Global illumination in sparse voxel octrees

Nelson Max¹

Accepted: 1 February 2021 / Published online: 8 March 2021
© The Author(s) 2021

Abstract

This paper enhances the voxel-to-voxel radiance shooting, propagation, and scattering algorithm of Max [20]. It reduces the memory requirements by storing the radiance only on the occupied cells of a sparse voxel octree and by sampling only 24 propagation direction bins instead of 96 or more. It gives a modification of the propagation algorithm to compensate for the larger solid angle of the direction bins and allows for position-dependent fully anisotropic scattering. For a volume of n^3 voxels, the computation time is $O(n^3(\log n)^{3/2})$.

Keywords global illumination · Octree · Multiple scattering

1 Introduction

Global illumination is an important aspect of realistic rendering. Here I consider multiple scattering global illumination in a volume with a participating medium. A popular technique for this is stochastic path tracing from the viewpoint into the scene. Such Monte Carlo methods must be repeated for each new viewpoint, and are often slow to converge. They are noisy unless there are many samples per pixel, although modern noise reduction methods can help. Other methods for participating media, *e. g.*, [19], compute and store photons or beams propagated stochastically from the light sources in an initial viewpoint-independent step, and then collect them for a final gather bounce into the viewing rays.

Another family of techniques applies finite element methods to precompute the direction-dependent radiance in each voxel, which can later be used for a final gather along viewing rays from any viewpoint, with traditional color/opacity compositing. The volume is discretized into voxels, and the light flow direction is discretized into direction bins. Max [20], which I enhance here, claimed that this global illumination precomputation can be accomplished in time $O(n^3 \log n)$ for a volume of n^3 voxels, by propagating the

flux in each direction bin, plane by plane through the volume, in repeated shooting passes which efficiently amalgamate the propagating flux from multiple shooting voxels. However, as discussed below, the computation time for a fixed allowed error is actually $O(n^3(\log n)^{3/2})$.

Max [20] stored 96 or more radiance direction bins per voxel, which requires an impractical amount of storage for high-resolution volumes. Here I decrease the storage by using a sparse voxel octree [11] and storing only 24 propagation direction bins at each octree node. The direction bins are formed by subdividing into four subsquares each square face of an axis-aligned direction cube centered at the shooting voxel. Flux from the shooting voxel propagates into the direction bin volume subtended by the subsquare, through successive receiving planes of voxels parallel to the bin's direction cube face, accumulating transparency/opacity along the way. In the absence of any opacity, the flux reaching a given receiving voxel should be proportional to its volume-to-volume form factor, which is approximately $1/r^2$, where r is the distance from the shooting to the receiving voxel.

The method of [20] is designed to propagate flux equally to all voxels in the receiving square in which a receiving plane intersects the direction bin volume. This is good approximation if the direction cube face is finely subdivided. But for 24 direction bins, where each cube face is divided into only 4 subsquares, this is a poor approximation, because the form factor of the receiving square corner voxel corresponding to a cube diagonal direction should be only $1/3$ as large as that for the corner voxel corresponding to an axis direction. One contribution of this paper is to modify the propagation method

Supplementary Information The online version contains supplementary material available at <https://doi.org/10.1007/s00371-021-02078-6>.

✉ Nelson Max
max@cs.ucdavis.edu

¹ Department of Computer Science, University of California, Davis, CA 95616, USA

so that it more closely approximates the correct form factors. This is done by replacing the fixed equal weights in [20], used to determine how a cell propagates flux to its direct neighbors, by unequal ones that depend on the distance between the shooting plane and the currently propagating plane.

Another contribution is implementation in the rendering system described in Max *et al.* [21], where arbitrarily complex polygonal models are recursively diced into sparse voxel octree cells. Polygons labeled as belonging to smooth surfaces have their area-weighted cell intersection fragments added into an averaged plane equation, while others labeled as contributing to the participating medium have their fragments used to build a distribution of the microflake normals of the cell, which determines the cell's extinction coefficient, scattering coefficient, and scattering phase function (see [21] and its "Appendix"). This allows fully anisotropic phase functions, depending on both the incoming and scattering directions, rather than just on the angle between them, as in the Mie, Raleigh, and Henyey Greenstein phase functions. For such phase functions, I use the spherical harmonics method of [21] for representing the surface normal distribution to effectively compute on the fly the bin-to-bin scattering matrix for each cell.

In summary, the contributions of this paper are:

- decreased storage costs from using a sparse voxel octree and only 24 direction bins,
- a revised propagation method to compensate for the larger solid angle of these direction bins,
- a correction of the computation time of both methods from $O(n^3 \log n)$ to $O(n^3 (\log n)^{3/2})$,
- implementation of fully anisotropic scattering using a spherical harmonics representation of the microflake normal distribution,
- automatic differentiation in the gradient-based optimization of the weighting coefficients for flux propagation to neighbors,
- a method for computing volume-to-volume form factor integrals for configurations that have $1/r^2$ singularities from touching volumes,
- propagation to an extra layer of voxels along each slanted face of the direction bin pyramid, and
- a more detailed description of the algorithm in [20], including code.

2 Related work

Cerezo *et al.* [3] give a good summary of the methods for rendering participating media that were developed by 2005, and Novak *et al.* [23] review more recent Monte Carlo methods. Although much recent work on participating media uses stochastic raytracing, this section concentrates on applica-

tions of the finite element method for thermal radiation heat transfer in mechanical engineering, since these are more relevant to the current paper.

In computer graphics, Goral *et al.* [10] first applied the finite element method to patches on diffusely reflecting surfaces, and Rushmeier and Torrance [26] first applied it to diffusely scattering volume elements. Immel *et al.* [12] discretized the light flow directions into solid angle bins for global illumination on non-diffuse surfaces, and Tan [28] and Kaplanyan *et al.* [16] represented the angular dependence of radiance in volume elements by spherical harmonics. Max *et al.* [21] gave various ways to represent a spatially varying distribution of normals for a participating medium of microflakes, one of which used spherical harmonics. Jacob *et al.* [13] also use spherical harmonics for the same purpose for the microstructure of yarn fibers, giving very realistic images of knitting. Kaplanyan *et al.* [16] use spherical harmonics in this way to compute volumetric occlusion.

Cohen *et al.* [5] describe a shooting method for radiosity on diffuse surfaces, where each surface patch maintains an unshot radiosity for future shooting to all other patches visible to it, as well as maintaining a total accumulated radiosity used in the final rendering. Max [20] used this shooting philosophy, with a way of amalgamating the flux propagating in a direction bin from multiple shooting cells to efficiently approximate the form factor and accumulated transparency integrals needed for the finite element method.

The method reported here combines many of the above ideas, using the sparse voxel octree structures of [11], [6] and [21], the discretization of the light flow directions of [12] and [20], the shooting scheme of [5], the propagation scheme of [20] using the solid angle bins of [12], and the spherical harmonics representation for the distribution of microflake normals of [13], [16], and [21].

