ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering

Ricardo Farias[1,*] Joseph S. B. Mitchell[1,†] Cláudio T. Silva[2,‡]

[1]University at Stony Brook [2]AT&T Labs-Research

Abstract

We present a simple new algorithm that performs fast and memory-efficient cell projection for (exact) rendering of unstructured datasets. The main idea of the "ZSweep" algorithm is very simple; it is based on sweeping the data with a plane parallel to the viewing plane, in order of increasing *z*, projecting the faces of cells that are incident to vertices as they are encountered by the sweep plane. The efficiency arises from the fact that the algorithm exploits the implicit (approximate) global ordering that the *z*-ordering of the vertices induces on the cells that are incident on them. The algorithm projects cells by projecting each of their faces, with special care taken to avoid double projection of internal faces and to assure correctness in the projection order. The contribution for each pixel is computed in stages, during the sweep, using a short list of ordered face intersections, which is known to be correct and complete at the instant that each stage of the computation is completed.

The ZSweep algorithm is simple enough to be readily adaptable to general (non-tetrahedral) cell formats. It is memory efficient, since its auxiliary data structures have only to store partial information taken from a small number of "slices" of the dataset. We also introduce a simple technique of data sparsification, which may be of interest in its own right.

Our implementation is hardware-independent and handles datasets containing tetrahedral and/or hexahedral cells. We give experimental evidence that our method is competitive, up to 5 times faster than the best previously-known exact algorithms that use comparable amounts of memory, while using much less memory than ray-casting.

1 Introduction

We study the problem of rendering unstructured volumetric data. In this paper, our focus is on direct volume rendering, a term used to define a particular set of rendering techniques which avoids generating intermediary (surface) representations of the volume data. Instead, the scalar field is generally modeled as a cloud-like material, and rendered by computing a set of lighting equations. In general, while evaluating the volume rendering equations [12], it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells intersected by the ray, so that the overall integral in the rendering equation can be evaluated.

Our main contribution in this paper is a very fast and memory-efficient algorithm for rendering unstructured grids. In particular, we propose a novel solution to the computation of the sorted order of the cells intersected by all the rays in a given image. The algorithm is simple and is based on the sweep paradigm. The algorithm has been fully implemented; our experiments show that we obtain significant improvements in speed, by up to a factor of 5 over the prior state-of-the-art. Further, with some new optimizations we introduce, based on an idea of "data sparsification" in storing the main dataset, we improve on the memory usage of prior sweep-based algorithms.

Related Work

Early work in adapting ray tracing techniques for rendering unstructured grids is described in Garrity [8] and Uselton [19]. These techniques are "exact", in the sense that in principle (i.e., without accounting for degeneracies), for each pixel, a correct cell stabbing order is computed. Unfortunately, these techniques tend to be relatively slow, despite the optimizations proposed in the respective articles.

Shirley and Tuchman [15] show how to exploit polygon-based graphics hardware for computing the volume rendering equations for one tetrahedron. By using the MPVO technique of Williams [20] to visibility sort the cells in back-to-front order, they propose a "projective" method for rendering unstructured grids. This particular projective technique had several limitations, including the fact that the MPVO technique of Williams is only able to generate "correct" visibility order for certain types of datasets, and the actual approximation proposed in [15] generates visual artifacts. Improving on the Shirley and Tuchman technique, Stein et al. [18] propose techniques to explore texture mapping to improve the visual quality, and an $O(n^2)$ sorting algorithm which is able to compute correct visibility order for general acyclic unstructured grids. Their work is further improved by Williams et al [21], Silva et al [17], and Comba et al [7], leading to almost linear-time (in practice) "exact" visibility sorting techniques. Max et al [13] proposed a different sorting technique based on "power" sorting. This technique is more restricted than the MPVO-sorted grids, in fact, it is only guaranteed to produce correct sorting results for Delaunay triangulations. Despite its shortcomings, this technique is quite useful, and has been used extensively in practice, leading to excellent rendering times (see Cignoni et al [6, 5, 4], and Wittenbrink [22]). Recently, Cignoni and De Floriani [3] have proposed a more general extension of power sorting, but provide little experimental results.

[*]Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, Stony Brook, NY 11794-3600; rfarias@ams.sunysb.edu.

[†]Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, Stony Brook, NY 11794-3600; jsbm@ams.sunysb.edu.

[‡]AT&T Labs-Research, 180 Park Ave., PO Box 971, Florham Park, NJ 07932; csilva@research.att.com.

Since "projective" methods work by projecting, in visibility order, the polyhedral cells of a mesh onto the image plane, and incrementally compositing the cell's color and opacity into the final image, it is crucial to these methods to compute a correct visibility ordering of the cells. Strictly speaking, the projective methods that do not use a provably correct visibility order algorithm are not exact, since incorrect projection leads to wrong images. Because these techniques render each tetrahedron one at a time, it is not possible to correctly handle grids that contain cycles. (Note that this is not a problem for ZSweep, and in general for ray casting based techniques.)

The plane sweep paradigm, which is based on processing geometric entities in an order determined by passing a line or a plane over the data, has been used widely in computational geometry for the design of efficient algorithms; see [14]. It has also been used in devising efficient volume rendering algorithms.

Giersten [9] pioneered the use of sweep algorithms in volume rendering. His sweep algorithm is based on a sweep-plane that is orthogonal to the viewing plane (in particular, orthogonal to the y-axis). Events in the sweep are determined by vertices in the dataset and by values of y that correspond to the pixel rows. When the sweep plane passes over a vertex, an "Active Cell List" (ACL) is updated accordingly, so that it stores the set of cells currently intersected by the sweep-plane. When the sweep plane reaches a y-value that defines the next row of pixels, the current ACL is used to process that row, casting rays, corresponding to the values of x that determine the pixels in the row, through a regular grid (hash table) that stores the elements of the ACL. This method has three major advantages: It is not necessary to store explicitly the connectivity between the cells; it replaces the relatively expensive operation of 3D ray-casting with a simpler 2D regular grid ray-casting, and it exploits coherence of the data between scanlines. The main disadvantage of the method is that the regular grid utilized in the 2D ray-casting may cause a loss of resolution in the rendering, while leading to possible aliasing effects (both spatial and temporal).

