Chapter ____ Lattice-Boltzmann Lighting Models

Robert Geist and James Westall
Clemson University

In this chapter, we present a GPU-based implementation of a photon transport model that is particularly effective in global illumination of participating media, including atmospheric geometry such as clouds, smoke, and haze, as well as densely placed, translucent surfaces. The model provides the "perfect" GPU application in the sense that the kernel code can be structured to minimize control flow divergence and yet avoid all memory bank conflicts and uncoalesced accesses to global memory. Thus the speedups over single-core CPU execution are dramatic. Example applications include clouds, plants, and plastics.

1 Introduction, Problem Statement and Context.

Quickly and accurately capturing the the process of light scattering in a volume filled with a participating medium remains a challenging problem, but often it is one that must be solved to achieve photo-realistic rendering. The scattering process can be adequately described by the standard volume radiative transfer equation

$$(\vec{\omega} \cdot \bigtriangledown + \sigma_t)\, L(\vec{x}, \vec{\omega}) = \sigma_s \int p(\vec{\omega}, \vec{\omega'}) L(\vec{x}, \vec{\omega'}) d\vec{\omega'} + Q(\vec{x}, \vec{\omega}) \tag{1}$$

where $\vec{x}$ is a position in space, $\vec{\omega}$ is a spherical direction, $p(\vec{\omega}, \vec{\omega'})$ is the phase function, σ_s is the scattering coefficient, σ_a is the absorption coefficient, $\sigma_t = \sigma_s + \sigma_a$ is the extinction coefficient, and $Q(\vec{x}, \vec{\omega})$ is any emissive field within the volume [1]. This simply says that the change in radiance along any path includes the loss due to absorption and out-scattering (coefficient $-\sigma_t$) and the gain due to in-scattering from other paths (coefficient σ_s).

Radiative transfer in volumes is a well-studied topic, and many methods for solving (1) have been suggested. Most are a variation on one of a few central themes. Rushmeier and Torrance [25] used a *radiosity* (finite-element) technique to model energy exchange between environmental zones and hence capture isotropic scattering. Their method requires computing form-factors between all pairs of volume elements. Bhate and Tokuta [2] extended this approach to anisotropic scattering, at the cost of additional form-factor computation.

The most popular approach is undoubtedly the *discrete ordinates method* [3] in which (1) is discretized to a spatial lattice with a fixed number of angular directions in which light may propagate. The need to compute only local lattice point interactions, rather than form-factors between all volumes, substantially reduces the computational complexity. Max [21] and Languénou *et al.* [20] were able to capture anisotropic effects using early variations on this technique. It is well-known, however, that this discretization leads to two types of potentially visible errors, light smearing caused by repeated interpolations, and spurious beams, called the ray effect. Fattal [8] has recently suggested a method that significantly ameliorates both. Successive iteration across six collections of 2D maps of rays, along which light is propagated, is employed. The 2D maps are detached from the 3D lattice, thus allowing a much finer granularity in the angular discretization. Execution time for grids of reasonable size, e.g., 128^3, is several minutes. Kaplanyan and Dachsbacher [17] have

recently provided a real-time variation on the discrete ordinates method that is principally aimed at single-bounce, indirect illumination. The method propagates virtual point lights along axial cells by projecting illumination to the walls of an adjacent cell and then constructing a new point light in the adjacent cell that would deliver exactly that effect. It is not immune to either light smearing or the ray effect, and thus the authors restrict its use to low-frequency lighting. The method is extremely fast, largely due to relatively low resolution, nested grids that are fixed in camera space and give higher resolution close to the viewer.

A third central theme, and that to which our own method may be attached, is the *diffusion approximation*. Stam [27], motivated by earlier work of Kajiya and Von Herzen [16] in lighting clouds, observed that a two-term Taylor expansion of diffuse intensity in the directional component only could be substituted into (1) to ultimately yield a diffusion process that is an accurate approximation for highly scattering (optically thick) media. Jensen *et al.* [15] showed that a simple, two-term approximation of radiance naturally leads to a diffusion approximation that is appropriate for a highly scattering medium. This was later extended to include thin translucent slabs and multi-layered materials [7].

In [11] we introduced an approach to solving (1) based on a lattice-Boltzmann technique and showed how this might be applied to lighting participating media such as clouds, smoke, and haze. In [12] we extended this approach to capture diffuse transmission, inter-object scattering, and ambient occlusion, such as needed in photo-realistic rendering of dense forest ecosystems. Most recently, we showed that the technique could be parameterized to allow real-time relighting of scenes in response to movement of light sources [28].

As originally described in [11], the technique was effective, but slow. The technique is grid-based, and a steady-state solution for a grid of reasonable size (128^3 nodes) originally required more than 30 minutes on a single CPU. Our goal here is to provide the details of a GPU-based implementation that, for this same size grid, will run on a single GPU in 2 seconds. Example lighting tasks will include clouds, plants, and plastics.

2 Core Methods.

Lattice-Boltzmann (LB) methods are a particular class of *cellular automata* (CA), a collection of computational structures that can trace their origin at least back to John Conway's famous *Game of Life* [10], which models population changes in a hypothetical society that is geographically located on a rectangular grid. In Conway's game, each grid site follows only local rules, based on nearest-neighbor population, in synchronously updating itself as populated or not, and yet global behavior emerges in the form of both steady-state colonies and migrating bands of "beings" who can generate new colonies or destroy existing ones.

Arbitrary graphs and sets of rules for local updates can be postulated in a general CA, but those that are most interesting exhibit a global behavior that has some provable characteristic. Lattice-Boltzmann methods also employ synchronous, neighbor-only update rules on a discrete grid, but the discrete populations have been replaced by continuous distributions of some quantity of interest. The result is that the provable characteristic is often quite powerful: the system is seen to converge, as lattice spacing and time step approach zero, to a solution of a targeted class of partial differential equations (PDEs).

Lattice-Boltzmann methods are thus often regarded as computational alternatives to finite-element methods (FEMs) for solving coupled systems of PDEs. The methods have provided significant successes in modeling fluid flows and associated transport phenomena [4, 13, 26, 29, 31]. They provide stability, accuracy, and

computational efficiency comparable to finite-element methods, but they realize significant advantages in ease of implementation, parallelization, and an ability to handle inter-facial dynamics and complex boundaries.

