a regular grid structure. And the angular integrals are approximated by using various quadrature schemes. DOM based numerical solution schemes, otherwise known as grid schemes, can be categorized based on their treatment of the *linear streaming operator*;  $\vec{\omega} \cdot \nabla$  (explained in the next section). Schemes which treat this operator as a directional derivative and hence compute an integral along a path segment or a characteristic segment are termed as the *characteristics* methods. It is to be noted that the directions of these characteristics are actually the discrete ordinate directions of a quadrature scheme. These methods can be further divided into *long* and *short characteristics* methods. In the *short characteristics* approach, we are primarily concerned about radiative transport within a voxel or only along a *short characteristic* segment, starting at one face and ending on another face of the same voxel. Thus, in this method, light is transported from incoming faces of a voxel to its outgoing faces along a particular direction (see Figure 1). The *step characteristics* method [Lathrop 1969] is one of the earliest techniques which uses the *short characteristics* approach; it assumes source functions to be constant over a short step along the characteristic. This method has also been widely used in the neutron transport domain [Dehart 1992]. In computer graphics, Languenou et. al [1994] employed this approach to embed participating media within a radiosity based global illumination solution. To achieve greater accuracy in such techniques a high resolution spatial grid is required. Our algorithm extends this approach. In the *long characteristics* method, the integral is solved along the entire length of the characteristic segment with its end points on the boundary of the volume, thus, we need to compute only the intersection points with the voxels along the characteristic. However, this method demands a very high density of characteristics. Fattal [2009] overcame this issue partially by detaching the spatial and angular resolution of the storage grid from the light propagation sweeps, known as *light propagation maps* (LPM). The LPMs propagate light along directions divided into 6 axially aligned bins. This LPM technique was extended by Gruson et. al [2012] to parallel graphics architecture.

The other interpretation of the *linear streaming operator* is an inner product of two vectors. This interpretation leads us to *integrato-interpolational* and *finite element* schemes. The reader is referred to [Nikolaeva et al. 2007] for more explanation on this classification.

The DOM method is much more efficient than the comparable stochastic methods and can easily account for non-homogeneity. However, it suffers from two main limitations viz, *ray effect* and *false scattering*. Ray effects are related to the angular discretization scheme. False scattering or numerical smearing is related to the spatial discretization and can be reduced by using a more refined and accurate grid scheme. The MDOM [Ramankutty and Crosbie 1997; Coelho 2004] reduces the ray effects by splitting the final solution into *direct* and *indirect* components. The direct component is calculated analytically and the DOM is used for computing the indirect component. It is important to emphasize that these limitations viz, ray effects and false scattering are interdependent such that reducing one may increase the other [P.J and Coelho 2002]. So, while the MDOM does reduce the ray effect it has a tendency to introduce numerical smearing. However, this can be mitigated by use of finer grid resolutions.

Diffusion approximation [Stam 1995] based solutions are most appropriate for uniform homogeneous media. Bernabei et. al [2012] implemented a parallel Lattice-Boltzmann solution [Geist et al.

2004] for the diffusion equation and combined it with *Eikonal* rendering equation [Ihrke et al. 2007] for rendering refractive participating media in real-time. Wang et. al [2010] proposed a parallel solution to diffusion equation which uses a tetrahedral mesh instead of a regular grid. This allowed them to render arbitrarily shaped objects. Other GPU based methods for rendering participating media include Kaplanyan et. al [2010]. They have used a coarse volumetric light propagation scheme for plausible rendering of participating media in real-time.

The *gigavoxel* technique [Crassin et al. 2009] for streaming volume blocks from the CPU uses a *sparse voxel octree* to decide which blocks are needed to be streamed. A similar octree-based approach is used by Gobbetti et. al [2008] as well. The gigavoxel approach avoids streaming in certain cases by utilizing a hierarchical Mip-Map like data structure, which is used to approximate volume data.

### 3 Background

For a given wavelength, the RTE for a scattering, absorbing and emitting medium can be written in the following form,

$$(\vec{\omega} \cdot \nabla)I(x, \vec{\omega}) + kI(x, \vec{\omega}) = E(x, \vec{\omega}) + \frac{\sigma}{4\pi} \int_{4\pi} I(x, \vec{\omega}') f(\vec{\omega}', \vec{\omega}) d\vec{\omega}' \quad (1)$$

where  $I(x, \vec{\omega})$  is the radiance at position  $x$  along the direction  $\vec{\omega}$ . The term  $E(x, \vec{\omega})$  expresses spontaneous emission at position  $x$  in the direction  $\vec{\omega}$ . The phase function  $f(\vec{\omega}', \vec{\omega})$  represents the fraction of radiance incoming from the direction  $\vec{\omega}'$ , scattered towards the direction  $\vec{\omega}$ . The scalar quantities  $\sigma(x)$  and  $k(x)$  are the scattering coefficients and the extinction coefficients respectively. As their dependence on position  $x$  is implied we simply refer to them as  $\sigma$  and  $k$ .

