Using Hierarchical Simplicial Meshes to Render Atmospheric Effects

F. Betul Atalay David M. Mount

Department of Computer Science
University of Maryland
College Park, Maryland 20742
{betul,mount}@cs.umd.edu

Abstract

A fundamental element of computer graphics is producing realistic visualizations of various natural phenomena. An important problem in this area is that of rendering scenes containing atmospheric effects, which arise from the absorbtion and scattering of light due to small particles such as dust and smoke. Generating photorealistic renderings through ray-tracing is computationally very demanding because of the time needed to perform numerical integration along the length of each ray to determine the opacity and color of the gaseous media.

We present a new data structure and algorithm for efficiently rendering atmospheric effects. Our approach is based on adaptively sampling a sparse set of the rays. Each ray is modeled as a point in a 4-dimensional space, and the results of the numerical integrations for these rays are computed accurately and stored in a 4-dimensional spatial index. The results for arbitrary rays are then computed through relatively inexpensive linear interpolations between neighboring sampled rays. In order for the interpolations to be continuous, it is important that the spatial index be based on a cell complex. This precludes the use of well-known structures such as kd-trees and octrees. Our spatial index is a recently discovered pointerless data structure, called a *simplex decomposition tree*, which is based on a hierarchical simplicial decomposition of space.

In addition to presenting the data structure, we discuss a number of issues in the efficient application of this tree for rendering. We present empirical evidence that our approach can produce renderings of high quality significantly faster than simple ray-tracing. Given the centrality of sampling and reconstruction in computer graphics, this data structure is likely to be useful other applications in this field.

Key words: Atmospheric effects, rendering, ray-tracing, geometric data structures, hierarchical meshes, interpolation.

1 Introduction

Realistic image synthesis is a fundamental problem in computer graphics. An important and challenging subproblem is that of accurately rendering atmospheric phenomena such as smoke and dust, which arise as a result of the absorbtion and scattering of light while passing through a participating medium. An accurate simulation of the interaction of light with a participating medium is quite complex, since it involves the use of radiative transport theory [11, 5]. However, for the purpose of rendering atmospheric effects, simpler models have been proposed, and have been shown to be quite satisfactory. A good survey of different optical models can be found in [16]. There has been considerable success in recent years in producing realistic physical models for smoke and related natural phenomena[8, 10, 15, 17, 19, 22]. Our interest here is not on how to model such phenomena, but rather on how to render them accurately and efficiently.

A number of hardware-based approaches for rendering smoke and other atmospheric phenomena have been proposed in the literature. In the context of rendering, Stam [22] and Fedkiw, et al. [7] propose the use of 3-dimensional texture maps to store the density of the atmospheric medium in each voxel of the texture map, and then render this texture map from back to front. Dobahsi, et al. [6] propose a similar approach based on computing a collection of preprocessed sample planes. For approaches involving other types of atmospheric phenomena, see also [4, 12, 23].

However, these methods suffer from a limited ability to model multiple scattering and other effects needed to render media with high albedo. Also, the use of grid-based representations, while amenable to hardware implementation, cannot readily adapt to variations in the density and color of the media. An alternative approach for generating realistic images, which can handle media with high albedo, is based on photon maps [10, 7]. This process is more computationally intensive, but achieves a high degree of realism by solving the full volume render-

ing equation for the medium. A significant component in the actual rendering time is the numerical integration performed by marching along the length of each ray in order to determine the overall opacity and color of the media.

We propose a new data structure and algorithm for accelerating this process. The basic idea is very simple, namely to replace wherever possible the computationally intensive numerical integration along each ray with a combination of sampling and interpolation. Each ray is modeled as a point in a 4-dimensional space. Rays are sampled adaptively, and the result of the numerical integration for each of these rays is computed accurately and stored in a 4-dimensional spatial index.

More formally, we can think of the medium as a function f, which maps rays to a pair consisting of the color and opacity. These are the net color and opacity obtained by marching the ray through the medium, until its collision with a solid object. Since we model rays as points in 4-dimensional space, the function f is a function over $\Re^4$:

$$f : \text{ray} \rightarrow (\text{color}, \text{opacity}).$$

In regions where f varies smoothly, we expect that ray coherence can be exploited, that is, nearby rays will pass through regions of similar color and density, and so the accumulated color and opacity will be close to each other. In order to generate the final rendering, rather than integrating along each ray through the volume, we avoid this expensive integration by collecting and storing relatively sparse set of sample rays, along with their associated colors and opacities, in a fast data structure. We can then use inexpensive interpolation methods to approximate the values of these sampled quantities for other input rays. Using an adaptive sampling strategy, regions with higher variations in color and density are sampled more densely, while avoiding oversampling in smooth areas. We dynamically maintain a *cache* of the most recently generated samples, in order to reduce the space requirements of the data structure.

