Fast Rendering of Irregular Grids

Cláudio T. Silva Joseph S. B. Mitchell Arie E. Kaufman

State University of New York at Stony Brook

Stony Brook, NY 11794

Abstract

We propose a fast algorithm for rendering general irregular grids. Our method uses a sweep-plane approach to accelerate ray casting, and can handle 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.

In this paper, we establish the practicality of our method through experimental results based on our implementation, and we also provide theoretical results, both lower and upper bounds, on the complexity of ray casting of irregular grids.

1 Introduction

Volume rendering methods are used to visualize scalar and vector fields by modeling volume as "cloud-like" cells composed of semi-transparent material that emits its own light, partially transmits light from other cells, and absorbs some incoming light [31, 17]. 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 [26]. 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* (non-curvilinear) irregular grids, in which there is no implicit connectivity information. Furthermore, in some applications *disconnected* grids arise.

In this paper, we present and analyze a new method for rendering general meshes, which include unstructured, possibly disconnected, irregular grids. Our method is based on ray casting and employs a sweep-plane approach, as proposed by Giertsen [14], but introduces several new ideas that permit a faster execution, both in theory and in practice.

Definitions and Terminology

A *polyhedron* is a closed subset of $\Re^3$ whose boundary consists of a finite collection of convex polygons (*2-faces*, or *facets*) whose union is a connected 2-manifold. The *edges* (*1-faces*) and *vertices* (*0-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 *3-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.

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 $\rho_{i,j}$. (Then, the number of cells intersected by $\rho_{i,j}$ is between $k_{i,j}/2$ and $k_{i,j}$.) 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*.

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 [30]. 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 [11] 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. [16] 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 [13] has proposed a preprocessing step that identifies the

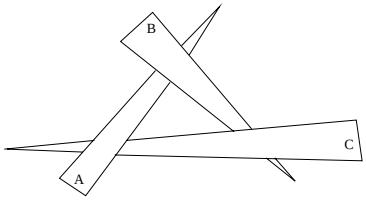

Figure 1: *3 triangles that have no depth ordering.*

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 $O(n^{4+\epsilon})$ preprocessing and storage [2, 4, 8, 20], or answer queries in worst-case time $O(n^{3/4})$, using a data structure that is essentially linear in n [3, 24]. 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 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. [18] 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 [23]), but with extra care needed to keep the space complexity low, while achieving the provably good query time.

Uselton [28] proposed the use a Z-buffer to speed up FirstCell; Ramamoorthy and Wilhelms [22] point out that this approach is only effective 95% of the time. They also point out that 35% of the time is spent checking for exit cells and 10% for entry cells. Ma [15] 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 [17, 32, 25, 27], 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, as shown in Figure 1. Computing and verifying depth orders is possible in $O(n^{4/3+\epsilon})$ time [1, 7, 9]. 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 [12, 19]. However, for some important classes of meshes (e.g., rectilinear meshes and Delaunay meshes [10]), it is known that a depth ordering always exists, with respect to any viewpoint. This structure has been exploited by Max et al. [17]. Williams [32] 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 [29], 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.

Two important references on rendering irregular grids have not yet been discussed here — Giertsen [14] and Yagel et al. [33]. We discuss Giertsen's method in the next section. For details on [33], we refer to their paper in these proceedings.

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.

Our Contribution

In this paper we propose a new 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 the 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 by Giertsen [14], 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 establish the practicality of our method through experimental results based on our implementation. We also discuss theoretical lower and upper bounds on the complexity of ray casting in irregular grids.

2 Sweep-Plane Approaches

A standard algorithmic paradigm in computational geometry is the "sweep" paradigm [21]. Commonly, a *sweep-line* is swept across the plane, or a *sweep-plane* is swept across 3-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.

Giertsen's Method

Giertsen's pioneering work [14] was the first step in optimizing ray casting by making use of coherency in order to speed up rendering.