Chandrasekhar [4] called the idea of propagating flux in a finite collection of discrete directions the "discrete ordinates method." It was applied to rendering clouds by Patmore [24], but suffers from the "ray effect" of missing propagation in other directions. Elek *et al.* [7] decrease this effect by propagating anisotropy, represented by the single Henyey Greenstein parameter g , as well as the flux, in a separate propagation grid with one axis along the external illumination direction, thus accurately representing that direction. Fattal [8] uses a large number of propagation directions (481 in his examples) to mitigate the ray effect.

Stam [27] introduced to computer graphics the diffusion approximation for multiple isotropic scattering in an optically thick medium, using a multigrid method to solve the resulting partial differential equation. Jacob *et al.* [13] extended the diffusion approximation to the case of anisotropic scattering. This method does not correctly propagate flux across empty regions in the volume. Koerner *et al.* [17] adjust the diffusion coefficient with an expression that

Fig. 1 Flow of flux data from the light source to the viewing ray. The rounded-corner boxes represent multiplication by the scalar, vector, or matrix quantity inside them

depends on the flux and the magnitude of its gradient to solve this problem. Max *et al.* [22] solved this problem instead by volume rendering the current multigrid diffusion approximation from the points of view of voxels at the edges of a cloud, to get the correct boundary conditions for the next iteration. The method here correctly propagates the flux across all voxels, even those in empty regions of the octree with zero extinction where no unshot or received flux is stored.

A recent group of methods fits multiple scattering approximation parameters to converged path traced training examples and then applies the approximation to a given cloud. Bouthors *et al.* [2] fit the size and location of, and the multiply-scattered transmission from, the surface region that collects most of the multiply scattered external illumination reaching a point x in a slab of participating medium of thickness t between two parallel planes, using a small number of 1D and 2D texture tables. Kallweit *et al.* [15] use a deep neural net to predict the radiance at a point x in a cloud from density samples at a scaled hierarchy of small volume sample grids surrounding x . The net was trained on 75 synthetic cloud models. Ge *et al.* [9] use a neural net to fit the path-traced multiply-scattered radiance at all positions and directions in a homogeneous medium resulting from a ray at the origin along the z axis, as a function of 6 parameters for position, direction, and material properties. This is used to compute the contribution of virtual ray lights, created by tracing random rays from the light source into the medium.

3 Shooting method

The overall radiance shooting computation is summarized in Fig. 1, whose central stack of four boxes represents quantities stored at each full voxel, with the top box storing one RGB color, and the lower three boxes storing a vector of 24 colors, one for each direction bin. The attenuated external illumination is found using the “slab method” of Kajiya and von Herzen [14] and stored in the octree as the Initial flux. At each full voxel, it is multiplied by the scattering coefficient and a ray-to-bin vector, computed by integrating the phase

function for the illumination direction over each bin of outgoing scattered directions, and stored as the initial Unshot flux vector. In the Flux propagation iteration, described in detail in the following section, and in “Appendix A”, the Unshot flux is spread per direction bin through the volume, layer by layer, and added to the total Received flux vector at each full voxel it passes through, and also stored in a temporary Treceived vector. This Treceived vector is then multiplied by the voxel’s scattering coefficient and a bin-to-bin scattering matrix, and the result is used to update the Unshot flux for the next shooting iteration.

The radiance propagation is iterated until convergence on a voxel grid corresponding to a chosen global illumination octree level and then interpolated to the lowest leaf level using joint bilateral upsampling [18] keyed to the directionally averaged extinction coefficient of the participating medium. The radiance values per direction bin at the leaves are averaged up the octree levels, stopping at the level just below the global illumination level, and then, the values at the global illumination level are averaged up the rest of the levels to the root.

If the phase function is constant for the whole volume, the bin-to-bin scattering matrix is precomputed by integrating the phase function over both the incoming and outgoing direction bins, using a 4D version of Simpson’s rule. For the spatially varying phase functions using spherical harmonics, the matrix for each spherical harmonic basis function is precomputed by integrating its corresponding phase function. When computing the scattering from a cell, the contributions of these precomputed matrices, weighted by the spherical harmonics coefficients stored in the cell, are added together.

In the final gathering pass, as each viewing ray passes through the voxels, accumulating volume attenuation along the way, it collects the Initial flux for single scattering, using the exact illumination-direction to viewing-direction phase function, giving voxel-precision volume shadows, and also collects the multiple scattered Received flux, multiplied by the current voxel’s scattering coefficient and element-wise (*i.e.*, dot product) by a bin-to-ray vector computed for this viewing ray. During the final gathering, the appropriate octree level is chosen at each ray position according to the ray differential, as in [6], increasing rendering efficiency and enhancing anti-aliasing.

4 Propagation method

To simplify things, think of the octree level at which the global illumination is computed as having cells of side 1, with vertices on an integer lattice, with the extinction coefficient adjusted to compensate for this change in length units. Voxels are named by their lowest index vertex. As stated in Introduction, the method of [20] is organized to propa-

Fig. 2 (top) 3D direction bin 0. (middle) 2D direction bin 0. The shooting pixel at (i_0, j_0) is outlined, the direction bin 0 wedge with vertex at the pixel center is in light blue dotted lines, and the values of tempwork are shown in column $i_0 + l$, for $l = 5$. (bottom) For 3D, the square of tempwork values at layer $l = 5$ from the shooting voxel

gate flux in a direction bin from one layer of voxels to the next. The first bin 0 of the 24 bins consists of directions (x, y, z) with $x > 0$, $0 \leq y \leq x$, and $0 \leq z \leq x$, whose un-normalized vectors fill up a semi-infinite direction pyramid with apex at the origin shown at the top of Fig. 2. Flux at voxel (i, j, k) flowing in directions in this bin will go mostly to the four dominant neighboring voxels in the next x layer, namely $(i + 1, j, k)$, $(i + 1, j + 1, k)$, $(i + 1, j, k + 1)$, and $(i + 1, j + 1, k + 1)$, but flux starting near the faces of the voxel, flowing in directions near the slanted $y = x$ and $z = x$ faces of the direction bin, can also reach voxels $(i, j + 1, k)$, $(i, j + 1, k + 1)$, and $(i, j, k + 1)$ in the shooting layer, and $(i + 1, j + 2, k)$, $(i + 1, j + 2, k + 1)$, $(i + 1, j + 2, k + 2)$, $(i + 1, j + 1, k + 2)$, and $(i + 1, j, k + 2)$ in the next layer. Max [20] only dealt with the first four dominant neighbors above, as will the rest of this section. Propagation to the other neighbors is discussed in “Appendices” A and B.

For a simpler example, consider first the 2D case code fragment below, which propagates the flux in pixel (i, j) half to pixel $(i + 1, j)$ and half to pixel $(i + 1, j + 1)$.

```
// Initialize by unshot flux from x = i0
for (j = 0; j < tsize; ++j)
    work[j] = unshot[i0, j];
for (i = i0 + 1; i < tsize; ++i) {
    for (j = 0; j < tsize; ++j) {
        tempwork[j] = 0.5*work[j];
        if (j > 0) // propagate on diagonal
            tempwork[j] += 0.5*work[j-1];
        received[i][j] = tempwork[j];
    }
}
// Update work array
// accounting for transparency
for (j = 0; j < tsize; ++j) {
    transp = exp(-len*extinction[i][j]);
    work[j] = tempwork[j]*transp;
}
```

Variable tsize is the dimension of the square of pixels, len is the average length of a ray within a layer slab, and received[i][j] represents flux passing through cell (i, j) , as shown in Fig. 1.