In order to support rapid access, a reasonable approach would be to store the sampled rays and their associated results in a data structure based on a hierarchical subdivision of 4-space, such as an octree or kd-tree [3, 20]. One significant problem with such an approach is that the resulting subdivisions are not guaranteed to be a cell complexes, which means that *cracks* may exist between neighboring cells and these may result in discontinuities in the interpolation. (See Fig. 1(a) and Fig. 2 to see the effect of cracks on the final image rendered by a related ray-tracing project involving the use of 4-dimensional interpolations to compute reflections.)

Grid-based subdivisions [9, 13], on the other hand, are not adaptive to variations. Thus, in order to achieve good quality, they have to sample very densely, resulting in very large data structures.

In contrast, our approach is based on a hierarchical subdivision of 4-space into a cell complex whose faces are simplices, that is, a *hierarchical simplicial complex*. (See Fig. 1(b) for a two-dimensional example.) We store the hierarchical mesh in a data structure called a *simplex decomposition tree*, or *sd-tree*. Our hierarchical decomposition is based on the longest-edge bisection method given by Maubach [14]. Nodes of this data structure can be accessed through a pointerless method by associating each node with a location code [2]. Because a simplicial complex is used, we can guarantee a C^0 continuous approximation of f. In addition, simplicial decompositions are much simpler than the octree-based subdivisions, in the sense that the interpolations are performed with a minimal number of samples (5 samples for the 4-dimensional case), and hence, this is much cheaper than the quadrilinear interpolation using 16 samples.

In Section 2, we present an overview of the simplex decomposition tree data structure and describe how it is used for answering interpolation queries. The simplex decomposition tree involves a subdivision of 4-dimensional space, and it is well known that the complexities of subdivisions tend to increase exponentially as a function of the dimension. Consequently, it is important to save space wherever possible. In Section 3, we discuss a number of issues involved in the use of the data structure for the purposes rendering, and how to minimize the size of the resulting data structure.

Section 4 presents an empirical analysis of the data structure. The data structure does not rely on any particular model or representation of the medium or a particular method of modeling light transport along the ray. It merely assumes that it is possible to determine the color and density of the medium at any point, and that we have access to a function for integrating this information along each ray to determine its contribution in terms of opacity and color. For our experiments, we applied a simple light model, which accounts for extinction of light due to absorption by particles (opacity) and for the addition of light by reflection of external illumination. We adapted a simple smoke volume shader given in [18] that is used by ray-tracers like RenderMan or BMRT. The general idea is to march along the viewing ray, choosing an appropriate step size, sampling illumination and accounting for atmospheric extinction based on the smoke density at every portion of the ray. The smoke density at any point is determined by a noise function.

Although we have motivated this data structure from the perspective of a volume shader, there are a number of applications having to do with lines in 3-space that can

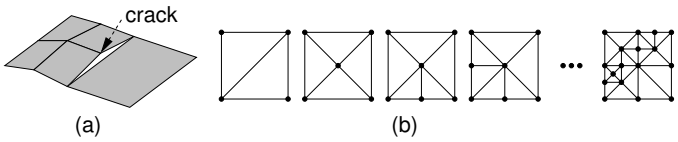


Figure 1: A crack (a) and a hierarchical simplicial mesh in the plane (b).

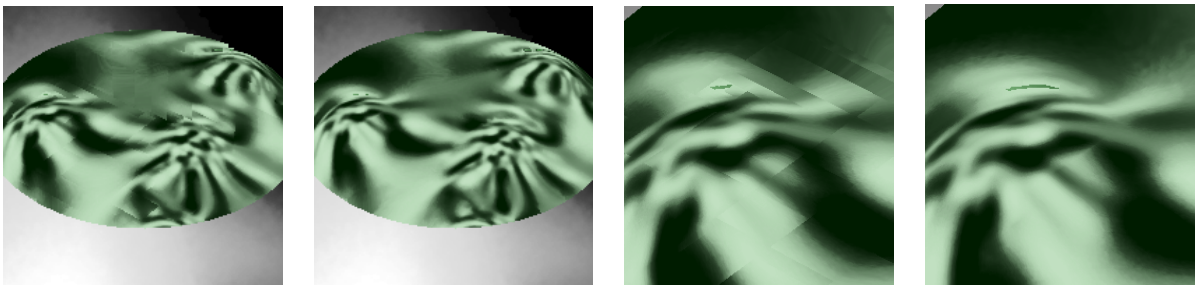


Figure 2: Results of a ray-tracing application to produce an 800×800 image based on 4-dimensional interpolations using (a) a kd-tree based on 14,492 samples and (b) a simplex decomposition tree based on 13,456 samples (CPU-times are equal for both). Details of these images are shown in (c) and (d), respectively. Note the blocky artifacts in the kd-tree approach (c).

benefit from this general approach.

2 The Simplex Decomposition Tree

In this section, we describe our data structure, the *sd-tree*. One *sd-tree* is built per volume. The volume is defined by an axis-aligned bounding box, and the data structure stores the attributes associated with some set of sample rays that intersect the volume.