Following the same basic idea of sweeping the data, Yagel presented a different approach to rendering unstructured grid data, which also allowed some further speed-up using hardware support, as he shows in [23]. His sweep algorithm is based (as is ours) on a sweep with a plane parallel to the viewing plane. Just like Giertsen's algorithm, Yagel's does not need to compute and keep explicit cell adjacency information, allowing it to be memory-efficient in its basic data structures. Graphics hardware can be used to accumulate the contributions of each slice to the final image. The main drawback of this algorithm is its memory consumption, which can be substantial, since it must store the polygons resulting from each slice. Also, its accelerated version requires graphics hardware support.

The Lazy Sweep algorithm [16] is the most recent algorithm based on the sweeping paradigm. It was shown to be faster and more memory efficient than its predecessors. Besides the array of vertices and the array of cells, the only other adjacency information used is a list (the "vertex use set"), for each vertex, that keeps the indices of all cells that are incident on the vertex.

We also briefly discuss the ray-casting algorithm of Bunyk et al. [1], which we use in our experimental comparisons. This is a very fast algorithm, but it requires a lot memory. It's basic idea is as follows:

- In preprocessing, identify all cells and faces that touch each vertex and identify all boundary faces.
- For the given new rotation angle, rotate all vertices.
  - By projecting all boundary faces on the screen, create for each pixel an ordered list of the boundary faces that a ray cast through the pixel crosses as it enters and exits the volume.
  - For each pixel, starting from the first boundary face intersected, use the cell adjacency information to find the next face intersected by the ray. (Each interior face points to its two neighboring cells, allowing one to go easily from cell to cell while computing the contribution of each cell.) Use the ordered list of boundary faces to determine the entry and exit points of the ray as it passes into and out of the volume.

One difficulty with the implementation of the algorithm is that when a ray exactly hits a vertex or an edge of the dataset, it may have difficulty resolving which the next cell is, potentially leaving the corresponding pixel *un*rendered. However, in most cases the implementation produces high-quality images and does so very quickly, making it a reasonable choice for our experimental comparisons.

Recently, Hong and Kaufman [11, 10] proposed a very fast ray casting technique for curvilinear grids. Their work is similar in some ways to [1], but optimized for curvilinear grids, which makes it faster and use far less memory than [1]. Their implementation has not been shown to work on unstructured datasets.

Finally, we mention that the new algorithm presented in this paper, is also related in some ways with the *A-buffer* [2] approach, optimized with the use of the order of the vertices to assure a *quasi-order* projection of the faces. The work the *A-buffer* has to perform to order the intersections between the ray cast and the faces projected is very small.

2 The ZSweep Algorithm

Our ZSweep algorithm is designed with the intent of combining accuracy and simplicity with speed and memory efficiency, building on the success of prior sweep approaches.

The algorithm is a simple sweep with a plane, Π, parallel to the viewing plane, in order of increasing z-coordinate. (This is the only similarity with Yagel et al's algorithm.) *Events* occur when Π encounters a vertex v, at which point we project the faces of cells that are incident to v and lie beyond v (in z-coordinate).

In order to facilitate our further discussion of the algorithm, we introduce some notation:

- The *vertex use set*, $U(v)$, associated with vertex v is a list of all cells that are incident on v.
- We say that vertex v is a *swept vertex* if it has already been swept over by Π (its z-coordinate, v_z, is less that the current sweep value, z.
- A face f is a *swept face* if at least one vertex of f is a swept vertex.

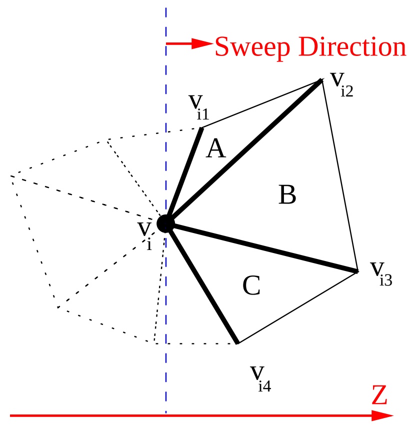

Figure 1: When the sweep plane Π encounters vertex v_i, the cells A, B, and C are first encountered, so the (highlighted) faces (v_i, v_{i_1}), (v_i, v_{i_2}), (v_i, v_{i_3}), and (v_i, v_{i_4}) are projected.

- A cell c is a *swept cell* if all of its faces are swept. Since our algorithm maintains the swept status explicitly only for vertices, we use the following observations to determine the swept status of a cell: A tetrahedral cell is swept if and only if (at least) two of its vertices have been swept; a hexahedral cell is swept if and only if (at least) five of its vertices have been swept.

We now describe the steps of the algorithm in greater detail.

The first step is to sort the vertices by z-coordinate into an *event list*; this determines the order of the events. We use a heap to efficiently sort the vertices. The heap keeps only indices to the vertex array. Using a heap, we can save some memory over quicksort, and, more importantly, it will allow us to adapt our algorithm to dynamic situations in which new vertices may be inserted.

Optionally, one can choose to sort and store in the event list only the *boundary* vertices (on the boundary of the dataset), and then to insert interior vertices into the event list only as they are discovered during the sweep algorithm. This optimization has the potential to save some memory; however, we have reported our results based on not using this option, as we have found that the event list is responsible for only about 4% of the total memory required for the vertices and cells (including the use sets).

The main loop of the algorithm is the sweep in the z-direction, which is performed simply by stepping through the event list. When the ith vertex, v_i, of the event list is encountered, we *project* [*] each face f that is incident on v_i for which v_i is the vertex having minimal z-coordinate. (Necessarily, such faces f have not been swept.) The faces to project are readily determined by examining the use set of v_i. Refer to Figure 1 for an illustration in two dimensions.

