Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering

Steven P. Callahan, Milan Ikits, *Student Member*, *IEEE*,
João L.D. Comba, and Cláudio T. Silva, *Member*, *IEEE*

Abstract—Harvesting the power of modern graphics hardware to solve the complex problem of real-time rendering of large unstructured meshes is a major research goal in the volume visualization community. While, for regular grids, texture-based techniques are well-suited for current GPUs, the steps necessary for rendering unstructured meshes are not so easily mapped to current hardware. We propose a novel volume rendering technique that simplifies the CPU-based processing and shifts much of the sorting burden to the GPU, where it can be performed more efficiently. Our hardware-assisted visibility sorting algorithm is a hybrid technique that operates in both object-space and image-space. In object-space, the algorithm performs a partial sort of the 3D primitives in preparation for rasterization. The goal of the partial sort is to create a list of primitives that generate fragments in nearly sorted order. In image-space, the fragment stream is incrementally sorted using a fixed-depth sorting network. In our algorithm, the object-space work is performed by the CPU and the fragment-level sorting is done completely on the GPU. A prototype implementation of the algorithm demonstrates that the fragment-level sorting achieves rendering rates of between one and six million tetrahedral cells per second on an ATI Radeon 9800.

Index Terms—Volume visualization, graphics processors, visibility sorting.

1 Introduction

Given a general scalar field in $\mathbb{R}^3$, a regular grid of samples can be used to represent the field at grid points $(\lambda_i, \lambda_j, \lambda_k)$, for integers i, j, k and some scale factor $\lambda \in \mathbb{R}$. One serious drawback of this approach is that, when the scalar field has highly nonuniform variation—a situation that often arises in computational fluid dynamics and partial differential equation solvers—the voxel size must be small enough to represent the smallest features in the field. Unstructured grids with cells that are not necessarily uniform in size have been proposed as an effective means for representing disparate field data.

In this paper, we are primarily interested in volume rendering unstructured scalar data sets. In volume rendering, the scalar field is modeled as a cloud-like material that both emits and attenuates light along the viewing direction [1]. To create an image, the equations for the optical model must be integrated along the viewing ray for each pixel (see Fig. 1). For unstructured meshes, this requires computing a separate integral for the contribution of the ray segment inside each cell. If the order of these segments is known, the individual contributions can be accumulated using front-to-back or back-to-front compositing.

- *S.P. Callahan, M. Ikits, and C.T. Silva are with the Scientific Computing and Imaging Institute, School of Computing, University of Utah, 50 S. Central Campus Dr., Salt Lake City, UT 84112. E-mail: {stevec, ikits}@sci.utah.edu, csilva@cs.utah.edu.*
- *J.L.D. Comba is with the Universidade Federal do Rio Grande do Sul, Instituto de Informatica, Av. Bento Goncalves, 9500, Campus do Vale-Bloco IV-Prédio 43425, Porto Alegre RS 91501-970, Brazil. E-mail: comba@inf.ufrgs.br.*

Manuscript received 18 Oct. 2004; revised 27 Dec. 2004; accepted 5 Jan. 2005; published online 10 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0131-1004.

On a practical level, the whole computation amounts to sampling the volume along the viewing rays, determining the contribution of each sample point, and accumulating the contributions in proper order. Given the increasing size of volume data sets, performing these operations in real-time requires the use of specialized hardware. Modern GPUs [2] are quite effective at performing most of these tasks. By coupling the rasterization engine with texture-based fragment processing, it is possible to perform very efficient volume sampling [3], [4]. However, generating the fragments in visibility order is still necessary.

For regular grids, generating the fragments in visibility order is straightforward. This is often accomplished by rendering polygons $p_1, p_2, \ldots, p_n$ perpendicular to the view direction at different depths. The polygons are used to slice the volume and generate the samples for the cells that intersect them. The fact that the polygons are rendered in sorted order and are parallel with each other guarantees that all the fragments generated by rasterizing polygon p_i come before those for p_{i+1}. In this case, compositing can be accomplished by blending the fragments into the framebuffer in the order in which they are generated. For details on performing these computations, see [5].

The sampling and compositing procedure for unstructured grids is considerably more complicated. Although the intrinsic volume rendering computations are similar, the requirement of generating fragments in visibility order makes the computations more expensive and difficult to implement. The Projected Tetrahedra (PT) algorithm [6] was the first to show how to render tetrahedral cells using the traditional 3D polygon-rendering pipeline. Given tetrahedra T and a viewing direction v, the technique first classifies the faces of T into front and back faces with respect to v. Next, for correct fragment generation, the faces are subdivided into regions of equal visibility. Note that the PT algorithm can properly handle only a single tetrahedral

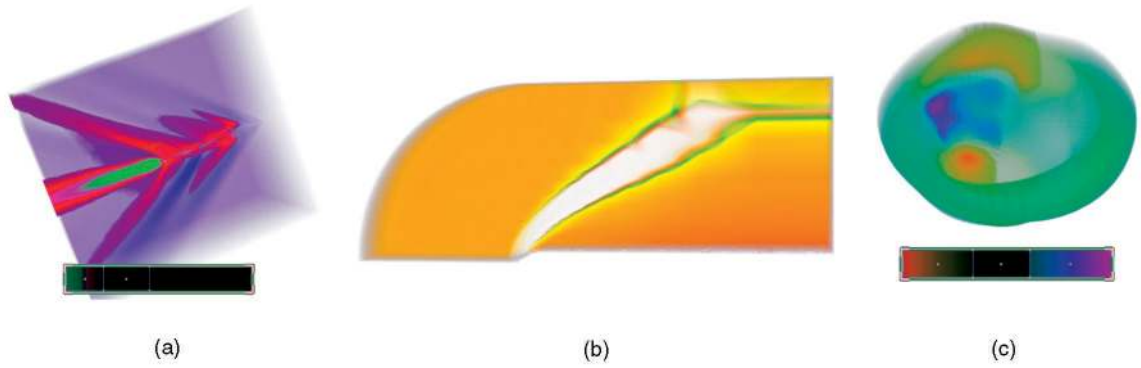

Fig. 1. Results of volume rendering the (a) fighter, (b) blunt fin, and (c) heart data sets with the HAVS algorithm.

cell. For rendering meshes, cells have to be projected in visibility order, which can be accomplished using techniques such as the Meshed Polyhedra Visibility Ordering (MPVO) algorithm [7]. For acyclic convex meshes, this is a powerful combination that leads to a linear-time algorithm that is provably correct, i.e., it is guaranteed to produce the right picture. When the mesh is not convex or contains cycles, MPVO requires modifications that significantly complicate the algorithm and its implementation, leading to slower rendering times [8], [9], [10], [11].

The necessity of explicit fragment sorting for unstructured grids is the main cause of the rendering-speed dichotomy between regular and unstructured grids. For regular grids, we are exploiting the fact that we can sort in object space (implicit in the order of the planes being rendered) and avoid sorting in image space (i.e., sorting fragments). Thus, on modern GPUs, it is possible to render regular volumes at very high frame rates. Unfortunately, performing visibility ordering for unstructured grids completely in image space has turned out to be quite expensive and complex [11], [12], [13].