2.1 Parameterizing Rays as Points

We model each ray by the directed line that contains the ray. Directed lines can be represented as a point lying on a 4-dimensional manifold in 5-dimensional projective space using Plücker coordinates [21], but we will adopt a simpler popular representation, called the *two-plane parameterization* [9, 13, 3]. A directed line is first classified into one of 6 different classes (corresponding to 6 *plane pairs*) according to the line's *dominant direction*, defined to be the axis corresponding to the largest coordinate of the line's directional vector and its sign. These classes are denoted $+X, -X, +Y, -Y, +Z, -Z$. The directed line is then represented by its two intercepts (s,t) and (u,v) with *front plane* and *back plane*, respectively, that are orthogonal to the dominant direction and coinciding with the volume's bounding box.

2.2 Construction of the *sd-tree*

The *sd-tree* is a collection of binary trees based on a regular simplicial decomposition of the 4-dimensional space of directed lines. Each plane-pair is associated with a 4-dimensional hypercube in line space containing all rays that pass through it. The 16 corner points of the hypercube represent the 16 rays from each of the four corners of the front plane to the each of the four corners of the back plane.

To simplify the presentation, assume that each hypercube has been scaled and translated to a reference hypercube of side length 2, centered at the origin, that is $[-1,1]^4$. Each hypercube is initially subdivided into $4! = 24$ congruent simplices that share the diagonal joining the vertices $(-1)^4$ and $(1)^4$. It is well known that the collection of these simplices fully subdivide the hypercube, and further that this subdivision is a simplicial complex[1]. These $4!$ simplices form the starting point of our simplicial decomposition. Simplices are then refined by a process of repeated subdivision, called *bisection*, in which a simplex is bisected by splitting its longest edge [14]. Hence, each coarse simplex at the highest level is the root of a separate binary tree, which are conceptually joined under a common super-root corresponding to the hypercube. The *sd-tree* then consists of 6 such trees, one for each plane-pair. A 4-dimensional simplex has 5 vertices, which is the minimum number of points required for linear interpolation in 4-dimensional space. The 5 vertices of a simplicial leaf cell in *sd-tree* constitute the ray samples which form the basis of our interpolation.

With every 4 consecutive bisections, the resulting simplices are similar copies of their 4-fold grandparent, subject to a uniform scaling by $1/2$. Thus, the pattern of

decomposition repeats every 4 levels. Moreover, all simplices at the same level are congruent to each other, and thus all similarity classes can be represented by 4 canonical simplices. All other simplices can be described by a permutation, a reflection, scaling and translation of these reference simplices. Using this fact, Atalay and Mount [2] have shown that, it is possible to define a location code that uniquely identifies a simplex of the hierarchy. This location code, called the $LPT\ code$, directly encodes the geometric relationship between a simplex and its associated reference simplex. All operations on the hierarchy, such as traversal, point location and neighbor finding are defined by means of the $LPT\ code$. Especially, locating any face neighbor of a simplex efficiently, given only its $LPT\ code$ is a valuable operation for compatibility refinement as will be explained shortly.

2.3 Adaptive and Dynamic Subdivision

The *sd-tree* grows and shrinks dynamically based on demand. Initially, only the 16 corners of each hypercube is sampled, and the initial 24 coarse simplices are constructed. A leaf simplex is subdivided by bisection along its longest edge, by sampling the midpoint of that edge. To avoid duplicate sampling, a hash table containing all the vertices of the subdivison is maintained.

Rays need to be sampled more densely in some regions than others, that is, in regions where the function f has greater variation. For this reason, the subdivision is carried out adaptively. The leaf cell is subdivided unless one of the following *termination conditions* is satisfied.

Termination Condition: We would like to compute an error of approximation associated with a leaf simplex. A good error measure for a leaf simplex S can be defined as the distance, $d(f(b), \tilde{f}(b))$, where b is the barycenter of S, $f(b)$ is the correct value of the function at b computed by ray marching, and $\tilde{f}(b)$ is the approximate value of the function at b computed by barycentric interpolation of the vertices of S. For the smoke volume application, the distance d is a weighted distance of color and opacity. However, this error measure is quite costly for simplicial decompositions, since the number of nodes grow faster than the number of vertices, and so, to sample a number of vertices, we would be forced to compute many more samples at node barycenters to compute the error during the construction. For this reason, we have chosen instead to use another heuristic to define the error associated with a leaf simplex S.

$$e(S) = \max\{d(f(v_i), f(v_j)) \mid 0 \leq i < j \leq 4\},$$

where v_0,...,v_4 are the vertices of S.

Pixel Resolution versus Depth Constraint: The tree could be allowed to grow until pixel resolution (i.e. projected leaf simplex width is less than the pixel width), or, in order to avoid excessive growth at strong discontinuity regions, the user may specify a *depth constraint*, such that the tree is not allowed to grow beyond that depth. If the subdivision is stopped due to the depth constraint, though, that leaf is not used for interpolation.