In order to perform face projection, we use a very fast scan conversion for triangles, which not only determines which pixels lie in the projection, but also determines the z-coordinate (depth) of each point of the (unprojected) triangle and computes the interpolated value (via bi-linear interpolation) for the scalar field data.

[*] Our face projection is different from the ones used in projective methods, such as the Shirley and Tuchman approach. During face projection, we simply compute the intersection of the ray emanating from each pixel, and store their z-value (and other auxiliary information). The actual lighting calculations are deferred to a later phase.

To guarantee accuracy in the rendering algorithm, it is important to make certain that the projection of the faces is done in a correct order for each pixel. The order in which faces are projected in our ZSweep is according to the z-coordinate of the first vertex encountered of the face. This order is *not*, however, sufficient to guarantee that faces are projected automatically in correct depth order for every pixel. For example, in Figure 1, faces (v_i, v_{i_1}) and (v_i, v_{i_2}) are each projected when we encounter v_i. While a local analysis of the faces at v_i would permit us to project (v_i, v_{i_1}) *before* (v_i, v_{i_2}), we would have to project also face (v_{i_1}, v_{i_2}) before (v_i, v_{i_2}) in order to have those pixels in the projection of (v_{i_1}, v_{i_2}) have the correct ordering of projected faces. We do not, however, project face (v_{i_1}, v_{i_2}) until Π reaches the vertex v_{i_1}. While in two dimensions it is possible to project faces (edges of triangles) in an order that is consistent in z, in three dimensions it is well known that the precedence relation induced by depth ordering can have cycles. (Even three triangles in space can form a cycle.)

Thus, our ZSweep algorithm keeps for each pixel a z-order list of intersections, projected on that pixel. Each time that a face is projected on the screen, we insert into the list, for each pixel under the projection, an element (with the z value of the intersection as well as the interpolated scalar value) into the corresponding pixel list. The insertions must preserve the correct z-ordering of each pixel list. If insertions were being made in “random” order, it would be important to store each pixel list in an efficient data structure (e.g., heap or balanced binary tree) to permit efficient insertion. However, the order in which we project faces in our ZSweep is such that the face we are projecting will most likely lie at the end of the list, or be “very close” to it. Thus, we have implemented a doubly-linked list structure for the pixel lists, and we perform insertion from the end (larger z-coordinate) towards the beginning of the list. Some experiments showed us that 70% of the insertions are performed at the end of the list. Also, about 12% of the insertions occur in the next-to-last position, 17% in the position before that, with less than 1% of insertions occuring more than two positions before the end. Thus, doing a simple insertion, starting at the end of the list, results in a significant time savings, since the ordering determined by the ZSweep face projection order is already so close to the depth order in most cases.

While the pixel lists allow us to ensure that each pixel gets the correct ordering of all projected faces, it comes at the cost of potentially increasing the memory requirement substantially. In order to avoid this, we use a technique we call *delayed compositing* to flush the pixel lists on a regular basis. In particular, at any given stage of the sweep we have a *z-target*, which represents the value of z at which we will next stop the sweep momentarily and compose the values that are in the pixel lists; the sweep continues beyond the z-target, after setting a new z-target appropriately.

Initially, we define the z-target to be the maximum z-coordinate among the vertices adjacent to the first vertex, v_0, encountered by Π. When the sweep reaches the z-target (say, corresponding to vertex v), we compose, in order, the entries of the pixel lists into the accumulated value being kept for each pixel, starting from the last z-coordinate where composition left off for that pixel, and ending when we reach the depth of the z-target. (Thus, we may not compose *all* entries of the pixel list; those corresponding to z-coordinates beyond the z-

target are not composed yet, as there is a chance that there are faces not yet projected whose z-coordinates will precede them.) This incremental composition is done for each pixel whose pixel list has more than one entry. We remove from the pixel lists all of the entries that we compose, except for the last one (since it will be needed in order to continue the composition later). After composing the values at all of the relevant pixels, we reset the z-target to be the maximum z-coordinate of the vertices adjacent to v, and continue the sweep. In the example of Figure 1, if v_i was the previous z-target, then, when it is encountered, the new z-target is set to the z-coordinate of v_{i_3}.

Our choice of z-target allows us to prove that the composition is always done in the correct order for each pixel; i.e., we never compose a face f at a pixel for which there is an *unprojected* face f' preceding f in the depth order at the pixel. For, if to the contrary such an f' existed, then f' must have a vertex with z-coordinate less than that of its depth at the pixel, and therefore less than that of the z-target. However, then the face f' would have been projected prior to reaching the z-target (by the invariant maintained by our ZSweep), giving us a contradiction.

There is another issue in our delayed compositing method: If the dataset has highly nonuniform cell sizes (and therefore edge lengths), it could be that the z-target is set to be "very far" away from the prior z-target, leading to some pixel lists growing quite large before we reach it. To avoid this, we set a second criterion for stopping the sweep and performing incremental composition: When any pixel list reaches a user-specified threshold K (the current default is $K = 16$) in size, we stop and do incremental compositing. In some rare cases, it may be that some of the pixel lists *need* to grow beyond any prespecified threshold before compositing can be done while guaranteeing correctness of the order (as we insist in our exact algorithm). (Such examples are purely contrived, having cells that are "slivers" or "needles", and have never been observed to exist in our experiments.) In order to address this rare (but possible) event, we allow the size of the threshold K to increase (to $2K$, $4K$, etc.), as needed, in case the pixel lists cannot be even partially flushed (as in pathological cases). (The need to increase the threshold has not arisen in any of our experiments so far.)

3 Implementation Details

Our implementation of the ZSweep algorithm is in C++, consisting of less then 4500 lines of code, and is available from the authors.

While our algorithm permits datasets having general cell formats, our implementation currently handles only tetrahedral and hexahedral cells (as well as datasets having a mixture of the two), since these are by far the most popular unstructured cell formats.

