A Hierarchical Volumetric Shadow Algorithm for Single Scattering

Ilya Baran Jiawen Chen Jonathan Ragan-Kelley Frédo Durand Jaakko Lehtinen

Computer Science and Artificial Intelligence Laboratory*
Massachusetts Institute of Technology

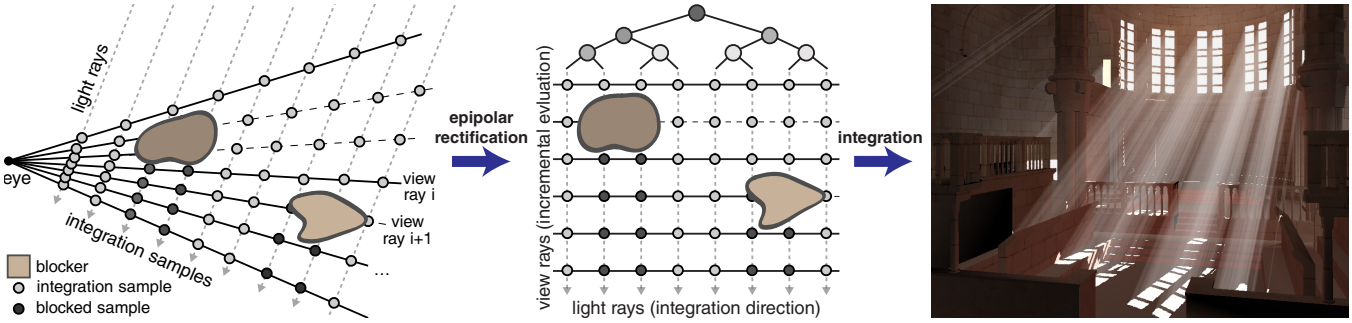


Figure 1: *Rendering volumetric shadows in participating media requires integrating scattering over view rays.* ***Left:*** *The visibility component of this integral has a special structure: once a light ray hits an occluder, that light ray does not contribute to the integral along any view ray past the occluder.* ***Middle:*** *Our method exploits this structure by computing the integrals in an epipolar coordinate system, in which light rays (dashed grey) and view rays (solid black) are orthogonal and the integration can be performed asymptotically efficiently using a partial sum tree.* ***Right:*** *This enables us to compute high-quality scattering integrals much faster than the previous state of the art.*

Abstract

Volumetric effects such as beams of light through participating media are an important component in the appearance of the natural world. Many such effects can be faithfully modeled by a single scattering medium. In the presence of shadows, rendering these effects can be prohibitively expensive: current algorithms are based on ray marching, i.e., integrating the illumination scattered towards the camera along each view ray, modulated by visibility to the light source at each sample. Visibility must be determined for each sample using shadow rays or shadow-map lookups. We observe that in a suitably chosen coordinate system, the visibility function has a regular structure that we can exploit for significant acceleration compared to brute force sampling. We propose an efficient algorithm based on partial sum trees for computing the scattering integrals in a single-scattering homogeneous medium. On a CPU, we achieve speedups of 17–120x over ray marching.

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

Keywords: volumetric scattering, global illumination

1 Introduction

Volumetric phenomena, such as "god rays" that appear when sunlight scatters through clouds, are important lighting effects that contribute greatly to the realism of a scene. They are caused by occlusion within the participating medium. Well-known techniques exist for rendering such images: ignoring multiple scattering, marching along each view ray and computing in-scattering modulated by visibility to the light source yields good results when using enough samples, but this method prohibitively slow due to the large number of samples necessary. In effect, ray marching requires sampling visibility in the entire 3D view volume. A well-known property of visibility is that along light rays, it is a 2D step function: light is unblocked until it hits a surface, and all points along the ray after that surface are shadowed. (This is the basis of shadow mapping.) Our main idea is to make use of this property and compute the scattering *incrementally* over view rays. To facilitate this, we use a coordinate transformation that maps the diverging view rays and light rays into a rectilinear grid (Fig.1, middle). This allows us to perform the integration for each view ray using a tree data structure that is maintained incrementally. Together, these steps reduce algorithmic complexity drastically by removing the need to sample visibility in the volume. We demonstrate 17–120x performance improvements on the CPU over brute force sampling.

The main contribution of this paper is a hierarchical algorithm for approximating the scattering integrals, which has significantly lower algorithmic complexity $O(s(r+d)\log d)$ (where rs is the image resolution and d is the effective number of integration samples) than ray marching's $O(rsd)$. We also present an epipolar sampling scheme that guarantees sufficient sampling for a rectangular image at a lower cost than uniform sampling.

In this work we do not consider complex light transport effects such as caustics [Sun et al. 2010], transparent blockers, or multiple scattering, which necessitates a more complex approach due to the diffusion within the volume [Jensen and Christensen 1998; Jarosz et al. 2008].

2 Overview and Related Work

In the presence of a single-scattering participating medium of uniform scattering cross-section σ, computing the in-scattered radi-

*e-mail: {ibaran,jiawen,jrk,fredo,jaakko}@csail.mit.edu

ance reaching the eye along a view ray is an integration problem. The radiance L scattered by the medium towards the eye over the entire ray is

$$L = \int_0^d e^{-\sigma s} L_i(s)\,\mathrm{d}s \tag{1}$$

where s is the path length over the ray, $L_i(s)$ is the radiance scattered from the light source towards the ray origin at each point along the ray, d is the distance at which the eye ray terminates into an opaque surface, and $e^{-\sigma s}$ is the extinction due to out-scattering between the eye and point s on the ray. The in-scattered illumination at point s on the ray due to a directional light is

$$L_i(s) = \sigma L_{\text{in}}(s, \mathbf{l})\rho(\mathbf{l}, \mathbf{v}),$$

where $L_{\text{in}}(s, \mathbf{l})$ is the radiance incident to point s from light direction $\mathbf{l}$, $\mathbf{v}$ is the direction towards the eye, and $\rho(\mathbf{l}, \mathbf{v})$ is the scattering phase function, which we assume to be uniform ($\rho \equiv 1/4\pi$). See Pharr and Humphreys [2004] for details. In the presence of occluders, the integral is difficult to compute analytically since arbitrary portions of the path may be in shadow and do not contribute a lighting component; more precisely, $L_{\text{in}}(s, \mathbf{l}) = V(s, \mathbf{l})\tilde{L}_{\text{in}}(s, \mathbf{l})$, where $V(s, \mathbf{l})$ is the visibility towards the light source and $\tilde{L}_{\text{in}}(s, \mathbf{l})$ is the incident radiance assuming no occlusion. The final integral we are evaluating is

$$L = \int_0^d e^{-\sigma s}\, V(s, \mathbf{l})\, \sigma\, \tilde{L}_{\text{in}}(s, \mathbf{l})\, \rho(\mathbf{l}, \mathbf{v})\, \mathrm{d}s. \tag{2}$$

