The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids

Claudio T. Silva and Joseph S.B. Mitchell

Abstract—*Lazy Sweep Ray Casting* is a fast algorithm for rendering general irregular grids. It is based on the sweep-plane paradigm, and it is able to accelerate ray casting for rendering irregular grids, including disconnected and nonconvex (even with holes) unstructured irregular grids with a rendering cost that decreases as the "disconnectedness" decreases. The algorithm is carefully tailored to exploit spatial coherence even if the image resolution differs substantially from the object space resolution.

Lazy Sweep Ray Casting has several desirable properties, including its generality, (depth-sorting) accuracy, low memory consumption, speed, simplicity of implementation, and portability (e.g., no hardware dependencies).

We establish the practicality of our method through experimental results based on our implementation, which is shown to be substantially faster (by up to two orders of magnitude) than other algorithms implemented in software.

We also provide theoretical results, both lower and upper bounds, on the complexity of ray casting of irregular grids.

Index Terms—Volumetric data, irregular grids, volume rendering, sweep algorithms, ray tracing, computational geometry, scientific visualization.

1 Introduction

For the visualization of three-dimensional data, whether scalar or vector, direct volume rendering has emerged as a leading, and often preferred, method. While the surface rendering method can be applied to visualize volumetric data, they require the extraction of some structure, such as isosurfaces or streamlines, which may bias the resulting visualization. In rendering volumetric data directly, we treat space as composed of semitransparent material that can emit, transmit, and absorb light, thereby allowing one to "see through" (or see inside) the data [43], [22], [21]. Volume rendering also allows one to render surfaces, and, in fact, by changing the properties of the light emission and absorption, different lighting effects can be achieved [18].

The most common input data type is a *regular (Cartesian) grid* of *voxels*. Given a general scalar field in $\Re^3$, one can use a regular grid of voxels to represent the field by regularly sampling the function at grid points $(\lambda i, \lambda j, \lambda k)$, for integers i, j, k, and some scale factor $\lambda \in \Re$, thereby creating a regular grid of voxels. However, a serious drawback of this approach arises when the scalar field is *disparate*, having nonuniform resolution with some large regions of space having very little field variation, and other very small regions of space having very high field variation. In such cases, which often arise in computational fluid dynamics and partial differential equation solvers, the use of a regular grid is infeasible, since the voxel size must be small enough to model the smallest "features" in the field. Instead, *irregular grids* (or *meshes*), having cells that are not necessarily uniform cubes, have been proposed as an effective means of representing disparate field data.

Irregular grid data comes in several different formats [37], [41]. One very common format has been *curvilinear grids*, which are *structured* grids in computational space that have been "warped" in physical space, while preserving the same topological structure (connectivity) of a regular grid. However, with the introduction of new methods for generating higher quality adaptive meshes, it is becoming increasingly common to consider more general *unstructured* (noncurvilinear) irregular grids, in which there is no implicit connectivity information. Furthermore, in some applications *disconnected* grids arise.

Rendering of irregular grids has been identified as an especially important research area in visualization [17]. The basic problem consists of evaluating a volume rendering equation [21] for each pixel of the image screen. To do this, it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells of the mesh along the ray. This information is used to evaluate the overall integral in the rendering equation.

In this paper, we present and analyze the *Lazy Sweep Ray Casting* algorithm, a new method for rendering general meshes, which include unstructured, possibly disconnected, irregular grids. A primary contribution of the Lazy Sweep Ray Casting (LSRC) algorithm is a new method for accurately calculating the depth-sorted ordering. LSRC is based on ray casting and employs a sweep-plane approach, as proposed by Giertsen [15], but introduces several new ideas that permit a faster execution, both in theory and in practice.

This paper is built upon the paper of Silva, Mitchell, and Kaufman [36], where the fundamentals of our method were developed. In the months since the writing of [36], we have made several improvements and extensions; as we report our latest results here, we will compare them to the results in the earlier work of [36].

• *The authors are with the Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, Stony Brook, NY. 11794-3600. E-mail: {csilva, jsbm}@ams.sunysb.edu.*

For information on obtaining reprints of this article, please send e-mail to: transvcg@computer.org, and reference IEEECS Log Number 104719.0.

1.1 Definitions and Terminology

A *polyhedron* is a closed subset of $\Re^3$ whose boundary consists of a finite collection of convex polygons (*two-faces*, or *facets*) whose union is a connected two-manifold. The *edges* (*one-faces*) and *vertices* (*zero-faces*) of a polyhedron are simply the edges and vertices of the polygonal facets. A convex polyhedron is called a *polytope*. A polytope having exactly four vertices (and four triangular facets) is called a *simplex* (*tetrahedron*). A finite set S of polyhedra forms a *mesh* (or an *unstructured, irregular grid*) if the intersection of any two polyhedra from S is either empty, a single common edge, a single common vertex, or a single common facet of the two polyhedra; such a set S is said to form a *cell complex*. The polyhedra of a mesh are referred to as the *cells* (or *three-faces*). If the boundary of a mesh S is also the boundary of the convex hull of S, then S is called a *convex* mesh; otherwise, it is called a *nonconvex* mesh. If the cells are all simplices, then we say that the mesh is *simplicial*.

We are given a mesh S. We let c denote the number of connected components of S. If $c = 1$, we say that the mesh is *connected*; otherwise, the mesh is *disconnected*. We let n denote the total number of edges of all polyhedral cells in the mesh. Then, there are $O(n)$ vertices, edges, facets, and cells.

For some of our theoretical discussions, we will be assuming that the input mesh is given in any standard data structure for cell complexes (e.g., a facet-edge data structure [10]), so that each cell has pointers to its neighboring cells, and basic traversals of the facets are also possible by following pointers. If the raw data does not have this topological information already encoded in it, then it can be obtained by a preprocessing step, using basic hashing methods.

Our implementation of the LSRC algorithm relies on only a very simple and economical structure in the input data. In particular, we store with each vertex v its "use set" (see [32]), which is simply a list of the cells of the mesh that "use" v (have v as a vertex of the cell). Note that this requires only $O(n)$ storage, since the total size of all use sets is bounded by the sum of the sizes of the cells.

The image space consists of a screen of N-by-N pixels. We let $\rho_{i,j}$ denote the ray from the eye of the camera through the center of the pixel indexed by (i, j). We let $k_{i,j}$ denote the number of facets of S that are intersected by $p_{i,j}$. Finally, we let $k = \sum_{i,j} k_{i,j}$ be the total complexity of all ray casts for the image. We refer to k as the *output complexity*. Clearly, $\Omega(k)$ is a lower bound on the complexity of ray casting the mesh. Note that $k = O(N^2 n)$, since each of the N^2 rays intersects at most $O(n)$ facets.

1.2 Related Work

A simple approach for handling irregular grids is to resample them, thereby creating a regular grid approximation that can be rendered by conventional methods [28], [42]. In order to achieve high accuracy, it may be necessary to sample at a very high rate, which not only requires substantial computation time, but may well make the resulting grid far too large for storage and visualization purposes. Several rendering methods have been optimized for the case of curvilinear grids; in particular, Frühauf [12] has developed a method in which rays are "bent" to match the grid deformation. Also, by exploiting the simple structure of curvilinear grids, Mao et al. [20] have shown that these grids can be efficiently resampled with spheres and ellipsoids that can be presorted along the three major directions and used for splatting.

A direct approach to rendering irregular grids is to compute the depth sorting of cells of the mesh along each ray emanating from a screen pixel. For irregular grids, and especially for disconnected grids, this is a nontrivial computation to do efficiently. One can always take a naive approach, and, for each of the N^2 rays, compute the $O(n)$ intersections with cell boundary facets in time $O(n)$, and then sort these crossing points (in $O(n \log n)$ time). However, this results in overall time $O(N^2 n \log n)$, and does not take advantage of coherence in the data: The sorted order of cells crossed by one ray is not used in any way to assist in the processing of nearby rays.

Garrity [14] has proposed a preprocessing step that identifies the boundary facets of the mesh. When processing a ray as it passes through interior cells of the mesh, connectivity information is used to move from cell to cell in constant time (assuming that cells are convex and of constant complexity). But every time that a ray exits the mesh through a boundary facet, it is necessary to perform a "FirstCell" operation to identify the point at which the ray first reenters the mesh. Garrity does this by using a simple spatial indexing scheme based on laying down a regular grid of voxels (cubes) on top of the space, and recording each facet with each of the voxels that it intersects. By casting a ray in the regular grid, one can search for intersections only among those facets stored with each voxel that is stabbed by the ray.

The FirstCell operation is in fact a "ray shooting query," for which the field of computational geometry provides some data structures: One can either answer queries in time $O(\log n)$, at a cost of $O(n^{4+\varepsilon})$ preprocessing and storage [2], [4], [8], [27], or answer queries in worst-case time $O(n^{3/4})$, using a data structure that is essentially linear in n [3], [33]. In terms of worst-case complexity, there are reasons to believe that these tradeoffs between query time and storage space are essentially the best possible. Unfortunately, these algorithms are rather complicated, with high constants, and have not yet been implemented or shown to be practical. (Certainly, data structures with super-linear storage costs are not practical in volume rendering.) This motivated Mitchell et al. [23] to devise methods of ray shooting that are "query sensitive"—the worst-case complexity of answering the query depends on a notion of local combinatorial complexity associated with a reasonable estimate of the "difficulty" of the query, so that "easy" queries take provably less time than "hard" queries. Their data structure is based on octrees (as in [31]), but with extra care needed to keep the space complexity low, while achieving the provably good query time.

Uselton [39] proposed the use a Z-buffer to speed up FirstCell; Ramamoorthy and Wilhelms [30] point out that this approach is only effective 95 percent of the time. They also point out that 35 percent of the time is spent checking for exit cells and 10 percent for entry cells. Ma [19] describes a parallelization of Garrity's method. One of the

disadvantages of these ray casting approaches is that they do not exploit coherence between nearby rays that may cross the same set of cells.