3.1 Preprocessing and Basic Structures

There are two main arrays that store the data: the vertex array and the cell array. Most often, these two arrays are responsible for 90% of all memory used by our code. To make the connectivity faster and easier we build the *use set* for each vertex, which gives a list of all cells that use the vertex. The use set can be built in linear time by a pass over the data. Another step in the preprocessing phase is to mark the boundary faces and vertices. This is currently performed in time quadratic in the length of the use sets k, or $O(nk^2)$ where n is the number of vertices in the data set; since the value of k is very small for well behaved data sets (the longest we observed was 32) this cost can be considered linear in the number of vertices.

3.2 Sweep

The sweep function expects the vertices to be in depth (z) order in the vertex array. We used a heap to order the vertices by their z-coordinates. The heap keeps the array indices for the vertices, instead of pointers, which makes it straightforward to modify our code to obtain a shared-memory version of our code that is memory-efficient.

Now we consider how the sweep function identifies which face must be sent to the projection function. If vertex v_i is the vertex with smallest z-coordinate (there can be many of them in degenerate cases), we know that this vertex does not have any cell in its **use set** that has already been projected. But this case is a special case of the general case, so we will only discuss the general case, as depicted in Figure 1. Suppose vertex v_i has just been obtained from the heap. We scan v_i's use set to visit all of its touching cells. Notice that all cells represented by dashed lines in the figure are considered *swept* cells (by our definition). Suppose that we find the cell B; we can project both faces, between cells $A-B$ and between cells $B-C$. Then suppose we now find cell A. To avoid projecting twice all the interior faces of the dataset, we make use of a very small hash table. (Below, we discuss a further optimization ("sparsification") that we perform in order to minimize, but not eliminate, such occurrences.) In practice, we have observed that the hash table holds at most 15 faces per vertex, for data sets up to half a million cells.

If the current cell is hexagonal the only extra care that must be taken is to create two triangular faces and send them to the *hash table*. (The projection function will not even know if the face came from a tetrahedral or hexahedral cell.) We have used some connection information to avoid generating intersecting triangular faces; see Figure 2. One problem that will happen if we have intersecting faces is that once the hash key is built based on the indices for the vertices, 4 different faces will be included into the hash table. This will not cause the code to fail, but it can cause undesirable artifacts in the final image if the four vertices are not coplanar, which is true in general.

To address this issue, we use the connection between the global index and relative indices for the points. The point p_i in the global array of points can appear as any of the L vertices of a given L-vertex cell. Given a point we can easily find out the faces of the cell that use it. Each internal face will be found twice by the algorithm. We use the unique global index as the identification for the vertices. Consider the face shown in Figure 2. Assume that the cells C_k and C_p are the ones that share this face. Suppose that for cell C_k, the current vertex appears as its first local vertex (v_0). This will lead us to find the two triangular faces shown in Figure 2(a). When we find the same face on another cell, suppose that the current vertex appears as its second local vertex. To find the same combination for the three vertices independent of their relative local position for both neighbor cells, we used a "wrapping" computation to find the global index of each vertex, starting with the current vertex. So if the current vertex is the first vertex in the hexahedral face of the cell C_k, the indices for the two triangular faces are given by:

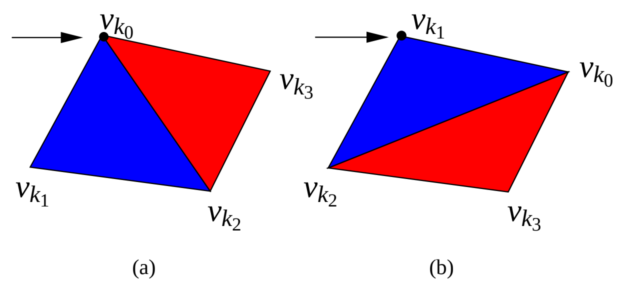

Figure 2: Left: The two triangular faces created when the current vertex v_{k_1} is the local vertex 0 for this face. Right: A case in which the current vertex is the second vertex for this same face, when found for the other cell that shares this face. This will happen if one always creates the faces starting from the local vertex 0 for all faces. Remember also that the hash key is generated based on the indices values, and that in this case four different faces will be included into the hash table.

$$\text{Triangle 1} = (v_{k_{0\%4}}, v_{k_{1\%4}}, v_{k_{2\%4}})$$
$$\text{Triangle 2} = (v_{k_{0\%4}}, v_{k_{2\%4}}, v_{k_{3\%4}})$$

When the cell C_p is found, to assure that the same two triangles will be generated we first find the relative position for the current vertex of the face for the other cell (it is 1 now). Then, we start getting the global vertex indices by the same integer division:

$$\text{Triangle 1} = (v_{k_{1\%4}}, v_{k_{2\%4}}, v_{k_{3\%4}})$$
$$\text{Triangle 2} = (v_{k_{1\%4}}, v_{k_{3\%4}}, v_{k_{4\%4}})$$

3.3 Projection

Before projecting, the code calls the composite function if either the current z-coordinate of the sweep plane has reached the target-z or if there is at least one pixel list with a length greater than a given threshold size.

This phase of the algorithm is simple. It gets the faces from the *hash table*, one by one, and projects them onto the screen. The projection is done by means of optimized intersection formulas. One detail is worth a remark: As the projections take place, the program keeps track of the bounding box of the screen region that contains pixels whose lists had some insertion. The composite function does not need to scan the entire screen looking for pixel lists to compose; it scans only the current bounding box.

3.4 Delayed Compositing

Now the last phase of the algorithm computes the color and brightness contribution from all faces projected so far. As the pixel lists contain the depth (z) and the interpolate scalar value of all faces that correspond to this pixel, the code must only go through each list and integrate the color and the opacity contributions; intersections that are summed to the pixel are removed from the pixel lists.