The principal drawback to the methods, compared to FEMs, is the counter-intuitive direction of the derivation they require. Differential equations describing the macroscopic system behavior are derived (emerge) from a postulated computational update (the local rules), rather than the reverse. Thus a relatively simple computational method must be justified by a derivation that is often fairly intricate.

In the Appendix we provide the derivation of the parameterized diffusion equation for our lighting model. On the one-hand, the derivation itself is totally irrelevant to the implementation of the LB lighting technique proposed here, and users of the technique may feel free to skip it entirely. On the other hand, if the processes of interest to the user vary, beyond simple parameter changes, from our basic diffusion process, a new derivation will be required, and it is extremely valuable to have such a sample derivation at hand.

3 Algorithms, Implementation, and Evaluation.

Lattice-Boltzmann methods are grid-based, and so we begin with a standard 3D lattice, a rectangular collection of points with integer coordinates in 3D space. A complication of LB methods in three dimensions is that isotropic flow, whether it is water or photon density, requires that all neighboring lattice points of any site (lattice point) be equidistant. A standard approach, due to d'Humières, Lallemand, and Frisch [6], is to use 24 points equidistant from the origin in 4D space and project onto 3D. The points are:

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

and projection is truncation of the fourth component, which yields 18 directions, i.e., the vectors from the origin in 3D space to each of the projected points. Axial directions then receive double weights. Representation of several phenomena, including energy absorption and energy transmission, is facilitated by adding a direction from each lattice point back to itself, which thus yields 19 directions, the non-corner lattice points of a cube of unit radius.

The key quantity of interest is the per-site photon density. We store this value at each lattice point and then simulate transport by synchronously updating all the values in discrete time steps according to a single update rule. Let $f_m(\vec{r}, t)$ denote the density at lattice site $\vec{r} \in \Re^3$ at simulation time t that is moving in cube direction $\vec{c}_m$, $m \in \{0, 1, ..., 18\}$. If the lattice spacing is λ and the simulation time step is τ, then the update (local rule) is:

$$f_m(\vec{r} + \lambda \vec{c}_m, t + \tau) - f_m(\vec{r}, t) = \Omega_m \cdot f(\vec{r}, t) \tag{2}$$

where Ω_m denotes row m of a 19×19 matrix, Ω, that describes scattering, absorption, and (potentially) wavelength shift at each site. If $\rho(\vec{r}, t) = \sum_m f_m(\vec{r}, t)$ denotes total site density, then the derivation in the Appendix shows that the limiting case of (2) as $\lambda, \tau \to 0$ is the diffusion equation

$$\frac{\partial \rho}{\partial t} = D \nabla^2_{\vec{r}} \rho \tag{3}$$

where the diffusion coefficient

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

As noted, this is consistent with previous approaches to modeling multiple photon scattering events [15, 27], which have invariably led to diffusion processes.

For any lattice-Boltzmann method, the choice of the so-called collision matrix, Ω, is not unique. Standard constraints are conservation of mass, $\sum_m(\Omega_m \cdot f) = 0$, and conservation of momentum, $\sum_m(\Omega_m \cdot f)\vec{v_m} = \tau\vec{F}$, where $\vec{v_m} = (\lambda/\tau)\vec{c_m}$ and $\vec{F}$ represents any site external force. Here, as in [11, 12, 13, 28], for the case of isotropic scattering, we specify Ω as follows:

For row 0:

$$\Omega_{0j} = \begin{cases} -1 & j = 0 \\ \sigma_a & j > 0 \end{cases} \tag{5}$$

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

$$\Omega_{ij} = \begin{cases} 1/12 & j = 0 \\ \sigma_s/12 & j > 0, \quad j \neq i \\ -\sigma_t + \sigma_s/12, & j = i \end{cases} \tag{6}$$

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

$$\Omega_{ij} = \begin{cases} 1/24 & j = 0 \\ \sigma_s/24 & j > 0, \quad j \neq i \\ -\sigma_t + \sigma_s/24, & j = i \end{cases} \tag{7}$$

Entry i, j controls scattering from direction $\vec{c_j}$ into direction $\vec{c_i}$, and thus the non-diagonal entries are scaled values of the scattering coefficient for the medium, σ_s. Diagonal entries must include out-scattering, $-\sigma_t$, and, as noted earlier, isotropic flow requires double weights (scale factors) on the axial directions. Directional density f_0 holds the absorption/emission component. On update, i.e., $\Omega \cdot f$, fraction σ_a from each directional density will be moved into f_0. The entries of Ω are then multiplied by the density of the medium at each lattice sight, so that a zero density yields a pass-through in (2), and a density of 1 yields a full scattering.

Of course, in general, scattering is anisotropic and wavelength-dependent. Anisotropic scattering is incorporated by multiplying σ_s that appears in entry $\Omega_{i,j}$ by a normalized phase function:

$$pn_{i,j}(g) = \frac{p_{i,j}(g)}{\left(\sum_{i=1}^{6} 2p_{i,j}(g) + \sum_{i=7}^{18} p_{i,j}(g)\right)/24} \tag{8}$$

where $p_{i,j}(g)$ is a discrete version of the Henyey-Greenstein phase function [14],

$$p_{i,j}(g) = \frac{1 - g^2}{(1 - 2g\vec{n_i} \cdot \vec{n_j} + g^2)^{3/2}} \tag{9}$$

Here $\vec{n_i}$ is the normalized direction, $\vec{c_i}$. Parameter $g \in [-1, 1]$ controls scattering direction. Value $g > 0$ provides forward scattering, $g < 0$ provides backward scattering, and $g = 0$ yields isotropic. Mie scattering [9] is generally considered preferable, but the significant approximations induced here by a relatively coarse grid render the additional complexity unwarranted. Note that (8), which first appeared in [12], differs from the treatment in [11], in that setting $\sigma_a = 0$ and $g = 1$ yields an effect that is identical to a pass-through.

If the media in the model are presumed to interact with all light wavelengths in an identical way, then $f_m(\vec{r}, t)$ can be considered to be the luminance density. In our cloud lighting example of section 3.2.1, this is the case.