2.1 Related Work

The traditional method for computing physically based volumetric shadows in a single-scattering medium is ray marching. Pharr and Humphreys offer a useful introduction [2004]. For each pixel in the output image, a ray is cast from the eye through the pixel. Equation (1) is then approximated by point sampling along the ray, casting a shadow ray from from each sample towards the light. If it is occluded, the sample is discounted from the integral. The ray terminates once it hits an opaque object or exits the scene bounding volume. For a scene with p rays (pixels) and where the visibility is evaluated at n samples along each ray, the complexity of the algorithm is $O(pn)$. In this work we concentrate on the problem of determining visibility within the medium. Scattering models and methods for evaluating them in the absence of occlusion (e.g. [Sun et al. 2005; Pegoraro and Parker 2009]) are orthogonal to our approach.

Ray marching can perform well enough in scenes of limited depth ranges, particularly when aided by limiting the computations to regions where volumetric shadows actually appear [Wyman and Ramsey 2008]. However, this is hard to do in scenes where interesting volumetric effects take place at multiple locations over a large distance. In addition to ray marching with jittered sampling, several earlier interactive techniques approximate the scattering integral using volume rendering techniques. This means sampling the integrand along parallel planes that slice the volume, effectively computing a Riemann sum [Dobashi et al. 2000]. Several methods use shadow volumes for determining visibility for view ray segments. For example, Billeter et al. [2010] render volumetric shadows by explicitly constructing visibility boundaries between lit and shadowed volumes. They avoid a strong dependence on scene complexity by extracting the boundaries from shadow maps, but like with all shadow volume algorithms, the complexity of the visibility function still determines the running time for each pixel. In all, no matter how the volume is sampled, obtaining high quality results in large scenes with complex visibilities with high image resolutions remains challenging due to the need to sample visibility over the entire 3D view volume. Our technique addresses this fundamental cost by performing an initial 2D sampling followed by incremental, hierarchical evaluation.

In recent work, Sun et al. [2010] describe a technique for rendering single scattering effects by gathering in the space of light and view rays, employing 6D hierarchies for culling. They do not target volumetric shadows, but rather more expensive effects for off-line rendering, such as volumetric caustics.

In addition to performing actual visibility sampling along rays, image space blurring techniques may also be employed for approximating light shafts [Mitchell 2008]. Such techniques can produce compelling results, but offer no guarantees on correctness. In particular, they require the light source to be visible on the screen, and do not handle occlusion properly.

It has been observed previously that the volumetric shadow problem can be simplified using epipolar geometry [Max 1986; Engelhardt and Dachsbacher 2010]. Max [1986] adapts a span-based scan converter and the shadow volume algorithm to epipolar coordinates, and incrementally updates the hidden portions of view rays along epipolar planes. He then computes the scattering integrals on the visible segments of the ray analytically, implying an integration cost proportional to the complexity of the visibility function. Furthermore, the shadow volume algorithm has a strong dependence on scene complexity. Recently, Engelhardt and Dachsbacher [2010] observed in a similar vein that the values of the scattering integral mostly vary continuously along epipolar image lines except at depth discontinuities. They take image space samples at discontinuities detected from a Z buffer using brute force ray marching, and use shadow maps for visibility. This may work well for scenes with limited amounts of depth discontinuities, but the dependence on their number can lead to poor performance with complex visibility, such as foliage. Furthermore, the dependence on depth discontinuities can lead to scattering features being missed (see Sec. 10). We also build on epipolar coordinates, but we avoid ray marching by using incremental integration with partial sum trees that enable us to compute the scattering integrals at all pixels at a significantly lower asymptotic cost that is independent of the scene complexity.

2.2 Algorithm Overview

We evaluate the scattering integral along epipolar lines in the output image. In practice, we first render a depth map from the point of view of the camera, and a shadow map from the light. We then transform both of these maps into a rectified epipolar coordinate system (Sec. 3). This coordinate system makes light and view rays mutually orthogonal. Our key contribution is the observation that the parameterization enables incremental evaluation of visibility between view rays (Fig. 4, Sec. 4). We incrementally maintain a hierarchical representation, a partial sum tree, of the integrand when iterating over the view rays. The tree enables us to integrate each view ray in logarithmic time, regardless of the complexity of the visibility function along it (Sec. 5-6). This enables us to avoid exhaustive sampling of visibility in the 3D view volume, leading to significant speedups. We also make use of depth distributions to cut down on the number of integration segments along each view ray (Sec. 7). Once the scattering integrals have been computed in the epipolar coordinate system, we resample the results back into the original image domain.

For simplicity, we will describe our algorithm for directional lights and discuss the changes necessary for spot lights in Section 9.

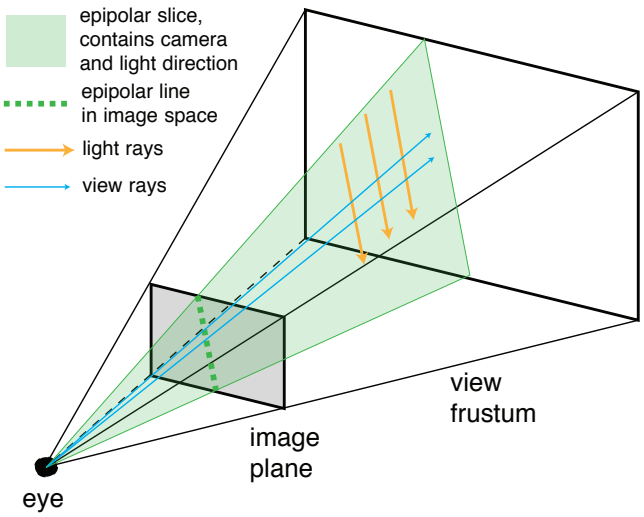


Figure 2: *Illustration of an epipolar slice, i.e., a planar section (green) that contains the eye point, view rays (blue) and the light direction (orange).*

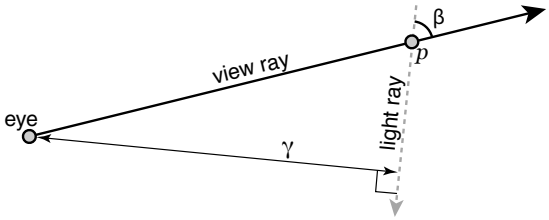


Figure 3: *Rectified coordinates of point p within a slice: the angle β specifies the view ray and the distance γ specifies the light ray.*

3 Epipolar Geometry

Consider the line that passes through the eye point and is parallel to the light direction. This line defines a family of halfplanes around it that we call epipolar slices. Figure 2 illustrates one such slice. The intersections of these halfplanes with the image plane are epipolar rays, along which streaks of lighting ("god rays") appear to emanate. The scattering results of all view rays within a slice only depend on the light rays within the same slice, which implies that we can process the slices independently.