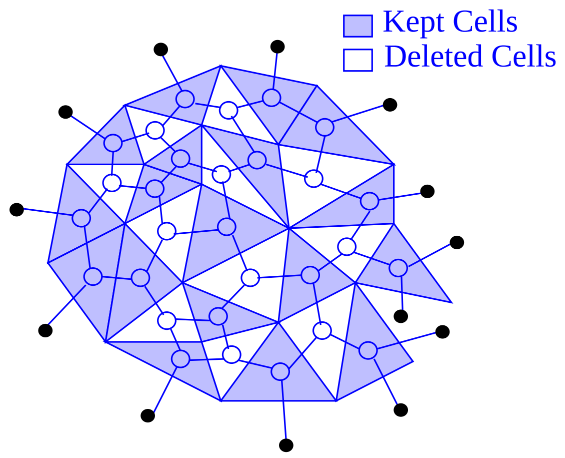

Figure 3: This drawing shows the sparse representation of a 2D mesh. Over the mesh, it is shown the equivalent graph with a edge covering. In the graph the nodes represent the cells and the edges represent the face between the cells. A ghost node must be included for each boundary face, to make it possible their representation in the graph.

3.5 Optimizations

Two other optimizations were included in our implementation and brought further efficiency in both speed and memory usage.

3.5.1 Sparse Data Representation

Since our algorithm ultimately performs face projections, it will project twice those faces that are shared by two face-neighboring cells. In order to avoid this as much as possible, we perform a "sparsification" step in which we keep only a subset of the cells that is sufficiently large to contain the set of all faces. In particular, we can "throw away" a large set of cells, provided that we do not throw away *both* cells that contain a given face. In the terminology of graph theory, we seek to find a maximum independent set in the dual graph of the cells. (Nodes correspond to cells and two cells are adjacent in the graph if the corresponding cells share a common face.) While finding maximum independent sets in graphs is a hard problem, we apply a greedy heuristic that works well in practice to eliminate a substantial fraction of the cells, leading to a substantial decrease in memory requirements (and some decrease in running time too).

In particular, we do the following. We keep all cells that have faces on the boundary, since they are essential for those boundary faces. We then iteratively mark cells for deletion. Each time we delete a cell, we mark its neighboring cells as "essential" (they are not permitted to be deleted. We continue until all cells are either deleted or marked "essential."

Our "sparsification" technique is related to the "chess-boarding" technique of Cignoni et al [4]. In [4], they save memory by avoiding the duplication of the "edges" of a regular grid dataset during isosurfacing.

Dataset Information						
Dataset	Vertices	Boundary Vertices	Faces	Boundary Faces	Cells	Cells in Sparse
Blunt Fin	41K	6.7K	382K	13.5K	187K	105K
Comb.Chamber	47K	7.8K	438K	15.6K	215K	121K
Oxygen Post	109K	27.7K	1040K	27.7K	513K	282K
Delta Wing	212K	20.7K	2032K	41.5K	1005K	541K
SPX	3K	1.4K	27K	2.8K	13K	8.6K
Hexahedral	2.7K	1.3K	6.4K	1.3K	1.9K	—-

Table 1: This table shows the number of cells (tetrahedra/hexahedra), the total number of vertices and faces, as well as the number of boundary vertices and faces for all datasets. The rightmost column shows the number of cells for each dataset after sparsification, which always results in at most 56% of the cells being kept.

***ZSweep* Preprocessing Time and Memory Usage**						
	Required Memory (MB)			*Preprocess (sec)*		
				K7-PC		SGI
Datasets	128^2	256^2	512^2	Original	Sparse	
Blunt Fin	13	16	24	2	2	7
Comb.Chamber	15	16	25	3	2	8
Oxygen Post	34	38	52	6	4	19
Delta Wing	64	68	80	13	8	37

Table 2: The first three columns show the total memory required by ZSweep to render each dataset in different resolutions. The fourth and fifth columns show the preprocessing times, on the K7-PC, for the original data sets and its sparse representation. The last column shows the preprocessing time on the SGI platform measured for the original data sets.

3.5.2 Use of Previous Heap Result

This optimization is only important if one wants to use the code to generate a sequence of images. Once the first image has been rendered, the heap class is able to keep a integral copy of itself. So supposing that the dataset is rotated by a small angle, it is usually true that the vertices are likely to have almost the same order than in the previous order. If instead of sending the vertices every time from the original global array, they are inserted into the heap in the order they had in the previous sweep, then the next ordering will be performed in linear time (in practice), since the vertices will be almost in order already. (A similar optimization is used in [22].)

Recall that the heap keeps only the indices for the points and the memory that it uses is very small compared to the memory used by the points and the cells. But if the amount of memory available is very small, this optimization can be omitted.

4 Experimental Results

As a first step in the experimental investigation of the ZSweep algorithm, we implemented a version that handled only tetrahedral grid datasets. Then, the simplicity of the algorithm allowed us, by a very simple modification, to make it handle also hexahedral grids data and mixed (tetrahedral and hexahedral) data. All of our experiments were conducted with this enhanced version of the software.

The data input may represent disconnected, concave datasets, even with “holes,” consisting of tetrahedral and hexahedral cells. The code reads the data from a file similar to the *Geomview*’s *off* format and is able to determine the type of each cell by its number of indices. The resulting image can be saved in *ppm* file format.

We ran our experiments on several popular datasets available from NASA, including *Blunt Fin*, *Combustion Chamber*, *Liquid Oxygen Post*, and *Delta Wing*. We use the tetrahedralized versions of these datasets, since our algorithm is intended to visualize unstructured grids. (For structured datasets one should opt for algorithms designed specifically to exploit the implicit representation of the grid, which allows for fast and highly memory-efficient algorithms; e.g., see [10].) We also perform our experiments on two other datasets, selected in order to verify the functionality of our implementation in the case of holes and hexahedral cells: *SPX*, which is a small tetrahedral dataset having *holes*, and *Hexahedral*, which is a small hexahedral dataset.

Table 1 gives basic information about all six datasets used in our experiments. In the rightmost column is shown the size of the data after sparsification, which eliminates, on average, about 53% of the cells. This savings allows the algorithm not only to reduce its total memory consumption, but also to reduce considerably the reading and preprocessing time.