Otherwise, as in our other examples, each discrete wavelength to be modeled is characterized by appropriate scattering parameters and phase function. The lattice-Boltzmann model is run once for each parameter set, and the results are combined in a post-processing (ray-tracing) step.

An alternative approach is to implement $\vec{f}_m(\vec{r}, t)$ as a vector of per-wavelength densities and run the model one time. This approach can be used to incorporate wavelength shifts during the collision process.

The model is thus completely specified by a choice of g, σ_a, and σ_s, any of which can be wavelength-dependent.

Initial conditions include zero photon density in all directions at all interior nodes in the lattice. For each node in the lattice boundary, directions that have positive dot products with the external light source direction will receive initial photon densities that are proportional to these dot products. These directional photon densities on boundary nodes are fixed for the duration of the execution. Any flow (movement of photon density) into a boundary node is ignored, as is any flow out of a boundary node that goes off-lattice.

3.1 An OpenCL Implementation.

The fundamental update (2) must be synchronous. Each lattice site must read 19 values and write 19 values at each time step, but the layout is particularly well-suited to GPU architectures in that there is no overlap in sites read or sites written between or among sites. We have implemented kernels for the update in both CUDA and OpenCL for the NVIDIA GTX 480, and we find, after optimizations of both, essentially no speed difference. Thus, in the interest of greater portability, we present the OpenCL solution here.

We use two copies of the directional densities, $f_m(\vec{r}, t)$, and ping-pong between them as we advance the time step, t. Experience indicates that a final time of $n \times \tau$, where n is twice the longest edge dimension of the lattice, will suffice to achieve near-equilibrium values.

Both the collision matrix, Ω, and the per-site density of the medium are constant for the duration of the computation. The Ω matrix, which is 19×19, easily fits in constant memory, and so we can take advantage of the caching effects there. The medium density is one float per lattice site, which is too large for constant memory, but each lattice site reads only one medium density value per update, and that value can be stored in a register.

In addition to storage for Ω and storage for the per-site medium density, GPU-side storage includes two arrays for the flows:

```
cl_mem focl[2][WIDTH*HEIGHT*DEPTH*DIRECTIONS];
```

and a single integer array

```
cl_mem dist[WIDTH*HEIGHT*DEPTH*DIRECTIONS];
```

which holds, for each lattice site and each direction, the flow array index where exiting flow should be placed. The values for *dist[]* can be calculated and stored by the CPU during initialization.

On the CPU side, the entire update, using kernel *mykrn_update*, command queue *mycq*, and lattice dimension *LDIM* is then given by:

```
void run_updates()
{
int t, from = 0;
cl_event wait[1];

for(t=0;t<FINAL_TIME;t++,from = 1-from){
   clSetKernelArg(mykrn_update,0,sizeof(cl_mem),(void *)&focl[from]);
   clSetKernelArg(mykrn_update,1,sizeof(cl_mem),(void *)&focl[1-from]);
   clEnqueueNDRangeKernel(mycq,mykrn_update,3,0,ws,lws,0,0,&wait[0]);
   clWaitForEvents(1,wait);
   }
clEnqueueReadBuffer(mycq,focl[from],CL_TRUE,0,LDIM*sizeof(float),&f[0],
   0,NULL,NULL);
return;
}
```

The work size, *ws*, is a three-dimensional integer vector with value (WIDTH,HEIGHT,DEPTH), the dimensions of the the grid. The local work size, *lws*, which specifies threads per scheduled block, is (1, 1, DEPTH), for reasons to be specified shortly.

A naive, but conceptually straightforward kernel is then given by

```
__kernel void update(__global float* from, __global float* to,__global
   int* dist, __global float* omega, __global float* density)
{
int i = get_global_id(0);
int j = get_global_id(1);
int k = get_global_id(2);
int m, n;
float medium_density, new_flow;

medium_density = density(i,j,k);
for(m=0;m<DIRECTIONS;m++){
   new_flow = 0.0f;
   for(n=0;n<DIRECTIONS;n++) new_flow += omega(m,n)*from(i,j,k,n);
   to[dist(i,j,k,m)] = from(i,j,k,m) + new_flow*medium_density;
   }
}
```

The *from* and *to* arguments hold addresses of the two copies of the flow values. Macros *from(i,j,k,n)*, *dist(i,j,k,m)*, and *omega(m,n)* simply provide convenient indexing into the arrays of the same names, which are necessarily one-dimensional in GPU storage. Both the *from* and *to* arrays hold an extra index, a "sink", and if the exiting flow from any site in any direction would be off-lattice, the value of the *dist(i,j,k,m)* index is this "sink".

This kernel is attractive in its simplicity, and it offers zero control flow divergence, since there are no conditionals. Nevertheless, although the kernel will produce correct results, its performance advantage over a CPU implementation is modest. For the $256 \times 128 \times 128$ array used in the first example of the next section, a CPU implementation running on an Intel i7 X980 3.33GHz processor required 19.5 minutes. The OpenCL implementation with this kernel required 2.8 minutes. With kernel modifications, the total execution time can be reduced to 6 seconds, of which 1 second is CPU initialization and 5 seconds are GPU execution.

The first important modification is to avoid re-loading the values, *from(i,j,k,n)*, from global device memory. Instead, we force storage into registers by declaring 19 local variables,

```
float f00, f01, ..., f18
```

We then load each directional value once, unroll the interior loop, and use these local variable values instead of *from(i,j,k,n)*.

The second important modification is to rearrange storage to facilitate coalescing of memory requests. Since we have WIDTH×HEIGHT×DEPTH lattice sites, and each of these has 19 DIRECTIONS, it is natural to access linear storage as

```
from[((((i)*HEIGHT+(j))*DEPTH+(k))*DIRECTIONS+(m))]
```

but this leads to relatively poor performance. Instead, since DEPTH is usually a large power of 2, we interchange DEPTH and DIRECTIONS so that the DEPTH index varies most rapidly in linear storage:

```
from[((((i)*HEIGHT+(j))*DIRECTIONS+(m))*DEPTH+(k))]
```

This rearrangement applies to both copies of the photon density values, *focl[2]*, and the integer *dist[]* array.