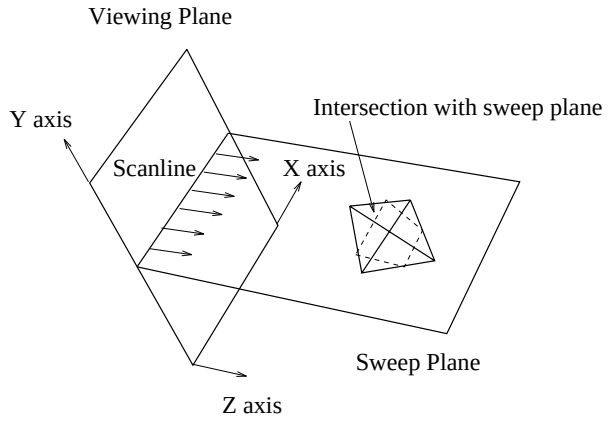

Figure 2: *A sweep-plane (perpendicular to the y-axis) used in sweeping 3-space.*

He performs a sweep of the mesh in 3-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 Figure 2. The algorithm consists of:

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 Tetrahedra List* (ATL) to empty. The ATL represents the sweep status; it maintains a list of the tetrahedra 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 ATL: Insert/delete, as appropriate, the tetrahedra incident on v. (Giertsen assumed that the tetrahedra are disjoint, so each v belongs to a single tetrahedron.)
   (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 3-space, Giertsen reduces the ray casting problem in 3-space to a 2-dimensional cell sorting problem.

Discussion

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 tetrahedral cells are all disjoint.) Second, the computationally expensive ray shooting in 3-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 being coarsed into a finite resolution buffer (the memory hashing buffer) for rendering, basically limiting the resolution of the rendering, and possibly creating an aliasing effect. Also, his memory scheme cannot be easily resolved by increasing the resolution of the buffer, as irregular grids have cell size variation of the order from 1:100,000, making it impractical to have a large enough buffer. In his paper, 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 form of collision 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 re-transforming a single copy, since floating point errors could accumulate.

3 Our 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 explore ray casting algorithms, as they have an inherent advantage for handling 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 *inter-scanline* and *inter-ray* 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 (e.g, no 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 subset of the mesh vertices.

A vertex v is *locally extremal* (or simply *extremal*, for short) if all of the edges incident to it lie in the (closed) halfspace above or below it (in y-coordinate). A simple (linear-time) pass through the data readily identifies the extremal vertices.

We initialize the event queue with the extremal 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 extremal 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 (non-extremal ones) will be inserted into and deleted from the event queue each time the sweep-plane hits a vertex of S.

The sweep algorithm proceeds in the usual way, processing events as they occur, as determined by the event queue and by the scanlines. 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; these are easily determined by checking the signs of the dot

products of edge vectors out of v with the vector representing the y-axis. Otherwise, the sweep-plane has encountered a scanline. And 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 encountered.

We remark here that, instead of doing a sort (in y) of all vertices of S at once, the algorithm is able to take advantage of the partial order information that is encoded in the mesh data structure. (In particular, if each edge is oriented in the $+y$ direction, the resulting directed graph is acyclic, defining a partial ordering of the vertices.) Further, by doing the sorting "on the fly", using the event queue, our algorithm can be run in a "lock step" mode that avoids having to sort and sweep over highly complex subdomains of the mesh. This is especially useful, as we see in the next section, if the slices that correspond to our actual scanlines are relatively simple, or the image resolution (pixel size) is large in comparison with some of the features of the dataset. (Such cases arise, for example, in some applications of scientific visualization on highly disparate datasets.)

3.2 Processing a Scanline

When the sweep-plane encounters a scanline, the current sweep status data structure gives us a "slice" through the mesh in which we must solve a two-dimensional ray casting problem. Let $\mathcal{S}$ denote the polygonal (planar) subdivision at the current scanline (i.e., $\mathcal{S}$ is the subdivision obtained by intersecting the sweep-plane with the mesh S.) In time linear in the size of $\mathcal{S}$, we can recover the subdivision $\mathcal{S}$ (both its geometry and its topology), just by stepping through the sweep status structure, and utilizing the local topology of the cells in the slice.