Another approach for rendering irregular grids is the use of projection ("feed-forward") methods [22], [45], [34], [38], in which the cells are projected onto the screen, one-by-one, in a *visibility ordering*, incrementally accumulating their contributions to the final image. One advantage of these methods is the ability to use graphics hardware to compute the volumetric lighting models in order to speed up rendering. Another advantage of this approach is that it works in object space, allowing coherence to be exploited directly: By projecting cells onto the image plane, we may end up with large regions of pixels that correspond to rays having the same depth ordering, and this is discovered without explicitly casting these rays. However, in order for the projection to be possible, a depth ordering of the cells has to be computed, which is, of course, not always possible; even a set of three triangles can have a cyclic overlap. Computing and verifying depth orders is possible in $O\left(n^{4/3+\varepsilon}\right)$ time [1], [7], [9], where $\epsilon > 0$ is an arbitrarily small positive constant. In case no depth ordering exists, it is an important problem to find a small number of "cuts" that break the objects in such a way that a depth ordering does exist; see [7], [5]. BSP trees have been used to obtain such a decomposition, but can result in a quadratic number of pieces [13], [26]. However, for some important classes of meshes (e.g., rectilinear meshes and Delaunay meshes [11]), it is known that a depth ordering always exists, with respect to any viewpoint. This structure has been exploited by Max et al. [22]. Williams [45] has obtained a linear-time algorithm for visibility ordering convex (connected) acyclic meshes whose union of (convex) cells is itself convex, assuming a visibility ordering exists. Williams also suggests heuristics that can be applied in case there is no visibility ordering or in the case of nonconvex meshes, (e.g., by tetrahedralizing the nonconvexities which, unfortunately, may result in a quadratic number of cells). In [40], techniques are presented where no depth ordering is strictly necessary and, in some cases, calculated approximately. Very fast rendering is achieved by using graphics hardware to project the partially sorted faces.

A recent scanline technique that handles multiple and overlapping grids is presented in [44]. They process the set of polygonal facets of cells, by first bucketing them according to which scanline contains the topmost vertex, and then maintaining a "y-actives list" of polygons present at each scanline, as they sweep from top to bottom (in y). Then, on each scanline, they scan in x, bucketing polygons according to their left extent, and then maintaining (via merging) a z-sorted list of polygons, as they scan from left to right. The method has been parallelized and used within a multiresolution hierarchy, based on a kD tree.

Two other important references on rendering irregular grids have not yet been discussed here—Giertsen [15] and Yagel et al. [47]. We elaborate on these in the next section, as they are closely related to our method.

In summary, projection methods are potentially faster than ray casting methods, since they exploit spatial coherence. However, projection methods give inaccurate renderings if there is no visibility ordering, and methods to break cycles are either heuristic in nature or potentially costly in terms of space and time.

2 Sweep-Plane Approaches

A standard algorithmic paradigm in computational geometry is the "sweep" paradigm [29]. Commonly, a *sweep-line* is swept across the plane, or a *sweep-plane* is swept across three-space. A data structure, called the *sweep structure* (or *status*), is maintained during the simulation of the continuous sweep, and at certain discrete *events* (e.g., when the sweep-line hits one of a discrete set of points), the sweep structure is updated to reflect the change. The idea is to localize the problem to be solved, solving it within the lower dimensional space of the sweep-line or sweep-plane. By processing the problem according to the systematic sweeping of the space, sweep algorithms are able to exploit spatial coherence in the data.

2.1 Giertsen's Method

Giertsen's pioneering work [15] was the first step in optimizing ray casting by making use of coherency in order to speed up rendering. He performs a sweep of the mesh in three-space, using a sweep-plane that is parallel to the x-z plane. Here, the viewing coordinate system is such that the viewing plane is the x-y plane, and the viewing direction is the z direction; see Fig. 1. The algorithm consists of the following steps:

1) Transform all vertices of S to the viewing coordinate system.
2) Sort the (transformed) vertices of S by their y-coordinates; put the ordered vertices, as well as the y-coordinates of the scanlines for the viewing image, into an event priority queue, implemented in this case as an array, A.
3) Initialize the *Active Cell List* (ACL) to empty. The ACL represents the sweep status; it maintains a list of the cells currently intersected by the sweep-plane.
4) While A is not empty, do:
   a) Pop the event queue: If the event corresponds to a vertex, v, then go to b; otherwise, go to c.
   b) Update ACL: Insert/delete, as appropriate, the cells incident on v. (Giertsen assumed that the cells are disjoint, so each v belongs to a single cell.)
   c) Render current scanline: Giertsen uses a memory hash buffer, based on a regular grid of squares in the sweep-plane, allowing a straightforward casting of the rays that lie on the current scanline.

By sweeping three-space, Giertsen reduces the ray casting problem in three-space to a two-dimensional cell sorting problem.

Giertsen's method has several advantages over previous ray casting schemes. First, there is no need to maintain connectivity information between cells of the mesh. (In fact, he assumes the cells are all disjoint.) Second, the computationally expensive ray shooting in three-space is replaced by a simple walk through regular grid cells in a plane. Finally, the method is able to take advantage of coherence from one scanline to the next.

However, there are some drawbacks to the method, including:

1) The original data is coarsened into a finite resolution buffer (the memory hashing buffer) for rendering,

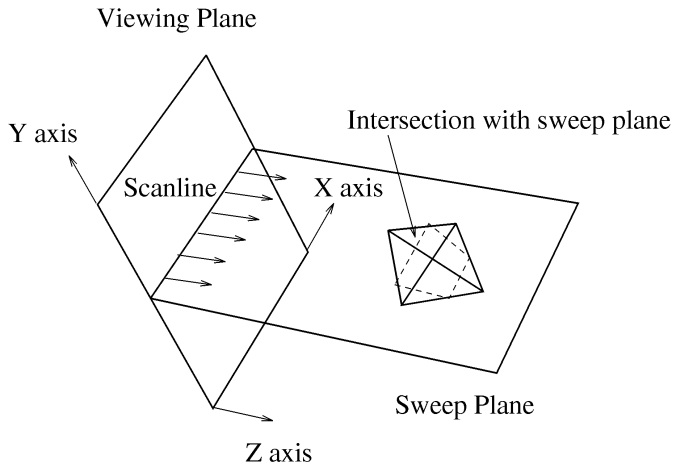


Fig. 1. A sweep-plane (perpendicular to the *y*-axis) used in sweeping three-space.

basically limiting the resolution of the rendering, and possibly creating an aliasing effect. While one could simply increase the size of the buffer, this approach is impractical in large datasets, where the cell size variation can be on the order of 1:100,000. Further, Giertsen mentions that when cells get mapped to the same location, this is considered a degenerate case, and the later cells are ignored; however, this resolution might lead to temporal aliasing when calculating multiple images.

2) Another disadvantage when comparing to other ray casting techniques is the need to have two copies of the dataset, as the transformation and sorting of the cells must be done before the sweeping can be started. (Note that this is also a feature of cell projection methods.) One cannot just keep retransforming a single copy, since floating point errors could accumulate.

2.2 Yagel et al.'s Method

In [46], [47], Yagel et al. proposed a method that uses a sweep-plane *parallel* to the viewing plane. At each position of the sweep-plane, the plane is intersected with the grid, resulting in a two-dimensional slice, each of whose cells are then scan-converted using the graphics hardware in order to obtain an image of that slice, which can then be composited with the previously accumulated image that resulted from the sweep so far. Several optimizations are possible. For example, instead of performing a full sort along the *z*-direction, a bucketing technique can be used. Also, the intersections of mesh edges with the slices can be accelerated by storing incremental step sizes (Δx and Δy) corresponding to the interslice distance (Δz); however, this speedup requires considerably more memory. Furthermore, the storage of the polygons in any given slice requires a significant amount of memory (e.g., 13.4 MB for the Blunt Fin [47]).

This method can handle general polyhedral grids without having to compute adjacency information, and, conceptually, it can generate high quality images at the expense of "slice oversampling." The simplicity of the method makes it very attractive for implementation and use. (Ideally, the user should have access to high-performance graphics hardware and an abundance of memory.)

3 The Lazy Sweep Ray Casting Algorithm

The design of our new method is based on two main goals:

1) The depth ordering of the cells should be correct along the rays corresponding to every pixel; and
2) The algorithm should be as efficient as possible, taking advantage of structure and coherence in the data.

With the first goal in mind, we chose to develop a new ray casting algorithm, in order to be able to handle cycles among cells (a case causing difficulties for projection methods). To address the second goal, we use a sweep approach, as did Giertsen, in order to exploit both *interscanline* and *interray* coherence. Our algorithm has the following advantages over Giertsen's:

1) It avoids the explicit transformation and sorting phase, thereby avoiding the storage of an extra copy of the vertices;
2) It makes no requirements or assumptions about the level of connectivity or convexity among cells of the mesh; however, it does take advantage of structure in the mesh, running faster in cases that involve meshes having convex cells and convex components;
3) It avoids the use of a hash buffer plane, thereby allowing accurate rendering even for meshes whose cells greatly vary in size;
4) It is able to handle parallel and perspective projection within the same framework, without explicit transformations.

3.1 Performing the Sweep

Our sweep method, like Giertsen's, sweeps space with a sweep-plane that is orthogonal to the viewing plane (the *x*-*y* plane), and parallel to the scanlines (i.e., parallel to the *x*-*z* plane).

Events occur when the sweep-plane hits vertices of the mesh *S*. But, rather than sorting all of the vertices of *S* in advance and placing them into an auxiliary data structure (thereby at least doubling the storage requirements), we maintain an event queue (priority queue) of an appropriate (small) subset of the mesh vertices.

A simple (linear-time) preprocessing pass through the data readily identifies the set of vertices on the boundary of the mesh. We initialize the event queue with these boundary vertices, prioritized according to the magnitude of their inner product (dot product) with the vector representing the *y*-axis ("up") in the viewing coordinate system (i.e., according to their *y*-coordinates). (We do *not* explicitly transform coordinates.) Furthermore, at any given instant, the event queue only stores the set of boundary vertices not yet swept over, plus the vertices that are the upper endpoints of the edges currently intersected by the sweep-plane. In practice, the event queue is relatively small, usually accounting for a very small percentage of the total data size. As the sweep takes place, new vertices (nonboundary ones) will be inserted into and deleted from the event queue each time the sweep-plane hits a vertex of *S*.

As the sweep algorithm proceeds, we maintain a *sweep status* data structure, which records the necessary information about the current slice through *S* in an "active

edge" list—see Section 5. When the sweep-plane encounters a vertex event (as determined by the event queue), the sweep status and the event queue data structures must be updated. In the main loop of the sweep algorithm, we pop the event queue, obtaining the next vertex, v, to be hit, and we check whether or not the sweep-plane encounters v before it reaches the y-coordinate of the next scanline. If it does hit v first, we perform the appropriate insertions/deletions on the event queue and the sweep status structure; these are easily determined by local tests (checking the signs of dot products) in the neighborhood of v. Otherwise, the sweep-plane has encountered a scanline. At this point, we stop the sweep and drop into a two-dimensional ray casting procedure (also based on a sweep) as described below. The algorithm terminates once the last scanline is processed.