The third modification, though least important in terms of performance gained, is probably the most surprising. Performance improves when we add conditionals and extra computation to this kernel! In particular, if we eliminate the *dist[]* array, instead pass only the 19 directions, and calculate the indices for the exiting photon density on the fly, performance improves by a small amount. The conditionals and the index computations, which are required to avoid indices that would be off-lattice, are less expensive than accesses to the global *dist[]* array. Since flow divergence then occurs only between boundary nodes and interior nodes, and we place each DEPTH stripe in a single thread block using local work size (1, 1, DEPTH), the effect of divergence is really minimal.

The final kernel is then given by

```
#define nindex(i,j,k)  (((i)*HEIGHT+(j))*DEPTH+(k))
#define bindex(i,j,k)  ((((i)*HEIGHT+(j))*DIRECTIONS*DEPTH)+(k))
#define dindex(i,j,k,m)  ((((i)*HEIGHT+(j))*DIRECTIONS+(m))*DEPTH+(k))
#define inbounds(q,r,s)  ((q>0)&&(q<(WIDTH-1))&&(r>0)&&(r<(HEIGHT-1))&&\
      (s>0)&&(s<(DEPTH-1)))
```

```
__kernel void update(__global float* from,__global float* to,__constant
      int* direction, __constant float* omega,__global float* density)
{
const int i = get_global_id(0);
const int j = get_global_id(1);
const int k = get_global_id(2);
const int lnindex = nindex(i,j,k);
const int lbindex = bindex(i,j,k);
int m;
float cloud_density, new_flow;
float f00, f01, f02, f03, ..., f18;
int outi, outj, outk;

cloud_density = density[lnindex];
f00 = from[lbindex+0*DEPTH];
f01 = from[lbindex+1*DEPTH];
f02 = from[lbindex+2*DEPTH];
f03 = from[lbindex+3*DEPTH];
...
f18 = from[lbindex+18*DEPTH];

for(m=0;m<DIRECTIONS;m++){
      outi = i+direction[3*m+0];
      outj = j+direction[3*m+1];
      outk = k+direction[3*m+2];
      if(inbounds(outi,outj,outk)){
            new_flow = 0.0f;
            new_flow += omega(m,0)*f00;
            new_flow += omega(m,1)*f01;
            new_flow += omega(m,2)*f02;
            new_flow += omega(m,3)*f03;
            ...
            new_flow += omega(m,18)*f18;
            to[dindex(outi,outj,outk,m)] = from[lbindex+m*DEPTH] +
                  new_flow*cloud_density;
            }
      }
}
```

Profiling this kernel shows 100% coalesced global memory reads and writes and an occupancy of 0.5. Transfer from/to global memory is measured at 67 GB/sec., which is a significant portion of the available bandwidth for the GTX 480 as reported by the *oclBandwidthTest* utility, 115 GB/sec. Note that this does not include transfer from (cached) constant memory, which is significantly larger, 718 GB/sec.

On devices of lower compute capability, this kernel can generate a significant number of uncoalesced global memory writes. Nevertheless, with a simple kernel modification we can force these writes to be coalesced as well. Since we use a local work size vector of (1, 1, DEPTH), we can collect DEPTH output values, one for each *outk*, in a local (shared) memory array of size DEPTH, synchronize the threads using a *barrier()*

command, and then write the entire DEPTH stripe in order. The final kernel, revised for this lower compute capability, is then:

```
#define nindex(i,j,k) (((i)*HEIGHT+(j))*DEPTH+(k))
#define bindex(i,j,k) ((((i)*HEIGHT+(j))*DIRECTIONS*DEPTH)+(k))
#define dindex(i,j,k,m) ((((i)*HEIGHT+(j))*DIRECTIONS+(m))*DEPTH+(k))
#define inbounds(q,r,s) ((q>0)&&(q<(WIDTH-1))&&(r>0)&&(r<(HEIGHT-1))&&\
      (s>0)&&(s<(DEPTH-1)))

__kernel void update(__global float* from,__global float* to,__constant
      int* direction, __constant float* omega,__global float* density)
{
const int i = get_global_id(0);
const int j = get_global_id(1);
const int k = get_global_id(2);
const int lnindex = nindex(i,j,k);
const int lbindex = bindex(i,j,k);
__local float getout[DEPTH];
float cloud_density, new_flow;
float f00, f01, f02, f03, ..., f18;
int m, outi, outj, outk;

cloud_density = density[lnindex];
f00 = from[lbindex+0*DEPTH];
f01 = from[lbindex+1*DEPTH];
f02 = from[lbindex+2*DEPTH];
...
f18 = from[lbindex+18*DEPTH];

for(m=0;m<DIRECTIONS;m++){
      outk = k+direction[3*m+2];
      if(outk>0 && outk<(DEPTH-1){
            new_flow = 0.0f;
            new_flow += omega(m,0)*f00;
            new_flow += omega(m,1)*f01;
            new_flow += omega(m,2)*f02;
            ...
            new_flow += omega(m,18)*f18;
            getout[outk] = from[lbindex+m*DEPTH] +
                  new_flow*cloud_density;
            }
      barrier(CLK_LOCAL_MEM_FENCE);
      outi = i+direction[3*m+0];
      outj = j+direction[3*m+1];
      if(inbounds(outi,outj,k))
            to[dindex(outi,outj,outk,m)] = getout[k];
      }
}
```

Testing this modified kernel on a Quadro FX 5600 (compute capability 1.0), shows that it entirely eliminates the uncoalesced writes generated by the original kernel. Execution time performance improves by slightly more than 25%. Nevertheless, on the GTX 480, where there were no uncoalesced writes, the modified kernel provides no improvement over the original. In fact, the overhead of the added conditional and the added barrier causes a small execution time penalty, and the modified kernel is actually slower than the original.

3.2 Examples.

We have applied this technique to a wide variety of rendering tasks. We find compelling lighting effects in those cases where the object geometry is both reflective and significantly transmissive across a reasonably wide band of the visible spectrum. We include three examples here.

3.2.1 Clouds.

In Figure 1 we show a synthetic cloud lighted with this technique. The cloud density here was also generated by a lattice-Boltzmann model, in this case a model of the interaction between water vapor and thermal energy. (Details of that model may be found in [13].) Key parameter values are the Henyey-Greenstein parameter, $g = 0.9$, which provides significant forward scattering, and $\sigma_a = 0.008$, and $\sigma_s = 0.792$, that together give a relatively high scattering albedo, $\sigma_s/\sigma_t = 0.99$, which is appropriate for clouds.

