SCI INSTITUTE
TECHNICAL REPORT

Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering

Steven P. Callahan, Milan Ikits, Joao L. D. Comba, Claudio T. Silva

UUSCI-2004-003

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

June 14, 2004

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. One major hurdle is that computing the volume rendering equation requires strict ordering of the fragments. While it is quite simple to generate the exact order for regular grids, the same computation is much more complex for unstructured meshes. Typically, relatively complex and costly object-space sorting techniques are needed to ensure that fragments are generated in sorted order for compositing by the GPU.
We propose a novel 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 objectspace and image-space. In object-space, the algorithm performs a partial sorting of the 3D primitives in preparation for rasterization. The goal of the partial sorting 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.

THE UNIVERSITY OF UTAH

Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering

Steven P. Callahan [1] Milan Ikits [1] João L. D. Comba [2] Cláudio T. Silva [1]

[1] Scientific Computing and Imaging Institute, University of Utah
[2] Federal University of Rio Grande do Sul, Brazil

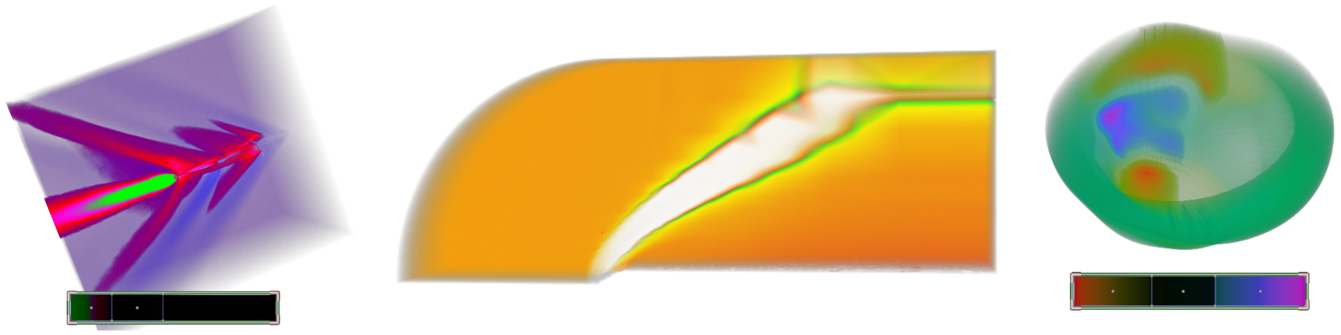
Figure 1: Results of the hardware-assisted visibility sorting algorithm (HAVS) for the fighter, blunt fin, and heart datasets. When running on a Pentium 4 with an ATI Radeon 9800 in a 512^2 viewport, the algorithm achieves rendering rates between one and six million cells per second for large datasets. HAVS can handle arbitrary non-convex meshes with very low memory overhead, and requires only minimal and completely automatic preprocessing of the data. Maximum data size is bounded by the available main memory of the system. The 3D pre-integrated transfer function lookup table needed for accurate rendering is built entirely on the GPU, allowing users to interactively design transfer functions with the user interface.

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.

One major hurdle is that computing the volume rendering equation requires strict ordering of the fragments. While it is quite simple to generate the exact order for regular grids, the same computation is much more complex for unstructured meshes. Typically, relatively complex and costly object-space sorting techniques are needed to ensure that fragments are generated in sorted order for compositing by the GPU.

We propose a novel 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 sorting of the 3D primitives in preparation for rasterization. The goal of the partial sorting 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.

[1] {stevec,ikits,csilva}@sci.utah.edu
[2] comba@inf.ufrgs.br

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 datasets. In volume rendering, the scalar field is modeled as a cloud-like material that both emits and attenuates light along the viewing direction [23]. To create an image, the equations for the optical model must be integrated along the viewing ray for each pixel. 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.

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 datasets, 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 [14, 27]. 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

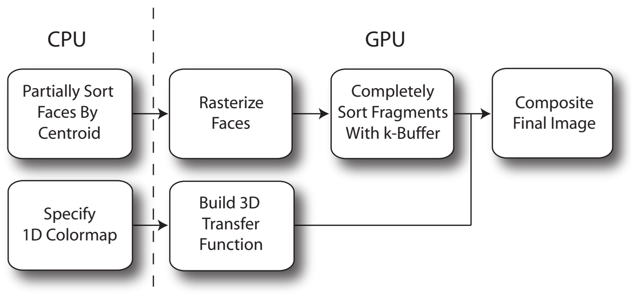
CPU
GPU
Partially Sort Faces By Centroid
Rasterize Faces
Completely Sort Fragments With k-Buffer
Composite Final Image
Specify 1D Colormap
Build 3D Transfer Function

Figure 2: Overview of the hardware-assisted visibility sorting algorithm. 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. The 3D pre-integrated transfer function table is updated entirely on the GPU using incremental computations.