In this paper, we build on the previous work of Farias et al. [14] and Carpenter [15] and propose a new volume rendering algorithm. Our main contributions are:

- We present a new algorithm for rendering unstructured volumetric data that simplifies the CPU-based processing and shifts much of the sorting burden to the GPU, where it can be performed more efficiently. The basic idea of our algorithm is to separate visibility sorting into two phases. First, we perform a partial visibility ordering of primitives in object-space using the CPU. Note that this first phase does not guarantee an exact visibility order of the fragments during rasterization. In the second phase, we use a modified A-buffer of fixed depth (called the k-buffer) to sort the fragments in exact order on the GPU.
- We show how to efficiently implement the k-buffer using the programmable functionality of existing GPUs.
- We perform a detailed experimental analysis to evaluate the performance of our algorithm using several data sets, the largest of which having over 1.4 million cells. The experiments show that our algorithm can handle general nonconvex meshes with very low memory overhead and requires only a light and completely automatic preprocessing step. Data size limitations are bounded by the available main memory on the system. The achieved rendering rates of over six million cells per second are, to our knowledge, the *fastest* reported results for volume rendering of unstructured data sets.
- We provide a detailed comparison of our algorithm with existing methods for rendering unstructured volumetric data. This includes render rates performed using optimized implementations of these algorithms using uniform test cases on the same machine.

The remainder of this paper is organized as follows: We summarize related work in Section 2. In Section 3, we describe our algorithm, define k-nearly sorted sequences, and provide further details on the functionality of the k-buffer. In Section 4, we show how to efficiently implement the k-buffer using the programmable features of current ATI hardware. Section 5 summarizes our experiments and results. In Section 6, we discuss different trade-offs of our approach. Finally, in Section 7, we provide final remarks and directions for future work.

2 Related Work

The volume rendering literature is vast and we do not attempt a comprehensive review here. Interested readers can find a more complete discussion of previous work in [5], [11], [16], [17], [18]. We limit our coverage to the most directly related work in the area of visibility ordering using both software and hardware techniques.

In computer graphics, work on visibility ordering was pioneered by Schumacker et al. and is later reviewed in [19]. An early solution to computing a visibility order was given by Newell, Newell, and Sancha (NNS) [20], which continues to be the basis for more recent techniques [21]. The NNS algorithm starts by partially ordering the primitives according to their depth. Then, for each primitive, the algorithm improves the ordering by checking whether other primitives precede it or not.

Fuchs et al. [22] developed the Binary Space Partitioning tree (*BSP-tree*)—a data structure that represents a hierarchical convex decomposition of a given space (typically, $\mathbb{R}^3$). Each node ν of a BSP-tree $\mathcal{T}$ corresponds to a convex polyhedral region, $P(\nu) \subset \mathbb{R}^3$ and the root node corresponds to all of

$\mathbb{R}^3$. Each nonleaf node ν is defined by a hyperplane, $h(\nu)$ that partitions $P(\nu)$ into two half-spaces, $P(\nu^+) = h^+(\nu) \cap P(\nu)$ and $P(\nu^-) = h^-(\nu) \cap P(\nu)$, corresponding to the two children, ν^+ and ν^- of ν. Here, $h^+(\nu)$ ($h^-(\nu)$) is the half-space of points above (below) the plane $h(\nu)$. Fuchs et al. [22] demonstrated that BSP-trees can be used for obtaining a visibility ordering of a set of objects or, more precisely, an ordering of the fragments into which the objects are cut by the partitioning planes. The key observation is that the structure of the BSP-tree permits a simple recursive algorithm for rendering the object fragments in back-to-front order. Thus, if the viewpoint lies in the positive half-space $h^+(\nu)$, we can recursively draw the fragments stored in the leaves of the subtree rooted at ν^-, followed by the object fragments $S(\nu) \subset h(\nu)$. Finally, we recursively draw the fragments stored in the leaves of the subtree rooted at ν^+. Note that the BSP-tree does not actually generate a visibility order for the original primitives, but for *fragments* of them.

The methods presented above operate in *object-space*, i.e., they operate on the primitives before rasterization by the graphics hardware [2]. Carpenter [15] proposed the A-buffer—a technique that operates on pixel fragments instead of object fragments. The basic idea is to rasterize all the objects into sets of pixel fragments, then save those fragments in per-pixel linked lists. Each fragment stores its depth, which can be used to sort the lists after all the objects have been rasterized. A nice property of the A-buffer is that the objects can be rasterized in any order and, thus, do not require any object-space ordering. A main shortcoming of the A-buffer is that the memory requirements are substantial. Recently, there have been proposals for implementing the A-buffer in hardware. The R-buffer [23] is a pointerless A-buffer hardware architecture that implements a method similar to a software algorithm described in [24] for sorting transparent fragments in front of the frontmost opaque fragment. Current hardware implementations of this technique require multiple passes through the polygons in the scene [25]. In contrast, the R-buffer works by scan-converting all polygons only once and saving the not yet composited or rejected fragments in a large unordered recirculating fragment buffer on the graphics card, from which multiple depth comparison passes can be made. The Z^3 hardware [26] is an alternative design which uses sparse supersampling and screen door transparency with a fixed amount of storage per pixel. When there are more fragments generated for a pixel than what the available memory can hold, Z^3 merges the extra fragments.

Because of recent advances in programmable graphics hardware, techniques have been developed which shift much of the computation to the GPU for increased performance. Kipfer et al. [27] introduce a fast sorting network for particles. This algorithm orders the particles using a Bitonic sort that is performed entirely on the GPU. Unstructured volume rendering has also seen a number of recent advances. Roettger and Ertl [28] demonstrate the efficiency of the GPU for compositing the ray integrals of arbitrary unstructured polyhedra. Their method uses an emissive optical model which does not require any ordering. Their technique is similar to HAVS without sorting. Recently, Wylie et al. have shown how to implement the Shirley-Tuchman tetrahedron projection directly on the GPU [29]. As mentioned before, the PT projection sorts fragments for a single tetrahedron only and still requires that the cells be sent to the GPU in sorted order. An alternative approach is to perform pixel-level fragment sorting via ray-casting. This has been shown possible by Weiler et al. for convex meshes [12] and, more recently, for nonconvex meshes [13].

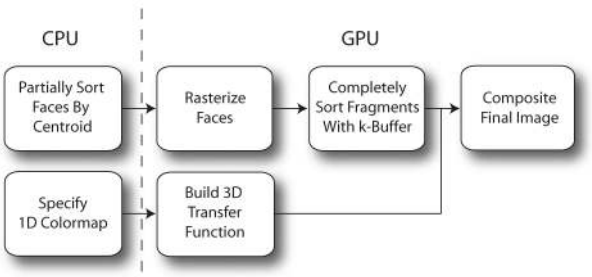

Fig. 2. Overview of the hardware-assisted visibility sorting algorithm (HAVS). Only a partial visibility ordering is performed on the CPU based on the face centroids. On the GPU side, a fixed size A-buffer is used to complete the sort on a per-fragment basis.