Suppose there is no extinction and the transparency is 1, so the work and tempwork arrays are the same. Assume a delta function unshot array, with $\text{unshot}[i][j] = \delta(i - i_0)\delta(j - j_0)$, meaning $\text{unshot}[i][j]$ is 1 when i is i_0 and j is j_0 , and is 0 otherwise. The columns of received[i][j], produced by iterations of the loop over column index i , are $(1, 0, 0, 0)^T$, $(1/2, 1/2, 0, 0)^T$, $(1/4, 1/2, 1/4, 0)^T$, and $(1/8, 3/8, 3/8, 1/8)^T$, for $tsize = 4$. In general, there are $l!/(k!(l - k)!)$ ways of choosing a path with l segments, k of which are diagonal, and $l - k$ of which are horizontal, to get from pixel (i_0, j_0) to pixel $(i_0 + l, j_0 + k)$, so received[i0 + l][j0 + k] will be given by the binomial distribution $B(k, l, .5)$, where

$$B(k, l, p) = \frac{l!}{k!(l - k)!} p^k (1 - p)^{l - k}. \quad (1)$$

Instead, the goal is to distribute the flux equally to all cells in column $i_0 + l$, as shown in the middle of Fig. 2. The following modification will transmit flux 1 to tempwork[i][j] for all the cells (i, j) fully within the semi-infinite direction wedge $j_0 + .5 \leq y \leq x - (i_0 + .5)$ shown in the middle of Fig. 2 with apex centered at $(i_0 + .5, j_0 + .5)$, flux 1/2 to cells sliced in half by the edges of that wedge, and nothing to cells completely outside that wedge.

```
1 // Initialize by unshot flux from x = i0
2 for (j = 0; j < tsize; ++j) {
3     work[j] = unshot[i0][j];
4     edge[0][j] = unshot[i0][j];
5     edge[1][j] = unshot[i0][j];
6 }
7 for (i = i0 + 1; i < tsize; ++i) {
8     float l = i - i0;
9     for (j = 0; j < tsize; ++j) {
10         tempwork[j] = 0.5*work[j];
```

```

11     if (j > 0) // propagate on diagonal
12         tempwork[j] += 0.5*work[j-1];
13     received[i][j] = tempwork[j]/l;
14 // Modify tempwork using edge arrays
15     tempwork[j] += 0.5*edge[0][j];
16     if(j > 0) // update by diagonal edge
17         tempwork[j] += 0.5*edge[1][j-1];
18 }
19 // Update work and edge arrays,
20 // accounting for transparency
21 for (j = 0; j < tsize; ++j) {
22     transp = exp(-len*extinction[i][j]);
23     work[j] = tempwork[j]*transp;
24     edge[0][j] *= transp;
25     if (j > 0)
26         edge[1][j] = edge[1][j-1]*transp;
27     else
28         edge[1][j] = 0.;
29 }
30 }

```

By line 13, the sum of the contributions to `tempwork` in column $i = i_0 + l$ is l , so when they are multiplied by the “2D form factor” $1/l$, energy is conserved. With $O(n^2)$ work, this code propagates the flux from n cells in layer i_0 to all the $\Omega(n^2)$ 2D cells to its right, thus approximating the effect of $\Omega(n^3)$ form factors and transparency integrals.

The two edge arrays `edge[0][j]` and `edge[1][j]` propagate the flux and accumulate the transparency along the horizontal and diagonal edges of the direction wedge. In the delta function no extinction case above, after line 12, column $i_0 + l$ will have the `tempwork` values $(1/2, 1, 1, \dots, 1, 1, 1/2)^T$, as shown in the middle of Fig. 2, and after line 17, these values will be modified by the edge arrays, becoming $(1, 1, 1, \dots, 1, 1, 1)^T$, ready for the next propagation step in lines 9 to 12.

A detailed description of the 3D version of this scheme is given in online “Appendix A”, together with the complete code for direction bin 0. Here is a brief summary. The goal is to propagate a unit flux in direction bin 0, so that before being divided by l^2 , the values of `tempwork` in a vertical slice of the pyramid at the top of Fig. 2 are as shown in the square at the bottom of that figure. If this pattern can be modified by restoring to 1 the values that are $1/2$ and $1/4$, then the next slice can be produced by adding 0.25 times the flux in a cell to each of its four dominant neighbors in the next slice, by analogy to lines 10 through 12 in the code above. The pattern modification is accomplished by propagating the flux along the four triangular faces of the direction bin pyramid at the top of Fig. 2 by the method in the 2D case above, and using the resulting values to restore the boundary values of the square to 1, by analogy to the way the two edge arrays `edge[0][j]` and `edge[1][j]` are used on lines 15 to 17 above.

5 Redistributing flux

For direction bin 0, where flux propagates from left to right, any given x -slice should receive flux from all the slices to its left, so the above method maintains $O(n)$ different propagating 2D layer arrays, each initialized by the unshot flux from a different x -slice, all moving together through each successive receiving slice. This approximates the effect of $\Omega(n^6)$ form factors and transparency integrals with $O(n^4)$ computation time. In [20] I claimed that this time cost can be decreased to $O(n^3 \log n)$ by taking the propagated flux originating as unshot at a slice at x -separation s from the receiving slice and redistributing it half each to the two layers storing the flux originating at slices at separations $s - a$ and $s + a$, where $a \ll s$. Aside from somewhat different transparency accumulation, the main error in the resulting received flux comes from different $1/s^2$ weights in the equation below:

$$\frac{1}{2(s-a)^2} + \frac{1}{2(s+a)^2} = \frac{s^2 + a^2}{(s^2 - a^2)^2} = \frac{1}{s^2} \left(\frac{1 + a^2/s^2}{(1 - a^2/s^2)^2} \right). \quad (2)$$

The expression in the large parentheses at the right approaches 1 as the ratio a/s approaches 0, so for any a and any $\epsilon > 0$ there is an s so that the relative error

$$\left(\frac{1}{2(s-a)^2} + \frac{1}{2(s+a)^2} - \frac{1}{s^2} \right) / \left(\frac{1}{s^2} \right) = \frac{1 + a^2/s^2}{(1 - a^2/s^2)^2} - 1 < \epsilon. \quad (3)$$

Start with $a = 1$, and let c be the smallest s for which this inequality is true. It is then also true for any $s > c$, so from c on, every other layer can be redistributed to its two adjacent layers. The relative error depends only on the ratio a/s , so from separation $2c$ on, we can use $a = 2$ and redistribute the flux in every other remaining layer to its two adjacent remaining layers.

```

0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 2 3 4 5
0 2 3 4 5 1
0 2 4 5 1 6
0 2 4 5 1 6 3
0 4 1 6 3 7
0 4 1 6 3 7 5
0 4 1 3 7 5 2
0 4 1 3 7 5 2 6
0 4 3 5 2 6 8
0 4 3 5 2 6 8 7
0 4 3 5 6 8 7 1
0 4 3 5 6 8 7 1 2
0 3 6 7 1 2 9