3.2 Processing a Scanline

When the sweep-plane encounters a scanline, the current (3D) sweep status data structure gives us a "slice" through the mesh in which we must solve a two-dimensional ray casting problem. Let S denote the polygonal (planar) subdivision at the current scanline (i.e., S is the subdivision obtained by intersecting the sweep-plane with the mesh S.) In time linear in the size of S, the subdivision S can be recovered (both its geometry and its topology) by stepping through the sweep status structure and utilizing the local topology of the cells in the slice. (The sweep status gives us the set of edges intersecting the sweep plane; these edges define the vertices of S, and the edges of S can be obtained by searching the set of triangular facets incident on each such edge.) In our implementation, however, S is not constructed explicitly, but only given implicitly by the sweep status data structure (a list of "active edges"), and then *locally* reconstructed as needed during the two-dimensional sweep (described below). The details of the implementation are nontrivial and they are presented in Section 5.

The two-dimensional ray casting problem is also solved using a sweep algorithm—now we sweep the plane with a sweep-line parallel to the z axis. (Or, in the case of perspective projection, we sweep with a ray eminating from the viewer's eye.) Events now correspond to vertices of the planar subdivision S, which occur at intersection points between an "active edge" in the (3D) sweep status and the current sweep-plane. These event points are processed in x-order; thus, we begin by sorting them. (An alternative approach, mentioned in Section 4, is to proceed as we did in 3D, by first identifying and sorting only the locally extremal vertices of S, and then maintaining an event queue during the sweep. Since a single slice has relatively few event points compared with the size of S, we opted, in our implementation, simply to sort them outright.) The *sweep-line status* is an ordered list of the segments of S crossed by the sweep-line. The sweep-line status is initially empty. Then, as we pass the sweep-line over S, we update the sweep-line status at each event point, making (local) insertions and deletions as necessary. (This is analogous to the Bentley-Ottmann sweep that is used for computing line segment intersections in the plane [29].) We also stop the sweep at each of the x-coordinates that correspond to the rays that we are casting (i.e., at the pixel coordinates along the current scanline), and output to the rendering model the sorted ordering (depth ordering) given by the current sweep-line status.

4 Analysis: Upper and Lower Bounds

We now proceed to give a theoretical analysis of the time required to render irregular grids. We begin with "negative" results that establish lower bounds on the worst-case running time:

THEOREM 1 (Lower Bounds). *Let S be a mesh having c connected components and n edges. Even if all cells of S are convex, $\Omega(k + n \log n)$ is a lower bound on the worst-case complexity of ray casting. If all cells of S are convex and, for each connected component of S, the union of cells in the component is convex, then $\Omega(c \log c)$ is a lower bound. Here, k is the total number of facets crossed by all N^2 rays that are cast through the mesh (one per pixel of the image plane).*

PROOF. It is clear that $\Omega(k)$ is a lower bound, since k is the size of the output from the ray casting.

Let us start with the case of c convex components in the mesh S, each made up of a set of convex cells. Assume that one of the rays to be traced lies exactly along the z-axis. In fact, we can assume that there is only one pixel at the origin in the image plane. Then, the only ray to be cast is the one along the z-axis, and k simply measures how many cells it intersects. To show a lower bound of $\Omega(c \log c)$, we simply note that any ray tracing algorithm that outputs the intersected cells, in order, along a ray, can be used to sort c numbers, z_i. (Just construct, in $O(c)$ time, tiny disjoint tetrahedral cells, one centered on each z_i.)

Now, consider the case of a *connected* mesh S, all of whose cells are convex. We assume that all local connectivity of the cells of S is part of the input mesh data structure. (The claim of the theorem is that, even with all of this information, we still must effectively perform a sort.) Again, we claim that casting a single ray along the z-axis will require that we effectively sort n numbers, $z_1, \ldots, z_n$. We take the unsorted numbers z_i and construct a mesh S as follows. Take a unit cube centered on the origin and subtract from it a cylinder, centered on the z-axis, with cross sectional shape a regular $2n$-gon, having radius less than $1/2$. Now remove the half of this polyhedral solid that lies above the x-z plane. We now have a polyhedron P of genus 0 that we have constructed in time $O(n)$. We refer to the n (skinny) rectangular facets that bound the concavity as the "walls." Now, for each point $(0, 0, z_i)$, create a thin "wedge" that contains $(0, 0, z_i)$ (and no other point $(0, 0, z_j)$, $j \neq i$), such that the wedge is attached to wall i (and touches no other wall). Refer to Fig. 2. We now have a polyhedron P, still of genus 0, of size $O(n)$, and this polyhedron is easily decomposed in $O(n)$ time into $O(n)$ convex polytopes. Further, the z-axis intersects (pierces) all n of the wedges, and does so in the order given by the sorted order of the z_is.

Thus, the output of a ray tracing algorithm that has one ray along the z-axis must give us the sorted order of the n wedges, and, hence, of the n numbers z_i. The $\Omega(n \log n)$ bound follows. □

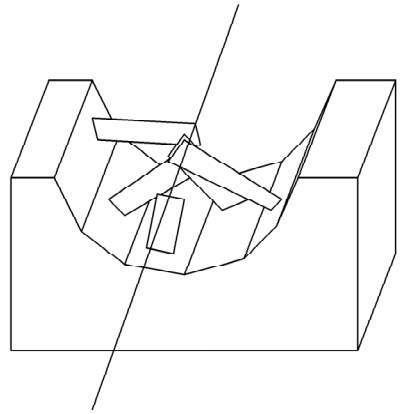


Fig. 2. Lower bound construction.

REMARK. *It may be tempting to think that if one is given a convex mesh (e.g., connected, with tetrahedral cells), that this information can be used to sort the vertices of the mesh (e.g., by x-coordinate) in linear time, thereby using topological information to make sweep algorithms more efficient. However, it is easy to show that, even in two dimensions, if we are given a triangulation with complete topological information, it still requires time* $\Omega(n \log n)$ *to sort the n vertices by their x-coordinates. (The proof, based on a reduction from sorting, is left to the reader.)*

4.1 Upper Bounds

The previous theorem establishes lower bounds that show that, in the worst case, any ray casting method will have complexity that is superlinear in the problem size—essentially, it is forced to do some sorting. However, the pathological situations in the lower bound constructions are unlikely to arise in practice.

We now examine upper bounds for the running time of the sweep algorithm we have proposed, and we discuss how its complexity can be written in terms of other parameters that capture problem instance complexity.

First, we give a worst-case upper bound. In sweeping three-space, we have $O(n)$ vertex events, plus N (presorted) "events" when we stop the sweep and process the two-dimensional slice corresponding to a scanline. Each operation (insertion / deletion) on the priority queue requires time $O(\log M)$, where M is the maximum size of the event queue. In the worst case, M can be of the order of n, so we get a worst-case total of $O(N + n \log n)$ time to perform the sweep of three-space.

For each scanline slice, we must perform a sweep as well on the subdivision S, which has worst-case size $O(n)$. The events in this sweep algorithm include the $O(n)$ vertices of the subdivision (which are intersections of the slice plane with the edges of the mesh, S), as well as the N (presorted) "events" when we stop the sweep-line at discrete pixel values of x, in order to output the ordering (of size $k_{i,j}$ for the ith pixel in the jth scanline) along the sweep-line, and pass it to the rendering module. Thus, in the worst case, this sweep of 2-space requires time $O\left(\sum_i k_{i,j} + n \log n\right)$ for slice j, for an overall cost, for all N slices, of $O\left(\sum_{i,j} k_{i,j} + Nn \log n\right) = O(k + Nn \log n)$.[1]

Now, the product term, Nn, in the bound of $O(k + Nn \log n)$ is due to the fact that each of the N slices might have complexity roughly n. However, this is a pessimistic bound for practical situations. Instead, we can let n_s denote the total sum of the complexities of all N slices; in practice, we expect n_s to be much smaller than Nn, and, potentially, n_s is considerably smaller than n. (For example, if the mesh is uniform, we may expect each slice to have complexity of $n^{2/3}$, as in the case of a $n^{1/3}$-by-$n^{1/3}$-by-$n^{1/3}$ grid, which gives rise to $n_s = O(Nn^{2/3})$.) If we now write the complexity in terms of n_s, we get worst-case running time of $O(k + n \log n + n_s \log n)$.

Note that, in the worst case, $k = \Omega(N^2 n)$; e.g., it may be that every one of the N^2 rays crosses $\Omega(n)$ of the facets in the mesh. Thus, the output size k could end up being the dominant term in the complexity of our algorithm. Note too that, even in the best case, $k = \Omega(N^2)$, since there are N^2 rays.

The $O(n \log n)$ term in the upper bound comes from the sweep of three-space, where, in the worst case, we may be forced to (effectively) sort the $O(n)$ vertices (via $O(n)$ insertions / deletions in the event queue). We now discuss how we can analyze the complexity in terms of the number, n_c, of "critical" vertices; this approach was used in the two-dimensional triangulation algorithm of Hertel and Mehlhorn [16].

Consider the sweep of three-space with the sweep-plane. We say that vertex v is *critical* if, in a small neighborhood of v, the number of connected components in the slice changes as the sweep-plane passes through v. (Thus, vertices that are locally min or max are critical, but also some "saddle" points may be critical.) Let n_c denote the number of critical vertices. Now, note that the lower bound construction that shows that, in the worst case, we must resort to sorting, is quite contrived: In particular, it has $n_c = \Omega(n)$, while one would expect in practice that n_c is very small (say, on the order of c, the number of connected components of the mesh).