Roughly speaking, all of the techniques described above perform sorting *either* in object-space *or* in image-space exclusively, where we consider ray casting as sorting in image-space and cell projection as sorting in object-space. There are also hybrid techniques that sort both in image-space and object-space. For instance, the ZSWEEP [14] algorithm works by performing a partial ordering of primitives in object-space followed by an exact pixel-level ordering of the fragments generated by rasterizing the objects. Depending on several factors, including average object size, accuracy and speed of the partial sort, and cost of the fragment-level sorting, hybrid techniques can be more efficient than either pure object-space or image-space algorithms. Another hybrid approach is presented in [30], where the authors show how to improve the efficiency of the R-buffer by shifting some of the work from image-space to object-space.

3 Hardware-Assisted Visibility Sorting

The hardware-assisted visibility sorting algorithm (HAVS) is a hybrid technique that operates in both object-space and image-space. In object-space, HAVS performs a partial sorting of the 3D primitives in preparation for rasterization; the goal here is to generate a list of primitives that cause the fragments to be generated in *nearly sorted order*. In image-space, the fragment stream is incrementally sorted by the use of a fixed-depth sorting network. HAVS *concurrently* exploits both the CPU and GPU such that the object-space work is performed by the CPU while the fragment-level sorting is implemented completely on the GPU (see Fig. 2). Depending on the relative speed of the CPU and the GPU, it is possible to shift work from one processor to the other by varying the accuracy of the two sorting phases, i.e., by increasing the depth of the fragment sorting network, we can use a less accurate object-space sorting algorithm. As

shown in Section 4, our current implementation uses very simple data structures that require essentially no topological information leading to a very low memory overhead. In the following sections, we present further details on the two phases of HAVS.

3.1 Nearly Sorted Object-Space Visibility Ordering

Visibility ordering algorithms (e.g., Extended Meshed Polyhedra Visibility Ordering [9]) sort 3D primitives with respect to a given viewpoint v in *exact* order, which allows for direct compositing of the rasterized fragments. In our work, we differentiate between the sorting of the 3D primitives and the sorting of the rasterized fragments to utilize faster object-space sorting algorithms.

To precisely define what we mean by nearly sorted object-space visibility ordering, we first introduce some notation. Given a sequence S of real values $\{s_1, s_2, \ldots, s_n\}$, we call the tuple of distinct integer values $(a_1, a_2, \ldots, a_n)$ the *Exactly Sorted Sequence* of S (or $\mathrm{ESS}(S)$) if each a_i is the position of s_i in an ascending or descending order of the elements in S. For instance, for the sequence $S = \{0.6, 0.2, 0.3, 0.5, 0.4\}$, the corresponding exactly sorted sequence is $\mathrm{ESS}(S) = (5, 1, 2, 4, 3)$. Extensions to allow for duplicated values in the sequence are easy to incorporate and are not discussed here. Similarly, we call a tuple $(b_1, b_2, \ldots, b_n)$ of distinct integer values a *k-Nearly Sorted Sequence* of S (or k-NSS(S)) if the maximum element of the pairwise absolute difference of elements in ESS(S) and k-NSS(S) is k, i.e., $\max(|a_1 - b_1|, |a_2 - b_2|, \ldots |a_n - b_n|)) = k$. For instance, the tuple $(4, 2, 1, 5, 3)$ is a 1-NSS(S) (i.e., $max(|5-4|, |1-2|, |2-1|, |4-5|, |3-3|) = 1$), while the tuple $(3, 1, 4, 5, 2)$ is a 2-NSS(S). In this work, we process sequences of fragment distances from the viewpoint. We relax the requirement of having exactly sorted sequences, which allows for faster object-space sorting, but leads to nearly sorted sequences that need to be sorted exactly during the fragment processing stage.

There are many techniques that implicitly generate nearly sorted sequences. For example, several hierarchical spatial data structures provide mechanisms for simple and efficient back-to-front traversal [31]. A simple way of generating nearly sorted object-space visibility ordering of a collection of 3D primitives is to use a BSP-tree, which has been shown to cause near-linear primitive growth from cutting [8]. The goal is to ensure that, after rasterization, pixel fragments are at most k positions out of order. In a preprocessing step, we can hierarchically build a BSP-tree such that no leaf of the BSP tree has more than k elements. Note that this potentially splits the original primitives into multiple ones. To generate the actual ordering of the primitives, we can use the well-known algorithm for back-to-front traversal of a BSP-tree and render the set of k primitives in the leaf nodes in any order. Since it is not strictly necessary to implement this approach, simpler sorting techniques are used in our work. In practice, most data sets are quite well-behaved and even simple techniques, such as sorting primitives by their centroid or even by their first vertex, are often sufficient to generate nearly sorted geometry. This was previously exploited in the ZSWEEP algorithm [14]. In ZSWEEP, primitives are sorted by considering a sweep plane parallel to the viewing plane. As the sweep plane touches a vertex of a face, the face is rasterized and the generated fragments are added to an A-buffer using insertion sort. It was experimentally observed that the insertion sort had nearly linear complexity because fragments were in almost sorted order. To avoid a memory space explosion in the A-buffer, ZSWEEP uses a *conservative* technique for compositing samples [14]. In our approach, we apply a more *aggressive* technique for managing the A-buffer.

3.2 The k-Buffer

The original A-buffer [15] stores all incoming fragments in a list, which requires a potentially unbounded amount of memory. Our k-buffer approach stores only a fixed number of fragments and works by combining the current fragments and discarding some of them as new fragments arrive. This technique reduces the memory requirement and is simple enough to be implemented on existing graphics architectures (see Section 4).

The k-buffer is a *fragment stream sorter* that works as follows. For each pixel, the k-buffer stores a constant k number of entries $\langle f_1, f_2, \ldots, f_k \rangle$. Each entry contains the distance of the fragment from the viewpoint, which is used for sorting the fragments in increasing order for front-to-back compositing and in decreasing order for back-to-front compositing. For front-to-back compositing, each time a new fragment f_{new} is inserted in the k-buffer, it dislodges the first entry (f_1). Note that boundary cases need to be handled properly and that f_{new} may be inserted at the beginning of the buffer if it is closer to the viewpoint than all the other fragments or at the end if it is further. A key property of the k-buffer is that, given a sequence of fragments such that each fragment is within k positions from its position in the sorted order, it will output the fragments in the correct order. Thus, with a small k, the k-buffer can be used to sort a k-nearly sorted sequence of n fragments in $O(n)$ time. Note that, to composite a k-nearly sorted sequence of fragments, $k+1$ entries are required because both the closest and second closest fragments must be available for the preintegrated table lookup. In practice, the hardware implementation is simplified by keeping the k-buffer entries unsorted (see Fig. 3).

Compared to ZSWEEP, the k-buffer offers a less conservative fragment sorting scheme. Since only k entries are considered at a time, if the incoming sequence is highly out of order, the output will be incorrect, which may be noticeable in the images. As shown in Section 5, even simple and inexpensive object-space ordering leads to fragments that are almost completely sorted.