4.1 ZSweep Performance

In this section we present the performance of ZSweep on two different platforms: an SGI machine (with a single 300MHZ MIPS R12000 processor and 512 Mbytes of memory) and a K7-PC (with a 900MHZ AMD K7 Athlon and 768 Mbytes of memory).

Table 2 shows *ZSweep*’s preprocessing times, which include reading and generating the *use set* for all vertices and memory usage required to create different image sizes. When larger images are required to be generated, more memory is necessary, since for each pixel the algorithm has to keep an ordered list (the pixel list) of intersected faces. The required memory grows sublinearly, however, since for an image 16 times larger, the memory goes up by less than a factor of 2.

Table 3 shows *ZSweep* rendering times on the SGI platform. The resolutions were chosen to allow us to compare our results with previous works. The compilation was performed in 32 bits with highest possible optimization ("-O3"). Table 3 shows the equivalent tests performed on the K7-PC platform.

The increase in the observed render time as the size of the image grows, particularly with larger datasets (e.g., *delta wing*), is largely due to time spent by the algorithm in keeping the pixel lists. Further care must be taken to avoid the render time to grow faster, if it is desired to visualize even larger datasets.

4.2 Comparison with Other Methods

We compare our results with two fastest and most recent algorithms available for unstructured grids, Lazy Sweep [16] and the ray-casting algorithm of [1]. (We do not compare here with hardware-accelerated algorithms, as we are studying the performance here of pure software implementations.) We compare the render costs, both in rendering time and total memory consumed for each of the three methods, for all four NASA datasets (those on which the other two methods apply); see Tables 5–8.

One last note we make is that *ZSweep*, just like *Bunyk et al.*, was implemented using a lighting model that, although simple, is computationally more expensive than the model used on the lazy sweep work. So even in the cases where *Lazy Sweep* compares in speed with *ZSweep*, keep in mind that the final image generated by *ZSweep* will be more accurate in terms of the lighting. †

5 Conclusion

The unstructured grid volume rendering algorithm (ZSweep) we presented in this paper has proven to be a very competitive option for both general and specific applications due to its relatively low memory requirements, high speed, accuracy and simplicity. It is considerably simpler and faster than the previous sweep-based rendering algorithms (without hardware assistance). As with the previous algorithms of [16] and [1], the accuracy of the final image does not depend on the characteristics of the dataset grid.

Also, as with the Lazy Sweep method of [16], ZSweep is memory efficient, even though a highly anomalous dataset could cause the pixel lists, maintained for each pixel of the screen, to become lengthy, making it necessary for further precautions (e.g., partitioning of the viewing plane into subimages) to be taken to avoid having these lists consume too much memory. We note, however, that for all tests on all datasets mentioned on Table 1, we did not notice an unexpected increase of memory usage. We did expect the memory allocation to increase for larger images, since as the image size increases, each face projected will insert intersection units into more and more pixel lists. While ZSweep is more than twice as fast as [16], it uses from 20% to 60% more memory, which is not enough even to slow down the reading/preprocessing step compared to the Lazy Sweep method. We have methods of reducing the memory requirements that we are exploring in our continuing investigations.

A possible parallelization can be obtained by dividing the image plane in a grid of rectangles, identifying all points of the data that lay inside each the parallelogram defined by a rectangle, as its base, and the depth as its height, and distribute the parallelograms to each processor to perform the ZSweep on its points.

On the other hand, even though ZSweep is slower than [1] in some cases, it uses considerably less memory and ZSweep does not have the difficulty that arises from having ray casts that hit degenerate points at vertices or edges of the grid.

We finally note that our current implementation suffers a small overhead of checking for the type of each cell to decide how to proceed, since it can handle tetrahedral and hexahedral cells together in the same dataset. Most other algorithm implementations do not offer this flexibility (two notable expections are the implementation of LSRC and HIAC). Due to the extreme simplicity of the ZSweep basic concept, this was easy to accomplish.

Besides parallelization, we are also exploring other improvements on ZSweep, including (1) further reducing the memory requirements by partitioning the image space and running the algorithm separately on subimages; (2) adapting ZSweep for walkthrough applications; currently, we assume

†Our lighting model is the same as that used in *Bunyk et al.*, based on integration of linearly-interpolated color and opacity values along each ray. Scalar values in the input dataset are shifted and scaled to fit the $[0,255]$ range. A user-specified piecewise-linear transfer function is read from a file; it specifies the mapping from this range to the set of opacity and RGB values. During ray casting, we calculate the z and interpolated scalar field values of the ray intersection points with the current and the next triangle and pass these values to the transfer function calculation module, which updates the RGB values of the current pixel.

The exact integration formulas follow. The following variables are used: z_c, z_n, z coordinates of intersection with the *current* and *next* triangles; Δz, distance between z_c and z_n; c_c, c_n, linearly-interpolated color component value in z_c and z_n; o_c, o_n, linearly-interpolated opacity in z_c and z_n; C_c, O_c, accumulated on the previous steps color and opacity values, initially 0; C_n, O_n, updated color and opacity values.

Color and opacity are linearly interpolated between their values in z_c and z_n:

$$o(z) = \frac{o_c(z_n - z) + o_n(z - z_c)}{\Delta z}$$

$$c(z) = \frac{c_c(z_n - z) + c_n(z - z_c)}{\Delta z}$$

These linear functions must be integrated from z_c to z_n to obtain O_n, C_n. We also need the opacity value in all intermediate points to use it in color computation:

$$O(z) = O_c + \int_{z_c}^{z} o(z)dz$$

$$C(z) = C_c + \int_{z_c}^{z} c(z)(1 - O(z))dz.$$

After computing these integrals analytically, we obtain the following values for O_n and C_n:

$$O_n = O_c + \frac{1}{2}(o_c + o_n)\Delta z$$

$$C_n = C_c - \frac{1}{2}(c_c + c_n)(O_c - 1)\Delta z - \frac{1}{24}(3c_c o_c + 5c_n o_c + c_c o_n + 3c_n o_n)\Delta z^2.$$

As a comparison, the lighting model used in Lazy Sweep amounts to a table lookup for each color channel, and multiplication by the transparency. It is possible to make the lighting model considerably more accurate and complex. A good example is the one used in the HIAC system, described in Williams et al [21]. Max [12] gives a good survey of optical models for volume rendering.