Now, if we conduct our sweep of three-space carefully, we can get away with only having to sort the critical vertices, resulting in total time $O(n + n_s + n_c \log n_c)$ for constructing all N of the slices. (Similarly, Hertel and Mehlhorn [16] were able to triangulate polygonal regions in the plane in time $O(n + n_c \log n_c)$, compared with the previous bound of $O(n \log n)$ based on plane sweep.) The main idea is to exploit the topological coherence between slices, noting that the number of connected components

1. The upper bound of $O(k + Nn \log n)$ should be contrasted with the bound $O(N^2 n \log n)$ obtained from the most naive method of ray casting, which computes the intersections of all N^2 rays with all $O(n)$ facets, and then sorts the intersections along each ray.

changes only at critical vertices (and their y-coordinates are sorted, along with the N scanlines). In particular, we can use depth-first search to construct each connected component of S within each slice, given a starting "seed" point in each component. These seed points are obtained from the seed points of the previous slice, simply by walking along edges of the grid (in the direction of increasing y-coordinate), from one seed to the next slice (in total time $O(n)$, for all walks); changes only occur at critical vertices, and these are local to these points, so they can be processed in time linear in the degree of the critical vertices (again, overall $O(n)$). This sweep of three-space gives us the slices, each of which can then be processed as already described. (Note that the extremal vertices within each slice can be discovered during the construction of the slice, and these are the only vertices that need to be sorted and put into the initial event queue for the sweep of a slice.)

In summary, we have

THEOREM 2 (Upper Bound). *Ray casting for an irregular grid having n edges can be performed in time $O(k + n + n_c \log n_c + n_s \log n)$, where $k = O(N^2 n)$ is the size of the output (the total number of facets crossed by all cast rays), $n_s = O(Nn)$ is the total complexity of all slices, and $n_c = O(n)$ is the number of critical vertices.*

REMARK. *The upper bound shows only linear dependence on n, while the lower bound theorem showed an $\Omega(n \log n)$ lower bound. This is not a contradiction, since, in the proof of the lower bound, the construction has $n_c = \Omega(n)$ critical vertices; this is in agreement with the upper bound term, $O(n_c \log n_c)$.*

Another potential savings, particularly if the image resolution is low compared with the mesh resolution, is to "jump" from one slice to the next, *without* using the sweep to discover how one slice evolves into the next. We can instead construct the next slice from scratch, using a depth-first search through the mesh, and using "seed" points that are found by intersecting the new slice plane with a critical subgraph of mesh edges that connects the critical vertices of the mesh. Of course, we do not know a priori if it is better to sweep from slice i to slice $i + 1$, or to construct slice $i + 1$ from scratch. Thus, we can perform *both* methods in parallel, on two processors, and use the result obtained by the first processor to complete its task. (Alternatively, we can achieve the same effect using a single processor by performing a "lock step" algorithm, doing steps in alternation between the two methods.) This results in an asymptotically complexity that is the minimum of the complexities of the two methods. This scheme applies not just to the sweep in three-space, but also to the sweeps in each slice.

As an illustration of how these methods can be quite useful, consider the situation in Fig. 3, which, while drawn only in two dimensions, can depict the cases in three-space as well. When we sweep from line 2 to line 3, a huge complexity must be swept over, and this may be costly compared to rebuilding from scratch the slice along line 3. On the other hand, sweeping from line 5 to line 6 is quite cheap (essentially no change in the geometry and topology), while constructing the slice along line 6 from scratch would be quite costly. By performing the two methods in parallel (or in "lock step"), we can take advantage of the best of both methods. The resulting algorithm exploits coherence in the data and has a running time that is sensitive, in some sense, to the complexity of the visualization task. Note that, in practice, when the image resolution is very low, one would probably prefer to oversample and then filter, rather than to use this method of "jumping" from slice to slice or from ray to ray.

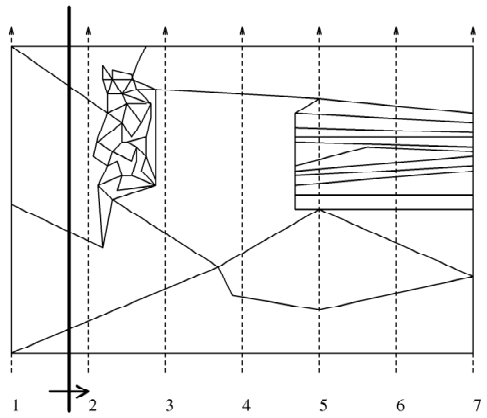


Fig. 3. Illustration of a sweep in one slice.

5 Implementation Details

We have implemented a version of the main LSRC algorithm with some simplifications. Here, we discuss some of the details of our implementation, concentrating on the most relevant issues unique to implementing LSRC. We try to present enough details so that an experienced graphics programmer can reproduce our results with minimal guess work.

Our current implementation handles general disconnected grids; however, it also assumes, for simplicity, that cells of the mesh are tetrahedra (simplices). The extension to more complex convex (or even nonconvex) cells is conceptually straightforward, while the details are somewhat tedious and do not contribute to the basic understanding of the algorithm.

There are other ways in which our implemented algorithm differs from the methods discussed previously, in the section on upper bounds. This is for two reasons—simplicity of coding and efficiency in practice (both in terms of running time and in terms of memory). In our discussions below, we point out how the implemented algorithm differs both for the 3D sweep (in inserting into the heap all boundary vertices of the grid, rather than just the critical vertices) and for the 2D sweep (in our maintaining of the sweep-line status).

Our implementation, in its entirety, consists of less than 5,000 lines of C code. We have not yet attempted to optimize the code, so we expect that it can be further improved.

The major modules of the program include:

- *3D sweep*, which sweeps the input mesh with a plane orthogonal to the viewing plane, while maintaining an active edge list (`AEdge`), and marking those tetrahedra that have been swept over;
- *2D sweep*, which sweeps a slice, producing the sorted intersections of cells along each ray of a scanline.

We also have a graphics module that handles computations of coordinates with respect to the viewing coordinate frame, manages the other modules, and computes the transfer function and the optical integration (or simple shading). When we speak below of *x*-, *y*-, or *z*-coordinates, these are all calculated using simple dot products (with the defining unit vectors of the viewing frame), and are not the result of a full coordinate transformation (which we seek to avoid).

5.1 Major Data Structures

Due to the large sizes of irregular grids, efficient data structures can substantially influence the performance and memory requirements of the implementation.

We basically have two "big" data structures:

- The `Vertex` list, which contains, for each vertex, its position and field value(s), its "use set" (list of tetrahedra containing it), and a couple of other utility data fields (e.g., a general-purpose flag).
- The `Tetrahedron` list, which contains, for each tetrahedron, pointers to its four vertices and one flag data field used to indicate if the sweep-plane has reached it yet in the 3D sweep.

In our experiments, these two main data structures typically occupy 95 percent of the overall space used by the algorithm. This organization of the data is memory efficient, while allowing the necessary connectivity information to be recovered quickly within the algorithm. Since each tetrahedron contains four vertices, the total amount of memory required by all of the "use sets" is bounded by $4\times$ the number of tetrahedra (this clearly extends to other cell complexes composed of cells of bounded complexity). We collect the vertices on the boundary of the meshes in lists so, during the sweep, we can preinsert them on the priority queues. It is important to note that not all points on the boundary need to be inserted.

In the 3D sweep, our sweep-plane (orthogonal to the *y*-axis) is moved from top to bottom, in the direction of decreasing *y*. As the sweep progresses, we need to be able to detect what is the next event, which corresponds to the closest vertex in the direction of the 3D sweep (*y*-axis). This is done by maintaining a priority queue that contains (some of the) vertices sorted along the *y*-axis. In particular, the priority queue contains those vertices not yet encountered by the sweep-plane, which are the bottom endpoints of the "active" edges of *S* intersected by the sweep-plane. The priority queue is implemented as a heap, `3DHeap`. Vertices are inserted as they are discovered (when a neighboring vertex above is encountered by the sweep-plane), and they are deleted as they are swept over.

For the sweep status data structure, we do not explicitly keep a list of active tetrahedra, as this is not necessary, but we do keep a list, `AEdge`, of which edges are currently active.

The `AEdge` (active edge) list is the central data structure in our implementation. Each `AEdge` element contains data fields used in several different phases of the algorithm. We have not yet attempted to optimize the storage space associated with the `AEdge` list; it typically does not contain a particularly large number of elements, since it represents only a cross section of the dataset. (The size of a cross section is typically only about $n^{2/3}$; e.g., in a regular *m*-by-*m*-by-*m* mesh, a cross section has complexity $O(m^2)$, while $n = m^3$.) An active edge entry in `AEdge` contains:

- pointers to its endpoints.
- a record of its intersection with the current position of the sweep-plane.
- pointers to its "top segment" and "bottom segment" (defined below, when we detail the 2D sweep).
- a few other data fields used for bookkeeping.

In addition to insertions and deletions, the `AEdge` list must support endpoint queries: Given a pair of vertices, *v* and *w*, determine the entry of `AEdge` that have the pair as endpoints. For this, we have implemented a simple-minded, hash-based dictionary data structure. We have experimented with using other data structures for keeping `AEdge`, such as a binary tree, but the overhead of keeping these more complex data structures seems to outweigh their advantages.

In the 2D sweep, we will also use the `Segment` data structure, which stores *pairs* of active edges that belong to the same facet of some cell. Such a pair of active edges determines a line segment in the current slice. Each `Segment` object also has two pointer fields to allow for the construction of double-linked lists of `Segment` objects, corresponding to the sweep-line status data structure, a depth-sorting of segments along each ray.

5.2 3D Sweep

In the 3D sweep, the events are determined by when the sweep-plane hits a vertex or arrives at a scanline. Since the *y*-coordinates of the scanlines are predetermined (and sorted), we have only to concern ourselves with the *y*-coordinates of the vertices. Since we are trying to be "lazy" about the sweep, we are interested in avoiding creating a single sorted list of *all* vertices, so we proceed as follows. First, in a single (preprocessing) pass over the `Vertex` list, we identify all of the vertices that lie on the boundary of the grid *S*; typically, this set of vertices is only a tiny fraction of the total set. Then, for a given viewing frame, we insert these boundary vertices into the `3DHeap`, based on *y*-coordinate key values. (Our current implementation does not take advantage of the fact that we can restrict attention to *critical* vertices, as discussed in Section 4; the boundary vertices, which can be identified in a preprocessing step (as opposed to critical vertices, which are defined with respect to a view-dependent *y*-axis) will be a (still small) superset of the critical vertices.) This aspect of the algorithm allows us to exploit nice structure that may be present in the input—grids that have few connected components, with each component being well-shaped (having relatively few boundary vertices), will allow our 3D sweep algorithm to run faster, as the only nonlinear time component of the algorithm is sensitive to the number of vertices on the boundary of the grid.