3.3 Volume Rendering Algorithm

The volume rendering algorithm is built upon the machinery presented above. First, we sort the *faces* of the tetrahedral cells of the unstructured mesh on the CPU based on the face centroids using the floating-point radix sort algorithm. To properly handle boundaries, the vertices are marked whether they are internal or boundary vertices of the mesh. Next, the faces are rasterized by the GPU, which completes the sort using the k-buffer and composites the accumulated color and opacity into the framebuffer (see Fig. 2). The complete pseudocode for our CPU algorithm is given below:

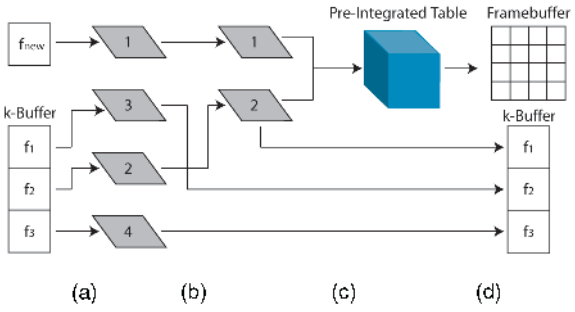

Fig. 3. Example of the k-buffer with $k = 3$ (see also Section 4). (a) We start with the incoming fragment and the current k-buffer entries and (b) find the two entries closest to the viewpoint. (c) Next, we use the scalar values (v_1, v_2) and view distances (d_1, d_2) of the two closest entries to look up the corresponding color and opacity in the preintegrated table. (d) In the final stage, the resulting color and opacity are composited into the framebuffer and the remaining three entries are written back into the k-buffer.

CPU-SORT
  Perform sort on face centroids
  for each sorted face sf
    Send sf to GPU for rasterization

4 K-Buffer Hardware Implementation

The k-buffer can be efficiently implemented using the *multiple render target* capability of the latest generation of ATI graphics cards. Our implementation uses the `ATI_draw_buffers` OpenGL extension, which allows writing into up to four floating-point RGBA buffers in the fragment shader. One of the buffers is used as the framebuffer and contains the accumulated color and opacity of the fragments that have already left the k-buffer. The remaining buffers are used to store the k-buffer entries. In the simplest case, each entry consists of the scalar data value v and the distance d of the fragment from the eye. This arrangement allows us to sort up to seven fragments in a single pass (six entries from the k-buffer plus the incoming fragment).

The fragment program is comprised of three stages (see Fig. 3 and the source code in the Appendices, which can be found on the Computer Society Digital Library at http://computer.org/tvcg/archives/htm). First, the program reads the accumulated color and opacity from the framebuffer. Program execution is terminated if the opacity is above a given threshold (early ray termination). Next, the program fetches the current k-buffer entries from the associated RGBA buffers and finds the two closest fragments to the eye by sorting the entries based on the stored distance d. For the incoming fragment, d is computed from its view-space position, which is calculated in a vertex program and passed to the fragment stage in one of the texture coordinate registers. The scalar values of the two closest entries and their distance is used to obtain the color and opacity of the ray segment defined by the two entries from the 3D preintegrated texture. Finally, the resulting color and opacity are composited with the color and opacity from the framebuffer, the closest fragment is discarded, and the remaining entries are written back into the k-buffer (see also Fig. 3). The complete pseudocode for our GPU fragment sorter ($k = 3$) is given below:

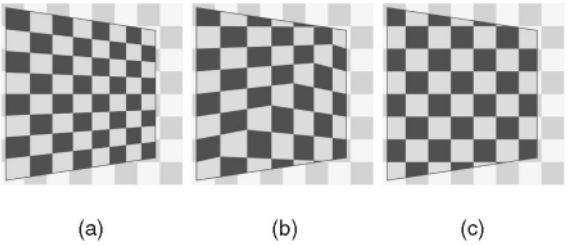

Fig. 4. Screen-space interpolation of texture coordinates. (a) The rasterizer interpolates vertex attributes in perspective space, which is typically used to map a 2D texture onto the faces of a 3D object. (b) Using the projected vertices of a primitive as texture coordinates to perform a lookup in a screen-space buffer yields incorrect results unless the primitive is parallel with the screen. (c) Computing the texture coordinates directly from the fragment window position or using projective texture mapping results in the desired screen-space lookup.

GPU-SORT
  for each fragment f_{new}
    Read color c_1 from framebuffer
    if c_1 is opaque **then**
      RETURN
    Read fragments f_1, f_2, and f_3 from k-buffer
    $n_1 \leftarrow$ closest fragment f_{new}, f_1, f_2, or f_3
    $(r_1, r_2, r_3) \leftarrow$ remaining fragments
    $n_2 \leftarrow$ closest fragment r_1, r_2, or r_3
    $d_1 \leftarrow$ depth of n_1, $d_2 \leftarrow$ depth of n_2
    $v_1 \leftarrow$ scalar of n_1, $v_2 \leftarrow$ scalar of n_2
    $\Delta d \leftarrow d_2 - d_1$
    Read c_2 from preintegrated table
      using v_1, v_2, and Δd
    Composite c_1 and c_2 into framebuffer
    Write r_1, r_2, and r_3 back into k-buffer

Several important details have to be considered for the hardware implementation of the algorithm. First, to look up values in a screen-space buffer, e.g., when compositing a primitive into a pixel buffer, previous implementations of volume rendering algorithms used the technique of projecting the vertices of the primitive to the screen, from which 2D texture coordinates are computed [32], [33]. As illustrated in Fig. 4, this approach produces incorrect results, unless the primitive is aligned with the screen, which happens only when view-aligned slicing is used to sample the volume. The reason for this problem is that the rasterization stage performs perspective-correct texture coordinate interpolation, which cannot be disabled on ATI cards [2]. Even if perspective-correct interpolation could be disabled, other quantities, e.g., the scalar data value, would still need to be interpolated in perspective space. Thus, to achieve the desired screen space lookup, one has to compute the texture coordinates from the fragment window position or use projective texture mapping [34]. Since projective texturing requires a division in the texture fetch stage of the pipeline, we decided to use the former solution in our implementation.

Second, strictly speaking, the result of simultaneously reading and writing a buffer is undefined when primitives

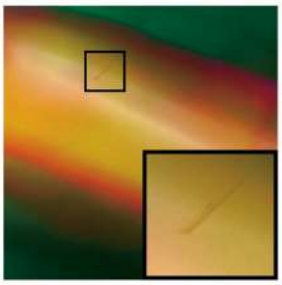

Fig. 5. Rendering artifacts resulting from the fragment-level race condition when simultaneously reading and writing the same buffer. In our experience, these artifacts are difficult to notice.

are composited on top of each other in the same rendering pass. The reason for the undefined output is that there is no memory access synchronization between primitives; therefore, a fragment in an early pipeline stage may not be able to access the result of a fragment at a later stage. Thus, when reading from a buffer for compositing, the result of the previous compositing step may not be in the buffer yet. Our experience is that the read-write race condition is not a problem as long as there is sufficient distance between fragments in the pipeline, which happens, e.g., when compositing slices in texture-based volume rendering applications [5]. Unfortunately, compositing triangles of varying sizes can yield artifacts, as shown by Fig. 5. One way to remedy this problem is to draw triangles in an order that maximizes the distance between fragments of overlapping primitives in the pipeline, e.g., by drawing the triangles in equidistant layers from the viewpoint. We advocate the addition of simultaneous read/write buffer access on future generation hardware to resolve this problem. We believe this feature will prove useful to a wide variety of GPU algorithms.