The two-dimensional problem is also solved using a sweep algorithm — now we sweep the plane with a sweep-line parallel to the z axis. Events now correspond to vertices of the planar subdivision $\mathcal{S}$. At the time that we construct $\mathcal{S}$, we identify those vertices in the slice that are locally extremal in $\mathcal{S}$ (i.e., those vertices that have edges only leftward in x or rightward incident on them.) These are inserted in the initial event queue. The *sweep-line status* is an ordered list, stored and maintained in a binary tree, of the edges of $\mathcal{S}$ crossed by the sweep-line. The sweep-line status is initially empty. Then, as we pass the sweep-line over $\mathcal{S}$, we update the sweep-line status and the event queue at each event when the sweep-line hits an extremal vertex, making insertions and deletions in the standard way. This is analogous to the Bentley-Ottmann sweep that is used for computing line segment intersections in the plane [21]. 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 (binary tree).

4 Analysis: Upper and Lower Bounds

We proceed now 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 $\Omega(k + c \log c)$ 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).*

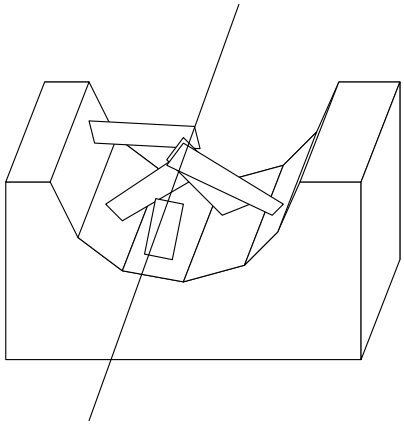

Figure 3: *Lower bound construction.*

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 Figure 3. 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_i's. 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. □

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 3-space, we have $O(n)$ vertex events, plus N "events" when we stop the sweep and process the 2-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 3-space.

For each scanline slice, we must perform a sweep as well, on the subdivision $\mathcal{S}$, 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 "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, for each scanline slice, requires overall time $O(\sum_{i,j} k_{i,j} + Nn \log n) = O(k + Nn \log n)$. Overall, then, we get $O(k + Nn \log n)$.*

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 $O(k + n \log n + n_s \log n)$.

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

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 3-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).

Consider the sweep of 3-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, if we conduct our sweep of 3-space carefully, then 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. The main idea is to exploit the topological coherence between slices, noting that the number of connected components 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 $\mathcal{S}$ 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 vertically ($+y$ direction) 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 3-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.)

*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.

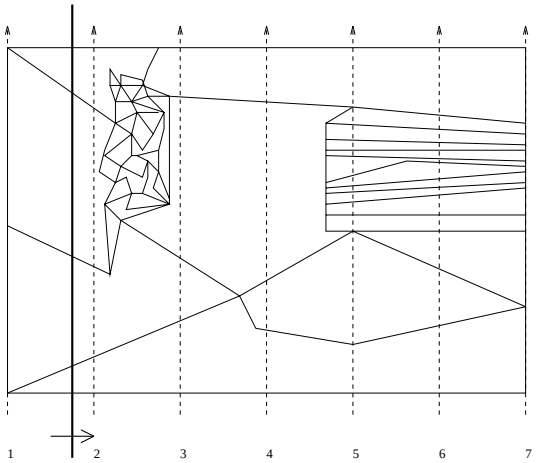

Figure 4: *Illustration of a sweep in one slice.*

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 a "lock step" algorithm (doing steps in alternation, between the two methods), to achieve asymptotically a complexity that is the minimum of the two. This scheme applies not just to the sweep in 3-space, but also to the sweeps in each slice.

As an illustration of how these methods can be quite useful, consider the situation in Figure 4, which, while drawn only in 2 dimensions, can depict the cases in 3-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 the 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 lock step (possibly in parallel, if a second processor is available), 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.

5 Experimental Results

We have implemented the main algorithm described in the previous sections. Our implementation handles general disconnected grids, and has most of the advantages of the complete algorithm described already, but we have not yet implemented the "lock step" idea (used

to avoid worst-case complexity in disparate data sets), and our code does not currently handle perspective projections. (The implementation of *perspective projection* will be done soon and is conceptually very simple, requiring only that the priority values in the queue be based on an appropriate dot product.) Further, in our initial implementation, we have assumed that cells of the mesh are tetrahedra (simplices). Our method does not require convex cells, even though they do make some of the implementation issues simpler.