For rendering, we use a standard, path integrating volume tracer that samples the medium density and the LB lighting solution along each ray. The medium density sample is used to determine opacity. We use tri-linear interpolation from the surrounding lattice point values for sampling both quantities. Watt and Watt [30] provide a detailed description of this algorithm.

3.2.2 Plants.

In [12] we first conjectured that a leafy plant or even a forest might be regarded as a "leaf cloud," to which LB lighting might be effectively applied. Of course, polygonal mesh models must first be voxelized [22, 23] to produce the lattice densities, which are then used as coefficient multipliers in the collision matrix.

The ray-tracer here differs from the path integrating volume tracer used for clouds. It is essentially a conventional ray-tracer that incorporates the LB lighting as an ambient source. For each fragment, it samples the lattice-Boltzmann solution at the nearest grid point to obtain an illumination value that is then modulated by the fragment's ambient response color, attenuated by viewer distance, and added to the direct diffuse and specular components to produce final fragment color.

The example plant model used here is a dogrose bush (Rosa canina) from the Xfrog Plants collection [24].

A visualization of the lattice-Boltzmann lighting alone is shown in Figure 3. Here each fragment is simply colored with the three-component photon density from the nearest grid node. Note that there is significant mid-band (green) illumination, and the right side, which is opposite the sun direction, is darker and than the left. Thus we are capturing both transmission and ambient occlusion.

Figure 1: Synthetic cloud with lattice-Boltzmann lighting.

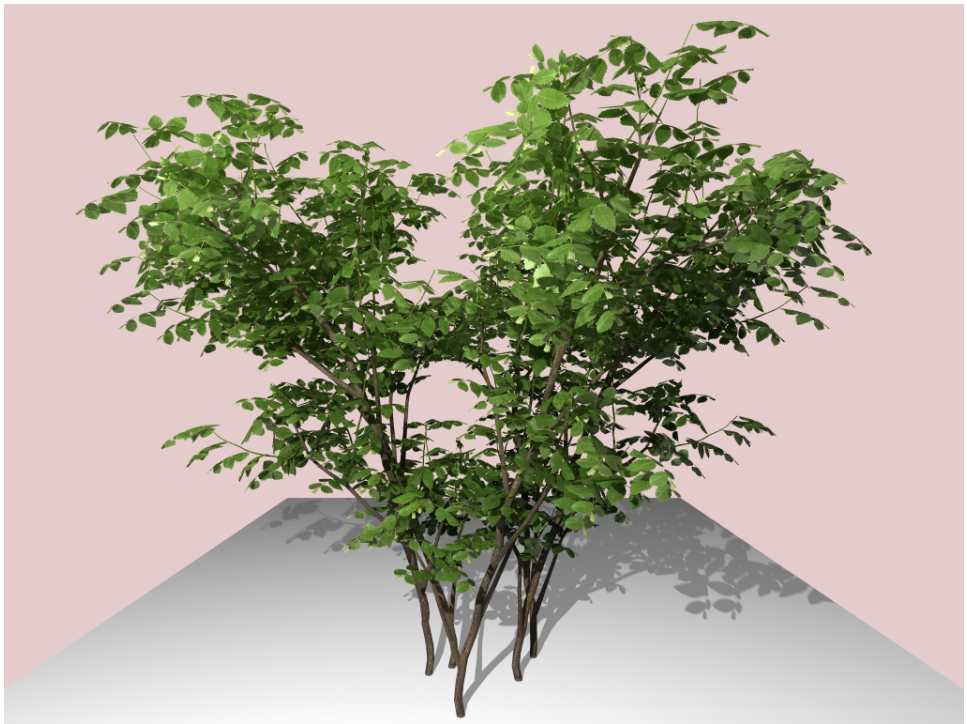

Figure 2: Synthetic plant (dogrose) with lattice-Boltzmann lighting.

Figure 3: Visualization of lattice-Boltzmann lighting for the dogrose plant.

	red	green	blue
$Henyey - Greenstein\ g$	-0.231	0.103	-0.750
σ_a	0.109	0.091	0.118
σ_s	0.891	0.909	0.882

Table 1: Parameters used in lighting the dogrose plant.

The key parameter values here were derived from the values reported in the remarkable study by Knapp and Carter [19] that showed that the reflective and transmissive properties of plants are essentially constant across species. The wavelength-dependent values used are shown in Table 1. Visible energy loss due to absorption was modeled by zeroing out the f_0 component at each site on each iteration.

3.2.3 Plastics.

The reflective and transmissive properties of polymethyl methacrylate (PMMA) are well-known. It is a light-weight, shatter-resistant alternative to glass that is often used in automobile taillights. To push the limits of our lighting technique, we fashioned a (hypothetical) PMMA teapot and applied LB lighting. The results are shown in Figure 4. The key model parameter values are shown in Table 2.

Although the effects of LB lighting here are admittedly minimal, we find that it offers a small, but believable "glow" that is not available with conventional lighting. In Figure 5 we show the same teapot, rendered with conventional lighting, where the ambient surface color is set equal to the diffuse surface color.