Consequently, if $e(S)$ exceeds a user-defined *distance threshold* and the depth of the cell in the tree is less than a user-defined *depth constraint*(or pixel resolution is not reached), the cell is subdivided. Otherwise, the leaf is said to be *final*.

Cache Behavior: If we were to expand all nodes in the tree until they are final, the resulting data structure could be very large, depending on the distance threshold and the depth constraint. For this reason, we only expand a node to a final leaf, if this leaf node is needed for some interpolation. If the size of the data structure exceeds a user-defined *cache size*, then the the tree is pruned to a constant fraction of this size by removing all but the most recently used nodes. In this way, the *sd-tree* behaves much like an LRU-cache.

2.4 Compatible Refinement and the Simplicial Complex

A subdivision in d-dimensional space is said to be *compatible*, if each simplex S in the subdivision shares a $(d-1)$-face with exactly one neighbor simplex. A simplex decomposition tree is a simplicial complex if all of its simplices are compatible. Compatibility is important, since otherwise, cracks occur along faces of the subdivision, which in turn present problems when performing interpolation. In order to keep the subdivision compatible at all times, whenever a simplex is bisected, a series of bisections will be triggered in other simplices.

```
compatBisect(S, e)
    mark S as pending
    for (S' ∈ N_e(S))
        if ( S' does not exist )
            compatBisect(parent(S'))
        // now S' exists
        if ( S' is a leaf and not marked as pending )
            compatBisect(S')
    simpleBisect(S)
```

Consider a simplex S which is about to be bisected, and let e denote the next edge of S to be split. The simplices of the subdivision that share this edge, denoted $E_e(S)$, must be bisected as well. Atalay and Mount [2] have defined neighbor rules to efficiently locate the neighboring simplices that share a common $(d-1)$-face with S, that is, the *facet neighbors* of S. Let $N_e(S)$ denote the facet neighbors of S that contain the edge e, or equivalently, the facet neighbors lying opposite all the $d-1$ vertices of S other than the endpoints of e. In

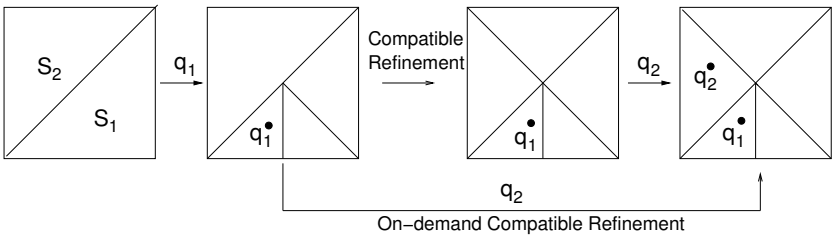


Figure 3: Compatible refinement and on-demand compatible refinement in 2D.

order to access all the simplices of $E_e(S)$ we compute facet neighbors recursively. The algorithm was given by Maubach [14], and is shown as the recursive function *compatBisect* in the code block. The procedure *simpleBisect* performs the basic bisection step of a simplex.

Maubach proved that in a compatible subdivision, the facet neighbors of S needed in this refinement either appear at the same depth as S or one level closer to the root [14]. For this reason, if the *compatBisect* procedure does not find a simplex S' in the tree, then it knows that its parent exists, and bisecting the parent will bring S' into existence. Note that the bisection of the parent may trigger recursive bisections on levels $\ell - 1$ and $\ell - 2$, and so on.

3 Rendering by Interpolation

Let us consider the interpolation of a given input ray **r**. In order to locate the leaf simplex containing **r**, after determining the appropriate hypercube depending on the dominant direction of **r**, we start our seach within the root simplex (one of the 24 simplices) containing **r**. The barycentric coordinates of **r** with respect to this root simplex is easily computed because of the special regular structure of the root simplices. After this initialization, we recursively descend the hierarchy until finding a leaf simplex, incrementally updating the barycentric coordinates with respect to each simplex along the way. Since the nodes of the tree are constructed only as needed, it is possible that **r** will reside in a leaf that is not marked as *final*. This means that, this particular leaf has not completed its recursive subdivision. In this case, the leaf is subdivided recursively, along the path **r** would follow, until the termination condition is satisfied, and the final leaf containing **r** is now marked as *final*. (Other leaves generated by this process are not so marked.) The color and opacity for **r** can now be interpolated by barycentric interpolation of the values associated with the 5 vertices of this *final* leaf simplex.