Next, we begin the sweep, using this `3DHeap` to identify vertex events. As the sweep progresses, we process vertex events in a natural way, by making insertions and deletions to the `3DHeap` and the `AEdge` list accordingly. Based on the "use set" of a vertex, we can determine the local geometry

about it, and thereby decide what insertions/deletions to make; see Fig. 4. The vertex event processing proceeds as follows:

While `3DHeap` is not empty, do

1) Remove from `3DHeap` the vertex v that has smallest key value (y-coordinate).
2) For each cell C that contains v,
   a) If v is the topmost vertex of C, insert the other vertices of C into the `3DHeap`, add the incident edges to the `AEdge` list, and mark C and its vertices as "visited."
   b) If v is the bottommost vertex of C, remove the incident edges from the `AEdge` list.
   c) Otherwise, make insertions and deletions from the `AEdge` list according to which edges incident on v are below or above it.

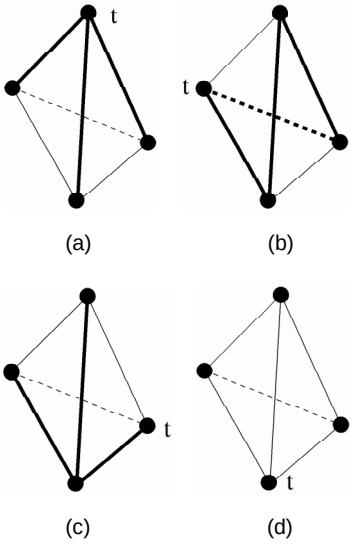


Fig. 4. Processing a cell C during the 3D sweep: (a) the sweep-plane hits the topmost vertex of C—the three incident edges are added to `AEdge`; (b) the sweep-plane hits an intermediate vertex of C—one edge is removed from `AEdge` and two edges are added; (c) the sweep-plane hits another intermediate vertex of C–two edges are removed from `AEdge` and one edge is added; (d) the sweep plane hits the bottommost vertex of C—three edges are removed from `AEdge`.

5.3 2D Sweep

The 3D sweep stops each time that the sweep-plane arrives at a scanline, at which point a 2D sweep occurs in the corresponding slice of S. Rather than explicitly constructing the slice (e.g., building a winged-edge data structure for the two-dimensional subdivision S), we use only the `AEdge` list to represent implicitly the structure of S. We refer to the line segments that are edges in the subdivision S as *segments* rather than "edges," in order to distinguish them from the edges of the three-dimensional mesh S (which are elements of `AEdge`). Since segments are determined by a pair of edges bounding a common face, the `Segment` data type simply stores such pairs. (The endpoints of a segment are determined by the intersection of the edges of the pair with the sweep-plane.)

In the 2D sweep, we maintain the ordering of segments intersected by a line, parallel to the z-axis, which is swept across the slice; this data structure is the *sweep-line status*. Typically, sweep-line algorithms utilize some form of balanced binary tree in order to affect efficient (logarithmic) insertion, deletion, and lookup in the sweep-line status structure. Indeed, in our first implementation of the 2D sweep, we too used a binary tree to store the sorted order of segment crossings; see [36]. However, through further experimentation, we have determined that a different (and simpler) approach works faster in practice, even though it cannot guarantee logarithmic worst-case performance. Thus, we describe here our current method of maintaining the sweep-line status structure.

Our 2D sweep begins by computing the intersections of the active edges (in `AEdge`) with the sweep-plane, caching them, and sorting them in x as we place them into the event priority queue, which is implemented as a heap—`2DHeap`. (Since a single slice is relatively small in size, we go ahead in this case with a full sorting for simplicity of implementation.) The sweep-line status structure is implemented as a doubly linked list of `Segment` objects, which represent the sorted list of segments intersecting the current sweep-line. When the sweep-line hits an active edge (i.e., hits a point p in the slice, where an active edge intersects the slice), we process this event, making updates to the sweep-line status structure and the `2DHeap` as necessary. The overall sweep algorithm proceeds as follows:

While `2DHeap` is not empty, do

1) Remove from `2DHeap` the active edge, (v_0, v_1), with the smallest key value (x-coordinate). Let v_0 be the vertex that is above (in y) the current slice.
2) For each cell C in the use set of v_0,
   a) If C is *not* in the use set of v_1, then we are done considering C (since (v_0, v_1) is not an edge of C); otherwise, proceed to (b).
   b) For each of the other vertices of C (exactly two, in the case of tetrahedral cells), determine if it forms an active edge (by querying the `AEdge` list) with one of v_0 or v_1; if so, then instantiate a `Segment` corresponding to that edge and (v_0, v_1). These `Segment` objects are inserted, as explained below, in a doubly linked (sorted) list that corresponds to the sweep-line status structure.

Step 2b above discovers the segments that are incident on the event point p, which is the intersection of the active edge (v_0, v_1) with the sweep-plane.

The updates to the sweep-line status structure are done in a manner that exploits the topological structure in the mesh (see Fig. 5). In particular, when point p is encountered, if there are leftward segments incident on p, then we identify them (using "top" and "bottom" pointers, described below), and delete them from the doubly-linked list. At the same time, we insert the rightward segments incident on p, after sorting them by angle (using only dot product computations) about p, using as insertion point the position in the list where the leftward segments had been. In this way, we need to do no searching in the sorted list of segments, *except* in the case that there were no leftward segments incident on p (in which case, we do a naive linear-time search in the linked list). While we could do these

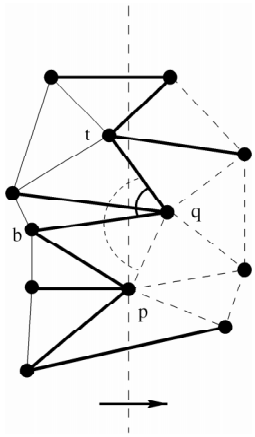


Fig. 5. Illustration of the action of the 2D sweep. The solid "thick" edges represent the elements of the `Segment` data structure currently in the sweep-line. The dashed elements have not been touched by the sweep-line yet. When the sweep-line encounters event point *p*, we discover edge (*p*, *q*), and therefore update the bottom segment of *q*, from (*b*, *q*) to (*p*, *q*). (The top segment of *q*, (*t*, *q*), remains unchanged.)

search and insertions more efficiently, in worst-case logarithmic time, we have found that the overhead associated with the data structures does not pay off in practice. Further, in the vast majority of cases, there is no linear search to do, since most event points have one or more leftward segments. (Indeed, those event points having no leftward segments are "critical" in the sense described earlier, as in Hertel and Mehlhorn [16].)

Specifically, we maintain, with each active edge, pointers to two additional `Segment` objects: a *top* segment and a *bottom* segment, representing the topmost and bottommost, respectively, among the leftward segments incident on the corresponding crossing point, *p*. These pointers are initialized to NULL. We maintain these pointers each time a new segment is added (when we discover its left endpoint), at which point we check its right endpoint and potentially update the top/bottom segment pointer of its corresponding active edge. If the active edge corresponding to event point *p* has a non-NULL top and bottom pointer, we know where to add the new segments (to the right of *p*), without having to search the whole sweep-line status structure. (If the pointers are NULL, then we must do a linear search to locate *p* in the linked list, since, in this case, *p* has no leftward segments.)

There are several advantages to our new approach (compared to the former binary-tree approach). Notice that now, we are only inserting edges where they share an endpoint. This allows for a much simpler and more robust ordering function. In our implementation, we use a 2D determinant method, which requires four subtractions, two multiplications, and one comparison in the general case, plus two extras comparisons to handle degeneracies when determining the correct ordering between two segments that share an endpoint. When performing the insertions into the sweep-line status, we still have to be careful in handling degeneracies, like in [36], but the case analysis is much simpler.

5.4 Final Rendering Issues

There is an issue of handling degeneracies when event points happen to coincide with *y*-coordinates of scanlines or with *x*-coordinates of pixels within a scanline. Thus, in our 3D sweep, we must be careful to process all event vertices that have the same *y*-coordinate before starting the processing of the 2D slice. Similarly, when sweeping a slice, we only perform the rendering along a ray once all event points that may have the same *x*-coordinate as the ray are processed.

5.4.1 Interpolation

Because the original scalar field is only provided at the original vertices and during rendering, we need to be able to evaluate the field at any given point, some form of interpolation is necessary. This is a nontrivial step in general, and considerable research has been devoted to this topic. We refer the reader to [24], [25]. In our current implementation, for tetrahedral cells, our approach is straightforward. To compute the value of the scalar field at the point *r*, where a ray crosses a segment (*p*, *q*) (in a 2D slice), we first use linear interpolation along each of the active edges (in `AEdge`) that define *p* and *q* to compute the values at *p* and *q*, and then do a third interpolation along (*p*, *q*) to determine the value at *r*.

5.4.2 Lighting Model

Once the stabbing order of the cells along a ray has been computed, any single-scattering lighting model can be applied. (See [21] for a survey.) We implemented the simple lighting model proposed by Uselton [39], in which cell size is not taken into consideration. The assumption is that each cell is as important as any other cell. We have been able to generate very good pictures with this method, but it does tend to overemphasize portions of the volume having particularly high cell density.

6 Experimental Results

6.1 Datasets

The code currently handles datasets composed of tetrahedral grids (possibly disconnected, with nonconvex boundary). The input format is very similar to the GeomView "off" file format: It simply has the number of vertices and tetrahedra, followed by a list of the vertices and a list of the tetrahedra, each of which is specified using the vertex locations in the file as an index. This format is compact, can handle general (disconnected) grids, and it is fairly simple (and fast) to recover topological information. Maintaining explicit topological information in the input file would waste too much space.

For our test runs, we have used tetrahedralized versions of four different datasets, all originally in NASA Plot3D format. For each dataset we broke each (hexahedral) cell into five tetrahedra. Information about the datasets are summarized in Table 1. (See volume-rendered images in Figs. 8-11.) Besides these, we tested LSRC on several artificial datasets for debugging purposes; in particular, we generated simple datasets that have disconnected components.

TABLE 1
A LIST OF THE DATASETS USED FOR TESTING

Name	Dimensions	# of Vertices	# of Cells
Blunt Fin	$40 \times 32 \times 32$	40,960	187,395
Liquid Oxygen Post	$38 \times 76 \times 38$	109,744	513,375
Delta Wing	$56 \times 54 \times 70$	211,680	1,005,675
Combustion Chamber	$57 \times 33 \times 25$	47,025	215,040

"Dimensions" are the original NASA Plot3D sizes. "# of Vertices" and "# of Cells" are the actual sizes used by LSRC during rendering.