The rendering algorithm consists of about 5,000 lines of C code. It is fairly naive in terms of optimization, so we expect that it can be further improved. An interesting aspect of the code is the way it cleanly handles geometric degeneracies. The major modules of the program include: *3D sweep*, which sweeps the vertices of the input mesh along a given direction, while maintaining two dynamic sweep status data structures — the active tetrahedra list (ATL) and the active edge list (AEL); *2D sweep*, which orders the 3D edge intersections, and is complemented with the code that incrementally depth sorts the segments along the current ray. We also have a graphics module that sets up the transformations and manages the other modules, and the transfer function and the optical integration (or simple shading) modules. We do not attempt to describe the implementation in detail, but we shall explain some of the most relevant issues.

Due to the large sizes of irregular grids, efficient data structures can substantially influence the performance of the implementation. For priority queues (we use two of them, one for the incremental 3D sweep sort, another for the 2D sweep), we use a simple *heap* implementation (the same code is shared for 3D/2D). Instead of performing the view-dependent $O(n)$ search for extremal vertices, we simply preprocess the external vertices of the grid and place them in the heap before starting the sweep (all the internal vertices are still inserted incrementally – see Figure 5 – in order to avoid the need for substantial extra storage). In order to keep the ATL and AEL, we need a dictionary data structure that allows efficient insertion/deletion. We have experimented using a hash table and binary trees. The hash tables performed much better than the binary trees in our examples, because of lower overhead, both in time and space.

During the 2D sweep, a binary tree stores the sweep status. Edges are inserted in depth order, and for rendering at the pixel locations, the binary tree is sent to the shader. The handling of the binary tree is tricky, since a consistent ordering of all the segments along each ray must be maintained as edges are inserted and deleted during the sweep. Due to degeneracies, geometric tests alone are not suficient to keep a consistent ordering; edges may have the same *geometrical* properties, but *topologically* they are different, which causes inconsistencies in the tree – for instance, an edge might be inserted along a certain binary tree path when its first endpoint is reached, but might not be found in the tree when its second endpoint is reached due to the insertion of another edge along that path, resulting in an inconsistent sweep-status state. This problem is solved by assigning a computational ordering, that is, explicitly using an ordering function that depends on the memory position of the edges (which are fixed for each scanplane), to break geometric "ties", therefore forcing a globally consistent ordering among edges.

Another place where degeneracies have to be avoided is during the final rendering. The problem arises because several vertices may lie on the same plane. This leads to intersections that may have non-closed and/or null primitives (e.g., a triangle with two coincident sides). The solution is to keep track of the current status of the priority queue and only perform the rendering once all the events with values lower than or equal to the current y-value (or x-value when in ray casting) have been processed. This solution is conceptually simple, correct and easy to implement.

Geomview, from the Geometry Center of the University of Minnesota, was instrumental in the development of our renderer, helping to create animations and visually debug our code. Without visual debugging it would have been virtually impossible to write this code.

Datasets

The code currently handles datasets composed of tetrahedral grids. The input format is analogous to the Geomview "off" file. 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 grids (including disconnected), 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 the well-known Blunt Fin and Liquid Oxygen Post datasets, originally in NASA Plot3D format. The Blunt Fin contains 40-by-32-by-32 data points (40,960 vertices), from which we create 187,395 tetrahedra by breaking each cell into 5 tetrahedra. Figure 7 depicts the decomposition used, and Figure 8 shows a running configuration of the algorithm. The Post dataset contains 38-by-76-by-38 data points (109,744 vertices) and 513,375 tetrahedra after conversion. We have generated several other artificial datasets for debugging purposes; in particular, we generated simple datasets that have disconnected components.

Memory Requirements

Our algorithm is very memory efficient. The dataset is stored as a collection of vertices and tetrahedra. Each tetrahedron only stores indices to its vertices, and a single flag that identifies the external faces (no topological information is saved at the tetrahedra). Each vertex contains, besides its position and scalar value, a flag, used during the algorithm for various purposes, and a list of the tetrahedra it belongs to. Because each tetrahedron contains four vertices, the overall increase in memory cost for the topological information is minimal.