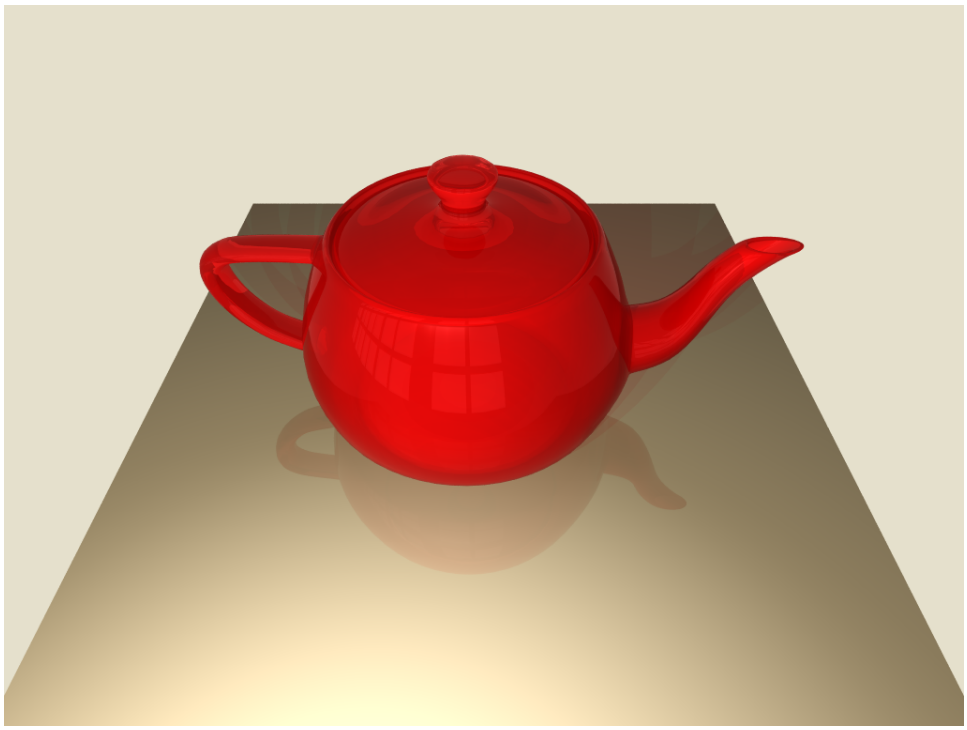

Figure 4: Lattice-Boltzmann lighting for the PMMA teapot.

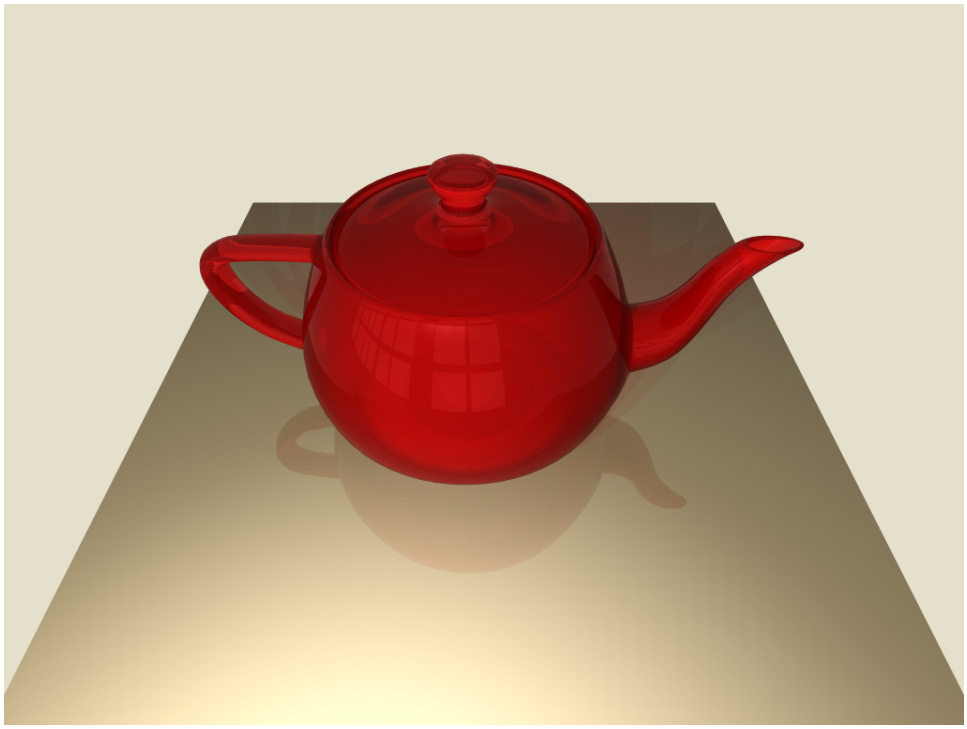

Figure 5: The PMMA teapot with conventional lighting.

	red	green	blue
$Henyey - Greenstein\ g$	0.80	-0.88	-0.88
σ_a	0.01	0.01	0.01
σ_s	0.99	0.99	0.99

Table 2: Parameters used in lighting the PMMA teapot.

4 Final Evaluation.

In Table 3 we compare timings for solutions of our LB lighting model as applied to the 256×128^2 grid used in the cloud lighting example of section 3.2.1. Since each line includes a 1-second CPU overhead for loading, the available speedup (without resorting to SSE instructions or multi-threading) is from 1169s to 5s, or 234x.

It is interesting to note that by resorting to SSE instructions and multi-threading on the i7 X980, we can achieve an execution time that is better than that of the naive kernel running on the GTX 480.

platform	time (seconds)
Intel i7 X980	1170
Intel i7 X980 with SSE	281
Intel i7 X980 with SSE and 4-way threading	100
NVIDIA GTX 480 naive kernel	167
NVIDIA GTX 480 final kernel	6

Table 3: Execution time for lighting the 256×128^2 cloud model.

Results scale linearly with available resources. It should be expected that doubling the longest edge dimension would increase execution time by a factor of four. There are twice as many nodes, but achieving steady-state requires twice as many iterations. In Table 4 we show execution times for a cloud model of size 128^3. Results are as expected. Of the 2 seconds required for the final kernel on the 128^3 problem, approximately 0.7 seconds may be attributed to CPU loading, and thus the expected factor of $\times 4$ holds for all entries.

Although the computational structure of our method is similar to that which would be used in a parallel implementation of the discrete ordinates method, it is important to remember that our simulation-like structure is actually a numerical solution of an approximating diffusion equation. Thus, like Stam's approach [27], its value is probably restricted to highly scattering geometry. Nevertheless, on the other side of this same coin,

platform	time (seconds)
Intel i7 X980	292
Intel i7 X980 with SSE	70
Intel i7 X980 with SSE and 4-way threading	26
NVIDIA GTX 480 naive kernel	41
NVIDIA GTX 480 final kernel	2

Table 4: Execution time for lighting a 128^3 cloud model.

one should not expect any ray effects, as seen in the discrete ordinates derivatives.

We believe that our LB lighting model is effective in quickly lighting scenes with highly scattering geometry, such as clouds or forests. It is best used to capture diffuse transmission, inter-object scattering, and ambient occlusion, all of which are important to photo-realism.