Our incremental integration algorithm relies on the ability to index the view rays and light rays independently. To achieve this, we "rectify" the coordinate system to one in which view rays are parallel to one axis, and light rays to another, essentially doing a change of variables. Each point in space p (except those on the light ray through the eye point) is the unique intersection of a view ray and light ray; we give it the coordinates (α, β, γ), as illustrated in Figure 3. The first coordinate, α, specifies the slice p is in. The β coordinate specifies the view ray by measuring the angle between the view ray through p and the light direction. Using an angle instead of distance measured along the screen makes all slices have the same geometry in world space, a key symmetry that we will exploit later (Sec. 6). It results in a fairly uniform sampling if the camera's field of view is not unreasonably wide. The γ coordinate specifies the light ray by measuring the distance from the light ray containing p to the eye point.

The first stage of our algorithm is to compute a depth map from the point of view of the camera and a shadow map from the light, and

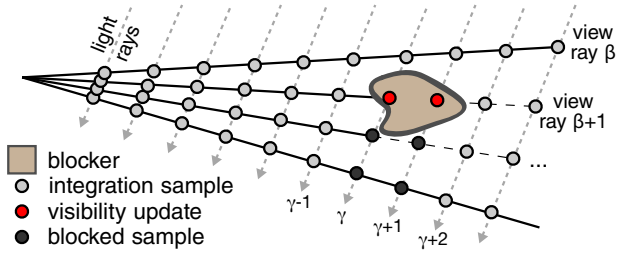


Figure 4: *A visibility update, stepping from view ray β to $\beta + 1$. Two light rays (γ and $\gamma + 1$) terminate at view ray $\beta + 1$. Their integrand entries $b[\gamma]$ and $b[\gamma + 1]$ are set to zero (and the partial sum tree updated) before computing the integral for view ray $\beta + 1$. The entries $b[\gamma]$ and $b[\gamma + 1]$ remain zero for all subsequent view rays.*

rectify these into the epipolar coordinate system. Note that this is not a mere resampling, as also the *values* of the functions change: for example, for each blocker depth in the shadow map, we compute the angle β that corresponds to the view ray direction of the blocker, and vice versa for camera depth maps.

The rectification (and the unrectification of the integrated result at the end) running times are proportional to the size of the relevant target image. For s slices, each with r camera rays on average, and d light rays, rectification of the camera depth map and the unrectification take $O(sr)$ time and the rectification of the shadow map takes $O(sd)$ time.

4 Incremental Integration

Once rectification has been performed, evaluation of the scattering integrals within a slice proceeds by looping over the view rays (β) and approximating the integrals along them as Riemann sums over the light rays (γ): for each β, we sum the product of visibility and the relevant scattering terms over all γ until the view ray terminates. The lengths of the view rays, $D[\beta]$, are read from the rectified depth map rendered from the point of view of the camera. Visibility at each step is determined by comparing the current β to a blocker distance $S[\gamma]$ that is read from the rectified shadow map, exactly as in traditional shadow mapping.

The above procedure can be written as a conditional summation as follows:

$$I[\beta] = \sum_{\gamma < D[\beta]} e^{-\sigma s(\gamma,\beta)}\, \sigma\, \tilde{L}_{\text{in}}\, \rho\, V(\gamma,\beta) \left|\frac{ds}{d\gamma}\right| \Delta\gamma = \sum_{\gamma < D[\beta]} I(\gamma,\beta) V(\gamma,\beta) = \sum_{\substack{\gamma < D[\beta] \\ S[\gamma] > \beta}} I(\gamma,\beta). \quad (3)$$

Here $\Delta\gamma$ is the step size in the γ direction, $s(\gamma,\beta)$ is the world-space path length along the view ray β, $ds/d\gamma$ is its derivative along the view ray, and $I(\gamma,\beta)$ contains all other terms in the integrand except visibility. This equation means that for each view ray (β), we look at all light rays from 0 to $D[\beta]$ and add the contributions of light rays whose stored blocker distance is greater than the β coordinate; these are exactly the segments of the view ray that are not occluded from the light.

This summation has a special structure: once we have considered a particular shadow-map value $S[\gamma]$, all corresponding segments for subsequent eye rays will be shadowed and will not contribute to the integrals of the view rays below. In Section 6, we will show

how to approximate $I(\gamma, \beta)$ as the sum of a few terms of the form $\Gamma(\gamma)\mathrm{B}(\beta)$. For now, assume that this can be done and consider the integration of one of these terms. As we sweep over the camera rays β from the light away, we incrementally maintain a function $b[\gamma]$ that for each γ stores $\Gamma(\gamma)V(\gamma, \beta)$. In the beginning, $\beta = 0$, and $b[\gamma] = \Gamma(\gamma)$ for all γ. Now, when we step from one view ray to the next, we set $b[\gamma] = 0$ for all light rays that terminate at the current view ray, i.e., for which $S[\gamma] = \beta$ (see Fig. 4). Whenever an entry of $b[\gamma]$ is set to zero, it will remain so for all subsequent view rays, indicating that the corresponding segments will be shadowed and thus their visibility need not be checked any more. The integral at β is then $\mathrm{B}(\beta) \sum_{\gamma < D[\beta]} b[\gamma]$.

Despite the fact that visibility is maintained incrementally, the above algorithm still has the same complexity as ray marching because each eye ray has to iterate over all γ until $D[\beta]$ to perform the sum. To reduce the complexity, we store $b[\gamma]$ in a partial sum tree as described below.

5 Hierarchical Integration

A partial sum tree is a complete binary tree where each node is the sum of its two children. The leaves of the tree store the n data elements and interior nodes store a hierarchy of partial sums (Fig. 5a). Partial sum trees feature an efficient query operation, which, in $O(\log n)$ time, lets us query for the sum of an arbitrary interval within the array of n elements. The tree also supports an update operation, which lets us modify any leaf element and the nodes above it in $O(\log n)$ time. The tree can be compactly stored as a contiguous array of size $2n$. It can be built in linear time by starting with the input array at the leaves and recursively summing up to the root.