Besides the input dataset, the only other memory consumption is in the priority queues, which are very small in practice. (For the Blunt Fin, the extra storage is below half a megabyte.) This low storage requirement is due to our incremental computations, which only touch a cross section of the dataset at a time. The overall memory consumption for rendering the Blunt Fin is about 8MB of memory total, of which over 95% is the dataset itself (about 36% is topology information). For the Post dataset, the storage requirement is a bit over 21MB, of which 97% is the dataset itself (about 35% is topology information).

Performance Analysis

Our primary system for measurements was a Sun UltraSparc-1. We present numbers for the tetrahedralized version of the Blunt Fin and Post datasets, described above. It is important to notice, that our rendering times will clearly be higher than algorithms that treat the either dataset as a curvilinear grid composed of hexahedral cells.

Reading the Blunt Fin dataset off a local disk takes 9.8 seconds on the UlraSparc. The Post dataset takes 27.32 seconds. Our ASCII input files require parsing; thus, processing time dominates, not the actual disk access time. (Our tetrahedralized Blunt Fin version has almost 6MB, and the Post has over 16MB.) The use of binary files would likely improve efficiency, but using ASCII files simplifies the manual creation of test samples. In a preprocessing phase, we recover the adjacency information of the grid, and separate the external vertices into a list (for the Blunt Fin, we classify 6,760 vertices

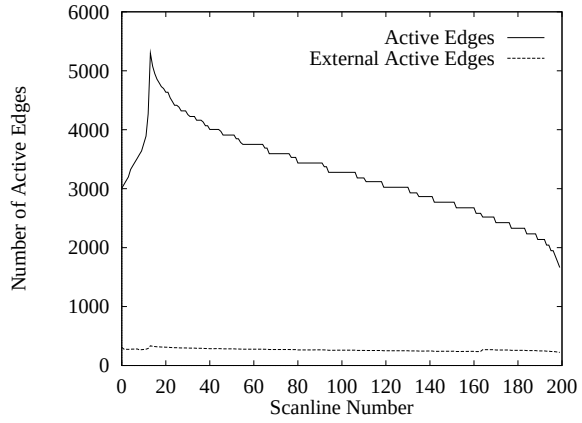

Figure 5: *Number of active edges as a function of the scanline: the active edges are the edges intersected at a given scanline by the current scanplane during the 3D sweep. The number of external active edges is also shown.*

as externals, for the Post, 13,840 vertices). The complete preprocessing takes 2.95 seconds for the Blunt Fin, and 8.48 seconds for the Post.

Rendering can be decomposed into several stages: 3D sweep, 2D sweep and 1D ray casting (including shading). All of them are embedded inside each other. The complexity of the 3D sweep is independent of the image size; it just depends on how many points need to be processed. For instance, without performing any rendering, just sweeping (the 3D sweep) the Blunt Fin grid takes about 3.5 seconds. During this time, the ATL and AEL are being updated at every event (the binary tree implementation takes over three times as long). The AEL is used during the 2D sweep for calculating and ordering the intersections for the final ray casting (see Figures 5 and 6).

Since the Blunt Fin projection is not square, it is not meaningful to give performance numbers on a square screen. Instead we give numbers for a 527-by-200 screen (105,400 pixels), which matches the aspect ratio of the Blunt Fin for the direction in which we are looking. Figure 5 contains the number of active edges for each scanline. It is easy to see that, because the structure of the grid is irregular, the number of edges varies quite a bit. For the Post, we used a 300-by-300 screen (90,000 pixels).

Rendering a scanline involves computing the intersection points, sorting them along the direction of the scanline, and then performing a 1D sweep (or sort) along each ray incrementally (which basically involves processing events and shading). Figure 6 shows the rendering times for the 2D sweep, for each scanline. The performance numbers indicate: the time to process a given scanline is directly correlated to the number of active edges on that plane; the cost per scanline varies depending on the complexity of the plane being rendered; (and most important for future optimization) the event handling time dominates the total time spent per scanline.