There are limitations. Since the technique models photon transport as a diffusion process, it is inappropriate for most direct illumination effects. Thus it must be augmented with other techniques to capture such. Another limitation is the memory requirement. Since the grid state is maintained as floats:

```
cl_mem focl[2][WIDTH*HEIGHT*DEPTH*DIRECTIONS];
```

a grid of size 256^3 would require 2.5GB, and larger grids would exceed available GPU memory. To light a scene with more than a single cloud or a single tree and yet capture inter-object, indirect illumination requires a grid hierarchy, such as suggested in [28].

5 Future Directions.

More effective use of grids of limited size is under investigation. The impressive, real-time performance achieved by Kaplanyan and Dachsbacher [17] can be attributed to their use of nested grids of varying resolution positioned in camera space. Their grid hierarchy is more sophisticated than our own [28], and could offer significant performance benefits. Another approach is the use of dynamic grids. In [5] we proposed a spring-loaded vertex model for 2D radiosity computations. The grids were represented as point masses interconnected by zero-rest-length springs. Spring forces were dynamically determined by the current best lighting solution until equilibrium, with an attendant deformed grid, was achieved. This approach allowed high quality lighting solutions with relative small grid sizes. Application of this approach to 3D lighting appears most promising.

6 Acknowledgments.

This work was supported in part by the National Science Foundation under Award 0722313 and by an equipment donation from NVIDIA Corporation.

References

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

[2] Neeta Bhate and A. Tokuta. Photorealistic Volume Rendering of Media with Directional Scattering. In *Third Eurographics Workshop on Rendering*, pages 227–245, Bristol, UK, May 1992.

[3] Subrahmanyan Chandrasekhar. *Radiative Transfer*. Clarendon Press, Oxford, UK, 1950.

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

[5] R. Danforth and R. Geist. Automatic mesh refinement and its application to radiosity computations. *Int. Journal of Robotics and Automation*, **15**:1:1 – 8, 2000.

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

[7] C. Donner and H. Jensen. Light diffusion in multi-layered translucent materials. *ACM Trans. on Graphics*, 24(3):1032–1039, 2005.

[8] Raanan Fattal. Participating media illumination using light propagation maps. *ACM Trans. Graph.*, 28(1):1–11, 2009.

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

[10] M. Gardner. Mathematical games: John conway's game of life. *Scientific American*, October 1970.

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

[12] R. Geist and J. Steele. A lighting model for fast rendering of forest ecosystems. In *Proc. of the IEEE Symposium on Interactive Ray Tracing (RT08)*, pages 99–106, and back cover, Los Angeles, California, August 2008.

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

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

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

[16] J. Kajiya and B. von Herzen. Ray tracing volume densities. *ACM Computer Graphics (SIGGRAPH '84)*, 18(3):165–174, July 1984.

[17] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 99–107, New York, NY, USA, 2010. ACM.

[18] I. Karlin, A. Ferrante, and C. Ottinger. Perfect entropy functions of the lattice boltzmann method. *Europhysics Letters*, 47:182–188, July 1999.

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

[20] E. Languénou, K. Bouatouch, and M. Chelle. Global illumination in presence of participating media with general properties. In *Fifth Eurographics Workshop on Rendering*, pages 69–85, Darmstadt, Germany, June 1994.

[21] N. L. Max. Efficient light propagation for multiple anisotropic volume scattering. In *Fifth Eurographics Workshop on Rendering*, pages 87–104, Darmstadt, Germany, June 1994.

[22] Patrick Min. Binvox. //http://www.cs.princeton.edu/ min/binvox, 2003.

[23] F. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Trans. on Visualization and Computer Graphics*, 9(2), April.

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

[25] H. Rushmeier and K. Torrance. The zonal method for calculating light intensities in the presence of a participating medium. In *Proc. SIGGRAPH '87*, pages 293–302, July 1987.

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

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

[28] J. Steele and R. Geist. Relighting forest ecosystems. In G. Bebis et al, editor, *ISVC 2009 Part I, LNCS 5875 (Proc.* 5^{th} *Int. Symp. on Visual Computing)*, pages 55–66, Las Vegas, Nevada, November 2009. Springer, Heidelberg.

[29] Nils Thürey, Ulrich Rüde, and Marc Stamminger. Animation of open water phenomena with coupled shallow water and free surface simulations. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 157–164, Vienna, Austria, 2006.

[30] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, Wokingham, England, 1992.

[31] X. Wei, W. Li, K. Mueller, and A. Kaufman. The lattice-boltzmann method for gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2), 2004.

7 Apppendix: Derivation of the Diffusion Equation.

The purpose of this section is to show that, as the lattice spacing, λ, and the time step, τ, both approach 0, the limiting behavior of the fundamental update (2) is the diffusion equation (3). A sketch of the derivation originally appeared in [11]. The present treatment includes significantly more detail. As noted earlier, the principal value of this derivation is as a prototype for related derivations.

We begin by expanding the left side of (2) in a Taylor series with respect to the differential operator

$$\nabla = (\partial/\partial\vec{r}, \partial/\partial t) = (\partial/\partial x, \partial/\partial y, \partial/\partial z, \partial/\partial t) \tag{10}$$

which handles 3 spatial dimensions and 1 temporal. This gives us

$$[(\lambda\vec{c_m}, \tau) \cdot \nabla] f_m(\vec{r}, t) + \frac{[(\lambda\vec{c_m}, \tau) \cdot \nabla]^2}{2!} f_m(\vec{r}, t) + ... = \Omega_m \cdot (f(\vec{r}, t)) \tag{11}$$

For the diffusion behavior we seek, it will be important for the time step to approach 0 faster than the lattice spacing. Specifically, we write

$$t = \frac{t_0}{\epsilon^2} \qquad \text{where} \quad t_0 = o(\epsilon^2)$$

that is, t_0 is a term that approaches 0 faster than ϵ^2, and

$$\vec{r} = \frac{\vec{r_0}}{\epsilon} \qquad \text{where} \quad \|\vec{r_0}\| = o(\epsilon)$$