```


0	3	6	7	1	2	9	8
0	3	6	7	2	9	8	5
0	3	6	7	2	9	8	5
0	3	6	2	8	5	1	4
0	3	6	2	8	5	1	4
0	3	6	2	8	1	4	9
0	3	6	2	8	1	4	9
0	3	6	2	8	1	4	9

This process can continue using higher powers of 2 for a , as shown in the pattern above. The horizontal axis represents the shooting layer j , the vertical axis represents the receiving layer k , counting downwards from the top, and the separation s is $k - j$. The numbers represent the index in an array of layers for the `work`, `edge`, and `corner` arrays used to propagate the flux. A blank space in column j means that shooting layer j 's flux has been redistributed, and its storage index can be reused for another shooting layer. Because of the recursive nature of this successive redistribution, using at most c layers at each power of two spacing, and at most $\log n$ powers of two, where n is the resolution of the octree level where the flux propagation is done, the total number of layers required is less than $c \log_2 n$. A closer estimate is $c(1 + \log_2(n/c)/2)$, but in any case, assuming ϵ and c are independent of n , the number of layers to be processed is $O(\log n)$ so I concluded in [20] that the total computation time is $O(n^3 \log n)$ instead of $O(n^4)$. However, the assumption that c does not depend on n is false, because the error, an increase in flux since the difference in Eq. (3) is positive, compounds as the extra flux received by a layer when its neighbors are redistributed is again increased if that layer itself is redistributed. Since $(1+x)^{\log n} = 1 + (\log n)x + O((\log n)^2 x^2)$, we must use $\epsilon/\log n$ on the right hand side of Eq. (3) instead of ϵ .

Using the fact that $1/(1-x) = 1+x + O(x^2)$, the expression in the large parentheses at the right of Eq. (2) is $1 + 3a^2/s^2 + O(a^4/s^4)$, so, taking $a = 1$, and $s = c$ in Eq. (3), we get the condition

$$\frac{1 + 1/c^2}{(1 - 1/c^2)^2} - 1 = \frac{3}{c^2} + O\left(\frac{1}{c^4}\right) < \frac{\epsilon}{\log n}. \quad (4)$$

So $c \approx \sqrt{3 \log n / \epsilon} = O(\sqrt{\log n})$, $c \log n = O((\log n)^{3/2})$, and the total cost is $O(n^3 (\log n)^{3/2})$.

6 Weights and their optimization

As discussed in Introduction, equal propagation to a cell's four primary neighbors will give incorrect form factors, since dividing by l^2 does not account for the diagonal distance between a shooting cell and a receiving cell. The correct diagonal distance cannot be used if the same flux propagation step is to combine the flux from all cells in a shooting layer. Instead, unequal weights are used. For the 2D case above, the four factors of 0.5 in lines 10, 12, 15, and 17

of the code above would be replaced, respectively, by four `weightswork_weight[0][1]`, `work_weight[1][1]`, for propagation, and `edge_weight[0][1]`, `edge_weight[1][1]`, for modification, which depend on the separation $l = i - i_0$ between the shooting and receiving layers. In the 3D version the factors of 0.25 are similarly replaced by weights. For direction bin 0 and propagating cell (i, j, k) , the neighbor $(i+1, j, k)$ will have the largest weight, `work_weight[0][1]`, the two neighbors $(i+1, j+1, k)$ and $(i+1, j, k+1)$ will have medium weight, `work_weight[1][1]`, and neighbor $(i+1, j+1, k+1)$ will have the smallest weight, `work_weight[2][1]`, so that cells in a layer which are nearer to the x axis will get more flux. When also considering the weights for propagating along the four triangular faces, and for doing the modifications in preparation for propagating to the next layer, there are 17 different weights per layer: `weight[k][l]`, for $k = 0, 1, \dots, 16$.

The goal of the optimization is to adjust these weights so that the computed flux shot from cell (i, j, k) in direction bin b , received at cell (l, m, n) , in the absence of extinction, is a good approximation to the volume-to-volume form factor

$$F(i, j, k; l, m, n; b) = \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{\chi(r, s, t; b)}{r^2 + s^2 + t^2} dx dy dz du dv dw \quad (5)$$

where $r = l + u - i - x$, $s = m + v - j - y$, $t = n + w - k - z$, and $\chi(r, s, t; b)$ is 1 if the ray direction vector (r, s, t) is in direction bin b and 0 otherwise. Note that the factor of π in the denominator of Eq. (33) in [26] is missing here because we are considering radiance rather than radiosity. Methods for computing these form factors are described in the next section and in "Appendix B". The 17 weights per layer, for a volume with, say, 256 layers, would be an unwieldy number to optimize. Instead, I have assumed that the weights vary more slowly as the distance l between layers gets large, so that they can be approximated by a truncated Laurent series

$$\text{weight}[k][l] = a_k + \frac{b_k}{l} + \frac{c_k}{l^2}. \quad (6)$$

To allow for more degrees of freedom for small layer separations of 1 or 2 where the form factors are larger and have more impact, I also defined

$$\text{weight}[k][1] = d_k \quad (7)$$

$$\text{weight}[k][2] = e_k \quad (8)$$

so that Eq. (6) is only applied if $l \geq 3$. For each weight type k , there are 5 constants a_k , b_k , c_k , d_k , and e_k , giving a manageable number of $5 \times 17 = 85$ variables to optimize.

Fig. 3 Form factor ratios for a 128^3 volume, with the desired ratio 1 at the break between lighter green and darker green. left: for 85 parameters, middle: for $17(n-1) = 2159$ parameters, right: for a 256^3 volume using the same 85 parameters as on the left, far right: color scale

The optimization minimizes a least squares cost

$$\text{Cost}(\{a_k\}, \{b_k\}, \{c_k\}, \{d_k\}, \{e_k\}) = \sum_{l=1}^{255} \sum_{m=0}^{l+1} \sum_{n=0}^{l+1} \left(\frac{\text{received}[l][m][n]}{l^2} - F(0, 0, 0; l, m, n; 0) \right)^2 \quad (9)$$

where the first direction bin $b = 0$ is used in Eq. (5), the shooting cell is at $(0, 0, 0)$, and $\text{received}[l][m][n]$ is what would be computed by the algorithm in “Appendix A” using the weight arrays specified in Eqs. (6), (7), and (8). In order to conserve energy and make the average brightness of an image correct, an adjustable weight times the square of the difference between the summed $\text{received}[l][m][n]/l^2$ terms and the summed $F(0, 0, 0; l, m, n; 0)$ terms was sometimes added to this cost, to force this difference toward zero. The cost was minimized by the L-BFGS gradient-based nonlinear optimization algorithm [1], with initial values given in the main program at the end of “Appendix A”.