To update an element in the array, one simply updates the leaf node and recursively updates each parent up to the root (Fig. 5b). To answer a prefix-sum query $\sum_{k=0}^{q} b[k]$, we observe that each node holds a partial sum of the form $\sum_{k=i}^{j} b[k]$. To retrieve the sum from the tree, we simply check if q is in the interval represented by the root node. If so, then the sum is the value of the left child plus the recursive prefix sum of the right child (Fig. 5c). Recursion terminates when q is outside the interval range, in which case we simply return the value of the node. Both UPDATE and QUERY make a single pass down the tree; thus, their runtimes are both $O(\log n)$.

Half-tree. The fact that we only ever integrate (sum) over the elements starting from index 0 (corresponding to the eye) leads to the interesting observation that the recursive query never uses the right child of any node, be it leaf or internal: if it did, the correct answer would already have been computed higher up in the tree. The reader may want to convince herself of this fact by looking at Figure 5c, where the right children are marked by dashed lines. This allows us to store only half of the elements in the tree and on average cuts in half the number of elements that need to be modified for an update. The observation that an efficient data structure is possible with only half of the tree nodes is due to Fenwick [1994].

In order to avoid aliasing artifacts from the discretization, $D[\beta]$ is stored as a float rather than an integer index, and we linearly interpolate the results of querying the tree at $\lfloor D[\beta] \rfloor$ and $\lceil D[\beta] \rceil$. The second query is done by storing the complete lowest level of the tree (instead of half, like the other levels) and looking up the next element, rather than doing a full tree query. In all, we can answer cumulative sum queries in $O(\log n)$ time with only $1.5n$ total elements of data.

Putting together the incremental evaluation and the partial sum tree, as shown in Figure 6, we are now able to render volumetric shadows

```
// D contains lengths of view rays along γ
// S contains lengths of light rays along β
proc INTEGRATESLICE( int[] D, int[] S )
  // Mark where light rays terminate
  termination_depths ← empty list
  for each light ray γ in S
    APPEND( termination_depths[S[γ]], γ )
  end for
  // Initialize the incremental counter, tree, and result
  tree ← BUILD-TREE( Γ_i )
  results ← ZEROS
  // Walk over view rays
  for each view ray β
    // Set the counter to zero for light rays that terminate at this β
    for all light rays γ in termination_depths[β]
      UPDATE-TREE( tree, γ ) // sets the γ node to zero
    end for
    // Query two nearest results
    res_1 ← QUERY-TREE( tree, ⌈D[β]⌉ )
    res_2 ← QUERY-TREE( tree, ⌊D[β]⌋ )
    // Lerp results and multiply by SVD factors B_i
    results[β]_i ← B_i[β] LERP( res_1, res_2, D[β] − ⌊D[β]⌋ )
  end for
```

Figure 6: *Pseudocode for incremental hierarchical integration.*

in $O(s(r + d) \log d)$ time, where s is the number of slices, r is the number of view samples on a slice (the resolution of the β axis), and d is the number of light rays (the resolution of the γ axis). The $sr \log d$ term is the number of queries, and $sd \log d$ is the number of updates made into the tree. These terms dominate the $O(s(r + d))$ complexity of the rectification. In comparison, the brute force ray marcher requires $O(srd)$ time.

6 Variation Along and Across Light Rays

The function $I(\gamma, \beta)$ encodes all of the integration terms other than visibility. However, to make our incremental algorithm work, we need to approximate it as a sum of outer products of functions that are constant along γ and β, respectively. For this, we sample $I(\gamma, \beta)$ at discrete values, and compute the singular value decomposition [Golub and Van Loan 1996] of the resulting matrix.

$$I(\gamma, \beta) \approx \sum_{i}^{N} \Gamma_i(\gamma)\, \mathrm{B}_i(\beta). \tag{4}$$

Our algorithm integrates the products of each $\Gamma_i(\gamma)$ and the visibility function, and then sums the results multiplied by $\mathrm{B}_i(\beta)$. In practice, this merely means that the tree stores N-dimensional vectors, rather than scalars. The greater N is, the better our approximation of I, but the slower the running time.

View rays and light rays are parallel at the epipole (the apparent location of the light source on the screen). As a result, the rectification mapping is singular, and derivative $ds/d\gamma$ goes to infinity. As a practical measure to avoid the SVD being thrown off by the large values, we render results within a small ring (about two degrees) around the epipole using a ray marcher, and run our algorithm on the remainder of the image. This makes the SVD more well-behaved at a performance cost (see Sec. 10 for analysis).

We compute the SVD using a 64×64 subsampled matrix, and linearly upsample the singular vectors $\Gamma_i(\gamma)$ and $\mathrm{B}_i(\beta)$ to the resolution of the actual integration grid. The singular values from the

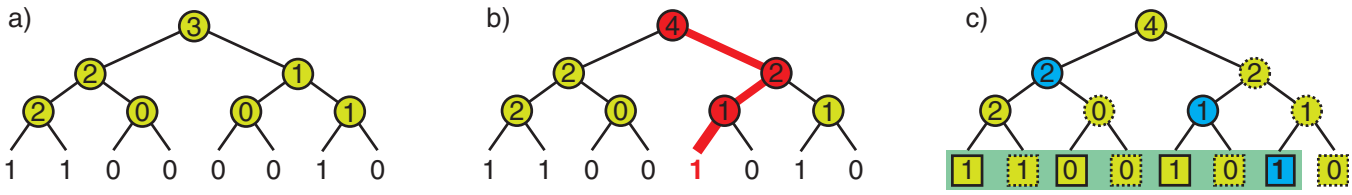


Figure 5: *Partial sum trees.* ***a)*** *Each tree node stores a sum of its child nodes.* ***b)*** *Update. When a leaf is updated, the nodes on the path to the root need to be updated.* ***c)*** *Query. Querying the sum of an interval can be performed in $O(\log n)$ time. The sum over the green interval is computed as the sum of the nodes marked in blue. Right child nodes (dashed) are not stored, because they are never needed.*

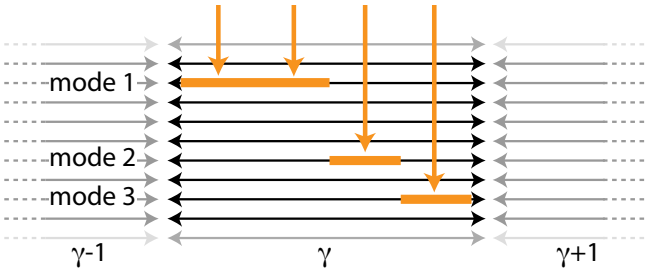


Figure 7: *Depth prefiltering. For each rectified γ coordinate, we sample blocker depth at four sub-intervals, cluster nearby depths in* modes*, and insert the modes into the* termination_depths *structure as fractional updates (Sec. 7).*