3.1 One-pass versus Two-pass Rendering

Notice that, even though the final tree constructed is compatible, this method does not totally avoid cracks in interpolation if the rendering and construction are done in the same single pass. Consider the two dimensional analogy in Fig. 3 following the link marked as *Compatible Refinement*. If q_1 arrives before q_2, there is no problem, since splitting of S_1 will force S_2 to split, and when q_2 arrives, the simplices will be compatible. However, assume that q_2 arrives before q_1 and that S_2 satisfies the termination condition, and marked as final. When q_2 arrives, it is answered by interpolation of the vertices of S_2. Then, when q_1 arrives, assume that the subdivision in the figure occurs splitting S_1 two more levels. And so, q_1 is answered by interpolating the vertices of a granchild of S_1. However, since q_2 is already answered at this point, there would be a crack in the interpolation, even though S_2 is forced to split by S_1's split.

One way of avoiding this is to render in two passes. In the first pass, the tree is constructed given all the query points, but without doing the interpolations. In the second pass, the queries are answered by performing the interpolations. Obviously, the two-pass rendering will be slightly more expensive, since the point location procedure will be done twice.

3.2 On-demand Compatible Refinement

Note that, there is a conflict between on-demand construction and compatible refinement for our purposes. To preserve compatibility, some simplices in the hierarchy will be refined, even though they will not be used for any interpolation query eventually. Thus, a lot of work done for construction of those simplices will be useless, unnecessarily reducing the efficiency of the overall algorithm, and increasing the size of the data structure. To prevent this, while keeping the compatibility property, we perform *on-demand compatible refinement*, which works as follows. The bisection of a simplex S does not trigger the bisection of a neighboring simplex, before that neighbor is actually required by some interpolation. Consider again Fig. 3, but following the *On-demand Compatible Refinement* link this time. S_1 and S_2 are neighbors of each other at the same level. Let the query point q_1 cause refinement of S_1 as shown. At the time q_1 caused this refinement, S_2 is not bisected to provide compatibility.

Unless another query needs S_2, the tree will remain non-compatible in fact, but still compatible for our purposes. However, if later, a query q_2 is located in S_2, before any termination condition is checked, we first check whether any neighbor of S_2 is refined by bisecting an edge shared by S_2 (even if S_2 is already marked as a final leaf, this check is performed, and might cause splitting of the final leaf). In Fig. 3, such a neighbor exists, that is S_1. Hence, S_2 will be bisected as well, and q_2 will continue its descend in the tree until no more splits are required, before being interpolated.

This method is much more efficient, since it generates a much smaller tree. But, for similar reasons with the original compatible refinement, it cannot avoid cracks totally. In this case, two-pass rendering corrects a substantial percentage of the cracks, but may not eliminate all the cracks. (Unlike the two-pass rendering explained in Section 3.1, the second pass as well, will induce subdivisions in the tree to correct the cracks.) Experimentally, we have seen that, among the final leaf nodes, less than 3% have cracks, and that the on-demand version performs comparably well with respect to the quality of the image generated. In fact, after a number of passes, the tree will converge to a crack-free tree.

4 Experimental Results

We have implemented the data structure and evaluated it using a simple model to render smoke. We have adapted the smoke volume shader code given in "PhotoRealistic RenderMan Application Note 20-Writing Fancy Volume Shaders"[18] to our own ray-tracer. The general idea is to ray march along the viewing ray choosing an appropriate step size, sampling illumination and accounting for atmospheric extiction based on smoke density at every portion of the ray. The smoke density at any point is determined by a noise function. This type of volume shaders that are used by ray-tracers like RenderMan or BMRT are very expensive, since reasonably small step sizes has to be chosen to avoid banding artifacts.

In general, this type of volume shaders must bind to surfaces, that is, there should be an objcet in the background, so that, the ray marching continues until the background object is hit. We model the smoke density as a finite volume, defined by an axis-aligned bounding box. The viewing ray enters the volume and the integration continues until the ray exits the volume (or hits an object that is within the volume). For simplicity, we have assumed that the smoke volume is designed to extend up to the background objects, and does not include any objects inside.

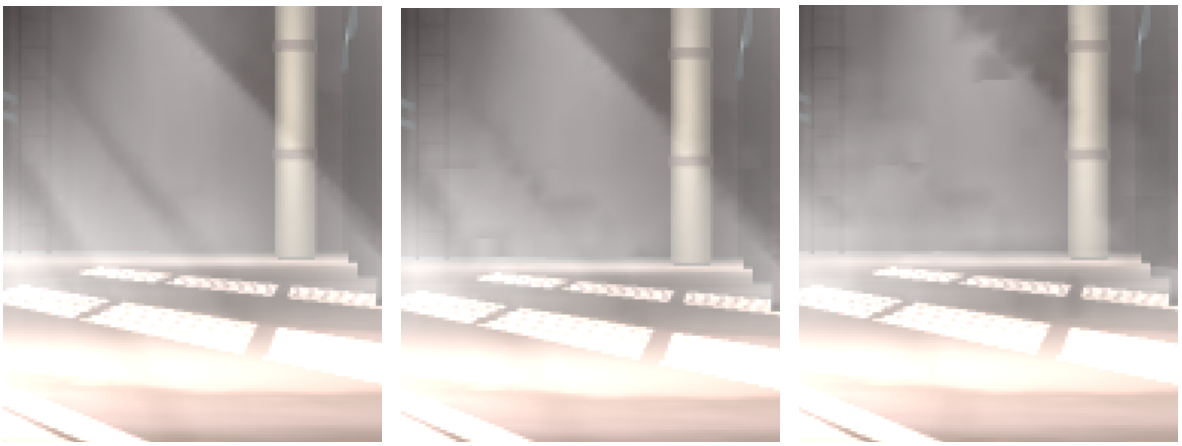
We have modeled the interior of a warehouse, with a number of windows letting sunlight in. The smoke volume covers the interior, extending from the left wall to the right wall, from the floor to the ceiling and from the back wall to the viewpoint. The viewpoint is slightly outside the volume. The step size we picked is 0.3 units (the shortest distance from the viewpoint to the back plane is 100 units). For smoke, we assume that all wavelengths are subject to same amount of scattering (color and opacity values have equal red, green, and blue components), thus, we store color and opacity as scalars. We have rendered images of size 800×600 anti-aliased (9 rays per pixel is shot.)