The event handling time is clearly the bottleneck of the rendering speed. This was puzzling at first, specially because it is just performing a sweep of a few thousand vertices (less than 5,500). In the 3D sweep, we handle over 40,000 vertices in about 3.5 seconds. Profiling the code showed that "CompareEdge" (a function that tells which of two edges is closer to the screen) is called over 68 million times, consuming over 40% of the *overall* rendering time. Further study shows that the reason for such a high number of calls to "CompareEdge" is related to the depth of the binary tree used to save the ordering. Because the Blunt Fin comes from a curvilinear grid, it has lots of vertices that lie (degenerately) on common planes, which causes extremely bad behavior in our binary tree sorting. This indicates that we can potentially obtain a dramatic improvement in performance, just by changing the data structure used (e.g., by employing a standard 2-3 tree or a Red-Black tree [6]). Another reason the 2D sweep is taking so long is the fact that there is a scanline component on its rendering time. As discussed later, the most time consuming parts of it can be eliminated by making incremental changes to the depth sorting on the segments.

Performance Comparisons

The total rendering time of our algorithm is 70 seconds for a 190,000 tetrahedra cell complex (the Blunt Fin), for a 527-by-200 image with almost complete pixel coverage (see Figure 9 – the picture was actually padded with a black frame after rendering for printing purposes). For the Blunt Fin, the performance of our current code is 373μs (microseconds) per tetrahedron, and 664μs per pixel. For the Post, a 500,000 tetrahedra cell complex, it takes 145 seconds (see Figure 10) to render a 300-by-300 image.

The most recent report on an irregular grid ray caster is Ma [15], 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, always starting with two nodes. With two nodes, for the *Cube*, he reports taking 2,415 seconds (2234 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 1593 (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. All his performance numbers reflect the use of 2 processors. Giertsen [14] 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 meaningful comparisons, nevertheless 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. [33] reports rendering the Blunt Fin, using an SGI with a Reality Engine^2 in just over 9 seconds, using a total of 21MB of RAM, using 50 "slicing" planes; with 100 planes, he reports the cost increases to 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 Post, it takes just over 20 seconds, using about 57MB RAM.

Optimizations

There are at least a couple of directions for optimization of the current code that may make it even more competitive. First, improvement in the data structures for keeping the sorted rays should lower the cost of using "CompareEdge". Second, at this time, we are starting the 2D sorting process over for every scanplane, not using the previously sorted information.

6 Conclusions and Future Work

In this paper we propose a new algorithm for rendering irregular grids. Our algorithm is carefully tailored to exploit spatial coherence even if the image resolution differs substantially from the object space resolution. We have also discussed some of the theoretical upper and lower bounds on ray casting approaches.

We have reported timing results showing that our method compares favorably with other ray casting schemes, and is, in fact, a couple orders of magnitude faster than other published ray casting results. Another advantage of our method is the fact that it is

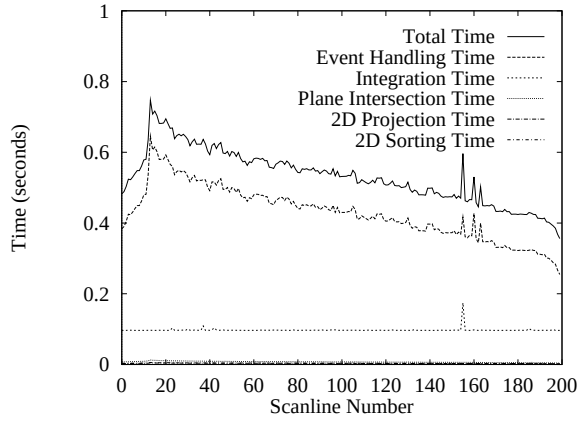

Figure 6: *Total rendering time as a function of the scanline: the intersection time is the time it takes to calculate the 2D point of intersection of the active edges with the scanplane; the event handling time is the time to process every active edge after sorting, including the 1D ray sorting necessary for integration. Note that event handling time dominates the total cost of a scanline.*

very memory efficient, making it suitable for use with very large datasets.

It is difficult to compare our method with hardware-based techniques (e.g., [33]), 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). Also, 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.

An interesting possible extension of our work would be to investigate issues involving out-of-core operation. The spatial locality of our memory accesses indicates that we should be able to employ *pre-fetching* techniques to achieve fast rendering when the irregular grids are much larger than memory. Also, our method is a natural candidate for parallelization.