The terms on the right hand side of the equation represent the change in radiance due to emission and due to in-scattering. Together, they are termed as the *source function*  $G(x, \vec{\omega})$ . In a non-emitting medium the emission component is zero, hence the source function takes the form:

$$G(x, \vec{\omega}) = \frac{\sigma}{4\pi} \int_{4\pi} I(x, \vec{\omega}') f(\vec{\omega}', \vec{\omega}) d\vec{\omega}'. \quad (2)$$

The boundary condition of RTE is

$$I(x, \vec{\omega}) = I_E(x, \vec{\omega}). \quad (3)$$

$I_E(x, \vec{\omega})$  is the radiance entering the volume. Integrating Equation 1 along a path from  $y$  to  $x$  we get the integral form of RTE

$$I(x, \vec{\omega}) = I_E(y, \vec{\omega}) e^{-\int_y^x k dx'} + \int_y^x G(x', \vec{\omega}) e^{-\int_{x'}^x k dx''} dx'. \quad (4)$$

In MDOM [Ramankutty and Crosbie 1997], the radiance is expressed as the sum of the *direct* component  $I_{dir}$  and the *indirect or diffuse* component  $I_{dif}$ . The *direct* component accounts for the extinction of the radiance entering the volume and satisfies the following equation

$$\vec{\omega} \cdot \nabla I_{dir}(x, \vec{\omega}) = -k I_E. \quad (5)$$

Hence, it can be expressed as

$$I_{dir}(x, \vec{\omega}) = I_E(y, \vec{\omega}) e^{-\int_y^x k dx'}. \quad (6)$$

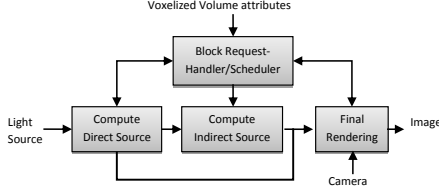


Figure 2: Functional overview of our pipeline.

The *indirect* component accounts for the diffusion of light inside the volume and satisfies the following equation

$$\vec{\omega} \cdot \nabla I_{dif}(x, \vec{\omega}) + k I_{dif}(x, \vec{\omega}) = G_{dir}(x, \vec{\omega}) + G_{dif}(x, \vec{\omega}), \quad (7)$$

where

$$G_{dir}(x, \vec{\omega}) = \frac{\sigma}{4\pi} \int_{4\pi} I_{dir}(x, \vec{\omega}') f(\vec{\omega}', \vec{\omega}) d\vec{\omega}', \quad (8)$$

and

$$G_{dif}(x, \vec{\omega}) = \frac{\sigma}{4\pi} \int_{4\pi} I_{dif}(x, \vec{\omega}') f(\vec{\omega}', \vec{\omega}) d\vec{\omega}'. \quad (9)$$

Note that

$$G(x, \vec{\omega}) = G_{dir}(x, \vec{\omega}) + G_{dif}(x, \vec{\omega}). \quad (10)$$

Approximating the above continuous spherical integrals using a quadrature scheme with directions  $m, m' \in [1, n_d]$  and expanding the directional derivative  $\vec{\omega} \cdot \nabla$  along the coordinate directions  $(x, y, z)$ , we get the DOM representation of Equations 7, 8 and 9 as:

$$\mu_m \frac{\partial I_{dif}^m}{\partial x} + \eta_m \frac{\partial I_{dif}^m}{\partial y} + \xi_m \frac{\partial I_{dif}^m}{\partial z} + k I_{dif}^m = G_{dir}^m + G_{dif}^m \quad (11)$$

$$G_{dir}^m = \frac{\sigma}{4\pi} \sum_{n'=1}^{n_r} I_{dir}^{n'} f_{n',m} w_{n'} \quad (12)$$

and

$$G_{dif}^m = \frac{\sigma}{4\pi} \sum_{m'=1}^{n_d} I_{dif}^{m'} f_{m',m} w_{m'} \quad (13)$$

where  $w_m$  are the quadrature weights,  $(\mu, \eta, \xi)$  are direction cosines of  $\vec{\omega}$  and  $n' \in [1, n_r]$  represents the incoming light directions at each voxel.  $G_{dif}^m$  and  $G_{dir}^m$  are the discrete components of the direct and the indirect source functions respectively.

## 4 Our Method

The functional overview of our pipeline is depicted in Figure 2. The inputs to this pipeline are the voxelized representation of the volume along with the extinction coefficients (scattering + absorption coefficients), a scattering albedo, a light source, view position and view direction information. The output is the realistic rendered image of the volume. Each stage of the pipeline further comprises of several passes which are implemented on the GPU. In the following sections, we give a detailed explanation of each stage of this rendering pipeline.

### 4.1 Direct component computation

The aim of this stage is to compute the direct source component ( $G_{dir}^m$ ) for each voxel. In this stage, rays emanating from the light source are marched through the volume. The average source function contribution of a light ray with radiance  $I_E^b$  at the volume

boundary point  $b$ , marching along direction  $m'$ , towards a direction  $m$  at a point  $p_v$  inside a voxel  $v$  is approximated using Equations 6 and 12 as

$$G_v^m = \frac{\sigma}{4\pi} f_{m',m} w_m I_E^b e^{-\int_b^{p_v} k dx'}, \quad (14)$$

#### 4.1.1 Parallelization

The light rays are inherently independent of each other. Hence, their marching can be easily parallelized to achieve higher performance. However, this parallelization of light rays poses a synchronization issue, as multiple rays could end up writing to the same voxel. Therefore, we need to serialize the writing of the increments to the source function generated by Equation 14 at each voxel. Moreover, as we have to handle volumetric data which is much larger than the size of the GPU memory, we need a serialization solution which will not only address these synchronization issues, but also efficiently integrate data streaming.