Computing the partial derivatives involved in the gradient is complicated, because the `propagate()` code in “Appendix A” has many multiplications and additions of terms involving the weighting coefficients. Iterating as the propagation progresses through the receiving layers, this would result in very complicated high degree polynomials of the 85 parameters determining these coefficients. Therefore, I used automatic differentiation to compute the partial derivatives with respect to these parameters. Even though I needed a gradient, rather than just a single derivative, I chose to use the forward method, because it was trivial to code. I made a `Value_and_Derivative` data structure type containing two doubles, called `value` and `derivative`, overloaded the normal arithmetic operators like `+` and `*` to operate on these structures according to the rules of calculus, and then made an overloaded copy of the `propagate()`

code of “Appendix A”, just changing the types of the input weights, the intermediate variables, and the output array received to `Value_and_Derivative`. For each of the 85 parameters, the derivatives of the input weights can be computed from Eqs. (6), (7), and (8), which are linear in these parameters. Automatic differentiation in the overloaded `propagate()` code then computes the partial derivatives of the `received` array entries with respect to that parameter, which are used to find the partial derivatives of the cost in Eq. (9).

I said above that it would be difficult to optimize all $17(n-1)$ weights for the $n-1$ possible layer separations in a volume of n^3 voxels, but in fact I was able to do this for $n = 128$ in several days using L-BFGS, in order to compare the errors in the approximate form factors. Figure 3 shows a 2D slice through the direction bin pyramid in Fig. 2 corresponding to triangle OAC. The color of each voxel slice rectangle indicates the ratio

$$\frac{\text{received}[l][m][n]}{l^2 F(0, 0, 0; l, m, n; 0)} \quad (10)$$

and an improvement is visible with the extra degrees of freedom. However, the 85 parameters in Eqs. (6), (7), and (8)), optimized for 256 layers, were used to produce the rendered images in Results section. These optimized parameters are given in online “Appendix C”. Figure 3 also shows the ratio from the 85 parameters, optimized for $n = 128$, but used for a volume with $n = 256$, demonstrating that the truncated Laurent series fit in Eq. (6) is valid beyond its fitting range.

7 Computing form factors

Because we are considering nearby, touching, and even coinciding shooting and receiving cells, we need to compute

the six dimensional integral in Eq. (5) accurately, instead of approximating it by $1/((l-i)^2 + (m-j)^2 + (n-k)^2)$ as [26] does. By translation invariance of the quantities r , s , and t defined after Eq. (5), it is sufficient to compute the form factors for $(i, j, k) = (0, 0, 0)$, and from rotation and reflection invariance, it is sufficient to consider only the first direction bin $b = 0$ discussed above. There are three cases to consider. First, if cell (l, m, n) lies completely within every semi-infinite direction pyramid with apex anywhere in cell $(0, 0, 0)$, which is the case for most receiving cells with non-zero form factors, then the 6D version of Simpson's rule can give accurate results.

In the second case, when the receiving cell does not touch the shooting cell but is only partially within some of these direction pyramids, Simpson's rule cannot be used, because the inequalities $0 \leq s \leq r$, $0 \leq t \leq r$ for the direction bin 0 pyramid where $\chi(r, s, t; 0)$ is 1 include only some of the weighted regular grid samples for the 6D Simpson's rule, destroying its good convergence properties derived from its position-varying weights. As an alternative, I use Riemann summation, taking samples $(x, y, z; u, v, w)$ at the subcube centers of regular grids subdividing the cells $(0, 0, 0)$ and (l, m, n) and evaluate the four inequalities on (r, s, t) . If any inequality is violated, the weight is 0. If all four inequalities are satisfied strictly, with $<$ instead of $=$, then I include the full term $\Delta^6/(r^2 + s^2 + t^2)$, where Δ is the side of the subdivision's subcubes. If exactly one of these inequalities is an equality, (r, s, t) lies on a face of the direction pyramid, and I weight this contribution by 1/2, because half of the corresponding 6D hypercube is shared by an adjacent direction bin. If $r = s = t > 0$, (r, s, t) is along the body diagonal edge of the direction pyramid, and I weight this contribution by 1/3, because the hypercube is shared equally by 3 direction bins. Finally, for the other three edges of the pyramid, when a different pair of inequalities become equalities, the hypercube is shared equally by 4 direction bins, so I weight the contribution by 1/4. I adapt the fineness of the subdivision to the length of the vector (l, m, n) , so that finer subdivisions with smaller Δ are used when the receiving cell is closer to the origin, and $1/(r^2 + s^2 + t^2)$ changes more rapidly, or more relevant to the convergence of the Riemann sums, has a larger second derivative.

In the third case, when the cells touch or coincide, it is necessary to use subcubes with varying Δ , to account for the infinitely increasing second derivative near the singularities of $1/(r^2 + s^2 + t^2)$. Ideally, infinitely many samples should be taken, and the method in online "Appendix B" effectively does this.

8 Serial and parallel versions

The algorithm in "Appendix A" maintains the work, edge, tempwork, tempedge, and corner propagation arrays for each of the $O((\log n)^{3/2})$ actively propagating layers for a direction bin, as discussed in Sect. 5, accessed via a list of active layer indices. In the bin-sequential serial version of the algorithm, for each bin in serial order, and for each receiving layer in the order of the main coordinate direction axis of the current bin, all the active layers are propagated one more step to the current receiving layer. Appropriate layers are redistributed to other active layers, and their memory layer indices freed. Next the sum of the received flux at the receiving layer, from all the actively propagating layers, is stored as Treceived in the occupied octree nodes, and used to increment their total accumulated received flux. Then Treceived is scattered via multiplication by the bin-to-bin scattering matrix to increment their unshot vectors. The unshot flux is then used to initialize the work, edge, and corner arrays for propagating the current direction bin's flux into the next receiving layer, using the first free layer memory index for the array storage, and also used for propagating unshot flux in other bins when they are processed in the serial order. The complete processing passes over all the bins are iterated in an outer loop over the number of bounces of multiple scattering, until the total unshot flux, which decreases by absorption or by exiting the octree volume, becomes small enough to indicate convergence. Because flux is scattered and ready for shooting as soon as it is received, this method converges faster than if the flux were scattered only at the end of a complete shooting iteration over all the bin directions. Assuming the initial voxel subdivision at the octree level at which this propagation is performed is fine enough that multiple scattering within a single voxel is unlikely, the number of complete shooting iterations until convergence should be almost independent of further increases in the resolution n , as is true in table 1, and is thus a constant in the $O(n^3(\log n)^{3/2})$ asymptotic computation time estimate.

For the parallel version of the algorithm, in each complete shooting iteration, a separate task is created for propagating the flux in each bin direction, and these tasks are executed whenever CPU threads are available. Each task allocates its own $O(n^2(\log n)^{3/2})$ temporary storage for the propagation arrays, and frees it on termination. Because of the hazard of potential concurrent attempts to increment the same bin element of the unshot flux vector at an octree node, it is not possible to scatter the flux into all bins as soon as it is received. This is why the temporary Treceived vectors in Fig. 1 are required. It is safe, however, to scatter the flux into the same direction bin component of the unshot vector as the bin that it is propagating in, since then other threads will not attempt to increment that component. This helps increase convergence somewhat, particularly when there is

Fig. 4 Cloud, rendered at 512^3 resolution. left: with direct illumination only, middle: with direct plus indirect illumination, with flux propagation on 256^3 voxels bilaterally upsampled to 512^3 , right: middle minus left, showing the contribution of indirect illumination

mainly forward scattering. After all the propagation tasks for a complete shooting iteration have completed, the scattering is computed at all full octree nodes for all non-equal pairs of direction bins. On the 3.6 GHz Intel I7 - 4790 CPU used for timing tests, with 4 cores and 8 threads, the speed up from parallel threads more than compensates for the slower convergence due to accumulating more received flux before scattering it for propagation in other direction bins.