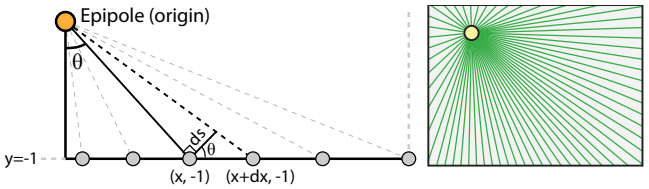


Figure 9: *Slice distribution. Note that we assume that ds and dx are infinitesimal. The epipolar slices are distributed non-uniformly in a way that guarantees a proper sampling over the image plane.*

SVD are multiplied into B_i. $N = 4$ singular values are sufficient for high-quality results in a uniform medium. The magnitude of the fourth singular value never exceeds $1/40$th of the magnitude of the first singular value, even under extreme view/light configurations.

Because the β axis is parametrized by angle, not pixel units, the slices are symmetric around the light direction (this is the "key symmetry" mentioned in Section 3). Therefore, for a uniform scattering phase function, I does not depend on α, and the same SVD may be used for all slices (although using different bounds, i.e., accessing a different subrectangle). Thus we only need to compute the SVD once per frame, rather than once per slice, which would be prohibitively expensive.

We have so far assumed that $\tilde{L}_{\text{in}}$, the incident radiance without occlusion, is constant, but for a textured light source (such as produced by stained glass), $\tilde{L}_{\text{in}}$ is a function of α and γ. We cannot bake $\tilde{L}_{\text{in}}$ into I because it depends on α, but because it does not depend on β, we can handle this case simply by initializing the tree with the element-wise product $\tilde{L}_{\text{in}}(\alpha, \gamma)\Gamma_i(\gamma)$ instead of just $\Gamma_i(\gamma)$.

7 Visibility Prefiltering

Our integration algorithm is based on computing Riemann sums. As is well known, this leads to correlated artifacts unless a very large number of samples is used. In contrast, Monte Carlo sampling converts these correlated errors to less objectionable noise. In practice, we observe that, in their basic form, our Riemann sums require approximately 10x the number of samples for equal visual quality when compared to Monte Carlo. To cut down on the number of light rays required for integration, we prefilter the visibility by not storing just a single blocker depth $S[\gamma]$ in each entry along the γ axis, but a distribution of depths as described below. Given such a distribution, we can approximate the percentage of visibility that changes along β for each integration sample instead of using a binary value. Note that the general idea is similar to shadow mapping techniques that store depth distributions to enable linear filtering of shadow maps [Donnelly and Lauritzen 2006].

More precisely, we sample the blocker depth four times along each $\Delta\gamma$ in the rectification loop. We approximate the depth distribution using one or more *modes* by grouping the depth values together that differ by at most a small constant (Fig. 7). For example, if the entire segment hits a single surface, the depth values will all be close to each other and there will be only one mode (this is the most common case). Each mode has a weight—the fraction of samples that comprise that mode. The modes are inserted into the *termination_depths* structure; when there is more than one mode, the tree node at the location of the segment gets updated once for every mode (at different β coordinates), with a fraction of the light corresponding to the mode's weight. This does not change the basic algorithm but results in clear quality improvements, particularly in animated sequences (cf. accompanying video); the differences are hard to spot in still images. Note that only light rays that terminate on depth discontinuities as seen from the light result in multiple modes.

8 Slice Distribution

The final result needs to be sampled sufficiently densely in the image plane after unrectification. We bound the maximum sampling error by requiring that *no screen pixel should be further than half a pixel away from an epipolar line*. The strategy of placing samples around the image perimeter at one-pixel intervals leads to guaranteed sufficient sampling. Engelhardt and Dachsbacher [2010] take this approach, although they relax the one-pixel criterion and compensate by joint bilateral upsampling [Kopf et al. 2007]. We view this relaxation as an orthogonal component to the scattering algorithm and do not discuss it further.

We found that uniform spacing around the image perimeter leads to unnecessary oversampling. A tighter distribution is obtained as follows. Without loss of generality, let the epipole (the apparent position of the light on the screen) be at the origin, and let an edge of the screen be the horizontal line $y = -1$. Recall that α is the coordinate that selects the epipolar slice and we now need to

Figure 8: *Scenes used in our tests. Left:* SIBENIK *(80k triangles). Middle:* TREES *(382k triangles). Right:* LANDSCAPE *(2M triangles).*

Figure 10: *A spotlight rendered using our algorithm.*

specify which slice corresponds to which α value so that uniformly sampling α yields a good slice distribution. We are looking for a function $x(\alpha)$ that gives the locations of the intersections of the epipolar slices with the edge $y = -1$, such that the sample density at the edge is uniform (Fig. 9). The angle θ formed by the y axis and the line segment from the origin to the point $(x, -1)$ is $\theta(\alpha) = \arctan x(\alpha)$. The orthogonal distance of the point $(x, -1)$ to the ray that intersects the x axis at the point $(x + \mathrm{d}x, -1)$ is $\cos(\arctan x(\alpha))\mathrm{d}x = \mathrm{d}x/\sqrt{1 + x^2}$ due to a trigonometric identity. We wish for this distance to always be one pixel when we increment α by a small amount $\mathrm{d}\alpha$, i.e., $P = \mathrm{d}\alpha$ where P is the (resolution-dependent) pixel size in normalized screen coordinates. We get the differential equation

$$\frac{\mathrm{d}x}{\sqrt{1 + x(\alpha)^2}} = \mathrm{d}\alpha, \quad x(0) = 0. \tag{5}$$

This equation integrates to $x(\alpha) = \sinh(\alpha)$. The desired slice distribution is obtained by stepping along α in equal increments of $\mathrm{d}\alpha = P$ until the entire edge is sampled. Other screen edges are treated analogously. The case where the epipole is not at the origin is handled by scaling geometry down uniformly, applying this formula, and scaling back.

Our slice distribution results in guaranteed sampling over the entire screen while using fewer slices than regular sampling of the perimeter. For example, when the epipole is on the screen near a corner, we get by with half the number of slices. We get the least savings when the epipole is in the center of the screen, resulting in 15% fewer slices (in 1280×960 resolution) than uniform sampling. The savings result from the fact that the spacing of the slices increases towards the corners. Note that this happens while always maintaining a half-pixel bound on sampling error.