The traditional serialization methods e.g. semaphores, though possible on the latest platforms, are still highly inefficient due to the obvious nature of the SIMD execution model. An OpenGL based blending solution is more expensive, as it requires an extra ray marching step to find out the volume data blocks needed for the actual light ray marching step. However, an intermediate approach implemented by Bernabei et. al [2012], where a GPGPU light ray marching pass writes radiance values to intermediate data structures and then an OpenGL blending shader gathers these values, is viable. But it does not integrate well with data streaming because of the additional book-keeping required for the OpenGL shader. We propose a unique and general purpose *sort-search-gather* technique to achieve this serialization. The data streaming integration is explained in detail in the next subsection 4.1.2.

This stage of the pipeline requires two main steps; 1) a light ray marching step, where each ray writes the increments to the source function and the corresponding voxel's *voxel-id*, in an intermediate buffer, and 2) a source function gathering step, where all increments to the source function of each voxel are gathered. Henceforth, we refer to these increments as *source\_deltas*. First, for a given light source we generate an array of rays emanating from that light source. We refer to this array of rays as *raymap*. In this *raymap* we maintain the ray position, direction and radiance value. The *raymap* also stores the status of the ray. An inactive status indicates that either the ray did not intersect the volume to start with, or upon ray marching has exited the volume. During the light ray marching step, a thread marching a unique light ray uses the *raymap* to maintain the current state of that ray between thread invocations. For the gathering step we do not know beforehand the number of voxels a ray will intersect. Therefore, each ray is allocated a fixed number of write-slots in the intermediate buffer. When the ray marching thread has exhausted all of its write-slots, it suspends itself. We then perform the gathering operation, clear the intermediate buffer and relaunch the ray marching threads. The threads then resume ray marching where they had left-off and reuse their allocated write-slots. We repeat this process until all light rays have exited the volume. Moreover, the cost of gathering these scattered *source\_deltas* is very high. Hence, we first perform a fast parallel sort of these *source\_deltas* by their corresponding *voxel-id* and then do a parallel gathering pass. Each gathering thread does a binary search for a unique *voxel-id* followed by a linear gathering of the associated *source\_deltas* with that *voxel-id*. Thus, the gathering step expands into three substeps sort, search and gather.

To further reduce the cost of sorting the *source\_delta\_buffer*, we store the *voxel-id* separately along with the corresponding location (buffer-offset) in the *source\_delta\_buffer* in another buffer called

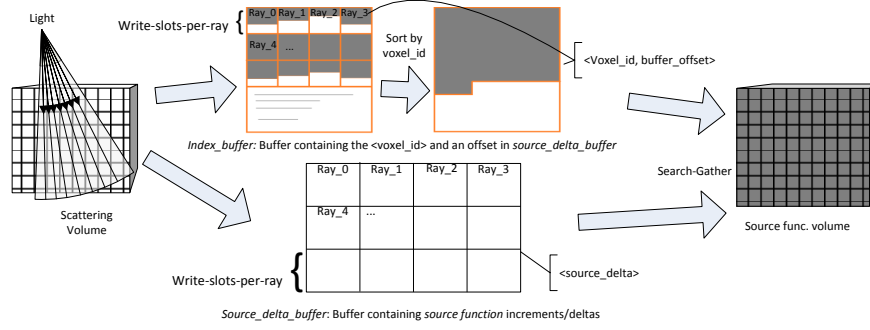


Figure 3: Overview of the sort-search-gather method.

the *index\_buffer*. In case of multiple *source\_delta\_buffers* this offset is same for all buffers. Thus for anisotropic phase functions, where the DOM storage angular resolution is greater than one, we use a separate *source\_delta\_buffer* for each storage direction ( $m \in n_d$ ). This allows us to execute the *sort-search-gather* pass in unison for all storage directions. To summarize, the *sort-search-gather* method is separated into two passes: 1) a pass for sorting of the *index\_buffer*, and 2) a *search-gather* pass for gathering the *source\_deltas* stored in the *source\_delta\_buffers*. This separation also allows us to perform step 1) on the GPU while uploading data for step 2) from the CPU memory to the GPU memory. Figure 3 depicts the various steps involved in this method. We use the parallel radix-sort method by Harris et. al [2007] to sort the *index\_buffer*.

For our hardware setting, our experiments have shown that the optimal value for *write-slots-per-ray* is 32. The *raymap* dimensions used for all our experiments were  $512 \times 512$ .

#### 4.1.2 Streaming

We use block-based streaming of the volumetric data to achieve scalability of our pipeline. Here, the entire voxel volume is broken into uniform blocks of voxels. We use two types of block data structures: 1) volume-blocks, which store the extinction coefficients, 2) source-function-blocks, which store the source function values. It is evident that we need an on-demand streaming interface for integrating streaming into the direct component stage. This interface is implemented in the block request-handler module.