We investigated the speedup and actual error committed by the interpolation algorithm, as well as the number of ray samples required, and the percentage of cracks in the data structure for the on-demand compatible refinement algorithm. Speedup is the ratio of the CPU-time for the traditional ray marching approach to the CPU-time for our interpolation algorithm. The error committed by the interpolation algorithm is measured as the average distance between the actual color and opacity, and the corresponding quantity for the interpolated case. We also report the maximum error committed among all the rays shot. The color and opacity values are normalized to the range [0,1]. For our test scene, the actual color values are in the range [0, 0.2953], and the opacity values are in the range [0,0.5045]. Average color is 0.05419 and the average opacity is 0.1249. Fig 4(a), (b) and (c) demonstrate how the variation in error reflects the change in the quality of the rendered image. Notice the artifacts in (b) and (c) when the data structure is not subdivided as densely as in (a).

The percentage of cracks is given both in terms of the percentage of the final leaves (the leaves used for interpolation) that have cracks, and the percentage of the rays that are interpolated using the leaves with cracks.

The number of ray samples is the number of rays that are sampled during the construction of the data structure at simplex vertices. Sampling is the dominating cost. For example, for the compatible, two-pass rendering, the first pass during which the construction is done takes 90% of the total time, while the second pass takes only 10% of the total time to do point location and interpolation. For more expensive smoke rendering models, or for smaller step sizes, the cost of sampling will be even more dominant, since the time taken by interpolation and point location will remain almost constant. Hence, the speedup is bounded by the ratio of the total number of rays shot while rendering by ray marching to the number of sample rays generated while rendering by the interpolation algorithm. This suggests that, for higher resolution images, the speedups will be much higher.

Table 1 shows sample results for rendering the im-



(a) (b) (c)

Figure 4: The following errors are with respect to color. (a) distance thr = 0.015, average error = 0.00233, max error = 0.02704. (b) distance thr = 0.035, average error = 0.00371, max error = 0.06754. (c) distance thr = 0.05, average error = 0.00545, max error = 0.13001.

Algorithm	Speed-up	Error (color)		Error(opacity)		#Ray samples	%Cracks	
		average	maximum	average	maximum		%leaf nodes	%rays
Ray-marching	1	0	0	0	0	4,320,000	-	-
Compatible, two-pass	6.20	0.00230	0.02704	0.00396	0.04163	334,438	-	-
On-demand compatible	18.24	0.00233	0.02704	0.00401	0.04163	101,605	2.494	2.131

Table 1: Sample results for the warehouse scene (800×600 *antialiased, distance threshold=0.015).*

age by the compatible, two-pass method, and by the on-demand compatible algorithms. We used a distance threshold of 0.015 which was found to perform well experimentally. Recall that the distance threshold, described in Section 2.3, is used to determine whether to terminate a subdivision process. The on-demand compatible algorithm performs as well as the compatible, two-pass algorithm in terms of quality, while sampling 69% fewer rays and creating 92% fewer nodes. Therefore, the on-demand compatible algorithm achieves a significant speedup of 18.24, which is 3 times the speedup achieved by the compatible, two-pass method. Even the speedup of 6.2 for the compatible, two-pass method is significant for expensive applications like this one. Corresponding images are given in Fig. 5. Part (a) shows the correct image generated by marching all rays, and part (b) shows the interpolated image generated using the on-demand compatible algorithm. (Since the on-demand compatible algorithm generates almost the same image as the compatible two-pass algorithm, we show only the image generated by the on-demand version.)

If the on-demand compatible algorithm is used to render in multiple passes as explained in Section 3.2, the percentage of cracks is reduced substantially as shown in Table 2, but of course reducing the speedup.

If desired, higher quality approximations can be rendered by lowering the distance threshold, at the potential expense of performance.

References

[1] E. Allgower and K. Georg. Generation of triangulations by reflection. *Utilitas Mathematica*, 16:123–129, 1979.