9 Results

Figure 4 shows a diffusely scattering cloud, whose density was modeled by a collection of ellipsoids and spheres, plus multiple cosine waves with different 3D wave vectors, and Perlin noise [25].

Table 1 gives computation times for different resolutions of the octree used for the flux propagation in this scene. The iterations were terminated when the unshot flux was less than 0.001 times the initial unshot flux from the attenuated external illumination. The attenuation of the external illumination, the global illumination shooting, and the bilateral upsampling to a rendering octree of twice the propagation resolution were all computed in parallel by 8 CPU threads. But the computation of the initial unshot flux from the attenuated external illumination was serial. From the table, one can confirm the $O(n^3(\log n)^{3/2})$ asymptotic computation time estimate. Not included are the $O(n^3)$ times for the octree construction, the slab method computation of the attenuated external illumination, and the final ray casting from the viewpoint, all of which are needed for even the single scattering contribution, and the $O(1)$ cost of computing the illumination-direction to direction-bin scattering vector, and the bin-to-bin scattering matrix.

Figure 5 shows a different view from further below of a diffusely scattering cloud, rendered at 128^3 resolution, by

Table 1 Resolution, maximum number of propagating layers, shooting iterations, and times in seconds for initial unshot flux, flux propagation, and bilateral upsampling

Resol.	Layers	Iter.	Unshot	Prop.	Upsample
16^3	14	10	.0027	.363	.0031
32^3	18	10	.0202	3.63	.0227
64^3	22	10	.152	38.2	.171
128^3	26	10	1.61	396	1.36
256^3	30	9	9.30	3908	20.3

the method here and by the method of Rushmeier and Torrance [26], both running in parallel on 8 CPU threads. The difference between the far left image and a corresponding image generated by the method of [26] is multiplied by a combined factor of 120 and, when added to a 0.5 gray level, still fits within the gray scale range [0, 1], so the errors are less in absolute value than $1/240$, about a single step in an 8 bit range of 255 values. This 128^3 voxel volume had only 434,110 occupied octree nodes out of the total of 2,396,745 nodes for a full octree (or 2,097,152 voxels in the grid). This shows the storage efficiency of the sparse octree.

Figure 6 (far left) shows a cloud, with a forward scattering Henyey Greenstein phase function, depending only on the scattering angle between the incident and scattered rays, with an average scattering angle cosine $g = 0.4$. It was rendered using 256^3 octree resolution and 300 by 300 image resolution by the methods of this paper, with the global illumination bilaterally upsampled from a 128^3 calculation. Note that due to this forward scattering, the light penetrates more deeply into the cloud. Since my (and also Rushmeier's) implementation of the algorithm in Rushmeier *et al.* [26] only handles diffuse scattering, I compared this result with a parallel version of recursive path tracing. The ray-to-bin scattering vector for the initial solar illumination, the bin-to-

Fig. 5 A diffusely scattering cloud. (far left) with direct illumination, and indirect illumination computed by the method of this paper, (medium left) with indirect illumination only, multiplied by a factor of 3, (medium right) with indirect illumination computed by the method

of Rushmeier *et al.* [26], multiplied by a factor of 3, (far right) middle left image minus middle right image, multiplied by a further factor of 40, and added to a medium gray

Fig. 6 A cloud with a forward scattering Henyey Greenstein phase function. (far left) with direct illumination, and indirect illumination computed by the method of this paper, (medium left) with indirect illumination only, multiplied by a factor of 3.443, (medium right) with

indirect illumination computed by recursive path tracing, multiplied by a factor of 3.443, (far right) middle left image minus middle right image, multiplied by a further factor of 10, and added to a medium gray

bin scattering matrices used in the intermediate bounces, and the bin-to-ray vector for the final scattering into the viewing ray all in effect implement a phase function which is constant over direction bins, and this spreads out the forward scattering peak. For a fairer comparison, I computed the average cosine of the scattering angle for the first scattering of the external solar illumination, using the integrated ray-to-bin scattering into the 24 direction bins. For $g = 0.4$, this average cosine was 0.366, so the recursive path tracing used $g = 0.366$.

The agreement is not as good as in the diffusely scattering case in Fig. 5, since the combined scaling factor for the far right image is only 34.4, but when viewed in the presence of the single-scattered external illumination, the difference is not very noticeable. The method in this paper took 77 minutes for the global illumination, and the recursive path tracing took 37 hours, with 31,300 samples per pixel, using Russian roulette, regular tracking (see [23]) through the octree cells, and inscattered attenuated external illumination at every cell (“next event estimation”) rather than just at the one where the recursive scattering takes place. Both timings used 8 CPU threads.

Figure 7 shows a cubical volume with a constant density of a diffusely scattering medium, and a blocking square prism extending from the front to the back, of much higher density and opacity. The top left image includes accurate single scattering, computed from the initial stored attenuated external illumination, plus the multiple scattering by the method here. The top right image shows just the multiple scattering, brightened factor of 5. The bottom left image is similarly brightened multiple scattering, computed by method of [26]. The bottom right image shows the top right image minus the bottom left one, brightened by another factor of 10, and added to a neutral gray, and indicates very good agreement. The streaks are the results of the imperfect shadowing due to the multiple propagation paths described in Max [20]. Both methods were run in parallel on 8 CPU threads, until the unshot flux was less than .0001 times its initial value. The method presented here took 75 seconds for the global illumination at resolution 64^3 , bilaterally upsampled to 256^3 , and Rushmeier’s took 20 hours at resolution 64^3 , repeating the $O(n^7)$ transparency computations for each of the 7 shooting passes, because it would have taken 137 gigabytes to store them.

Fig. 7 A cube of a participating medium with a light blocker

A reviewer of this paper asked if the method could include surface reflections, and it can, using the average plane equation representation for all surfaces inside each octree cell. The surface representation data size is then independent of the complexity of the polygonal surface model, depending only on the number of octree cells it intersects. In each propagation pass, the flux increment received in each volume cell is reflected off the surfaces back into the volume flux direction bins by the average surface plane, accounting for the volume-to-surface irradiation, and the surface-to-volume reflection. There is no need for surface-to-surface form factors, because the volume flux propagation carries the surface-to-surface flux. During the final rendering pass, both the attenuated direct illumination and the total received directional volume flux are used to shade the surfaces.