TABLE 2
MEMORY CONSUMPTION DURING RENDERING

Data Structure	Blunt Fin	Liquid Oxygen Post	Delta Wing	Combustion Chamber
Dataset Size	7.8MB	21.3MB	41.8MB	9MB
`AEdge`	390KB	675KB	2.14MB	375KB
`Segment`	8KB	8KB	20KB	4KB

"Dataset Size" includes the memory necessary to keep all the vertices (including their "use set") and tetrahedra. The ***`AEdge`*** *row gives the space used in storing the list of active edges cut by the current sweep-plane. The* ***`Segment`*** *row gives the storage requirement for the sweep-line status, representing the stabbing order of the cells along each ray.*

6.2 Memory Requirements

LSRC is very memory efficient. (See Section 5 for details about the data structures.) Besides the input dataset, the only other memory consumption is in the priority queues, and the `AEdge` and `Segment` data structures, which are very small in practice. This low storage requirement is due to our incremental computations, which only touch one cross section of the dataset at a time. See Table 2 for details about the overall memory consumption during the rendering of each dataset. These numbers are independent of the screen size being rendered, although they do depend on the "view," given that different cross sections of the datasets might lead to different memory usage patterns.

6.3 Performance Analysis

Our primary system for measurements was a Silicon Graphics Power Challenge, equipped with 16 processors (R10,000 195MHz), and three GB of RAM. We only used one of the processors during our experiments. All of the disk I/O numbers reflect reading off a local disk. We present rendering figures for the tetrahedralized version of the datasets described in Table 1. (We expect our rendering times to be considerably less if we work directly with the hexahedral cells without first tetrahedralizing them; however, the current implementation assumes tetrahedral cells.) The LSRC code was compiled with the native SGI compiler (for IRIX 6.2) and optimization level "–O3." All times reported are in seconds and represent measured wall-clock times.

In Table 3, we present the times to read and preprocess the datasets. Our input files are currently ASCII, which requires some amount of parsing upon reading; thus, the "Reading" time is dominated by this parsing time, not by disk access time. (The use of binary files would likely improve efficiency, but using ASCII files simplifies the manual creation of test samples.)

TABLE 3
TIME SPENT READING AND PREPROCESSING THE DATA

Operation	Blunt Fin	Liquid Oxygen Post	Delta Wing	Combustion Chamber
Reading	3.86s	10.48s	20.69s	4.51s
Connectivity	3.47s	9.62s	18.98s	4.02s
Boundary vertices	6,760	13,840	20,736	7,810

"Reading" accounts for the time spent reading and parsing the dataset off the disk. "Connectivity" represents the time spent recovering the adjacency and boundary information. The "Boundary vertices" row gives the number of vertices we classified as being on the boundary of the dataset.

TABLE 4
RENDERING RESULTS FOR THE FOUR DATASETS

	Blunt Fin	Liquid Oxygen Post	Delta Wing	Combustion Chamber
Image Size	530×230	300×300	300×300	300×200
Rendering Time	22s	37s	64s	19s
Full Pixels	83,264	70,503	48,009	33,947

Table 4 presents the rendering times for the different datasets. Each dataset has been rendered at a different resolution, primarily because it would not make sense to present square images for all of them, since their projections do not cover a square region. We also present the pixel coverage (number of "Full Pixels") for each image. These rendering times are about three to four times faster than the ones presented earlier in [36]. (In [36], only the Blunt Fin and Liquid Oxygen Post were used; there, it was reported that it took 70 seconds to render the Blunt Fin, while the new results reported here obtain a time of 22 seconds, an improvement by a factor of 3.1; for the Post dataset, the improvement has been from 145 seconds to 37 seconds, a factor of 3.9.)

We also tested how our algorithm scales with the image size: We rendered the Liquid Oxygen Post in three different resolutions: 300×300 (70,503 full pixels), 600×600 (282,086 pixels), and 900×900 (634,709 pixels), and the rendering times were 37 seconds, 82 seconds, and 136 seconds, respectively. This indicates that the cost per pixel actually decreases as the image size increases. This matches our intuition, because the larger the image, the less "useless sorting" we have to do per scanline. That is, in 2Dsweep, we basically get all the sorting information in a continuum along the scanline, but we only use that information along each pixel actually rendered. As the image size gets larger, the less "sorting" work 2Dsweep has to do per pixel rendered. For very large images, the shading cost should dominate. At this point, the sorting becomes essentially "free," as it has constant cost for a given dataset and view.

So far, we have shown that the new method is over three times as fast as the one presented in [36]. It is important to understand where the speedup was achieved. In order to be able to analyze the differences, we will recalculate Figs. 5 and 6 from [36] using our new method. We are using the same dataset (e.g., the Blunt Fin), in order to make direct performance comparisons possible. Fig. 6 illustrates

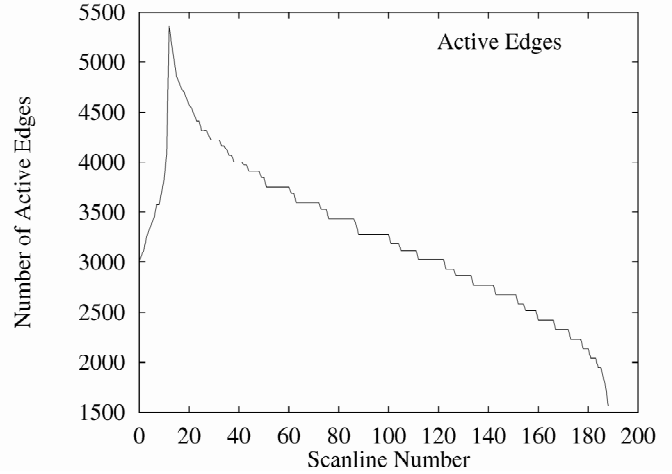


Fig. 6. The size of the `AEdge` list as a function of the scanline (*y*-coordinate).

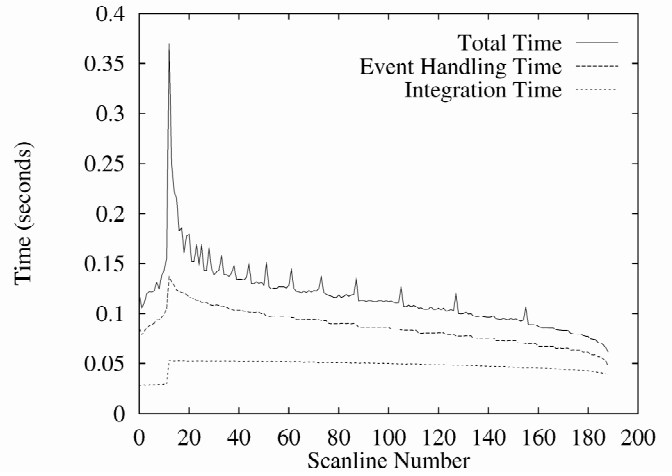


Fig. 7. An illustration of the breakdown of the total rendering time per scanline. The "Total Time" represents the actual time each scanline required for rendering. In order to avoid clutter in the plot, only the two major components of the rendering time are shown: the "Event Handling Time" (which is the time to process each active edge as it enters and exists the sweep-line status), and "Integration Time" (which is the time necessary for the shading calculations).

how the number of active edges varies in *y* during the 3D sweep. (This figure corresponds to Fig. 5 in [36].)

Fig. 7 illustrates how the rendering time breaks down by task, as a function of the scanline (again, for the Blunt Fin dataset so we can compare with the earlier results presented in [36]). Rendering a scanline involves computing the intersection points, sorting them along the direction of the scanline, performing a 1D sweep (or sort) along each ray incrementally (which basically involves processing events), and finally shading (or intergration time). The two components presented in Fig. 7 correspond to over 85 percent of the overall time spent in rendering. (The "Event Handling Time" is approximately 50 percent of the time and "Integration Time" is about 30 percent).

The results in Fig. 7 should be compared to Fig. 6 [36]. Our improvements to the 2D sweep, as explained in the previous section, resulted in several changes. First, the processing of each scanline is about three times as fast. Second, the event handling time is much lower (previously, it accounted for over 80 percent of the rendering time). Because of the lowering of the cost of handling the events, we can now clearly see the relative increase in the cost for the shading phase. (Before, the event handling cost was so dominating that all of the other processing time was negligible and did not appear clearly on the graph.)

The performance numbers indicate that:

1) The time to process a given scanline is directly correlated to the number of active edges corresponding to that slice;
2) The cost per scanline varies depending on the complexity of the slice being rendered; and,
3) The event handling time still dominates the total time spent per scanline.

In [36], the event handling time was clearly the bottleneck of the rendering speed. Now, it still accounts for about 50 percent of the overall rendering time. Future improvements may be possible based on reuse of intersweep planes sorting information or the use of some form of "jumping" over complexity between pixels (as in the lockstep idea proposed before).

6.4 Performance Comparisons

The most recent report on an irregular grid ray caster is that of Ma [19], from October 1995. Ma is using an Intel Paragon (with superscalar 50MHz Intel i860XPs). He reports rendering times for two datasets—an artificially generated *Cube* dataset with 130,000 tetrahedra and a *Flow* dataset with 45,500 tetrahedra. He does not report times for single CPU runs; his experiments use two processing nodes. For the *Cube*, he reports taking 2,415 seconds (2,234 seconds for the ray casting—the rest is parallel overhead) for a 480-by-480 image (approximately 230,000 pixels), for a total cost of 10.5 (9.69) milliseconds per pixel. The cost per tetrahedron is 18.5 (17.18) milliseconds. For the *Flow* dataset he reports 1,593 (1,585) milliseconds (same image size), for a cost of 6.9 (6.8) milliseconds per pixel, and 35.01 (34.8) milliseconds per tetrahedron.

Giertsen [15] reports running times of 38 seconds for 3,681 cells (10.32 milliseconds per cell). His dataset is too small (and too uniform) to allow direct and meaningful comparisons; however, our implementation handles a cell complex that has over 100 times the number of cells he used at a fraction of the cost per cell.

Yagel et al. [47] report rendering the Blunt Fin, using an SGI with a Reality Engine^2, in just over nine seconds, using a total of 21MB of RAM, using 50 "slicing" planes; with 100 planes, they report a rendering time of 13-17 seconds. (Their rendering time is dependent on the number of "slicing" planes, which, of course, affects the accuracy of the picture generated.) For a 50-slice rendering of the Liquid Oxygen Post, it takes just over 20 seconds, using about 57MB of RAM. For the Delta Wing, it takes almost 43 seconds and uses 111.7MB of RAM.