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 they are generated. For details on performing these computations, see [15].

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 [30] 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 cell. For rendering meshes, cells have to be projected in visibility order, which can be accomplished using techniques such as the MPVO cell-sorting algorithm [36]. 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 [5, 6, 19, 31].

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). On modern GPUs, it is possible to render regular volumes at very high frame rates. Unfortunately, performing visibility ordering for unstructured grids completely in object space has turned out to be quite expensive and complex [6].

We build on the previous work of Farias *et al.* [10], Jouppi and Chang [16], and Carpenter [4], 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 (see Figure 2).

- We show how to efficiently implement the *k*-buffer using the programmable functionality of existing GPUs.

- We perform detailed experimental analysis to evaluate the performance of our algorithm using several datasets, the largest of which having over 1.4 million cells. The experiments show that our algorithm can handle general non-convex 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 unstructured datasets.

- We build the 3D pre-integrated transfer function lookup table from the user specified 1D colormap entirely on the GPU using incremental computations [34], achieving highly interactive update rates ($>$ 10 fps).

The remainder of this paper is organized as follows. We summarize related work in Section 2. In Section 3, we describe our algorithm, provide the definition of *k*-nearly sorted sequences, and further details on the functionality of the *k*-buffer. In Section 4, we describe how to efficiently implement the *k*-buffer using the programmable features of current ATI hardware. Section 5 describes our experiments and Section 6 presents our results. In Section 7, we discuss different trade-offs of our approach. Finally, in Section 8, 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 treatment of previous work in [6, 11, 13, 15, 22]. We limit our coverage to the most directly related work in visibility ordering in both software and hardware.

In computer graphics, work on visibility ordering was pioneered by Schumacker *et al.* and is later reviewed in [33]. An early solution to computing a visibility order was given by Newell, Newell, and Sancha (NNS) [25], which continues to be the basis for more recent techniques [32]. 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, Kedem, and Naylor [12] 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 v of a BSP-tree $\mathscr{T}$ corresponds to a convex polyhedral region, $P(v) \subset \mathbb{R}^3$, and the root node corresponds to all of $\mathbb{R}^3$. Each non-leaf node v is defined by a hyperplane, $h(v)$ that partitions $P(v)$ into two half-spaces, $P(v^+) = h^+(v) \cap P(v)$ and $P(v^-) = h^-(v) \cap P(v)$, corresponding to the two children, v^+ and v^- of v. Here, $h^+(v)$ ($h^-(v)$) is the half-space of points above (below) the plane $h(v)$. Fuchs *et al.* [12] 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 $h^+(v)$, we can recursively draw the fragments stored in the leaves of the subtree rooted at v^-, followed by the object fragments $S(v) \subset h(v)$. Finally, we recursively draw the fragments stored in the leaves of the subtree rooted at v^+. Note that the BSP-tree does not actually generate a visibility order for the original primitives, but for *fragments* of them.

The work presented above operates in *object-space*, *i.e.*, they operate on the primitives before they are rasterized by the graphics hardware [2]. Carpenter [4] 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 [37] is a pointerless A-buffer hardware architecture that implements a method similar to a software algorithm described in [21] for sorting transparent fragments in front of the front-most opaque fragment. Current hardware implementations of this technique require multiple passes through the polygons in the scene [9]. 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 [16] is an alternative design which devotes a fixed amount of memory to the A-buffer. When there are more fragments generated for a pixel than what the available memory can hold, Z^3 merges the extra fragments.

Hardware-accelerated unstructured volume rendering has seen a number of recent advances. Recently, Wylie *et al.* has shown how to implement the Shirley-Tuchman tetrahedron projection directly on the GPU [38]. As mentioned before, the PT projection sorts fragments for a single tetrahedron only and still requires that the cells are 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 for convex meshes by Weiler *et al.* [34]. For non-convex meshes the authors rely on a manual convexification algorithm, *i.e.*, editing a non-convex mesh by adding extra primitives until the boundary of the resulting mesh is convex and the mesh contains no holes.