Figure 8 shows a cube of constant density diffusely scattering participating media with several polygons: a white horizontal rectangle near the top, and four red side faces of a rectangular prism below it. It has sRGB tone mapping to emphasize the dim global illumination effects and also uses $k_d = 1$ in the Lambert's law diffuse surface shading formula, instead of the maximum energy conserving value of $1/\pi$. The bottom side of the upper white polygon appears pink, from the white contributions of the volume scattering below it, and the red contributions of reflection from the red surfaces below it. The red glow in the participating media above the red prism is from surface to volume scattering, and the red shading inside the shadowed region of the top surface of the prism is due partly to double scattering paths, from the light source to the volume, then to that surface, and finally to the eye, but also to triple scattering, from the light source to that red surface, then to the bottom of the white polygon, then back to that red surface, and finally to the eye.

Figure 9 shows a cottonwood tree, scan converted into a sparse voxel octree of resolution 512^3 , with direct illumination rendered using the methods of Max *et al.* [21], and indirect illumination rendered as described here, using 85 weights optimized for 256^3 and extrapolated to 512^3 . The distribution of microflake normals was approximated by spherical harmonics up to and including order 4, resulting in 25 coefficients, which can represent the angular variation in the phase function with about the same precision as the 24 direction bins can store the directionality of the resulting scattering. The resulting phase function, computed by the methods in "Appendix" of [21], is fully anisotropic, depending on both the incoming and scattering directions, not just the angle between them.

When the global illumination was computed at 512^3 resolution, which took 250 minutes, most non-empty voxels contained a single (botanical) leaf. The first scattering of sunlight in such a voxel was mostly in the direction of the

Fig. 8 Cube of participating media with polygons. left: with direct illumination only, middle: with direct plus indirect illumination, right: difference showing the contribution of indirect illumination

Fig. 9 Cottonwood tree. left: with direct and indirect illumination, computed at 512^3 resolution; middle: indirect illumination only, computed at 512^3 resolution, multiplied by a factor of 5; right: indirect illumination only, computed at 64^3 resolution and interpolated, multiplied by a factor of 5

normal of the side of the leaf facing the sun. That scattered flux in the same voxel then illuminated the opposite side of the leaf, making it too bright, and the next scattering similarly made the illuminated side too bright. One cannot simply neglect the scattering within the shooting cell, because as shown in “Appendix B”, the form factor from a cell to itself is larger than that to any other cell, so significant flux would be lost. Figure 9 (right) shows the indirect illumination as computed in only 2 seconds on octree level 3, of resolution 64^3 , and then interpolated to lower levels without bilateral upsampling. Now the microflake assumptions are more valid because voxels contain multiple leaves, and the global illumination is smoother.

Several reviewers asked for a comparison with the method of [20]. I no longer have access to this code, nor memory of its details, but I tried to reconstruct it from the description in [20], parallelizing it for multithreading to compare fairly with the timings above. It divided each face of the direction cube into 16 equal squares, making 96 direction bins, while the method in this paper divides each cube face into 4 squares, making 24 bins. It used the propagation method in “Appendix A” with the constant weights in the main() program at the end of “Appendix A” instead of the layer dependent weights described above, which allow more accurate approximations to the true form factors. It only propagated flux to every other successive layer. The skipped layers were basically copied from the previous propagated layer, with just a different $1/l^2$ factor. Multiplying the received flux vector by the 96 by 96 bin-to-bin scattering matrix to get the unshot flux took 16 times as long as multiplying by a 24 by 24 matrix and became a more significant portion of the computation time. So this step, which was not parallelized in the results above, was also parallelized over different subtrees of the octree grid, the same way the bilateral upsampling and downsampling was.

Figure 10 shows comparisons of the light blocker scene of Fig. 7, with a Henyey Greenstein scattering phase function, rendered by path tracing, by the 24 direction bin method of this paper, and by the 96 direction bin method of [20], all running on a 128^3 voxel volume. For the path tracing, the Henyey Greenstein average cosine parameter g was set to 0.4. For the 24 bin method, the closest results were obtained with $g = 0.55$, while for the 96 bin method, whose finer directional resolution can better approximate a given phase function, g was 0.41. The recursive pathtracing image took 28 hours at 500 by 500 pixel resolution, with 7200 samples per pixel. The 24 bin method used 2.96 Gbytes of CPU memory and took 50 minutes, and the 96 bin method used 9.12 Gbytes of memory and took 182 minutes, both including octree creation, external illumination, 6 global illumination passes, and final image rendering. Figure 10 shows that the 96 bin method more closely approximates the path traced image, due to its better ability to approximate the scattering phase function. However, even though each of its three different direction bin shapes used a different constant multiple of the propagated flux when computing the received flux, to better approximate the form factors, these three parameters could not give the continuous variation in the effective form factors produced by the method of this paper, so artifacts of the direction bin boundaries are visible in the figure near the cast shadow, as spurious edges between regions of different brightness.

10 Discussion, limitations, and future work

The $O(n^3(\log n)^{3/2})$ CPU computation times are too slow for real time rendering. But off-line animation could profit from the precomputation of global illumination, when rendering a static scene from a moving viewpoint. This can eliminate

Fig. 10 Light blocker scene with Henyey Greenstein phase function: (top left) path traced, (top right) path traced with single scattering subtracted, leaving only multiple scattering, multiplied by 2.55, (middle left) multiple scattering via the 24 bin method of this paper, (middle right) multiple scattering via the 96 bin method of [20], (bottom left) middle left image minus top right image, multiplied by 5 and added to a neutral gray, (bottom right) middle right image minus top right image, multiplied by 5 and added to a neutral gray

the noise from Monte Carlo sampling, but gives imprecise shadowing in the participating medium.

As discussed in “Appendix A”, the variables *side* and *diagonal* have the purpose of creating an extra shell of propagation adjacent to the slanted faces of the bin direction pyramid. These variables were included in the optimization, and it produced a negative value for *side*, presumably using it to get better approximations to the larger form factors inside the bin direction pyramid, rather than to closely match the small positive form factors just outside it. As a result, some values of *received* can become negative and are replaced by zero. By adding an extra C^1 penalty for negative *received* values to the cost function, these negative values could be suppressed.

If the density of the participating medium is so high that multiple scattering within a single voxel is likely, it may take many iterations for this shooting method to converge. Max

Fig. 11 Cube of participating media with polygons, one of which is not axis aligned

[20] handled this case by using precomputed powers of the bin-to-bin scattering matrix. These powers could be precomputed for the Henyey Greenstein scattering phase function used for the cloud image here, but for the spherical harmonics used on the tree image here, the bin-to-bin scattering matrix at each voxel is a different linear combination of the matrices for the spherical harmonic basis functions, so these matrix powers would have to be computed on the fly.

The indirect illumination of polygons by the flux stored in discrete octree cells can lead to light and shadow leaks if the surfaces are not aligned with octree cell boundaries, as explained in Kaplanyan *et al.* [16]. The current method uses discrete direction bins, so there can also be surface shading artifacts if a surface orientation is not aligned with direction bin boundaries, as seen on the upper white polygon in Fig. 11.

The 24 direction bins do not give a fine enough subdivision of the unit sphere to capture narrowly and highly peaked phase functions. A potential solution is to use more direction bins, for example, the 96 bins in the comparison above with Max [20], but with layer-dependent weights as in this paper, fit separately for each of the tree direction bin shapes, to get a better and smoother approximation to the form factors. This requires significantly more memory, but the octree data compression features of [21], not used here, could save some memory. If the propagation algorithm were implemented on a GPU, it would be much faster. Atomic adds could be used, either in C++ on a CPU or in Cuda on a GPU, to scatter the received flux directly into the unshot flux, eliminating the need for the *Treceived* vector per octree node.