[2] F. B. Atalay and D. M. Mount. Hierarchical simplicial meshes and multidimensional interpolation. Unpublished Manuscript, 2003.

[3] K. Bala, J. Dorsey, and S. Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Trans. on Graph.*, 18(3), August 1999.

[4] U. Behrens and R. Ratering. Adding shadows to a texture-based volume renderer. In *1998 Volume Visualization Symposium*, pages 39–46, 1998.

[5] S. Chandrasekhar. *Radiative Transfer*. Dover, New York, 1960.

[6] Y. Dobashi, T. Yamamoto, and T. Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. In *Graphics Hardware 2002*, pages 99–108, 2002.

Algorithm	%Cracks		Speed-up
	%leaf nodes	%rays	
On-demand compatible, one-pass	2.494	2.131	18.24
On-demand compatible, two-pass	0.204	0.122	13.43
On-demand compatible, three-pass	0.041	0.026	10.66

Table 2: Percentage of cracks for multiple passes of the on-demand compatible algorithm

[7] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In *Proc. of SIGGRAPH 2001*, pages 15–22, 2001.

[8] N. Foster and D. Metaxas. Modeling the motion of a hot, turbulent gas. In *Proc. of SIGGRAPH 97*, pages 181–188, 1997.

[9] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 43–54, August 1996.

[10] H. W. Jensen and P. H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proc. of SIGGRAPH 98*, pages 311–320, 1998.

[11] W. Krueger. The application of transport theory to visualization of 3d scalar fields. In *Proc. IEEE Visualization '90*, pages 273–280, 1990.

[12] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proc. IEEE Visualization '99*, pages 355–362, 1999.

[13] M. Levoy and P. Hanrahan. Light field rendering. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 31–42, August 1996.

[14] J. M. Maubach. Local bisection refinement for N-simplicial grids generated by reflection. *SIAM J. Sci. Stat. Comput.*, 16:210–227, 1995.

[15] N. Max. Atmospheric illumination and shadows. *Computer Graphics(Proc. of SIGGRAPH 86)*, 20(4):117–124, 1986.

[16] N. Max. Optical models for direct volume rendering. *IEEE Trans. on Visualization and Comp. Graph.*, 1(2):99–108, 1995.

[17] T. Nishita, Y. Miyawaki, and E. Nakamae. A shading model for atmospheric scattering considering luminous intensity of light sources. *Computer Graphics (Proc. of SIGGRAPH 87)*, 21(4):303–310, 1987.

[18] PhotoRealistic RenderMan Application Note#20. Writing fancy atmosphere shaders. http://graphics.stanford.edu/lab/soft/prman/Toolkit/-AppNotes/appnote.20.html.

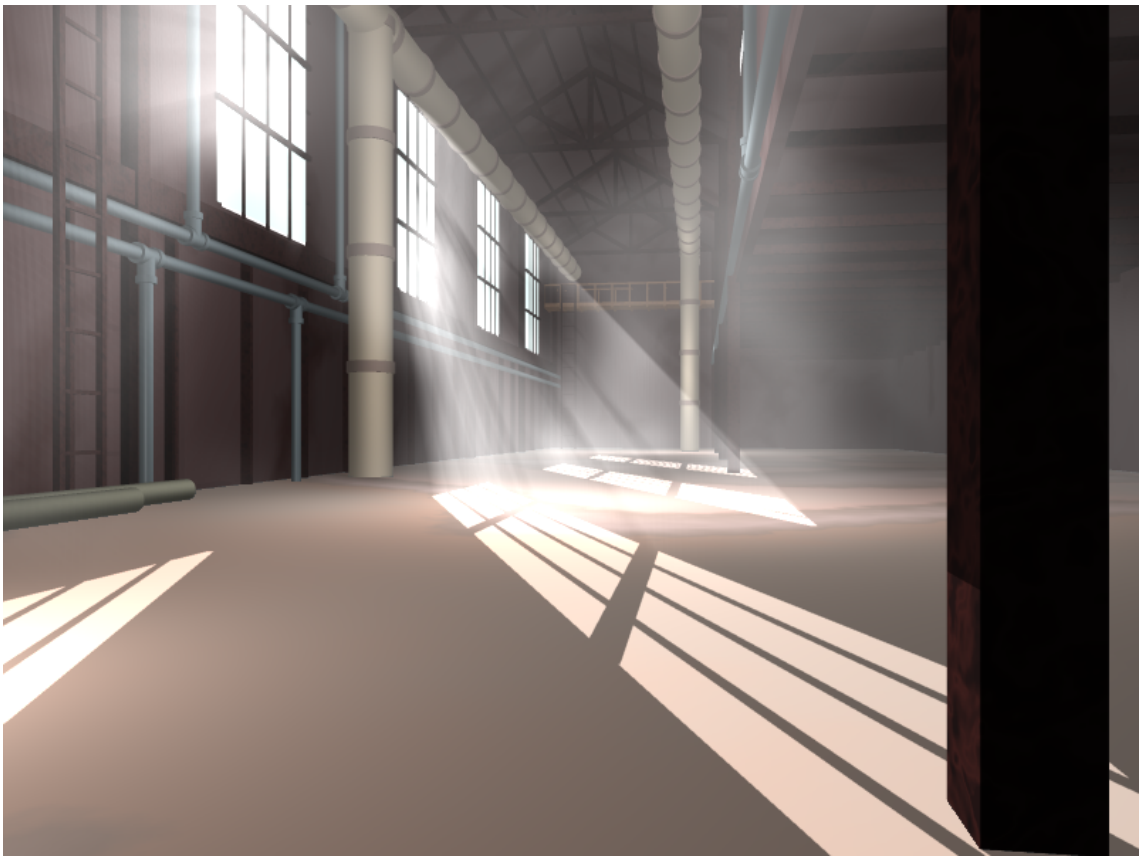
[19] A.J. Preetham, P.Shirley, and B. Smits. A practical analytic model for daylight. In *Proc. of SIGGRAPH 99*, pages 91–100, 1999.