First, we look at the data structures used by the streaming module. For every block-based volumetric data structure to be streamed, we maintain the following block tables on the CPU and the GPU: *block\_request\_table*, *block\_index\_table* and *block\_dirty\_table*. The *block\_request\_table* and *block\_dirty\_table* are bit arrays which store one bit per block. The *block\_request\_table* is used to track block requests whereas the *block\_dirty\_table* indicates if a block has been written into i.e. is *dirty*. As we write only to the source-function-blocks the *block\_dirty\_table* is not used for the volume-blocks. The *block\_index\_table* stores the offset of the actual block in the block store; *block\_pool*. The size of the *block\_pool* is dependent on the available GPU memory. Additional data structures are required on the CPU for tracking: a) unoccupied, b) non-dirty but occupied, and c) dirty occupied blocks in the *block\_pool* of each of the volumetric data structures. These additional data structures are used for deciding block replacements in the *block\_pool*. The priorities for these replacements in the descending order are: a) an unoccupied block, b) a non-dirty occupied block, and c) a dirty occupied block.

**Block request-handler:** The input to every light-ray marching GPU thread includes the block tables and the *block\_pool* for the volume-blocks. If a GPU thread encounters a missing block in the *block\_index\_table*, it sets a request bit in *block\_request\_table* and quits. In case, the thread has used up all of its write-slots and is required to exit, it sets the request bit for the current block before exiting. Thus, it lets the block request-handler know that it is yet to be done with the current block. When the control returns to the CPU, the block request-handler updates the *block\_index\_table* and the *block\_pool* based upon the *block\_request\_table*. As for the storage of source function values ( $G_v^m$ ) for each voxel. The search-gather pass writes the gathered *source\_deltas* to the voxel's location in its source-function-block. It is obvious that the volume-blocks visited by the light rays during the ray marching pass correspond to the source-function-blocks to be written during the search-gather pass. Hence, we use the *block\_index\_table* of the volume-blocks for pre-loading the source-function-blocks in its *block\_pool* before the search-gather pass. This pre-loading is performed simultaneously with the *index\_buffer* sorting pass further reducing the cost of synchronization.

## 4.2 Indirect Component

Here we describe the iterative computation of the indirect or diffuse source function component;  $G_{dif}^m$ . First, we detach the storage and propagation angular resolution of the DOM Equation 11. Then, we derive the equations which when solved iteratively will compute the indirect source function contributions at each voxel. The detachment of the storage and propagation angular resolution of the DOM allows us to reduce its storage requirements. We achieve this detachment by storing only a subset of the propagation directions for each voxel, usually eight directions for an anisotropic medium or a single direction in case of an isotropic medium. We follow a similar approach as Fattal's [2009] for this detachment. But unlike Fattal, we do so in a *short characteristic* setting. So, if  $m \in [1, n_d]$  represents the storage directions and  $n \in [1, N_d]$  represents propagation directions, we can rewrite the Equation 11 as

$$\mu_n \frac{\partial I_{dif}^n}{\partial x} + \eta_n \frac{\partial I_{dif}^n}{\partial y} + \xi_n \frac{\partial I_{dif}^n}{\partial z} + k I_{dif}^n = G^n \quad (15)$$

where,

$$G^n = \frac{\sigma}{4\pi} \sum_{m=1}^{n_d} \frac{f_{m,n}}{F_{m,n}} w_m G^m \quad (16)$$

refers to the source function along the propagation direction  $n$  and is reconstructed from the stored source function values  $G^m = G_{dir}^m + G_{dif}^m$ . We maintain energy conservation by normalizing the phase function with pre-computed normalization factors,  $F_{m,n}$ ,

while going back and forth between these two angular resolutions as:

$$F_{m,n} = \frac{1}{4\pi} \sum_{m=1}^{n_d} f_{m,n} w_m \quad (17)$$

It is to be noted that the final solution  $G^m$  is initialized to  $G_{dir}^m$  during the *direct component* stage and  $G_{dif}^m$  is the remaining part of the solution which we add during the *indirect component* stage.

We now derive an approximate integral of the differential Equation 15 over a voxel. We need to solve this integral to get the average radiance  $I_{dif}^n$  along the propagation direction  $n$ . Then, we use this average radiance  $I_{dif}^n$  to compute the indirect source function contribution at a voxel. First, we derive the equation for radiance propagation within a voxel. We assume that the extinction coefficient is constant ( $K$ ) within a voxel. Additionally, we ignore the in-scattering contribution from the current sweep. These assumptions mean that the source function is constant within a voxel. Solving Equation 4 under these assumptions for a path of length  $s$  within a voxel leads us to the following equation:

$$I_s^n = I_0^n e^{-Ks} + \frac{G_v^n}{K} (1 - e^{-Ks}) \quad (18)$$

where  $G_v^n$  is source function contribution from the previous iteration or the initial value in case of the first iteration. Based on this formulation, we can represent the outgoing radiance at each of the outgoing voxel faces as a linear combination of radiances arriving from all of the incoming faces as explained by Dehart [1992] and Langenou et. al [Langenou et al. 1994].