Acknowledgements This work was begun while the author was on sabbatical at Pixar Animation Studios, and the model for the cottonwood tree in Fig. 9 came from there. The reviewers made suggestions which improved the paper.

Declaration

Conflicts of interest The author declares that he has no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Bochkanov, S.: www.alglib.net/optimization/lbfgsandcg.php
- Bouthors, A., Neyret, F., Max, N., Bruneton, E., Crassin, C.: Interactive multiple anisotropic scattering in clouds. In: Proceedings of ACM symposium on Interactive 3D Graphics and Games, pp. 173–182. ACM (2008)
- Cerezo, E., Perez-Cazorla, F., Pueyo, X., Seron, F., Sillion, F.X.: A survey on participating media rendering techniques. *Visual Comp.* **21**(5), 303–328 (2005)
- Chandrasekhar, S.: *Radiation Transport*. The Clarendon Press, Oxford (1960)
- Cohen, M., Chen, S.E., Wallace, J., Greenberg, D.: A progressive refinement approach to fast radiosity image generation. In: *Computer Graphics (SIGGRAPH '88 proceedings)* **22**(4), 75–84 (1988)
- Crassin, C., Neret, F., Sainz, M., Green, S., Eisemann, E.: Interactive indirect illumination using voxel cone tracing. *Comp. Graph. Forum* **30**(7), 1921–1930 (2011)
- Elek, O., Ritschel, T., Dachsbacher, C., Seidel, H.P.: Principal ordinates propagation for real-time rendering of participating media. *Comput. Graph.* **45**, 28–39 (2014)
- Fatal, R.: Participating media illumination using light propagation maps. *ACM Transactions on Graphics* **28**(1, Article no. 7) (2009)
- Ge, L., Wang, B., Wang, L., Meng, X., Holzschuch, N.: Interactive simulation of scattering effects in participating media using a neural network model. *IEEE Trans. Visual. Comput. Graph.* (2019). <https://doi.org/10.1109/TVCG.2019.2963015>
- Goral, C., Torrance, K., Greenberg, D.: Modeling the interaction of light between diffuse surfaces. In: *Computer Graphics (SIGGRAPH '84 proceedings)* **18**(3), 312–222 (1984)
- Heitz, E., Neyret, F.: Representing appearance and pre-filtering subpixel data in sparse voxel octrees. In: C. Dachsbacher, J. Munkberg, J. Pantaleoni (eds.) *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association (2012). <https://doi.org/10.2312/EGGH/HPG12/125-134>
- Immel, D., Cohen, M., Greenberg, D.: A radiosity method for non-diffuse environments. In: *Computer Graphics (SIGGRAPH '86 proceedings)* **20**(4), 133–142 (1986)
- Jakob, W., Arbee, A., Moon, J.T., Bala, K., Marschner, S.: A radiative transfer framework for rendering materials with anisotropic structure. *ACM Transactions on Graphics (TOG)—Proceedings of ACM SIGGRAPH 2010* **29**(4) (2010). <https://doi.org/10.1145/1778765.1778790>
- Kajiya, J., Herzen, B.V.: Ray tracing volume densities. In: *Computer Graphics (SIGGRAPH '84 proceedings)* **18**(3), 165–174 (1984)
- Kallweit, S., Müller, T., McWilliams, B., Gross, M., Novák, J.: Deep scattering: rendering atmospheric clouds with radiancepredicting neural networks. *ACM Trans. Graph.* **36**(6), Article 231 (2017)
- Kaplanyan, A., Dachsbacher, C.: Cascaded light propagation volumes for real time indirect illumination. In: *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pp. 99–107 (2010)
- Koerner, D., Portsmith, J., Sadlo, F., Ertl, T., Eberhardt, B.: Flux-limited diffusion for multiple scattering in participating media. *Comp. Graph. Forum* **33**(6), 178–189 (2014)
- Kopf, J., Cohen, M., Lischinski, D., Uyttendaele, M.: Joint bilateral upsampling. In: *ACM Transactions on Graphics* **26**(3, Article No. 96) (2007)
- Krivanek, J., Georgiev, I., Hachisuka, T., Vevoda, P., Sik, M., Nowrouzezaharai, D., Jarosz, W.: Unifying points, beams, and paths in volumetric light transport simulation. *ACM Transactions on Computer Graphics* **33**(4), 103 (2014)
- Max, N.: Efficient light propagation for multiple anisotropic volume scattering. In: G. Sakas, P. Shirley (eds.) *Proceedings of the Fifth Eurographics Workshop on Rendering*, pp. 87–104 (1994)
- Max, N., Duff, T., Mildenhall, B., Yan, Y.: Approximations for the distribution of microflake normals. *Vis. Comp.* **34**(3), 443–457 (2018). <https://doi.org/10.1007/s00371-017-1352-2>
- Max, N., Schussman, G., Miyazaki, G., Iwasaki, K., Nishita, T.: Diffusion and multiple anisotropic scattering for global illumination in clouds. *J. WSCG* **22**(3), 277–284 (2004)
- Novak, J., Georgiev, I., Hanika, J., Jarosz, W.: Monte carlo methods for volumetric light transport simulation. *Comp. Graph. Forum State Art Rep.* **37**(2), 551–576 (2018)
- Patmore, C.: Simulated multiple scattering for cloud rendering. In: S. Mudur, S. Pattaniak (eds.) *Graphics, Design, and Visualization (Proceedings of ICCG93)*, pp. 29–40. Elsevier (1993)
- Perlin, K.: Improving noise. In: *Proceedings of ACM SIGGRAPH '02*, pp. 681–682 (2002)
- Rushmeier, H., Torrance, K.: The zonal method for calculating light intensities in the presence of participating media. In: *Computer Graphics (SIGGRAPH '86 proceedings)* **21**(4), pp. 293–302 (1987)
- Stam, J.: Multiple scattering as a diffusion process. In: Hanrahan, P., Purgathofer, W. (eds.) *Rendering Techniques '95*, pp. 41–50. Springer, Vienna (1995)
- Tan, Z.: Radiative heat transfer in multidimensional emitting, absorbing, and scattering media—mathematical formulation and numerical method. *J. Heat Trans.* **111**, 141–147 (1979)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Nelson Max received his Ph.D. in Mathematics from Harvard University in 1967 and is currently a Distinguished Professor Emeritus of Computer Science at the University of California, Davis. His research interests are in the areas of scientific visualization, computer animation, realistic computer graphics rendering, multi-view stereo reconstruction, and augmented reality. He has rendered realistic lighting effects in clouds, trees, and water waves and has produced numerous computer animations, shown at the annual ACM SIGGRAPH conferences, and in OMNIMAX stereo at the Fujitsu Pavilions at Expo'85 in Tsukuba Japan, and Expo'90 in Osaka Japan. He is a fellow of the Association for Computing Machinery (ACM), and the winner of the ACM SIGGRAPH 2007 Steven A. Coons award.