Roughly speaking, all the work described above performs 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 [10] 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 sorting, 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 [1], where the authors show how to improve the efficiency of the R-buffer by shifting some of the work to object-space.

3 HARDWARE-ASSISTED VISIBILITY SORTING

Our hardware-assisted visibility sorting (HAVS) algorithm 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 available CPU and GPU on current hardware, where the object-space work is performed by the CPU while the fragment-level sorting is implemented completely on the GPU (see Figure 2). Depending on the relative speed of the CPU/GPU, it is possible to shift work from one processor to the other by varying the accuracy of the different 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 rest of the section, we concentrate on presenting further details of the two phases of HAVS.

3.1 Nearly-sorted object-space visibility ordering

Visibility ordering algorithms (*e.g.* , XMPVO [31]) sort 3D primitives with respect to a given viewpoint v in *exact* order, which allows for direct blending and compositing of the rasterized fragments. In our work, we differentiate the sorting of the 3D primitives and the rasterized fragments to utilize faster object-space sorting algorithms.

To 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 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 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 the distances of the fragments 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 fragment processing.

Several techniques implicitly generate nearly sorted sequences. For example, several hierarchical spatial data structures provide mechanisms for simple and efficient back-to-front traversal [28]. A simple way of generating nearly-sorted object-space visibility ordering of a collection of 3D primitives is to use a BSP-tree. 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 datasets 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 [10]. 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 space explosion in the A-buffer, ZSWEEP uses a *conservative* technique for com-

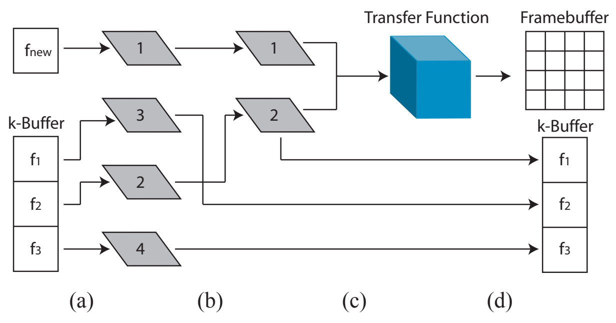

Figure 3: Example of the k-buffer with $k = 3$ (see also Section 4). (a) We start with the incoming fragment and the current k-buffer entries and (b) find the two entries closest to the viewpoint. (c) Next, we use the scalar values (v_1, v_2) and view distances (d_1, d_2) of the two closest entries to look up the corresponding color and opacity in the transfer function 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.

positing samples [10]. In our approach, we apply a more *aggressive* technique for managing the A-buffer.

3.2 The *k*-buffer

The original A-buffer [4] stores all incoming fragments in a list, which requires a potentially unbounded amount of memory. Our approach is closely related to the Z^3 architecture [16], which stores only a fixed number of fragments and works by combining the current fragments and discarding some of them as new fragments arrive. The k-buffer operates in a similar fashion 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 k entries $< f_1, f_2, \ldots, f_k >$. 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, 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, a k-buffer of size $k+1$ is required, because the first two entries in the buffer (f_1 and f_2) need to be in sorted order. In practice, the hardware implementation is simplified by keeping the k-buffer entries unsorted (see Figure 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 grid on the CPU. To properly handle boundaries, the vertices are marked whether they are internal or external. In the k-buffer, we store both the scalar value and view distance for each fragment generated by rasterizing the faces. When a fragment is discarded from the k-buffer, we look up the color and opacity of the ray segment defined by the closest and second closest entries from a 3D pre-integrated lookup table and composite them with the accumulated color and opacity in the framebuffer (see Figure 2).

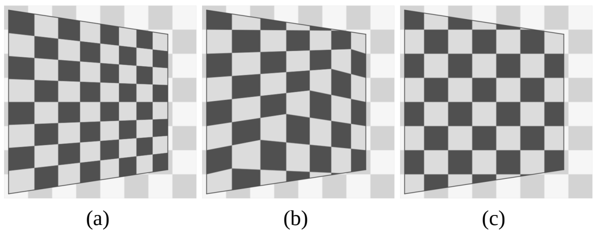

Figure 4: Screen-space interpolation of texture coordinates. (a) The rasterizer interpolates vertex attributes in perspective space, which is typically used to map a 2D texture onto the faces of a 3D object. (b) Using the projected vertices of a primitive as texture coordinates to perform a lookup in 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.