$$I_{out}^n = \sum_{in} weight_{in \rightarrow out} \cdot \left( I_{in}^n e^{-Ks_{in}} + \frac{G_v^n}{K} (1 - e^{-Ks_{in}}) \right) \quad (19)$$

where

$$weight_{in \rightarrow out} = \frac{\text{Projected Area}}{\text{Incoming Face Area}} \quad (20)$$

and  $s_{in}$  represents the average distance between the respective faces. The classification of voxel faces is relative to the direction of propagation and is pre-computed for every DOM direction. These outgoing radiances ( $I_{out}^n$ ) act as incoming radiances ( $I_{in}^n$ ) for their adjacent voxels. A similar formulation can be derived for the computation of the average radiance  $I_{dif}^n$  at the center of the voxel as:

$$I_{dif}^n = \sum_{in} \sum_{out} weight_{avg.in \rightarrow out} \cdot \left( I_{in}^n e^{-Ks_{avg.in}} + \frac{G_v^n}{K} (1 - e^{-Ks_{avg.in}}) \right) \quad (21)$$

where

$$weight_{avg.in \rightarrow out} = \frac{\text{Projected Volume}}{\text{Voxel Volume}} \quad (22)$$

$s_{avg.in}$  here represents the average distance between the respective faces and the voxel center. It is approximated as  $\frac{s_m}{2}$ . Having computed the radiance term we compute the storage source function contribution as:

$$G_v^m = \frac{\sigma}{4\pi} \frac{f_{n,m}}{F_{n,m}} w_n I_{dif}^n \quad (23)$$

where  $F_{n,m}$  is computed similar to Equation 17.

We solve Equations 19, 21 and 23 iteratively for each voxel and every propagation direction. This ultimately results in each voxel containing source function values for each DOM direction from every scattering event. We continue these iterations until convergence i.e. while  $\max |G_v^m| > \epsilon$ .

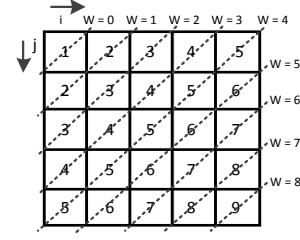


Figure 4: A 2D illustration of the wavefront algorithm,  $w$  is the wavefront step number and  $0 \leq i \leq \text{grid\_size.x}$ ,  $0 \leq j \leq \text{grid\_size.y}$  are cell indices. The cells satisfying  $w = i + j$  lie on that wavefront step.

#### 4.2.1 Parallelization

Since we need to propagate radiances from a voxel to its adjacent ones through their common faces, there is a diagonal dependency. This dependency can be best seen in Figure 1, where an incoming radiance for a voxel is same as the outgoing radiance for a previous voxel along the propagation direction. Thus, a diagonal propagation scheme will avoid synchronization issues. This implies that we need to start propagation at the eight volume corners. However, we need to propagate along only the directions which fall in that particular octant. Hence, we divide the propagation directions ( $n \in [1, N_d]$ ), into octants and propagate all the directions in an octant simultaneously. This induces a diagonal propagation sweep starting from each corner of the volume. This sweep when propagated in parallel, diagonally, is akin to propagating a wavefront starting from a corner voxel of the volume. These wavefronts scatter light throughout the volume by essentially propagating light from one voxel to its adjacent ones. Eight such wavefronts starting from each corner of the volume constitute one scattering iteration. Figure 4 illustrates a two dimensional version of the wavefront algorithm. For the 3D version, this is similar to a 3D hyperplane propagating diagonally from a grid corner to its opposite corner. A voxel  $v_{ijk}$  belongs to this 3D hyperplane, if it satisfies the equation.

$$w = i + j + k \quad (24)$$

Here  $i, j, k$  are voxel indices relative to the starting corner of the current wavefront and  $w$  ( $0 \leq w \leq (\text{grid\_size.x} + \text{grid\_size.y} + \text{grid\_size.z} - 3)$ ) is the wavefront step number.

Apart from maintaining the source function values  $G_v^m$  for each direction at every voxel, we also need to maintain source function contributions from the previous iteration and the current iteration,  $G_v^m$  and  $G_v^{m'}$ . We initialize  $G_v^m = G_v^m$  and  $G_v^{m'} = 0$ . It is to be noted that the values of  $G_v^m$  are equal to  $G_{dir}^m$  after the *direct component* stage. Since we have already extinguished the boundary radiance during the *direct component* stage, the incoming radiances ( $I_{in}^n$ ) for all the boundary voxels during this stage are zero.

#### 4.2.2 Streaming

We treat each block of volume as a separate entity which makes it compatible with the block-centric streaming mechanism. Therefore, we propagate a wavefront of blocks which is further decomposed into a wavefront of voxels for each block. It is to be noted that Equation 24 applies to these wavefronts as well. This decomposition happens on the GPU, where a *workgroup* of GPU threads operate on a single block. Figure 5 illustrates a 2D version of this approach. We can notice that the incoming radiances for the boundary voxels of a block is the outgoing radiances from the ones be-

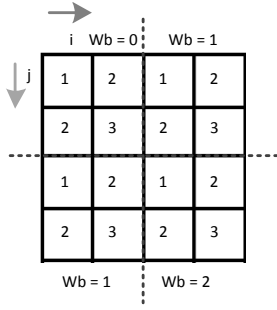


Figure 5: A 2D illustration of the nested wavefronts approach,  $W_b$  is the wavefront step number for the wavefront of blocks and the numbers in each cell represent the step number for the nested wavefront in each block.

longing to the blocks of the previous step. To store these outgoing radiances we maintain a 2D interface of voxels. These interfaces store outgoing radiance for each outgoing face along the directions in the current octant. Hence, each interface voxel stores  $\frac{N_d}{8} \times 3$  radiance values. These interfaces are cleared and reused for the next block-wavefront. We use the GPU shared memory for propagation between voxels within a block.