Third, to properly handle holes (concavities) in the data, vertices need to be tagged whether they belong to the boundary or not. Ray segments with both vertices on the boundary are assigned zero color and opacity. Unfortunately, this approach removes cells on the edges of the boundary as well. To solve this problem, a second tag is required that indicates whether a k-buffer entry is internal or external. This classification information is dynamically updated at every step such that, when the two closest entries are internal and the second closest entry is on the boundary, all k-buffer entries are marked external. When two external fragments are chosen as closest, the k-buffer entries are reversed to internal and the color and opacity from the preintegrated table is replaced with zero. Fortunately, these two tags can be stored as the signs of the scalar data value v and view distance d in the k-buffer. A further advantage of tagging fragments is that the classification allows for initializing and flushing the k-buffer by drawing screen aligned rectangles. Unfortunately, the number of instructions required to implement the logic for the two tags and to initialize and flush the buffer exceeds current hardware capabilities. Thus, currently, we use only a single tag in our implementation for initializing the k-buffer and do not handle holes in the data properly (the holes in Fig. 7b are visible because of the smaller number of fragments composited along the viewing rays going through them). Since the algorithm described above can handle holes properly, complete handling of holes will be added once next generation hardware becomes available.

TABLE 1
Analysis of Sorting Algorithms

Algorithm	Time
shellsort	584 ms
heapsort	507 ms
quicksort	281 ms
radixsort	64 ms

5 Experimental Results

Our implementation was tested on a PC with a 3.2 GHz Pentium 4 processor and 2,048 MB RAM running Windows XP. We used OpenGL in combination with an ATI Radeon 9800 Pro with 256 MB RAM. To assess the quality of our implementation, we ran extensive tests on several data sets to measure both the image quality and the interactive rendering rates.

5.1 CPU Sorting

We tested several commonly used sorting algorithms described in [35], [36] for generating nearly sorted sequences. Table 1 shows the performance results of various routines that sort an array of one million floating-point numbers. Given slight changes in the viewing direction, one approach would be to use an algorithm optimized for resorting previously sorted sequences (e.g., mergesort). We found, however, that resorting the face centroids using a faster sort is more efficient in practice because the ordering can change significantly from frame to frame.

In our implementation, we used an out-of-place sorting algorithm that achieves linear time complexity at the expense of auxiliary memory. The algorithm chosen was the Least Significant Digit (LSD) radix sort [36], which sorts numbers at individual digits one at time, from the least to the most significant one. As described, the algorithm does not work for floating-point numbers. However, floating-point numbers using the IEEE 754 standard (excluding NaN-values) are properly ordered if represented as signed magnitude integers. In order to use integer comparisons, a transformation is applied such that negative numbers are correctly handled. Discussion on the topic and several valid transformations are described in [37]. We used the following C++ function to convert floating-point numbers into 32-bit unsigned integers:

```
inline unsigned int float2fint(unsigned int f) {
  return f ^ ((-(f >> 31)) | 0x80000000);
}
```

Instead of performing radix sort individually at each bit, we worked on four blocks of 8-bits each. Sorting within each block uses a counting sort algorithm [36] that starts by counting the occurrences of the 256 different numbers in each 8-bit block. A single pass through the input suffices to count and store the ocurrences for all four blocks. The radix

TABLE 2
k-Buffer Analysis

Dataset	Fragments	Max A	Max k	$k>2$	$k>6$
f117	2,517,674	481	15	7632	71
kew	2,813,532	481	3	0	0
spx2	6,615,778	476	22	10,626	512
torso	7,223,435	649	15	43,317	1683
fighter	5,414,884	904	3	1	0

sort performs four passes through the input, each pass sorting numbers within a single block. Starting from the LSD block (0-7 bits), the counting results for that block are used to correctly position each number in an auxiliary array with the same size as the input. Once this block is sorted, a similar pass is issued to sort bits 8-15, this time using the auxiliary array as input and the original input array as output. Finally, two additional counting sorts are used to sort bits 16-23 and 24-31. Overall, five passes through the array are necessary to correctly sort the input, establishing the linear complexity.

Our code was written in C++ without any machine-level work, thus improvements can potentially be made to increase the performance of CPU sorting even further. In any case, our current sorting technique can sort upward of 15 million faces per second.

5.2 k-Buffer Analysis

As a measure of image quality, we implemented a software version of our algorithm that uses an A-buffer to compute the correct visibility order. As incoming fragments are processed, we insert them into the ordered A-buffer and record how deep the insertion was. This gives us a k size that is needed for the data set to produce accurate results. We also gain insight on how well our hardware implementation will behave for given k sizes. This analysis is shown in Table 2. For each data set, we show the number of total fragments generated when rendering them at 512^2 resolution, the maximum length of any A-buffer pixel list, the maximum k (i.e., the number of positions any fragment had to move to its correct position in the sorted order minus one for compositing), and the number of pixels that require k to be larger than two or six, which are the values currently supported by the hardware used. These results represent the *maximum* values computed from 14 fixed viewpoints on each data set.

Further analysis provides insight into the source of the problem. In particular, by generating an image for each fixed viewpoint of the data sets that reflect the distribution of the degeneracies, we can consider the distribution of the areas in which a small k size is not sufficient. Fig. 6 contains a sample of these images. This analysis shows that the problematic areas are usually caused by sliver cells, those which are large but thin (i.e., have a bad aspect ratio). This problem can be solved by finding the degenerate cells and subdividing them into smaller, more symmetric cells. Inspired by the regularity of Delaunay tetrahedralizations [38, Chapter 5], we tried to isolate these bad cells by analyzing how much they "differ" locally from a DT in the following sense: A basic property that characterizes DT is

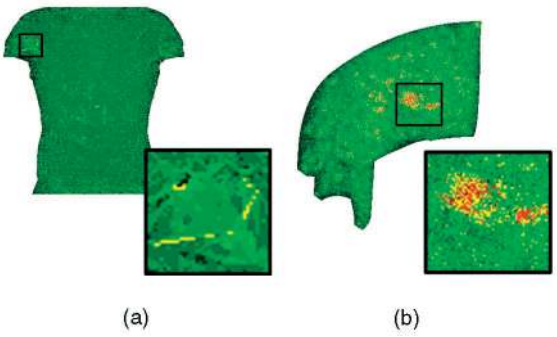

Fig. 6. Distribution of k requirements for the (a) torso and (b) spx2 data sets. Regions denote k size required to obtain a correct visibility sorting, for $k>6$ (red), $2<k\leq 6$ (yellow), and $k\leq 2$ (green).

the fact that a tetrahedron belongs to the DT of a point set if the circumsphere passing through the four vertices is empty, meaning no other point lies inside the circumsphere. By finding the degenerate cells of a data set that digress most from this optimal property and subdividing them, we can thereby lower the maximum k needed to accomplish a correct visibility ordering. Another approach is to perform mesh smoothing [39] operations on the data set. These operations attempt to eliminate cells with bad shape without creating additional tetrahedra. In our preliminary experiments, we were able to reduce the maximum k required to correctly render the f117 data set from 15 to 6 using these techniques.