Acknowledgements

We thank Ashish Tiwari for implementing the 3D sweep code, as part of a graphics course project. Juliana Freire helped with our rendering code. Brian Wylie helped with the preparation of the paper for publication. Dirk Bartz, Pat Crossno, George Davidson and Dino Pavlakos provided useful criticism on the paper. Blunt Fin and Post datasets courtesy of NASA. C. Silva is partially supported by CNPq-Brazil under a Ph.D. fellowship, by Sandia National Labs, and by the National Science Foundation (NSF), grant CDA-9626370. J. Mitchell is partially supported by NSF (CCR-9504192) and by Hughes Aircraft and Boeing. A. Kaufman is partially supported by NSF (CCR-9205047, DCA 9303181 and MPI-9527694) and by the Dept. of Energy under the PICS grant.

References

[1] Agarwal, P., M. Katz, M. Sharir. Computing depth orders and related problems. In *Proc. 4th Scand. Workshop Algorithm Theory*, pages 1–12, 1994.

[2] Agarwal, P. and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.

[3] Agarwal, P. and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

[4] Agarwal, P. and M. Sharir. Applications of a new partition scheme. *Discrete Comput. Geom.*, 9:11–38, 1993.

[5] Chazelle, B., H. Edelsbrunner, L. Guibas, R. Pollack, R. Seidel, M. Sharir, J. Snoeyink. Counting and cutting cycles of lines and rods in space. *Comput. Geom. Theory Appl.*, 1:305–323, 1992.

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

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

[8] de Berg, M,, D. Halperin, M. Overmars, J. Snoeyink, M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994.

[9] de Berg, M., M. Overmars, O. Schwarzkopf. Computing and verifying depth orders. *SIAM J. Comput.*, 23:437–446, 1994.

[10] Edelsbrunner, H. An acyclicity theorem for cell complexes in d dimensions. *Combinatorica*, 10:251–260, 1990.

[11] Fruhauf, T. Raycasting of nonregularly structured volume data. *Computer Graphics Forum (Eurographics '94)*, 13(3):294–303, 1994.

[12] Fuchs, H., Z. Kedem, B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[13] Garrity, M. Ray tracing irregular volume data. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:35–40, November 1990.

[14] Giertsen, C. Volume visualization of sparse irregular meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[15] Ma, K-L. Parallel volume rendering for unstructured-grid data on distributed memory machines. In *IEEE/ACM Parallel Rendering Symposium '95*, pages 23–30, 1995.

[16] Mao, X., L. Hong, A. Kaufman. Splatting of curvilinear grids. In *IEEE Visualization '95*, pages 61–68, 1995.

[17] Max, N., P. Hanrahan, R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:27–33, November 1990.

[18] Mitchell, J., D. Mount, S. Suri. Query-sensitive ray shooting. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 359–368, 1994.

[19] Paterson, M. and F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[20] Pellegrini, M. Ray shooting on triangles in 3-space. *Algorithmica*, 9:471–494, 1993.

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

[22] Ramamoorthy S. and J. Wilhelms. An analysis of approaches to ray-tracing curvilinear grids. Report UCSC-CRL-92-07, U. of California, Santa Cruz, 1992.

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

[24] Sharir, M. and P. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, 1995.

[25] Shirley, P. and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:63–70, November 1990.

[26] Speray, D. and S. Kennon. Volume probes: Interactive data exploration on arbitrary grids. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:5–12, November 1990.

[27] Stein, C., B. Becker, N. Max. Sorting and hardware assisted rendering for volume visualization. In *ACM Volume Visualization Symposium '94*, pages 83–89, 1994.

[28] Uselton, S. Volume rendering for computational fluid dynamics: Initial results. Tech Report RNR-91-026, Nasa Ames Research Center, 1991.

[29] Van Gelder, A. and J. Wilhelms. Rapid Exploration of Curvilinear Grids Using Direct Volume Rendering. In *IEEE Visualization '93*, pages 70–77, 1993.

[30] Wilhelms, J., J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. Direct volume rendering of curvilinear volumes. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:41–47, November 1990.

[31] Wilhelms, J. and A. Van Gelder. A coherent projection approach for direct volume rendering. *Comput. Graph.*, 25:275–284, 1991. Proc. SIGGRAPH '91.