4 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 comprises three stages (see Figure 3 and the source code in the supplementary document). 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). Unfortunately, we cannot exploit the early z-test on current generation hardware, because depth write is not possible with multiple floating-point render targets. This feature will be available on next generation hardware [3]. 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 pre-integrated transfer function 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 Figure 3).

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 [18, 20]. As illustrated in Figure 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, still would 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 [29]. 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 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 [15]. Unfortunately, compositing triangles of varying sizes can yield artifacts, as shown by Figure 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. Temporary solutions such as this can remove many of the artifacts until simultaneous read/write buffer access is available on future generation hardware.

Third, to properly handle holes 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 transfer function 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 Figure 7(b) 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.

4.1 Transfer Function Update

Interactive transfer function design is an important component of modern volume visualization systems. Computing the 3D transfer function lookup table required for accurate volume rendering of unstructured meshes is time-consuming, because numerical integration of the whole table is of $O(n^4)$ complexity [26]. Incremental computation reduces the complexity to $O(n^3)$ and significantly speeds up the construction of the table [34]. Previous unstructured volume rendering implementations either computed each slice of the 3D pre-integrated texture by numerical integration on the GPU [26], or performed incremental pre-integration on the CPU [34]. Both approaches, however, provide only limited interactivity (~ 5 s/update and ~ 1.5 s/update for $n = 128$, respectively). In the former case, numerical integration for each slice requires compositing a large number of screen aligned rectangles into the framebuffer, which is the same complexity as volume rendering the whole table with a very high sampling rate. In contrast, the latter approach, although it requires much fewer operations, does not exploit the inherent parallelism of GPU computations. In addition, the lookup table from the CPU side has to be downloaded to GPU memory after every update, which can be an expensive operation that further reduces the achievable level of interactivity.

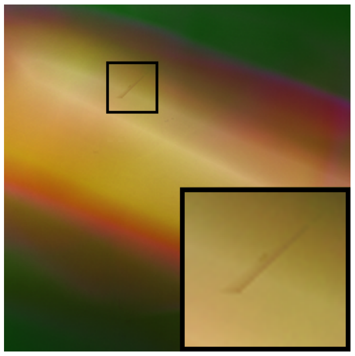
Figure 5: Rendering artifacts resulting from the fragment level race condition when simultaneously reading and writing the same buffer. These artifacts are generally less noticeable than the artifacts due to the limited k-buffer size in the implementation.

In this paper we combine the advantages of both approaches and compute the 3D lookup table using incremental pre-integration on the GPU. Thus, the table is immediately available for rendering after it is updated. The update procedure is divided into two parts. First, we compute the *base* slice of the 3D table using a modified version of the algorithm presented in [26]. The difference is that we interpolate opacity-corrected associated colors instead of chromacities and use front-to-back compositing. A floating-point pixel buffer is used to minimize round-off errors during the computation. Note, that linear interpolation of the colormap entries has to be performed explicitly in the fragment program, since current generation hardware supports only nearest neighbor interpolation for floating point texture lookups.

In the second stage, we incrementally compute each remaining slice using the base slice and the result from the previous step. To maximize performance, we use two sub-buffers (current and previous) of a single pixel buffer and alternate their role after each step (ping-pong rendering). Before the current slice becomes the previous slice, its contents are copied to the corresponding location in the 3D lookup table texture. Note that the slices are computed at floating point accuracy, but are quantized to 8 or 16 bits during the copy.

5 Experiments

In our system, we do not use the object-space visibility ordering scheme that provides a limit on k as described in Section 3.1. Instead, we implemented two simple sorting schemes based on either sorting faces by their centroid, or the order induced by the front-to-back ordering of the first vertex of a face. To assess the quality of these heuristics and determine the required k-buffer size for a given dataset, we ran extensive tests on several datasets. 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 dataset to produce accurate results. We also gain an insight on how well our hardware implementation will do for given k sizes. This analysis is shown in Table 1. These results represent the *maximum* values computed from fourteen fixed viewpoints on each dataset.

Dataset	Fragments	Max A	Max k	$k > 2$	$k > 6$
kew	2,634,400	481	2	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

Table 1: Analysis of k-buffer accuracy for given datasets. For each dataset, 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 our hardware implemention on the ATI Radeon 9800.