Note that the artifacts caused by a limited k size in the implementation are hard to notice. First, they are less pronounced when using a transparent transfer function. Also, even in our worst example (torso), only 0.6 percent of the pixels *could* be incorrect. For a pixel to be incorrect, a somewhat complex combination of events needs to happen, it is not simply enough that the k-buffer ordering fails. Thus, users normally do not notice any artifacts when interacting with our system.

5.3 Render Performance

Table 3 and Table 4 show the performance of our hardware-assisted visibility sorting algorithm on several data sets using the average values of 14 fixed viewpoints. Table 3 reflects the GPU-based portion of the algorithm, which includes the time required to rasterize all the faces, run the fragment and vertex programs, composite the final image, and draw it to the screen using `glFinish`. Table 4 includes the time required to sort the faces on the CPU as well as the GPU with $k=2$. This represents the rendering rates

TABLE 3
Performance of the GPU Sorting and Drawing

		$k=2$		$k=6$	
Dataset	Cells	Fps	Tets/sec	Fps	Tets/sec
f117	240,122	9.71	2331 K	4.42	1062 K
kew	416,926	5.45	2267 K	3.75	1561 K
spx2	827,904	2.07	1712 K	1.70	1407 K
torso	1,082,723	3.13	3390 K	1.86	1977 K
fighter	1,403,504	2.41	3387 K	1.56	2190 K

TABLE 4
Total Performance of HAVS

Dataset	CPU	GPU	Total	Fps	Tets/sec
f117	45 ms	103 ms	148 ms	6.8	1622 K
kew	79 ms	188 ms	267 ms	3.7	1562 K
spx2	160 ms	368 ms	528 ms	1.9	1568 K
torso	210 ms	390 ms	600 ms	1.7	1805 K
fighter	268 ms	505 ms	773 ms	1.3	1816 K

achieved while interactively rotating and redrawing the data set. All rendering was done with a 512^2 viewport and a 128^3 8-bit RGBA preintegrated table. In addition, a low opacity colormap was used and early ray termination was disabled, thus every fragment is rasterized to give more accurate timings. With early ray termination enabled, we have been able to achieve over six million cells per second with the fighter data set using a high opacity colormap due to the speedup in fragment processing. In addition, we can improve the performance of our algorithm by exploiting pipeline parallelism between the CPU and GPU, i.e., sorting is performed on the CPU for the next frame while the current frame is being rendered by the GPU. Through the use of this parallelism, we have been able to effectively remove the CPU time and reduce the total rendering time to the GPU time.

Our technique requires no preprocessing, and it can be used for handling time-varying data. In addition, our implementation allows interactive changes to the transfer function. These operations are only dependent on the GPU for sorting and rendering (see Table 3), therefore, the CPU portion of the algorithm is not performed. The user interface consists of a direct manipulation widget that displays the user specified opacity map together with the currently loaded colormap (see Fig. 7 and Fig. 1). Modifying the opacity or loading a new colormap triggers a preintegrated table update which renders the data set using the GPU sort only. We found that, in general, a 128^3 preintegrated table is sufficient for high quality rendering.

5.4 Comparison

Table 5 compares the timing results of our algorithm with those of other methods. All results were generated using 14 fixed viewpoints and reflect the total time to sort and draw the data sets in a 512^2 window. We used an optimized version of the Shirley-Tuchman PT algorithm [6] implemented by Max et al. that uses the MPVO with nonconvexities (MPVONC) algorithm for visibility ordering [7]. The bottleneck of the PT algorithm is the time required to compute the polygonal decomposition necessary for rendering the tetrahedra. Another limitation is that the vertex information needs to be dynamically transferred to the GPU with every frame. We avoid this problem in our method because we can storing the vertices in a vertex array on the GPU. This difference results in similar GPU timings with the two methods even though we are using vertex and fragment programs. Wylie et al. [29] describe a GPU implementation of the PT algorithm, called GPU Accelerated Tetrahedra Rendering (GATOR), in which the tetrahedral decomposition is accomplished using a vertex shader. However, the complexity of this vertex shader limits the performance on current GPUs. The results generated for the GATOR method were accomplished using their code, which orders the tetrahedra using the MPVONC algorithm. Our rendering rates are much faster

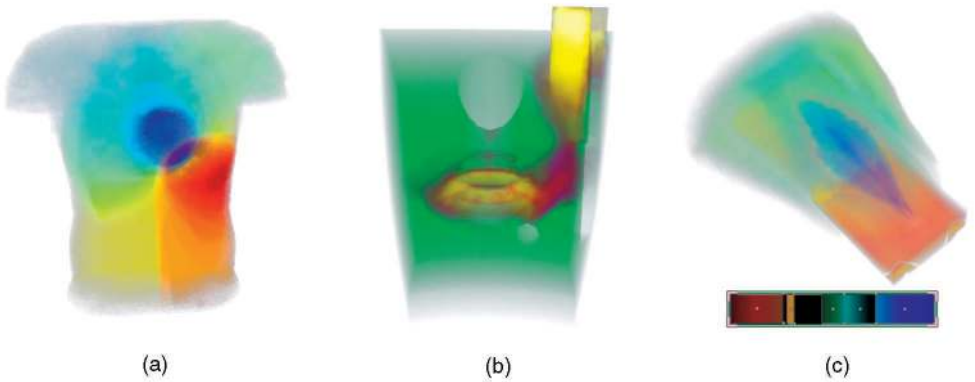

Fig. 7. Results of rendering the (a) torso, (b) spx, and (c) kew data sets with the HAVS algorithm.

TABLE 5
Time Comparison in Milliseconds with Other Algorithms

	f117			spx2			fighter		
Algorithm	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU	Total
HAVS	45	103	148	160	368	528	268	505	773
HW Ray Caster	N/A	498	498	N/A	1410	1410	N/A	N/A	N/A
PT	195	103	298	655	326	981	1799	664	2463
GATOR	85	185	270	276	702	978	861	1314	2175
ZSWEEP	3278	N/A	3278	10119	N/A	10119	12556	N/A	12556

then these PT methods, while producing higher quality images through the use of a 3D preintegrated table. Another recent technique is the GPU-based ray casting of Weiler et al. [13], [40]. The image quality of this technique is similar to that of our work, but the algorithm has certain limitations on the size of the data sets that make it less general than cell-projection techniques. In fact, the fighter data set we used for comparison could not be loaded properly due to hardware memory limitations. The results for this algorithm were generated using a DirectX-based ray caster described in Bernardon et al. [41], which is approximately twice as fast as the original technique reported by Weiler et al.. The ZSWEEP method [14] uses a hybrid image and object-space sorting approach similar to our sorting network, but does not leverage the GPU for better performance. The code that was used in this portion of our timing results was provided by Ricardo Farias. All of these approaches require a substantial amount of connectivity information for rendering, resulting in a higher memory overhead than our work. Another advantage of our algorithm is the simplicity of implementation in software and hardware. The software techniques described above (PT and ZSWEEP) require a substantial amount of code for sorting and cell projection. Implementations of the hardware techniques (GATOR and HW Ray Caster) involve developing long and complex fragment and vertex programs which can be difficult to write and debug.