**Block Scheduler:** We avoid any wavefront collision handling by executing only one sweep at a time. This also enables us to pre-compute the loading sequence for the volume-blocks and source-function-blocks ( $G_v^m$ ,  $G_v'^m$  and  $G_v''^m$ ) along the direction of sweep. This implies that instead of taking the on-demand streaming approach, we can simply schedule the propagation of block-wavefronts based on the pre-computed sequence. This scheduling of the wavefronts is handled by the block scheduler. The blocks in the current wavefront step are independent of each other and can be loaded or operated upon in any order. This implies that, if there is not enough GPU memory available, we can do the propagation only for the blocks which will fit in the GPU memory. After this is done, we load the next set of blocks and continue the process till the wavefront propagation is finished.

We still need a *block\_index\_table* and a *block\_pool* for de-referencing a *block-id*. We also need a *block\_dirty\_table* for keeping track of *dirty* blocks of  $G_v^m$ ,  $G_v'^m$  and  $G_v''^m$ . The GPU shared memory usage for storing the outgoing radiances within a block is determined by the dimensions of the thread-workgroup. As there is a limited shared memory available (usually 16 KB or 48KB) per workgroup it limits the workgroup size. So, for larger blocks, the wavefront of blocks can be further decomposed into wavefront of thread-workgroups which then gets decomposed into wavefront of voxels.

### 4.3 Final Rendering

Once the source function computation is complete, we use a parametric form of Equation 4 for final rendering. An on-demand streaming mechanism similar to the direct component stage is used for this stage as well. We store the ray parameter and the accumulated radiance for each view-ray between GPU thread invocations.

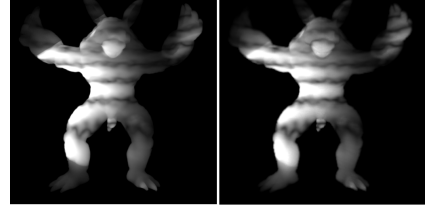


Figure 6: (l to r) Results rendered by a ground truth Monte Carlo based raytracer and our method.

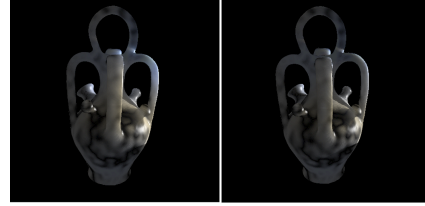


Figure 7: (l to r) multiple and single scattered renderings of a marble vase with surface lighting.

## 5 Results and Discussion

We implemented our pipeline using C++ and OpenCL. The results presented in this section are from our implementation executed on a machine with an Intel i7 CPU with 16 GB RAM and a GeForce GTX 580 GPU with 3 GB's of physical memory. In this section, we illustrate the correctness of our method by comparing it with ground truth results. We also present results for various participating media under different lighting scenarios. Then, we evaluate and analyze the performance of our method.

### 5.1 Comparisons

First, we compare our method with a volumetric path tracer which solves the full RTE. We choose a heterogeneous isotropic medium for this comparison; depicted in Figure 6. The ground truth result was rendered with path lengths of 12 and 64K samples per pixel and it took several hours to render. The corresponding result was produced from our method using spatial grid resolution of  $256^3$  and propagation angular resolution of 48 directions. Our method took 68 seconds to converge. Except for some *numerical smearing* noticeable on the right hand and right foot of the armadillo in our result, they are similar.

Next, we compare our method with the GPU LPM method by Gruson et. al [2012]. The LPM method is based on regular DOM, so it cannot readily handle directional light sources such as spotlight



Figure 8: (l to r) Results rendered by the LPM method and our method.



or point light. This would require discretizing the light source at the boundary and hence cause loss of accuracy. So we compare the environment lighting results from both methods. For using the environment lighting in our method, we skip the direct component stage. This is similar to solving the standard DOM equation with detached angular resolutions. We present the results of this comparison in Figure 8. The images depict isotropic renderings of a smoke simulation inside a  $128^3$  grid. The timings recorded for 3 iterations for both methods were 3 seconds for the GPU LPM method, and 3.05 seconds for our method. The directional resolutions were  $9 \times 9 \times 6$  directions for the GPU LPM method and 120 directions in our case. The LPM method needs high density of *characteristics* or directions, but all of its directions are divided in 6 maps and are propagated simultaneously. Since we employ *short characteristic* approach we need less number of directions than the LPM method. However, this also requires us to divide directions into 8 wavefronts which are propagated simultaneously. This means the computational performances of both methods are comparable.

We illustrate the difference between the multiple and the single scattering results from our method in Figure 7. A heterogenous and isotropic marble vase model is rendered using a  $256^3$  grid with propagation angular resolution of 24 directions. The difference is more pronounced in areas with shorter optical depth such as the handles, due to lesser exponential fall-off.