9 Spot Lights

We can extend the preceding discussion on rectifying perspective cameras and distant lights to perspective cameras and local spot lights (Fig. 10). The rectification mapping becomes slightly more complicated, but the main algorithm is unchanged.

The slices and rectified camera rays remain the same, but instead of indexing the light rays within a slice by the distance to the camera, we set the γ coordinate to be the angle the light ray makes with the direction from the light to the camera. This results in a nonuniform sampling density along the camera ray, and therefore the $ds/d\gamma$ term in $I(\gamma, \beta)$ varies also with γ, not just β like in the case of directional lights.

10 Results and Evaluation

Our implementation is a direct translation of the algorithm in Fig. 6. The brute force comparison algorithm is a simple ray marcher that uses shadow maps for visibility queries. We use a ray tracer to compute the depth maps required by both algorithms, but do not include the time taken by their construction in the results and only report time required for computing the scattering integrals. As a cache coherence optimization, the shadow map used for visibility tests by the brute force ray marcher is laid out in 8×8 blocks such that the texels in each block are scanned in Z-order curve.

Methodology. We study the performance of our algorithm in comparison to ray marching and epipolar sampling [Engelhardt and Dachsbacher 2010] in two settings, equal time and equal quality, focusing on a CPU implementation. The running times and speedup factors are computed only for the portion of the code related to the volumetric scattering integration: ray marching for brute force and epipolar sampling, and rectification, incremental hierarchical integration, unrectification, and the brute force around the epipole for our algorithm, as other parts of the rendering loop are the same. Although our algorithm supports variable lighting (Sec. 6), we use a single-color light source in all tests to focus on the quality of the scattering. The code is single-threaded. All tests are run on a PC with an Intel Core i7 960 CPU at 3.2 GHz with 12GB of RAM. All images are rendered at 1280×960 using 4 singular value coefficients. The number of epipolar slices chosen by the adaptive distribution algorithm varies between 2116-2264. The γ resolution of the integration grid (d, the number of light rays in each slice) is 4096 for two of the scenes and 2048 for the third. All scenes and both algorithms use a 4096×4096 shadow map for visibility queries. The ray marcher is tuned to use the smallest number of samples to give visually equal quality.

In addition to the CPU implementation, we have also implemented a parallel version of our algorithm for the GPU using CUDA. Our implementation is divided into three kernels: rectification, sorting, and integration. Rectification is a straightforward data-parallel op-

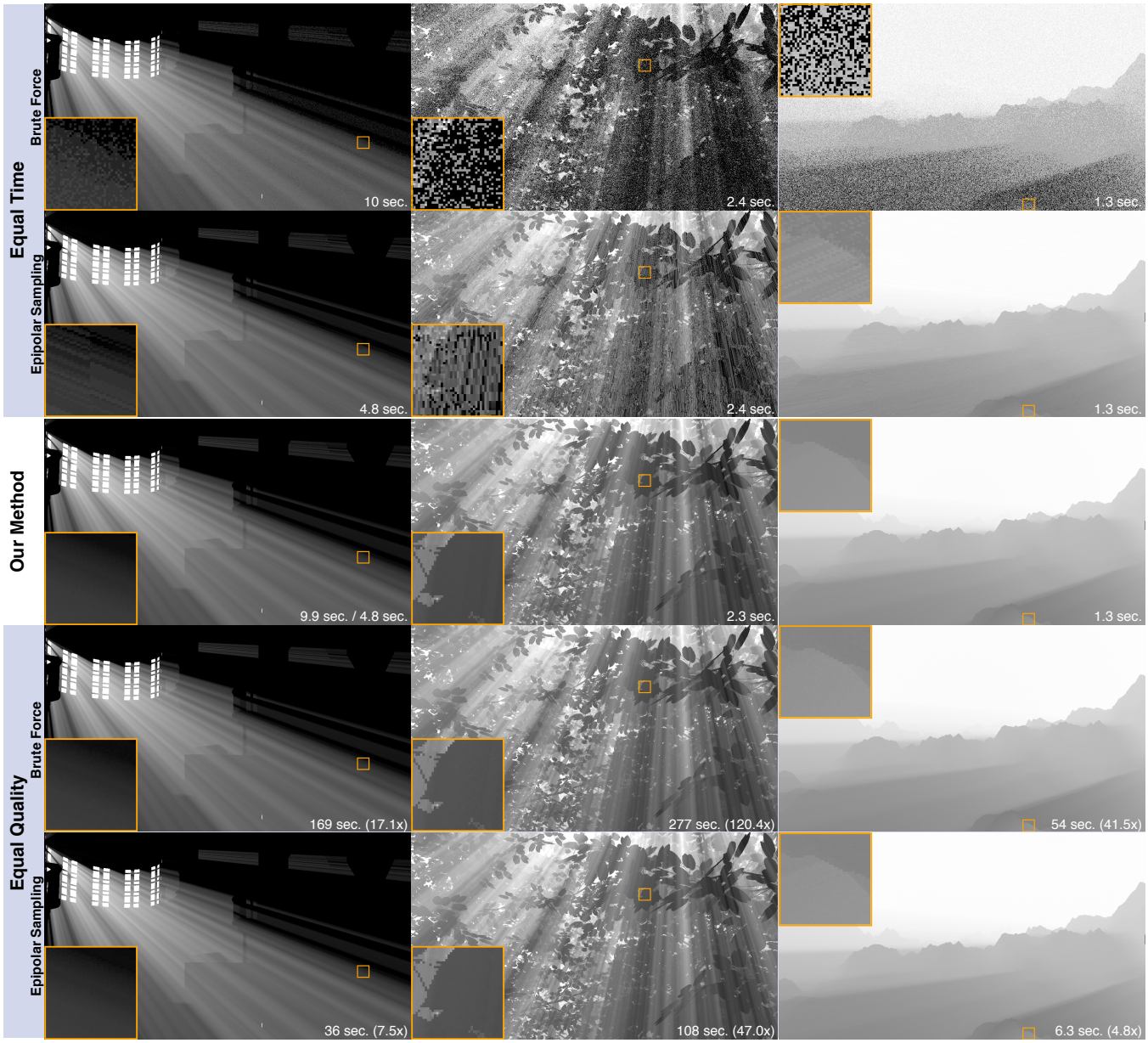


Figure 11: *Quality/speed comparison. Our method (middle row) is compared to ray marching and Engelhardt & Dachsbacher's epipolar sampling at* 1280×960 *resolution, given equal time (top rows) and at equal quality (bottom rows). For fairness in comparison to epipolar sampling, we used their method (instead of ray marching) for the region around the epipole on the* SIBENIK *scene where the epipole is on the screen, making our method take 4.8 secs (vs. 9.9 secs with brute force around the epipole).*

eration on the depth and shadow maps, using one thread for each output pixel. We combine visibility prefiltering and sorting into a single kernel that coalesces groups of 4 rays. We allocate one block of 32 threads per slice, and use a modified GPU radix sort that coalesces rays with similar depths as it walks over the input shadow map.