6 Discussion

When we started this work, we were quite skeptical about the possibility of implementing the k-buffer on current GPUs. There were several hurdles to overcome. First, given the potentially unbounded size of pixel lists, it was less than obvious to us that small values of k would suffice for large data sets. Another major problem was the fact that reading and writing to the same texture is not a well-defined operation on current GPUs. We were pleasantly surprised to find that, even on current hardware, we get only minor artifacts.

There are several issues that we have not studied in depth. The most important goal is to develop techniques that can refine data sets to respect a given k. Currently, our experiments show that, when the k-buffer is not large enough, a few pixels are rendered incorrectly. So far, we have found that most of our computational data sets are well-behaved and the wrong pixels have no major effect on image quality. In a practical implementation, one could consider raising the value of k or increasing the accuracy of the object-space visibility sorting algorithm once the user stops rotating the model. Using the smallest possible k is required for efficiency.

Some of our speed limitations originate from limitations of current GPUs. In particular, the lack of real conditionals forces us to make a large number of texture lookups that we can potentially avoid when next generation hardware is released. Furthermore, the limit on the instruction count has forced us into an incorrect hole handling method. With more instructions, we could also incorporate shaded isosurface rendering without much difficulty.

Finally, there is plenty of interesting theoretical work remaining to be done. It would be advantageous to develop input and output sensitive algorithms for determining the object-space ordering and estimation of the minimum k size for a given data set. We have preliminary evidence that, by making the primitives more uniform in size, k can be lowered. We believe it might be possible to formalize these notions and perform proofs along the lines of the work of Mitchell et al. [42] and de Berg et al. [43].

7 Conclusions and Future Work

In this paper, we presented a novel algorithm for volume rendering unstructured data sets. Our algorithm exploits the CPU and GPU for sorting both in object-space and image-space. We use the CPU to compute a partial ordering of the primitives for generating a nearly sorted fragment stream. We then use the k-buffer, a fragment-stream sorter of constant depth, on the GPU for complete sorting on a per-fragment basis. Despite limitations of current GPUs, we show how to implement the k-buffer efficiently on an ATI Radeon 9800.

Our technique can handle arbitrary nonconvex meshes with very low memory overhead. Similarly to the HW-based ray caster, we use a floating-point-based framebuffer that minimizes rendering artifacts and we can easily handle both parallel and perspective projections. But, unlike those techniques, the maximum data size is bounded only by the available main memory of the system. In our paper, we provide a comparison of our technique with previous algorithms for rendering unstructured volumetric data and enumerate the advantages in speed, ease of implementation, and adaptability that our work provides.

We would like to reemphasize the simplicity of our technique. At the same time that our technique is shown to be faster and more general than others, the implementation is very compact and easy to code. In fact, the rendering code in our system consists of less than 200 lines. We believe these qualities are likely to make it the method of choice for rendering unstructured volumetric data.

There are several interesting areas for future work. Further experiments and code optimization are necessary for achieving even faster rendering rates. In particular, we hope that next-generation hardware will ease some of the current limitations and will allow us to implement sorting networks with larger k sizes. Real fragment program conditionals will allow us to reduce the effective number of texture lookups. On next generation hardware, we will also be able to implement a more efficient early ray termination strategy. Another interesting area for future research is rendering dynamic meshes. We intend to explore techniques that do not require any preprocessing and can be used for handling dynamic data. Finally, we would like to devise a theoretical framework for analyzing the direct trade-off between the amount of time spent sorting in object-space and image-space.

ACKNOWLEDGMENTS

The authors thank Joe Kniss for suggesting that the k-buffer could be implemented efficiently on ATI hardware. They thank Ricardo Farias for the ZSWEEP code and the insightful discussions that helped shape many ideas presented in this paper. Carlos Scheidegger and Huy Vo provided invaluable code contributions. In particular, Vo wrote the fast radix sort used in their system. They thank Fábio Bernardon for the use of his HW Ray Caster code. The Teem toolkit [44] proved very useful for processing the data sets and results. They thank Mark Segal from ATI for the donated hardware and his prompt answers to their questions. They are grateful to Patricia Crossno, Shachar Fleishman, Nelson Max, and Peter Shirley for helpful suggestions and criticism. The authors also acknowledge Bruce Hopenfeld and Robert MacLeod (University of Utah) for the heart data set, Bruno Notrosso (Electricite de France) for the spx data set, Hung and Buning (NASA) for the blunt fin data set, and Neely and Batina (NASA) for the fighter data set. Steven P. Callahan is supported by the US Department of Energy (DOE) under the VIEWS program. Milan Ikits is supported by the US National Science Foundation (NSF) grant ACI-0078063 and the DOE Advanced Visualization Technology Center. Cláudio T. Silva is partially supported by the DOE under the VIEWS program and the MICS office, the NSF under grants CCF-0401498, EIA-0323604, and OISE-0405402, and a University of Utah Seed Grant. The work of João L.D. Comba is supported by a CNPq grant 540414/01-8 and FAPERGS grant 01/0547.3.

REFERENCES

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

[2] ATI, "Radeon 9500/9600/9700/9800 OpenGL Programming and Optimization Guide," 2003, http://www.ati.com.

[3] S. Roettger, M. Kraus, and T. Ertl, "Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection," *Proc. IEEE Visualization,* pp. 109-116, Oct. 2000.

[4] S. Guthe, S. Roettger, A. Schieber, W. Straßer, and T. Ertl, "High-Quality Unstructured Volume Rendering on the PC Platform," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware,* pp. 119-126, Sept. 2002.

[5] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics,* pp. 667-692, Addison Wesley, 2004.

[6] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Proc. San Diego Workshop Volume Visualization,* vol. 24, no. 5, pp. 63-70, Nov. 1990.

[7] P.L. Williams, "Visibility-Ordering Meshed Polyhedra," *ACM Trans. Graphics,* vol. 11, no. 2, pp. 103-126, Apr. 1992.

[8] J. Comba, J.T. Klosowski, N. Max, J.S.B. Mitchell, C.T. Silva, and P.L. Williams, "Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids," *Computer Graphics Forum,* vol. 18, no. 3, pp. 369-376, Sept. 1999.

[9] C.T. Silva, J.S. Mitchell, and P.L. Williams, "An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes," *Proc. IEEE Symp. Volume Visualization,* pp. 87-94, Oct. 1998.

[10] M. Kraus and T. Ertl, "Cell-Projection of Cyclic Meshes," *Proc. IEEE Visualization,* pp. 215-222, Oct. 2001.

[11] R. Cook, N. Max, C. Silva, and P. Williams, "Efficient, Exact Visibility Ordering of Unstructured Meshes," *IEEE Trans. Visualization and Computer Graphics,* vol. 1, no. 6, pp. 695-707, Nov./Dec. 2004.

[12] M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-Based Ray Casting for Tetrahedral Meshes," *Proc. IEEE Visualization,* pp. 333-340, Oct. 2003.

[13] M. Weiler, P.N. Mallón, M. Kraus, and T. Ertl, "Texture-Encoded Tetrahedral Strips," *Proc. Symp. Volume Visualization 2004,* pp. 71-78, 2004.