## 5.2 Performance evaluation

First, we discuss the memory requirements of our pipeline. The volume-blocks require  $n^3$  storage and each of the source-function-blocks ( $G_v^m, G_v'^m$  and  $G_v''^m$ ) require  $mn^3$  storage. For a  $256^3$  grid with each voxel containing 3 channel extinction coefficients stored as *float4* values, the volume-blocks storage amounts to 256 MB. For isotropic media, the source-function-blocks together require a  $256 \times 3$  MB storage, making the total volumetric storage requirement equal to 1 GB. For a  $512^3$  grid this total increases to 8 GB. These allocations happen on the CPU memory. The corresponding GPU memory allocations are determined by the sizes of their respective *block\_pools*. The *block\_pool* sizes are configured in terms of number of blocks they can accommodate and are same for all the *block\_pools*. For our hardware setting, our experiments have shown that the optimal block size is  $32^3$ .

Since we use OpenCL for our implementation, we can also compare timings for GPU and CPU executions of our pipeline. Graph 9 plots the speedup achieved by the GPU executions over the CPU executions of our pipeline for different grid sizes, and combined *block\_pool* sizes. The combined *block\_pool* size is analogous to the size of the total streaming cache. The CPU used for these experiments had an effective 12 execution cores (6 core Intel CPU with hyper-threading enabled) which execute the OpenCL kernels in parallel. As the graph 9 shows, when the combined *block\_pool* size is equal to the total memory requirement of a particular grid size, the overhead of our streaming approach over a non-streaming solution is negligible. It should be mentioned here that an 8X speedup is achieved for a  $512^3$  grid with a combined *block\_pool* size of just 256 MB, which is less than 4 percent of the total volumetric memory requirement. These results further corroborate the need for a GPU based parallel technique for solving the DOM equations.

## 5.3 Discussion

We begin the discussion by examining the effect of the storage angular resolution on the quality of results. In case of isotropic media,

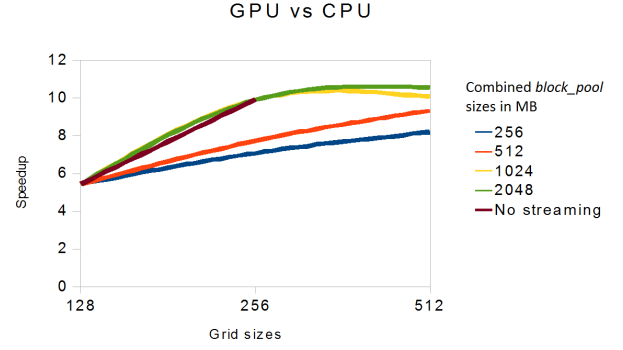


Figure 9: GPU vs CPU speedup: This graph plots the speedup of the GPU executions over the CPU (12 execution cores) executions as a function of increasing grid sizes and combined *block\_pool* allocations. Also, plotted are speedups for non-streaming cases (this excludes  $512^3$  grid, because there is not enough GPU memory available).

as the phase function is constant in all directions, the storage resolution does not impact the quality of the results. However, the capability of our method to handle anisotropic media is limited by the storage angular resolution. For the 8 direction storage our method can effectively handle Henyey-Greenstein phase functions with values of the Henyey-Greenstein constant ( $g$ ) upto 0.4. For values of  $g$  greater than 0.4 the results exhibit ray effects as shown in Figure 10. It shows the renderings of an anisotropic vase with homogeneous scattering coefficients and increasing values of  $g$ , which is lit by a beam of light. The storage angular resolution in this case is 8 and the propagation angular resolution is 48. If the direction of the light beam aligns with one of the storage directions (top images in Figure 10), then as expected, the medium exhibits higher forward scattering effect as  $g$  increases. But when the direction of light does not align with any of the storage directions (bottom images in Figure 10), then the ray effect increases as  $g$  increases beyond 0.4. Though, this can be mitigated by increasing the storage angular resolution, it is a matter of *quality vs performance* trade-off.

The propagation direction resolution can also be considered as a *quality vs performance* trade-off parameter. In case of a mostly convex geometry such as the bunny shown in Figure 11 (top images), the differences in the 24 and 48 propagation direction results are very subtle. This indicates that we can afford a low resolution propagation in this case and thus a better performance without sacrificing too much on quality. Whereas, in case of the armadillo shown in Figure 11 (bottom images), the results for 24 directions differ significantly from the 48 or 120 direction results. However, in both the cases there are negligible differences between the results of 48 and 120 propagation directions.

## 6 Conclusion

In this paper we presented a novel parallel pipeline for physically based rendering of participating media. The MDOM based short characteristic approach to solve the RTE, adapted in our method is robust, parallel and scalable. As a part of our pipeline, we implemented a unique and general purpose method for serialization of data generated during parallel ray marching of direct light. We use an intermediate data buffer to collect the generated data, and then perform a three step serialization operation viz. *sort-search-gather*.

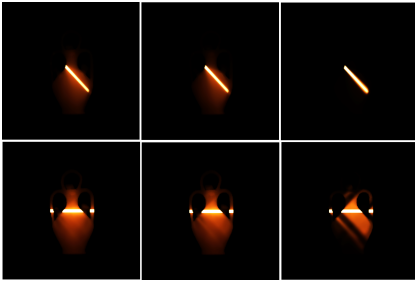


Figure 10: (l to r) Multiple scattering results with  $g = 0.4, 0.6$  and  $0.9$ . When light direction is aligned with one of the storage directions (t) and not aligned (b) with any of the storage directions.