For integration, because slices are independent and the partial sum trees are too large to fit into shared memory on current hardware, we assign one thread per slice and store trees in global memory. Observe that the inner loop of the integration algorithm only performs two operations: UPDATE-TREE and QUERY-TREE. Therefore, to reduce instruction divergence between threads, we use the *while-if-if* strategy of Aila and Laine [2009], which guarantees that at least half the threads in a warp are doing useful work.

Scenes. We test our algorithm on three scenes (Fig. 8). SIBENIK is an indoor scene with directional sunlight shining through the windows into hazy air. TREES is a smaller scene with complex visibility and many depth discontinuities. LANDSCAPE demonstrates our ability to render large scenes.

10.1 CPU Results

Figure 11 shows renderings and timings measured from the CPU implementation. It achieves high speeds in comparison to ray marching (17.1x–120.4x for similar quality) and our implementa-

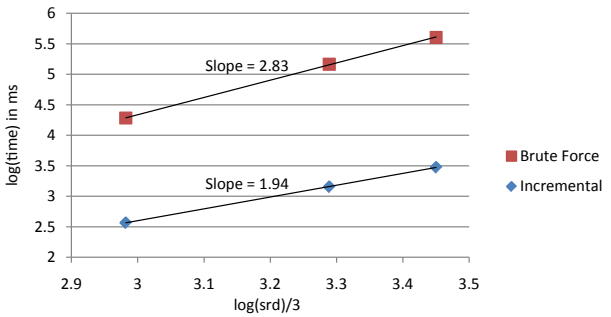


Figure 12: *This chart show the performance of brute force and our method as resolution increases. It demonstrates cubic scaling of brute force and essentially quadratic scaling of our method.*

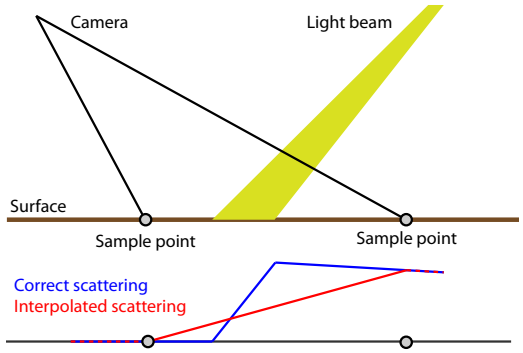
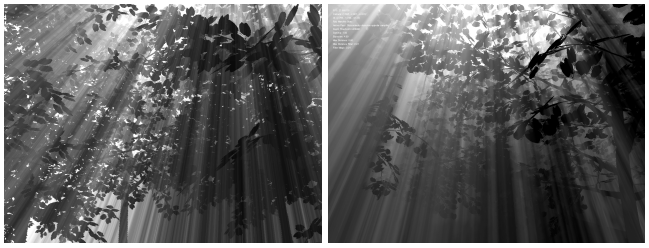


Figure 13: *A case where interpolating along depth discontinuities on epipolar lines leads to erroneous results. Here a narrow beam of scattered light illuminates a planar surface. Because there are no depth discontinuities from the camera's point of view, the sampling does not capture the features in the result, leading to blurring.*



Our GPU prototype, 33 fps | Engelhardt and Dachsbacher, 24 fps

Figure 14: *Results rendered using our GPU prototype compared to epipolar sampling at* 1280×960 *resolution, using 1024 epipolar slices for both algorithms. The difference in viewpoint and scattering intensity are due to manual matching of parameters between our codes (see text).*

tion of epipolar sampling (4.8x–47.0x for similar quality), clearly demonstrating the advantages of our algorithm's lower complexity and small working set. More precisely, the partial sum tree is so compact that it fits entirely into caches and makes the frequent accesses cheap. As expected from the complexity, our algorithmic advantages are particularly apparent in difficult shadowing conditions that require many ray marching samples.

Rectification is about two times as costly as the hierarchical integrator. In addition to rectification and integration, we also ray march in a region around the epipole when it is on-screen. Although the region is small ($\approx 2.5\%$ of total screen area), the ray marching takes about three quarters of the total rendering time because the rest of the image is processed so quickly. Fortunately, this cost only has to be paid when the epipole is on or close to the screen. For example, when the camera is rotated slightly in the SIBENIK scene so that the epipole moves away, the total speedup over ray marching jumps from 17.1x to approximately 73x. For comparison with epipolar sampling, we used epipolar sampling around the epipole instead of ray marching, which allowed us to render the SIBENIK scene in 4.8s instead of 9.9s.

The running time of epipolar sampling is bounded from below by the number of depth discontinuities on the screen. In a complex image like the TREES scene, the number of discontinuity pixels may be as high as 20%. This limits the potential speedup that can be obtained by epipolar sampling over brute force. Furthermore, while depth discontinuities along epipolar lines provide a useful heuristic for adaptive sampling, the interpolated results are still an approximation that can fail in some circumstances. Figure 13 illustrates a planar surface lit by a tight light beam. Because there are no depth discontinuities, the samples will not line up with the derivative discontinuities in the scattered illumination, leading to erroneous interpolation. Due to the binary sample-or-interpolate decisions that epipolar sampling makes, these artifacts are time-varying and result in undesirable "popping" during animation.

To verify our asymptotic complexity, we ran our algorithm on the Trees scene, scaling the resolution of both the screen (the number of slices s and the rays per slice r) and the light (the effective number of depth samples d). The log-log plot in Figure 12 illustrates our results. Brute force has a slope of 2.83 relative to $\log(srd)/3$, indicating roughly cubic scaling. Our algorithm has slope 1.94, indicating that we scale approximately quadratically, an order of magnitude better than ray marching.