[14] R. Farias, J. Mitchell, and C.T. Silva, "ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Proc. IEEE Volume Visualization and Graphics Symp.*, pp. 91-99, 2000.

[15] L. Carpenter, "The A-Buffer, an Antialiased Hidden Surface Method," *Computer Graphics (Proc. SIGGRAPH 84),* vol. 18, no. 3, pp. 103-108, July 1984.

[16] L. Guibas, "Computational Geometry and Visualization: Problems at the Interface," *Scientific Visualization of Physical Phenomena,* N.M. Patrikalakis, ed., pp. 45-59, Springer-Verlag, 1991.

[17] N.L. Max, "Sorting for Polyhedron Compositing," *Focus on Scientific Visualization,* pp. 259-268, Springer-Verlag, 1993.

[18] R. Farias and C.T. Silva, "Out-of-Core Rendering of Large, Unstructured Grids," *IEEE Computer Graphics and Applications,* vol. 21, no. 4, pp. 42-51, July/Aug. 2001.

[19] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys,* vol. 6, no. 1, pp. 1-55, Mar. 1974.

[20] M. Newell, R. Newell, and T. Sancha, "A Solution to the Hidden Surface Problem," *Proc. ACM Ann. Conf.,* pp. 443-450, 1972.

[21] C. Stein, B. Becker, and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Proc. IEEE Symp. Volume Visualization,* pp. 83-89, Oct. 1994.

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

[23] C. Wittenbrink, "R-Buffer: A Pointerless A-Buffer Hardware Architecture," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware,* pp. 73-80, 2001.

[24] A. Mammen, "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique," *IEEE Computer Graphics and Applications,* vol. 9, pp. 43-55, July 1984.

[25] C. Everitt, "Interactive Order-Independent Transparency," NVIDIA, technical report, 2001, http://developer.nvidia.com.

[26] N.P. Jouppi and C.-F. Chang, "Z3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware,* pp. 85-93, Aug. 1999.

[27] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: A GPU-Based Particle Engine," *Eurographics Symp. Proc. Graphics Hardware 2004,* pp. 115-122, 2004.

[28] S. Roettger and T. Ertl, "Cell Projection of Convex Polyhedra," *Proc. 2003 Eurographics/IEEE TVCG Workshop Volume Graphics,* pp. 103-107, 2003.

[29] B. Wylie, K. Moreland, L.A. Fisk, and P. Crossno, "Tetrahedral Projection Using Vertex Shaders," *Proc. IEEE/ACM Symp. Volume Graphics and Visualization,* pp. 7-12, 2002.

[30] T. Aila, V. Miettinen, and P. Nordlund, "Delay Streams for Graphics Hardware," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 792-800, July 2003.

[31] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys,* vol. 16, no. 2, pp. 187-260, 1984.

[32] J.M. Kniss, S. Premože, C.D. Hansen, P. Shirley, and A. McPherson, "A Model for Volume Lighting and Modeling," *IEEE Trans. Visualization and Computer Graphics,* vol. 9, no. 2, pp. 150-162, 2003.

[33] J. Krüger and R. Westermann, "Acceleration Techniques for GPU-Based Volume Rendering," *Proc. IEEE Visualization,* pp. 287-292, 2003.

[34] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping," *Proc. ACM SIGGRAPH,* pp. 249-252, July 1992.

[35] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed., pp. 40, 127-173. McGraw-Hill, 2001.

[36] R. Sedgewick, *Algorithms in C,* third ed., pp. 298-301, 403-437. Addison-Wesley, 1998.

[37] H. Warren Jr., *Hacker's Delight,* pp. 261-265. Addison-Wesley, 2002.

[38] H. Edelsbrunner, *Geometry and Topology for Mesh Generation.* Cambridge Univ. Press, 2001.

[39] L. Freitag, P. Knupp, T. Munson, and S. Shontz, "A Comparison of Optimization Software for Mesh Shape-Quality Improvement Problems," *Proc. 11th Int'l Meshing Roundtable,* pp. 29-40, 2002.
[40] M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-Based View-Independent Cell Projection," *IEEE Trans. Visualization and Computer Graphics,* vol. 9, no. 2, pp. 163-175, 2003.
[41] F.F. Bernardon, C.A. Pagot, J.L.D. Comba, and C.T. Silva, "GPU-Based Tile Ray Casting Using Depth Peeling," Technical Report UUSCI-2004-006, SCI Inst., 2004.
[42] J.S.B. Mitchell, D.M. Mount, and S. Suri, "Query-Sensitive Ray Shooting," *Int'l J. Computational Geometry and Applications,* vol. 7, no. 4, pp. 317-347, Aug. 1997.
[43] M. de Berg, M.J. Katz, A.F. van der Stappen, and J. Vleugels, "Realistic Input Models for Geometric Algorithms," *Proc. Ann. Symp. Computational Geometry,* pp. 294-303, 1997.
[44] G.L. Kindlmann, "The Teem Toolkit," 2003, http://teem.sourceforge.net.

Steven P. Callahan received the BS degree in computer science from Utah State University in 2002. He is currently pursuing the MS degree in computational engineering and science at the University of Utah, where he works as a research assistant in the Scientific Computing and Imaging Institute. His research interests include graphics, visualization, and large-scale scientific computing.

Milan Ikits is a PhD candidate in the School of Computing at the University of Utah. His research interests lie in the areas of computer graphics, scientific visualization, immersive environments, and human-computer interaction. He received the Diploma in computer science from the Budapest University of Technology and Economics in 1997. He joined Immersion Medical in 2004, where he has been involved with the company's research and development efforts toward advancing the state-of-the-art in surgical simulation. He has published several papers and book chapters on using immersive environments for scientific visualization. He is also the creator of the popular OpenGL Extension Wrangler library. He is a student member of the IEEE.

João L.D. Comba received the Bachelor's degree in computer science from the Federal University of Rio Grande do Sul, Brazil, the MS degree in computer science from the Federal University of Rio de Janeiro, Brazil, and the PhD degree in computer science from Stanford University. He is an associate professor of computer science at the Federal University of Rio Grande do Sul, Brazil. His main research interests are in graphics, visualization, spatial data structures, and applied computational geometry. His current projects include the development of algorithms for large-scale scientific visualization, data structures for point-based modeling and rendering, and general-purpose computing using graphics hardware. He is a member of the ACM Siggraph.

Cláudio T. Silva received the Bachelor's degree in mathematics from the Federal University of Ceara, Brazil, and the MS and PhD degrees in computer science from the State University of New York at Stony Brook. He is an associate professor of computer science at the University of Utah and a member of the Scientific Computing and Imaging (SCI) Institute. His main research interests are in graphics, visualization, applied computational geometry, bioinformatics, and high-performance computing. His current projects include the development of out-of-core and streaming algorithms for large-scale scientific visualization, techniques for point-based modeling and rendering, and efficient algorithms for modern graphics hardware. He has published more than 60 publications in international conferences and journals, holds five US patents, and presented tutorials at ACM SIGGRAPH, Eurographics, and IEEE Visualization conferences. He serves on numerous program committees and is papers cochair of the IEEE Visualization 2005 conference. He is a member of the ACM, Eurographics, and IEEE.