In order to facilitate comparisons, Table 5 summarizes all the performance results with the available data for each reported algorithm. Comparing these numbers with

TABLE 5
PERFORMANCE SUMMARY OF SEVERAL ALGORITHMS (INDICATED IN THE LAST COLUMN)

Dataset	# of Cells	Ren. Time	μ/Pixel	μ/Cell	Image Size	Memory	Algorithm
Blunt Fin	187,395	22s	180μs	117μs	530×230	8MB	LSRC
Post	513,375	37s	411μs	72μs	300×300	22MB	LSRC
Post	513,375	82s	227μs	159μs	600×600	22MB	LSRC
Post	513,375	136s	167μs	264μs	900×900	22MB	LSRC
Delta Wing	1,005,675	64s	711μs	63μs	300×300	44MB	LSRC
Chamber	215,040	19s	316μs	88μs	300×200	9MB	LSRC
Blunt Fin	187,395	70s	373μs	664μs	527×200	8MB	[36]
Post	513,375	145s	1,611μs	282μs	300×300	22MB	[36]
Cube	130,000	2,415s	10,500μs	18,500μs	480×480	N/A	Ma
Flow	45,500	1,593s	6,900μs	35,010μs	480×480	N/A	Ma
Blunt Fin	187,395	9.11s	N/A	48μs	N/A	21MB	Yagel
*Blunt Fin	187,395	13s–17s	N/A	69–91μs	N/A	21MB	Yagel
Post	513,375	20.45s	N/A	40μs	N/A	57MB	Yagel
Delta Wing	1,005,675	42.97s	N/A	42μs	N/A	112MB	Yagel
N/A	3,681	38s	144μs	10,320μs	512×512	2.7MB	Giertsen

"LSRC" are for results for the lazy sweep ray casting algorithm proposed in this paper; "[36]" are for the results we obtained in our previous work; "Yagel" are for results reported in [47]; "Ma" are for results reported in [19]; and "Giertsen" are results reported in [15]. The table includes columns indicating the datasets used, their sizes, and, when possible, the cost per pixel and per cell, and the memory usage of each algorithm. (For Yagel et al., 50-plane rendering times are reported, with the exception of the row marked with a "," which represents the rendering times using 100 planes.)*

those in Table 4, we see that LSRC is much faster than the other ray casting algorithms. Furthermore, it is comparable in performance to Yagel et al.'s method for 100-slice rendering, but it uses less than half of the memory used by their technique. By looking at the increase in rendering times as the datasets get larger, we see that the larger the dataset the more advantageous it is to use LSRC over these other techniques.

7 ALGORITHM EXTENSIONS

In this section, we mention some of the possible extensions to this work:

1) While our current implementation assumes tetrahedral cells, it is conceptually simple to extend it to arbitrary cells. The method itself applies in general.
2) It is straightforward to generalize our method to the case of multiple grids: We simply perform the sweep independently in each of the several grids and do a merge sort of the results along each ray, just before rendering.
3) We are now investigating some possible methods to improve our algorithm so that it exploits more of the coherence between scanline slices. It is reasonable to expect us to be able to reuse much of the slice information from one scanline to the next. In particular, the order of the (2D) event points is nearly the same for two consecutive slices. An improvement here could help to address the current bottleneck in the code.
4) An interesting possible extension of our work that we are now investigating is its application in "out-of-core" cases, in which the dataset is too large to fit in main memory, and we must be careful to control the number of paging operations to disk. The spatial locality of our memory accesses indicates that we should be able to employ *prefetching* techniques to achieve fast rendering when the irregular grids are much larger than memory.
5) Finally, our method is a natural candidate for parallelization. See Silva [35, Chapter 5], for further discussion on parallelization issues.

8 CONCLUSIONS

In this paper, we have proposed a fast new algorithm, termed the "Lazy Sweep Ray Casting" (LSRC) algorithm, for rendering irregular grids based on a sweep-plane approach. Our method is similar to other ray casting methods in that it does not need to *transform* the grid; instead, it uses (as do projection methods) the adjacency information (when available) to determine ordering and to attempt to optimize the rendering. An interesting feature of our algorithm is that its running time and memory requirements are sensitive to the complexity of the rendering task. Furthermore, unlike the method of Giertsen [15], we conduct the ray casting within each "slice" of the sweep-plane by a sweep-line method whose accuracy does not depend on the uniformity of feature sizes in the slice. Our method is able to handle the most general types of grids without the explicit transformation and sorting used in other methods, thereby saving memory and computation time while performing an accurate ray casting of the datasets. We established the practicality of our method through experimental results based on our implementation. We have also discussed theoretical lower

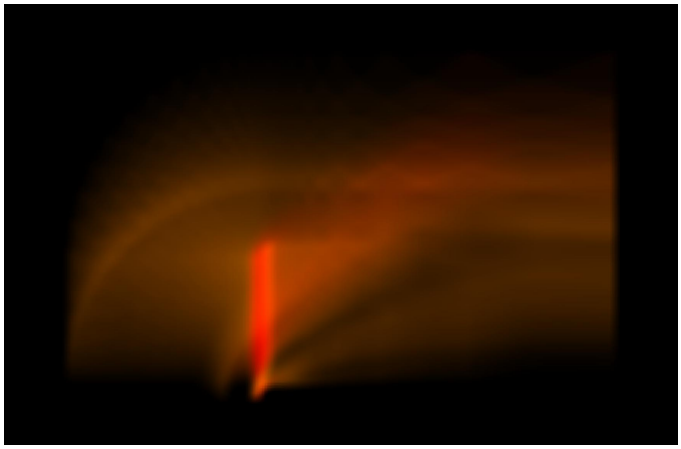


Fig. 8. Blunt Fin.

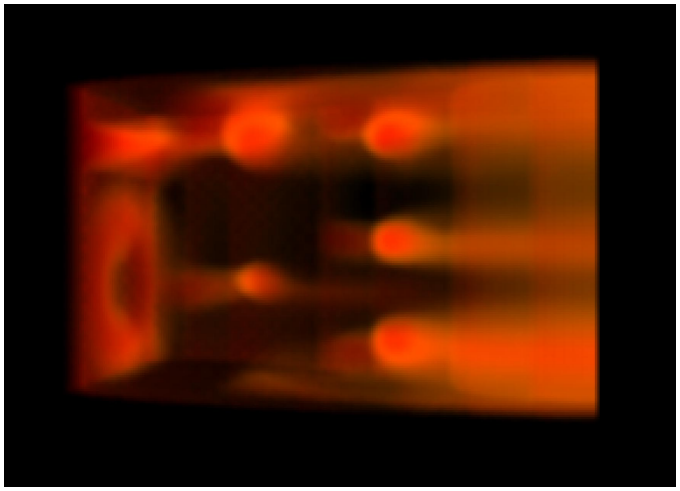


Fig. 9. Combustion Chamber.

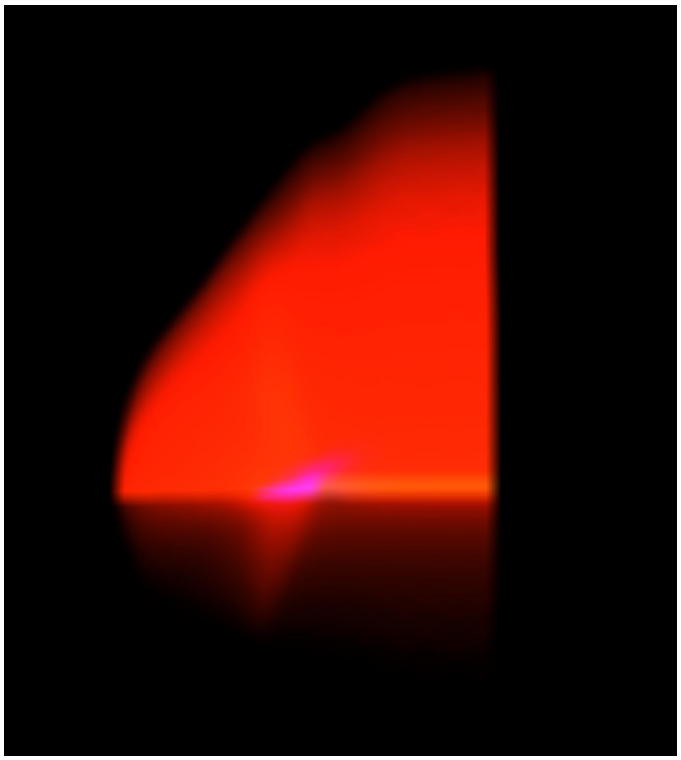


Fig. 10. Delta Wing.

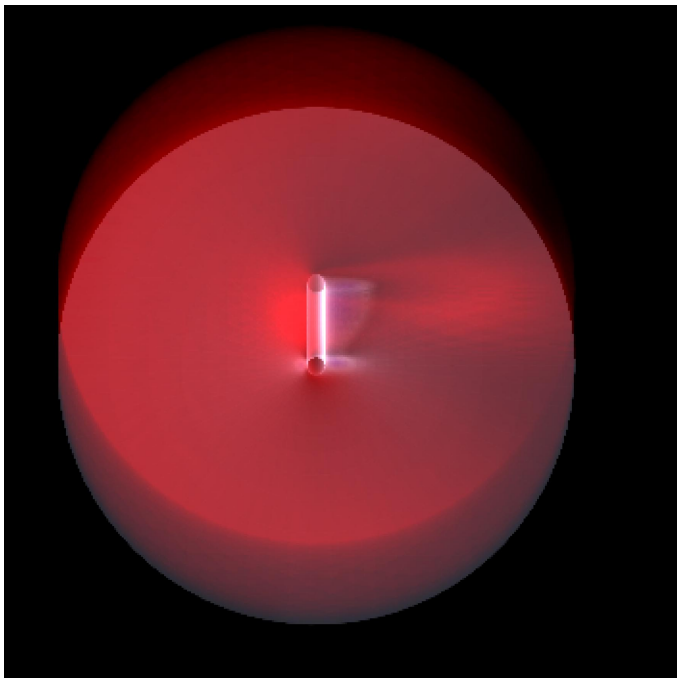


Fig. 11. Liquid Oxygen Post.

and upper bounds on the complexity of ray casting in irregular grids.

We have reported timing results showing that our method compares favorably with other ray casting schemes, and is, in many instances, two orders of magnitude faster than other published ray casting results. Another advantage of our method is that it is very memory efficient, making it suitable for use with very large datasets.