***ZSweep* Rendering Time on the SGI**						
Datasets	128^2		256^2		512^2	
Blunt Fin	2	4453	6	17858	33	71508
Comb.Chamber	4	5032	7	20234	32	81544
Oxygen Post	7	6254	16	25160	62	101034
Delta Wing	14	4396	23	17684	76	71062

Table 3: This table shows the render time (in seconds) and the number of pixels processed for each dataset and each image size on the SGI.

***ZSweep* Rendering Time on the K7-PC**						
Datasets	128^2		256^2		512^2	
Blunt Fin	2	4453	5	17858	20	71508
Comb.Chamber	2	5032	6	20234	21	81544
Oxygen Post	5	6254	11	25160	40	101034
Delta Wing	9	4396	16	17684	43	71062

Table 4: This table shows the render time (in seconds) and the number of pixels processed or each dataset and each image size on the K7-PC compatible machine.

Blunt Fin dataset comparison					
				ZSweep Results	
Method	Image Size	Time(s)	Memory (MB)	Time(s)	Memory (MB)
Lazy Sweep	530x230	22	8	5	16
Bunyk et al.	128^2	2	76	2	13
Bunyk et al.	256^2	8	77	6	16
Bunyk et al.	512^2	27	81	33	24

Table 5: While ZSweep uses about twice the memory that *Lazy Sweep* requires, it is 4.4 times faster. As the image size grows, *Bunyk et al.* becomes slightly faster than ZSweep, but at the cost of much higher memory consumption.

Combustion Chamber dataset comparison					
				ZSweep Results	
Method	Image Size	Time(s)	Memory (MB)	Time(s)	Memory (MB)
Lazy Sweep	300x200	19	9	5	16
Bunyk et al.	128^2	4	88	4	15
Bunyk et al.	256^2	10	89	7	16
Bunyk et al.	512^2	37	93	32	25

Table 6: In this case ZSweep is about 4.75 times faster than *Lazy Sweep*, while again using about twice the memory. For this dataset, *ZSweep* was faster than the *Bunyk et al.* method for all image sizes considered, while using substantially less memory.

Liquid Oxygen Post dataset comparison					
				ZSweep Results	
Method	Image Size	Time(s)	Memory (MB)	Time(s)	Memory (MB)
Lazy Sweep	300x300	37	22	22	35
Lazy Sweep	600x600	82	22	87	54
Bunyk et al.	128^2	5	208	7	34
Bunyk et al.	256^2	19	209	16	38
Bunyk et al.	512^2	72	214	62	52

Table 7: To create an image of size 300^2 *ZSweep* requires 60% more memory than *Lazy Sweep*. But, while *Lazy Sweep* maintains its memory requirements essentially the same even for larger images, ZSweep needs to allocate more and more memory, because of the pixel lists. Again ZSweep is comparable to *Bunyk et al.* in speed, but much more memory efficient.

Delta Wing dataset comparison					
				ZSweep Results	
Method	Image Size	Time(s)	Memory (MB)	Time(s)	Memory (MB)
Lazy Sweep	300x300	64	44	27	67
Bunyk et al.	128^2	4	406	14	64
Bunyk et al.	256^2	13	407	23	68
Bunyk et al.	512^2	43	411	67	80

Table 8: Delta Wing is a medium-size dataset with over a million tetrahedra. For this dataset it becomes clear that the pixel lists are slowing down the algorithm. But it is still 2.4 times faster than *Lazy Sweep*. And even though *ZSweep* became slower than *Bunyk et al.* algorithm, the memory this last one needs is still a big problem nowadays.

that views are from outside the datasets, and clipping the outside can be performed efficiently within our framework; (3) exploring the development of a hardware-assisted version of the ZSweep; (4) add other cell formats (including nonconvex cells).

Acknowledgements

We thank Brian Wylie (Sandia) for his collaboration on this research, and Paul Bunyk (Stony Brook) for access to and help with his code. We thank Peter Williams (LLNL) for the SPX dataset, and Constantine "Dino" Pavlakos (Sandia) for the Hexahedral dataset. NASA has gracefully provided the Blunt Fin, Liquid Oxygen Post, and Delta Wing datasets. The Combustion Chamber dataset is from the Visualization Toolkit (Vtk).

This work was made possible with the generous support of Sandia National Labs and the Dept of Energy Mathematics, Information and Computer Science Office. R. Farias acknowledges partial support from CNPq-Brazil under a PhD fellowship. J. Mitchell acknowledges support from HRL Laboratories, the National Science Foundation (CCR-9732221), NASA Ames Research Center, Northrop-Grumman Corporation, Sandia National Labs, Seagull Technology, and Sun Microsystems.

References

[1] P. Bunyk, A. Kaufman, and C. Silva. Simple, fast, and robust ray casting of irregular grids. In G. Nielson H. Hagen and F. Post, editors, *Scientific Visualization*. IEEE Computer Society Press, 1999.

[2] L. Carpenter. The A-buffer, an antialiased hidden surface method. In *SIGGRAPH '84*, pages 103–108, 1984.

[3] P. Cignoni and L. De Floriani. Power diagram depth sorting. In *10th Canadian Conference on Computational Geometry*, 1998.

[4] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In *Visualization in Scientific Computing '95*, pages 58–71. Springer Verlag, 1995.

[5] P. Cignoni, C. Montani, and R. Scopigno. Tetrahedra based volume visualization. In *Mathematical Visualization – Algorithms, Applications, and Numerics*, pages 3–18. Springer Verlag, 1998.

[6] P. Cignoni, C. Montani, and R. Scopigno. Tetrahedra based volume visualization. *Mathematical Visualization*, pages 3–18, 1998.