It follows from the chain rule for differentiation that

$$\frac{\partial}{\partial t} = \epsilon^2 \frac{\partial}{\partial t_0}$$

and

$$\frac{\partial}{\partial r_\alpha} = \epsilon \frac{\partial}{\partial r_{0\alpha}} \qquad \text{for} \quad \alpha \in \{x, y, z\}$$

As is standard practice in lattice-Boltzmann modeling, we also assume that we can write $f(\vec{r}, t)$ as a small perturbation on this same scale about some local equilibrium, $f^{(0)}$, i.e.,

$$f(\vec{r}, t) = f^{(0)}(\vec{r}, t) + \epsilon f^{(1)}(\vec{r}, t) + \epsilon^2 f^{(2)}(\vec{r}, t) + ... \tag{12}$$

where the local equilibrium carries the total density, i.e., $\rho(\vec{r}, t) = \sum_{m=0}^{18} f_m^{(0)}(\vec{r}, t)$. Equation (12) is the Chapman-Enskog expansion from statistical mechanics [4], wherein it is assumed that any flow that is near equilibrium can be expressed as a perturbation in the so-called Knudsen number, ϵ, which represents the mean free path (expected distance between successive density collisions) in lattice spacing units.

Equation (11) now becomes:

$$\left[[\epsilon\lambda(\vec{c_m} \cdot \frac{\partial}{\partial \vec{r_0}}) + \epsilon^2 \tau \frac{\partial}{\partial t_0}] + \frac{[\epsilon\lambda(\vec{c_m} \cdot \frac{\partial}{\partial \vec{r_0}}) + \epsilon^2 \tau \frac{\partial}{\partial t_0})]^2}{2} + ... \right] (f_m^{(0)} + \epsilon f_m^{(1)} + ...) = \Omega_m \cdot (f^{(0)} + \epsilon f^{(1)} + ...) \tag{13}$$

Equating coefficients of ϵ^0 in (13), we obtain:

$$0 = \Omega_m \cdot (f^{(0)}(\vec{r}, t)) \tag{14}$$

i.e., $f^{(0)}$ is indeed a local equilibrium. In general, a local equilibrium need not be unique, and the choice can affect the speed of convergence [18]. Nevertheless, in this case it turns out that Ω has a one-dimensional null space. We observe that

$$\vec{v} = (\sigma_a, 1/12, ..., 1/12, 1/24, ..., 1/24)$$

(where entries 1 - 6 are $1/12$) satisfies $\Omega_m \cdot \vec{v} = 0$, all m, and so we must have

$$f_m^{(0)} = K v_m$$

where the scaling coefficient, K, is determined by the requirement that $\rho = \sum_m f_m^{(0)} = K \sum_m v_m = K(1 + \sigma_a)$. Thus we have:

$$f_m^{(0)}(\vec{r}, t) = \frac{v_m}{1 + \sigma_a} \rho(\vec{r}, t) \tag{15}$$

Similarly, equating coefficients of ϵ^1 in (13), we obtain:

$$\lambda(\vec{c_m} \cdot \frac{\partial}{\partial \vec{r_0}}) f_i^{(0)}(\vec{r}, t) = \Omega_m \cdot f^{(1)}(\vec{r}, t) \tag{16}$$

that is, after substituting (15),

$$\frac{\lambda v_m}{1+\sigma_a}(\vec{c_m}\cdot\frac{\partial}{\partial\vec{r_0}})\rho(\vec{r},t)=\Omega_m\cdot f^{(1)}(\vec{r},t) \tag{17}$$

We would like to solve (17) for $f^{(1)}$, but we cannot simply invert Ω, since it is singular. Nevertheless, we can observe that any vector comprising lattice-direction components,

$$(c_{0_\alpha},c_{1_\alpha},...,c_{18_\alpha}) \quad \text{where } \alpha\in\{x,y,z\}$$

as well as any of

$$(v_0c_{0_\alpha},v_1c_{1_\alpha},...,v_{18}c_{18_\alpha}) \quad \text{where } \alpha\in\{x,y,z\}$$

is an eigenvector of Ω with eigenvalue $-\sigma_t$. Thus, if we write

$$f_m^{(1)}(\vec{r},t)=Kv_m(\vec{c_m}\cdot\frac{\partial}{\partial\vec{r_0}})\rho(\vec{r},t)$$

and substitute into (17), we can determine that $K=-\lambda/((1+\sigma_a)\sigma_t)$ and so

$$f_m^{(1)}(\vec{r},t)=\frac{-\lambda v_m}{(1+\sigma_a)\sigma_t}(\vec{c_m}\cdot\frac{\partial}{\partial\vec{r_0}})\rho(\vec{r},t) \tag{18}$$

Finally, we need to equate ϵ^2 terms in (13), but here it will suffice to sum over all directions. We obtain:

$$\sum_{m=0}^{18}\left[\tau\frac{\partial f_i^{(0)}}{\partial t_0}+\lambda(\vec{c_m}\cdot\frac{\partial}{\partial\vec{r_0}})f_m^{(1)}+\frac{\lambda^2}{2}(\vec{c_m}\cdot\frac{\partial}{\partial\vec{r_0}})^2f_m^{(0)}\right]=0 \tag{19}$$

Substituting expressions (15) and (18) into equation (19) and observing that

$$\sum_{m=0}^{18}v_mc_{m_\alpha}c_{m_\beta}=(1/2)\delta_{\alpha\beta} \quad \text{for } \alpha,\beta\in\{x,y,z\}$$

we obtain

$$\frac{\partial\rho}{\partial t_0}-\frac{\lambda^2(1/\sigma_t-1/2)}{2\tau(1+\sigma_a)}\left(\frac{\partial^2\rho}{\partial r_{0_x}^2}+\frac{\partial^2\rho}{\partial r_{0_y}^2}+\frac{\partial^2\rho}{\partial r_{0_z}^2}\right)=0 \tag{20}$$

which, if we multiply through by ϵ^2, yields the standard diffusion equation,

$$\frac{\partial\rho}{\partial t}=D\nabla_{\vec{r}}^2\rho \tag{21}$$

with diffusion coefficient

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

The critical step is clearly a choice of Ω that both represents plausible collision events and has readily determined eigenvectors that can be used to solve for the first few components in the Chapmann-Enskog expansion.