It is difficult to give a direct comparison of our method with hardware-based techniques (e.g., [47]), which can yield impressive speed-ups over purely software-based algorithms. On the other hand, software-based solutions broaden the range of machines on which the code can run; e.g., much of our code was developed on a small laptop, with only 16MB of RAM. Further, we are optimistic that implementation of the optimizations suggested in the last section will further improve the performance of our software. More experimentation should help us quantify exactly how our algorithm compares with other methods.

ACKNOWLEDGMENTS

We are indebted to Arie Kaufman for extensive discussions and encouragement on this research, as well as contributions to this paper; a precursor [36] to this paper was prepared jointly with him. We also thank the Center for Visual Computing (A. Kaufman, Director), for use of the computing resources in our experiments. We thank Dirk Bartz, Pat Crossno, George Davidson, Juliana Freire, Dino Pavlakos,

Ashish Tiwari, and Brian Wylie for useful criticism and help in this work. The Blunt Fin, the Liquid Oxygen Post, and the Delta Wing datasets are courtesy of NASA. The Combustion Chamber dataset is from Vtk [32].

This paper was supported in part by Sandia National Labs, by U.S. National Science Foundation grant CDA-9626370, U.S. National Science Foundation grant CCR-9504192, Hughes Aircraft, Boeing, and Sun Microsystems. Part of this work was conducted while C. Silva was partially supported by CNPq-Brazil on a PhD fellowship.

REFERENCES

[1] P.K. Agarwal, M.J. Katz, and M. Sharir, "Computing Depth Orders and Related Problems," *Proc. Fourth Scandinavian Workshop Algorithm Theory*, pp. 1-12, Lecture Notes in Computer Science, vol. 824, Springer-Verlag, 1994.

[2] P.K. Agarwal and J. Matousek, "Ray Shooting and Parametric Search," *SIAM J. Computing*, vol. 22, no. 4, pp. 794-806, 1993.

[3] P.K. Agarwal and J. Matousek, "On Range Searching with Semi-Algebraic Sets," *Discrete Computer Geometry*, vol. 11, pp. 393-418, 1994.

[4] P.K. Agarwal and M. Sharir, "Applications of a New Partition Scheme," *Discrete Computer Geometry*, vol. 9, pp. 11-38, 1993.

[5] B. Chazelle, H. Edelsbrunner, L.J. Guibas, R. Pollack, R. Seidel, M. Sharir, and J. Snoeyink, "Counting and Cutting Cycles of Lines and Rods in Space," *Computer Geometry Theory Applications*, vol. 1, pp. 305-323, 1992.

[6] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, Mass: The MIT Press, 1990.

[7] M. de Berg, *Ray Shooting, Depth Orders and Hidden Surface Removal*, Lecture Notes in Computer Science, vol. 703, Berlin: Springer-Verlag, 1993.

[8] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld, "Efficient Ray Shooting and Hidden Surface Removal," *Algorithmica*, vol. 12, pp. 30-53, 1994.

[9] M. de Berg, M. Overmars, and O. Schwarzkopf, "Computing and Verifying Depth Orders," *SIAM J. Computing*, vol. 23, pp. 437-446, 1994.

[10] D.P. Dobkin and M.J. Laszlo, "Primitives for the Manipulation of Three-Dimensional Subdivisions," *Algorithmica*, vol. 4, pp. 3-32, 1989.

[11] H. Edelsbrunner, "An Acyclicity Theorem for Cell Complexes in *d* Dimensions," *Combinatorica*, vol. 10, pp. 251-260, 1990.

[12] T. Frühauf, "Raycasting of Nonregularly Structured Volume Data," *Computer Graphics Forum (Eurographics '94)*, vol. 13, no. 3, 1994.

[13] H. Fuchs, Z.M. Kedem, and B. Naylor, "On Visible Surface Generation by A Priori Tree Structures," *Computer Graphics*, vol. 14, no. 3, pp. 124-133, 1980. *Proc. SIGGRAPH '80*.

[14] M.P. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 35-40, Nov. 1990.

[15] C. Giertsen, "Volume Visualization of Sparse Irregular Meshes," *IEEE Computer Graphics and Applications*, vol. 12, no. 2, pp. 40-48, Mar. 1992.

[16] S. Hertel and K. Mehlhorn, "Fast Triangulation of the Plane with Respect to Simple Polygons," *Information and Control*, vol. 64, nos. 1-3, pp. 52-76, Jan. 1985.

[17] A.E. Kaufman et. al., "Research Issues in Volume Visualization," *IEEE Computer Graphics and Applications*, vol. 14, no. 2, pp. 63-67, Mar. 1994.

[18] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29-37, May 1988.

[19] K-L. Ma, "Parallel Volume Rendering for Unstructured-Grid Data on Distributed Memory Machines," *Proc. IEEE/ACM Parallel Rendering Symp. '95*, pp. 23-30, 1995.

[20] X. Mao, L. Hong, and A. Kaufman, "Splatting of Curvilinear Grids," *Proc. IEEE Visualization '95*, pp. 61-68, 1995.

[21] N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99-108, June 1995.

[22] N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 27-33, Nov. 1990.

[23] J.S.B. Mitchell, D.M. Mount, and S. Suri, "Query-Sensitive Ray Shooting," *Proc. 10th Ann. ACM Symp. Computational Geometry*, pp. 359-368, June 1994. (To appear, *Int'l J. Computational Geometry & Applications*)

[24] G.M. Nielson, "Scattered Data Modeling," *IEEE Computer Graphics and Applications*, vol. 13, no. 1, pp. 60-78, Jan. 1993.

[25] G.M. Nielson, T.A. Foley, B. Hamann, and D. Lane, "Visualizing and Modeling Scattered Multivariate Data," *IEEE Computer Graphics and Applications*, vol. 11, no. 3, pp. 47-55, May 1991.

[26] M.S. Paterson and F.F. Yao, "Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modeling," *Discrete Computational Geometry*, vol. 5, pp. 485-503, 1990.

[27] M. Pellegrini, "Ray Shooting on Triangles in 3-Space," *Algorithmica*, vol. 9, pp. 471-494, 1993.

[28] C.E. Prakash, "Parallel Voxelization Algorithms for Volume Rendering of Unstructured Grids," PhD thesis, Supercomputer Centre, Indian Inst. of Science, 1996.

[29] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985.

[30] S. Ramamoorthy and J. Wilhelms, "An Analysis of Approaches to Ray-Tracing Curvilinear Grids," Technical Report UCSC-CRL-92-07, Univ. of California, Santa Cruz, 1992.

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

[32] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*. Upper Saddle River, N.J.: Prentice Hall, 1996.

[33] M. Sharir and P.K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*. New York: Cambridge Univ. Press, 1995.

[34] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 63-70, Nov. 1990.

[35] C. Silva, "Parallel Volume Rendering of Irregular Grids," PhD thesis, Dept. of Computer Science, State Univ. of New York at Stony Brook, 1996.

[36] C. Silva, J. Mitchell, and A. Kaufman, "Fast Rendering of Irregular Grids," *Proc. IEEE-ACM Volume Visualization Symp.*, pp. 15-22, Nov. 1996.

[37] D. Speray and S. Kennon, "Volume Probes: Interactive Data Exploration on Arbitrary Grids," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 5-12, Nov. 1990.

[38] C. Stein, B. Becker, and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Proc. Symp. Volume Visualization*, pp. 83-90, Oct. 1994.

[39] S. Uselton, "Volume Rendering for Computational Fluid Dynamics: Initial Results," Technical Report RNR-91-026, NASA Ames Research Center, 1991.

[40] A. Van Gelder and J. Wilhelms, "Rapid Exploration of Curvilinear Grids Using Direct Volume Rendering," *Proc. IEEE Visualization '93*, pp. 70-77, 1993.

[41] J. Wilhelms, "Pursuing Interactive Visualization of Irregular Grids," *Visual Computer*, vol. 9, no. 8, 1993.

[42] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri, "Direct Volume Rendering of Curvilinear Volumes," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 41-47, Nov. 1990.

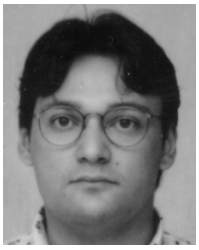
[43] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics (SIGGRAPH '91 Proc.)*, vol. 25, pp. 275-284, July 1991.

[44] J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs. "Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids," *Proc. IEEE Visualization '96*, pp. 57-64, 1996.

[45] P. Williams, "Visibility Ordering Meshed Polyhedra," *ACM Trans. Graphics*, vol. 11, no. 2, 1992.

[46] R. Yagel, "Volume Rendering Polyhedral Grids by Incremental Slicing," Technical Report OSU-CISRC-10/93-TR35, 1993.

[47] R. Yagel, D. Reed, A. Law, P-W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. IEEE-ACM Volume Visualization Symp.*, pp. 55-62, Nov. 1996.



Claudio T. Silva received a BS (1990) in mathematics from the Federal University of Ceara (UFC, Brazil), and an MS (1993) and PhD (1996) in computer science from the State University of New York at Stony Brook. He is currently a research associate in the Department of Applied Mathematics and Statistics, State University of New York at Stony Brook. Dr. Silva won a U.S. National Science Foundation CISE Postdoctoral Research Associateship and is partially funded by Sandia National Labs. His recent work focuses on the visualization of large datasets, and includes the development of a large interactive parallel volume rendering system; algorithms for volume rendering irregular grids; and simplification methods for terrain, 3D surface, and volumetric models. His main research interests are in computer graphics, scientific visualization, and high performance computing.

Joseph S.B. Mitchell received a BS (1981) in physics and applied mathematics, and an MS (1981) in mathematics from Carnegie-Mellon University. He received a PhD (1986) in operations research from Stanford University, while on a Howard Hughes doctoral fellowship and serving on the technical staff at Hughes Research Labs. From 1986 to 1991, Dr. Mitchell served on the faculty of Cornell University. In 1991, he joined the faculty of the State University of New York at Stony Brook, where he is currently an associate professor of applied mathematics and statistics and a research professor of computer science. Dr. Mitchell has received various research awards (U.S. National Science Foundation Presidential Young Investigator, Fulbright Scholar) and numerous teaching awards, including the President's and Chancellor's Awards for Excellence in Teaching. His primary research area is computational geometry applied to problems in computer graphics, visualization, manufacturing, and geographic information systems.