[7] J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, and P Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum*, 18(3):369–376, September 1999.

[8] M. Garrity. Raytracing irregular volume data. In *Computer Graphics*, pages 35–40, November 1990.

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

[10] L. Hong and A. Kaufman. Fast projection-based ray-casting algorithm for rendering curvilinear volumes. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):322–332, October - December 1999.

[11] L. Hong and A. Kaufman. Accelerated ray-casting for curvilinear volumes. *IEEE Visualization '98*, pages 247–254, October 1998.

[12] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[13] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics*, pages 27–33, November 1990.

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

[15] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics*, pages 63–70, November 1990.

[16] C. Silva and J. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), April–June 1997.

[17] C. Silva, J. Mitchell, and P. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. *1998 Volume Visualization Symposium*, pages 87–94, October 1998.

[18] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *1994 Symposium on Volume Visualization*, pages 83–90, October 1994.

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

[20] P. Williams. Visibility ordering meshed polyhedra. *ACM Transaction on Graphics*, 11(2):103–125, April 1992.

[21] P. Williams, N. Max, and C. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, January-March 1998.

[22] C. Wittenbrink. Cellfast: Interactive unstructured volume rendering. In *Proceedings IEEE Visualization'99, Late Breaking Hot Topics*, pages 21–24, 1999. Also available as Technical Report, HPL-1999-81R1.

[23] R. Yagel, D. Reed, A. Law, P.-W. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *1996 Volume Visualization Symposium*, pages 55–62. IEEE, October 1996.

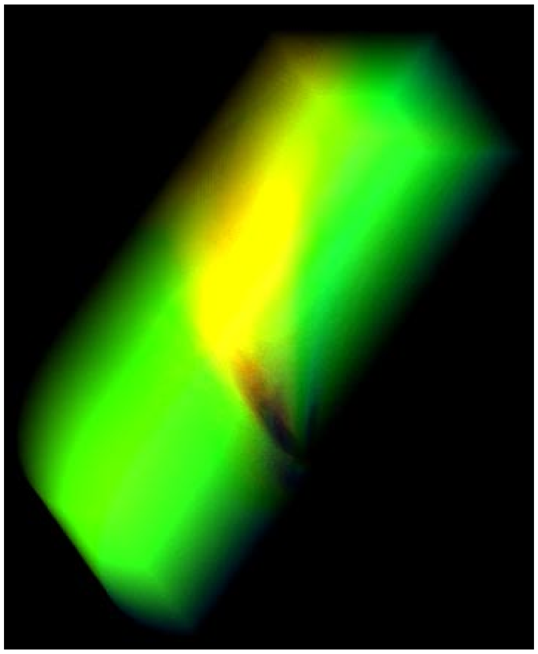

Figure 4: Image of Blunt Fin created in 512*x*512.

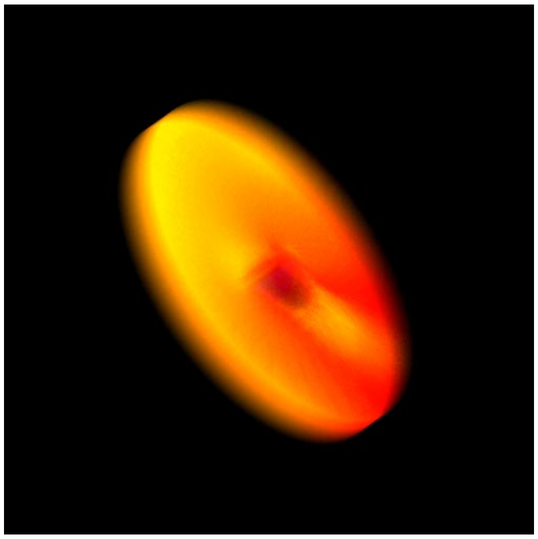

Figure 6: Image of Liquid Oxygen Post created in 512*x*512.

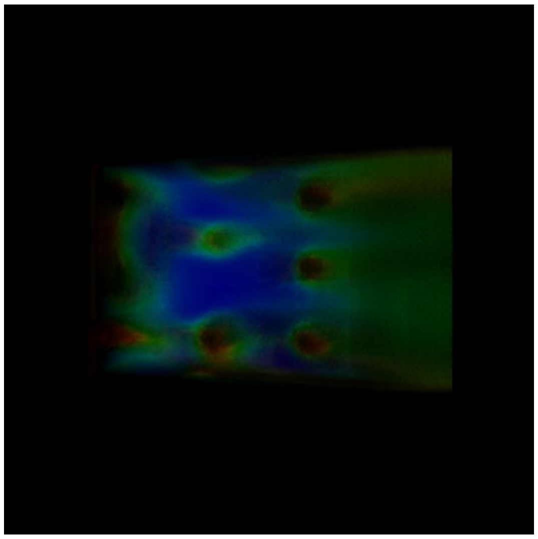

Figure 5: Image of Combustion Chamber created in 512*x*512.

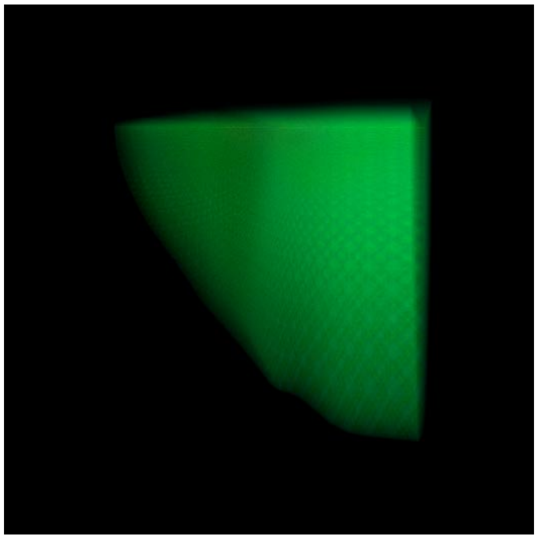

Figure 7: Image of Delta Wing created in 512*x*512.