10.2 GPU Results

Engelhardt and Dachsbacher kindly provided an executable of their epipolar sampling implementation on the GPU, and to compare to it, we have implemented a GPU version of our algorithm. We compared the methods on the TREES and SIBENIK scenes at 1280×960 resolution (Fig. 14). The test was run on an NVIDIA GeForce GTX 480 GPU. Because the implementations are different and we do not have access to their source code, we approximate the timing for the scattering-only portion of the code by modifying the shaders to disable direct illumination and colored lighting (we verified that this results in expected performance increases for their code), and report final frames per second (fps) numbers only. We matched camera positions and scattering model parameters manually, and adjusted their algorithm's depth discontinuity threshold to not fire on extraneous edges. Both algorithms compute results along epipolar planes; we use the maximum number of slices (1024) supported by their code to focus on the integration performance for both their and our algorithm. (Our adaptive distribution algorithm would only require $700 - 800$ slices to obtain the same sampling density in these scenes, resulting in another 15% speed improvement.) On the SIBENIK scene, both implementations ran at approximately 37 fps. On the TREES scene, epipolar sampling ran at 24 fps, while our implementation ran at 33 fps, a 1.4x speedup. While we achieve approximately equal quality on a static image, our quality is better in animation because we compute results for every sample without relying on interpolation. In particular, our video demonstrates the temporal coherence artifacts that epipolar sampling exhibits. Comparing to our GPU ray marcher implementation at equal quality (and using enough slices to get pixel-sized sampling), on SIBENIK the ray marcher runs at 9 fps, while we run at 32 fps (3.6x). On TREES the ray marcher runs at 5.6 fps, while we run at 23 fps (4.1x).

The speed of our CPU implementation relies on the fact that the memory accesses made by the integrator are extremely localized. While our GPU implementation performs on par with the state of the art, we see room for improvement. The data caches even in current high-end GPUs are insufficient to retain all the trees on-chip, incurring a large bandwith penalty for the CUDA implementation that keeps the trees in off-chip memory. While the depth prefiltering technique drastically reduces bandwidth, the requirements are still substantial. As future work, we aim to formulate a version of the algorithm where the trees are kept in on-chip local memory, and prefetcher threads load the streaming data onto the chip.

10.3 Limitations and Discussion

To enable the performance improvements, we make some assumptions about the scattering medium. The symmetry across slices that enables us to only use a single SVD per frame precludes media whose density varies spatially or that have anisotropic phase functions. For smoothly varying media, however, we could sample the SVD on a small number of slices and interpolate.

We also tested our algorithm with a textured light source with three color channels, simulating a stained glass window effect on a scene with the Sibenik cathedral with the epipole off the screen. Compared to a uniform light source, our algorithm was 1.5x slower. We did not test a textured light source on the GPU, but we expect that our bandwidth-limited algorithm would take the full 3x performance hit because it would need to integrate three color channels.

Like all algorithms based on shadow mapping, we are limited by the resolution of the shadow map. Cascading shadow maps (e.g., [Lloyd et al. 2006]) are used in practice to increase the effective resolution. Investigating how our algorithm can interact with such optimizations is an interesting direction for future work. In its basic form, our algorithm does not support translucent blockers that cast semi-transparent shadows in the medium. However, we believe it is possible to include multiplicatively transparent blockers as special types of tree updates, similar in spirit to how our depth prefiltering technique already treats blocker edges.

11 Conclusions

We have described an efficient algorithm for rendering shadowed single scattering effects in a uniform participating medium. Our main contributions are to perform scattering computations in a rectified coordinate system that allows incremental evaluation of visibility, removing the need to densely sample visibility in the view volume. In addition, combining incremental evaluation with a tree structure yields a lower asymptotic complexity than ray marching techniques. The advantages are particularly pronounced in difficult shadowing situations that require large numbers of samples: our algorithm performs significantly faster than brute force ray marching at equal quality, and delivers a large increase in quality given equal time. Furthermore, our preliminary GPU implementation delivers improved quality at similar performance compared to the state-of-the-art real-time technique.

Acknowledgements

We thank Timo Aila and Samuli Laine for invaluable assistance with our GPU implementation. Thanks to Thomas Engelhardt and Carsten Dachsbacher for giving us access to their prototype on very short notice. Thanks to the anonymous reviewers for their helpful feedback on improving the writing. We thank David Luebke and NVIDIA for a hardware donation. Jonathan and Jiawen were supported by Intel PhD fellowships. This work was partially supported by grants from Intel Corporation and the Singapore-MIT GAMBIT Game Lab.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, 145–149.

BILLETER, M., SINTORN, E., AND ASSARSSON, U. 2010. Real time volumetric shadows using polygonal light volumes. In *Proc. High Performance Graphics 2010*.

DOBASHI, Y., YAMAMOTO, T., AND NISHITA, T. 2000. Interactive rendering method for displaying shafts of light. In *Proc. Pacific Graphics*, IEEE Computer Society, Washington, DC, USA, 31.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *Proc. 2006 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 161–165.

ENGELHARDT, T., AND DACHSBACHER, C. 2010. Epipolar sampling for shadows and crepuscular rays in participating media with single scattering. In *Proc. 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, 119–125.

FENWICK, P. M. 1994. A new data structure for cumulative frequency tables. *Software, practice & experience 24*, 3, 327–336.

GOLUB, G. H., AND VAN LOAN, C. F. 1996. *Matrix Computations, 3rd. ed.* The Johns Hopkins University Press.

JAROSZ, W., ZWICKER, M., AND JENSEN, H. W. 2008. The beam radiance estimate for volumetric photon mapping. *Computer Graphics Forum 27*, 2 (Apr.), 557–566.

JENSEN, H. W., AND CHRISTENSEN, P. H. 1998. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proc. SIGGRAPH 98*, 311–320.

KOPF, J., COHEN, M. F., LISCHINSKI, D., AND UYTTENDAELE, M. 2007. Joint bilateral upsampling. *ACM Trans. Graph. 26*, 3.

LLOYD, D. B., TUFT, D., YOON, S.-E., AND MANOCHA, D. 2006. Warping and partitioning for low error shadow maps. In *Proc. Eurographics Workshop on Rendering 2006*, 215–226.

MAX, N. L. 1986. Atmospheric illumination and shadows. In *Computer Graphics (Proc. SIGGRAPH '86)*, ACM, New York, NY, USA, 117–124.

MITCHELL, K. 2008. Volumetric light scattering as a post-process. In *GPU Gems 3*, Addison-Wesley.

PEGORARO, V., AND PARKER, S. 2009. An analytical solution to single scattering in homogeneous participating media. *Computer Graphics Forum 28*, 2.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann.

SUN, B., RAMAMOORTHI, R., NARASIMHAN, S., AND NAYAR, S. 2005. A practical analytic single scattering model for real time rendering. *ACM Trans. Graph. 24*, 3.

SUN, X., ZHOU, K., LIN, S., AND GUO, B. 2010. Line space gathering for single scattering in large scenes. *ACM Trans. Graph. 29*, 4.

WYMAN, C., AND RAMSEY, S. 2008. Interactive volumetric shadows in participating media with single-scattering. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 87–92.