[20] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

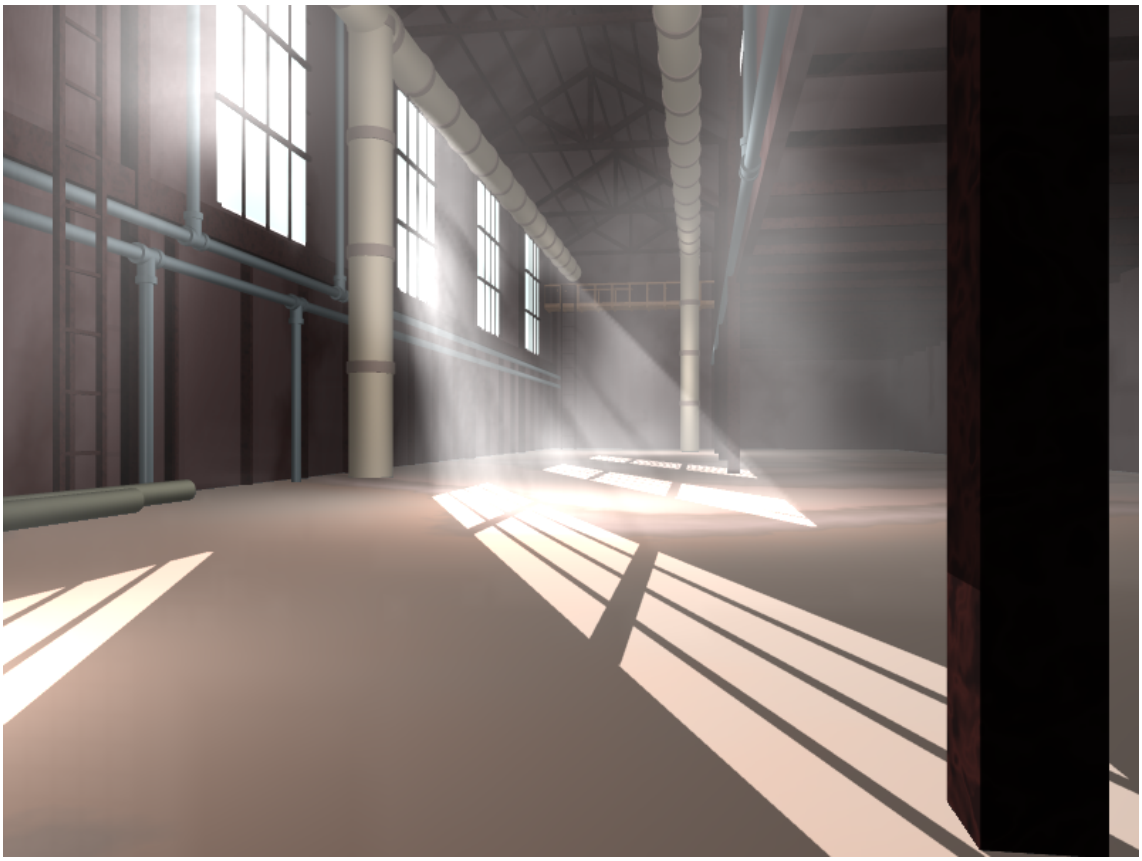
[21] D. M. Y. Sommerville. *Analytical Geometry in Three Dimensions*. Cambridge University Press, Cambridge, 1934.

[22] J. Stam. Stable fluids. In *Proc. of SIGGRAPH 99*, pages 121–128, 1999.

[23] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proc. of SIGGRAPH 98*, pages 169–178, 1998.



(a)



(b)

Figure 5: (a) Ray-marched image (b) Interpolated image using the on-demand compatible algorithm (800x600, anti-aliased, distance thr = 0.015).