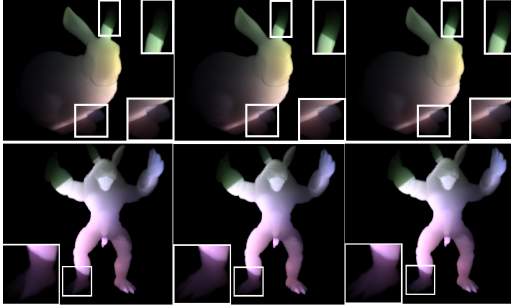


Figure 11: (l to r) Armadillo rendered with 24, 48 and 120 propagation directions.

By carefully splitting these serialization steps into two passes - 1) a pass for sorting of *index\_buffer* and 2) a *search-gather* pass for collecting the intermediate radiance data, we reduced the cost of serialization while efficiently integrating our streaming module. Furthermore, we employed an efficient block-based wavefront propagation and streaming mechanism for indirect light propagation. We demonstrated the ability of our method to render various heterogeneous volumes with different lighting conditions. This pipeline not only allows us to render high quality multiple scattering results, but also to view intermediate single scattering results. In future, we would like to extend our method to take in to account the refraction of light in the medium.

## References

- BERNABEI, D., HAKKE-PATIL, A., BANTERLE, F., DI BENEDETTO, M., GANOVELLI, F., PATTANAIK, S., AND SCOPIGNO, R. 2012. A parallel architecture for interactively rendering scattering and refraction effects. *Computer Graphics and Applications, IEEE* 32, 2, 34–43.
- CEREZO, E., PREZ, F., PUEYO, X., SERON, F. J., AND SILLION, F. X. 2005. A survey on participating media rendering techniques. *The Visual Computer* 21, 303–328.
- CHANDRASEKHAR, C. 1950. *Radiative Transfer*. Dover Publications.
- COELHO, P. J. 2004. A modified version of the discrete ordinates method for radiative heat transfer modelling. *Computational Mechanics* 33, 375–388.
- CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, 15–22.
- DEHART, M. D. 1992. *A discrete ordinates approximation to the neutron transport equation applied to generalized geometries*. PhD thesis, College Station, TX, USA.
- FATTAL, R. 2009. Participating media illumination using light propagation maps. *ACM Transactions on Graphics* 28, 1, 7:1–7:11.
- GEIST, R., RASCHE, K., WESTALL, J., AND SCHALKOFF, R. J. 2004. Lattice-boltzmann lighting. In *15th EG Workshop on Rendering Techniques, Norkping, Sweden, June 21-23, 2004*, A. Keller and H. W. Jensen, Eds., 355–362.
- GOBBETTI, E., MARTON, F., AND IGLESIAS G, J. A. 2008. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer* 24, 7, 797–806.
- GRUSON, A., HAKKE PATIL, A., COZOT, R., BOUATOUCH, K., AND PATTANAIK, S. N. 2012. Light propagation maps on parallel graphics architectures. In *EGPGV 2012*, H. Childs, T. Kuhlen, and F. Marton, Eds., 81–88.
- HARRIS, M., SENGUPTA, S., AND OWENS, J. D. 2007. Parallel prefix sum (scan) with cuda. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley.
- IHRKE, I., ZIEGLER, G., TEVS, A., THEOBALT, C., MAGNOR, M., AND PETER SEIDEL, H. 2007. Eikonal rendering: efficient light transport in refractive objects. *ACM Transactions on Graphics* 26, 3.
- KAPLANYAN, A., AND DACHSBACHER, C. 2010. Cascaded light propagation volumes for real-time indirect illumination. In *I3D 2010*, I3D '10, 99–107.
- LANGUENOU, E., BOUATOUCH, K., AND CHELLE, M. 1994. Global illumination in presence of participating media with general properties. In *5th EG Workshop on Rendering*, 69–85.
- LATHROP, K. 1969. Spatial differencing of the transport equation: Positivity vs. accuracy. *J. of Computational Physics* 4, 475 – 498.
- NIKOLAEVA, O., BASS, L., GERMOGENOVA, T., KUZNETSOV, V., AND KOKHANOVSKY, A. 2007. Radiative transfer in horizontally and vertically inhomogeneous turbid media. In *Light Scattering Reviews 2*, A. Kokhanovsky, Ed., Springer Praxis Books. Springer Berlin Heidelberg, 295–347.
- P.J. AND COELHO. 2002. The role of ray effects and false scattering on the accuracy of the standard and modified discrete ordinates methods. *J. of Quant. Spectroscopy and Radiative Transfer* 73, 2-5, 231 – 238.
- RAMANKUTTY, M. A., AND CROSBIE, A. L. 1997. Modified discrete ordinates solution of radiative transfer in two-dimensional rectangular enclosures. *J. of Quant. Spectroscopy and Radiative Transfer* 57, 1, 107 – 140.
- STAM, J. 1995. Multiple scattering as a diffusion process. In *EG Rendering Workshop*, 41–50.
- WANG, Y., WANG, J., HOLZSCHUCH, N., SUBR, K., YONG, J.-H., AND GUO, B. 2010. Real-time rendering of heterogeneous translucent objects with arbitrary shapes. *Computer Graphics Forum*, 2, 497–506.