[32] Williams, P. Visibility ordering meshed polyhedra. *ACM Trans. Graph.*, 11:103–126, 1992.

[33] Yagel, R., D. Reed, A. Law, P-W. Shih, N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Volume Visualization Workshop*, these proceedings, 1996.

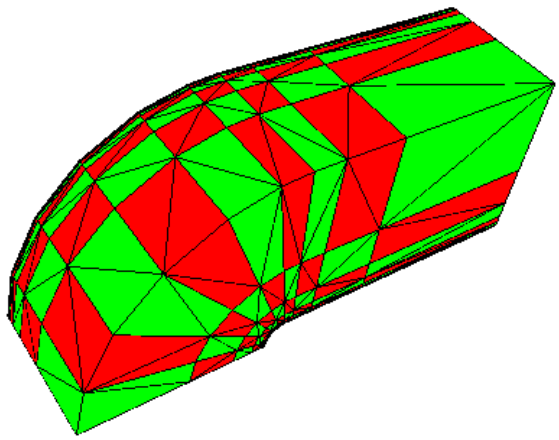

Figure 7: *Outside faces of a lower resolution version of the Blunt Fin are shown to demonstrate the tetrahedralization process. Red and green cells have to be tetrahedralized in opposite direction to allow for correct matching between cells.*

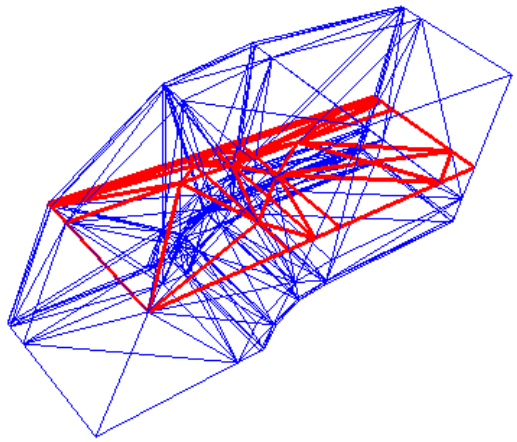

Figure 8: *A typical configuration during the sweep is shown in red. (A lower resolution version of the Blunt Fin is used to avoid excessive cluttering.)*

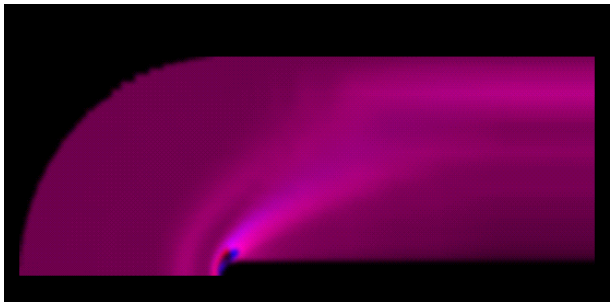

Figure 9: *A volume rendering of the Blunt Fin dataset generated with our method.*

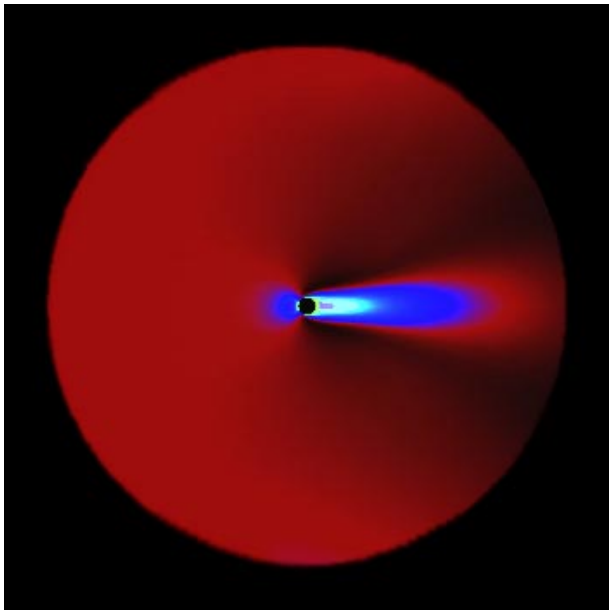

Figure 10: *A volume rendering of the Liquid Oxygen Post dataset generated with our method.*