However, the results of Table 1 alone do not completely describe the error in using a small k. In addition, it is necessary to consider the distribution of these areas in which a small k size is not sufficient. This is done by generating a set of images for each fixed viewpoint of a dataset that reflect the distribution of the degeneracies. Figure 6 contains a sample of these images. This analysis shows that the problematic areas are usually caused by degenerate cells, those which are large but thin, *i.e.*, have a bad aspect ratio. We believe 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 [8, 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 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 dataset that digress most from this optimal property, and subdividing them, we can thereby lower the maximum k needed to accomplish a correct visibility ordering. Note that the artifacts caused by a limited k size in the implementation are less noticeable when using a transparent transfer function. Thus, users normally do not notice these artifacts when interacting with our system.

6 Results

Our implementation was tested on a PC with a 3.0 GHz Pentium 4 processor and 1024 MB RAM running Windows XP. We used OpenGL in combination with an ATI Radeon 9800 Pro in order to take advantage of multiple render targets. Table 2 shows the performance of our hardware-assisted visibility sorting algorithm on several datasets using the average values of fourteen fixed viewpoints. The results reflect the GPU-based final sorting and do not include the partial ordering of the faces done on the CPU. This 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`. All rendering was done with a 512^2 viewport and a 128^2 8-bit RGBA transfer function. These numbers represent the time required to render the datasets with a low opacity colormap. Due to the speedup in fragment processing while using early ray termination, we have been able to achieve over six million cells per second with the fighter dataset using a high opacity colormap.

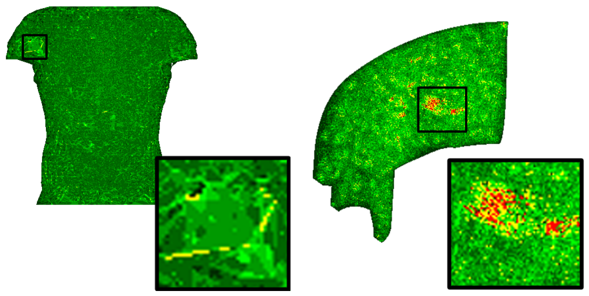

Figure 6: Distribution of k requirements for the (a) torso and (b) spx2 datasets. Regions denote k size required to obtain a correct visibility sorting, for $k > 6$ (red), $2 < k \leq 6$ (yellow), and $k \leq 2$ (green).

		$k = 2$		$k = 6$	
Dataset	Cells	Fps	Tets/sec	Fps	Tets/sec
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 2: Performance of the GPU sorting of our algorithm. The results show average values for fourteen fixed viewpoints.

Though our main focus was not to optimize the partial sort done on the CPU, the interactivity of rotating the volume depends on this step. We briefly attempted two different methods for sorting the faces. First, we sorted by face centroid to obtain a more correct visibility ordering, and second we sorted by vertex for better performance. The image quality achieved by the centroid sort is superior to the vertex sort for smaller k sizes, but the slowdown is substantial. The time required to do a complete re-sort of all the faces for every view change ranges from ~ 0.02 to ~ 2.5 seconds for the given datasets. Though the decrease in frame rate for small datasets is not considerable, it is much more costly for the larger datasets. Sorting by vertex yields results in the range of ~ 0.005 to ~ 1.1 seconds. This increase is due to the large disparity in the number of vertices and the number of faces in the datasets. A hybrid approach to achieve optimal frame rates from these methods would be to sort by vertex during rotation and to sort by face otherwise.

Size	Base	Incremental	Total
$64 \times 64 \times 128$	9.3 ms	19 ms	28.3 ms
$128 \times 128 \times 128$	35.6 ms	31.6 ms	67.2 ms
$128 \times 128 \times 256$	35.6 ms	63.2 ms	98.8 ms

Table 3: Timing results of incremental pre-integration of the transfer function on the GPU (in milliseconds).

In addition to displaying large datasets, our implementation allows interactive changes to the transfer function. The user interface consists of a direct manipulation widget that displays the user specified opacity map together with the currently loaded colormap (see Figure 1 and Figure 7). Modifying the opacity or loading a new colormap triggers a transfer function update on the GPU. The performance of hardware-assisted incremental pre-integration is given in Table 3. The results represent the time required to regenerate an 8-bit RGBA texture. In general, a 128^3 transfer function is sufficient for high quality rendering.

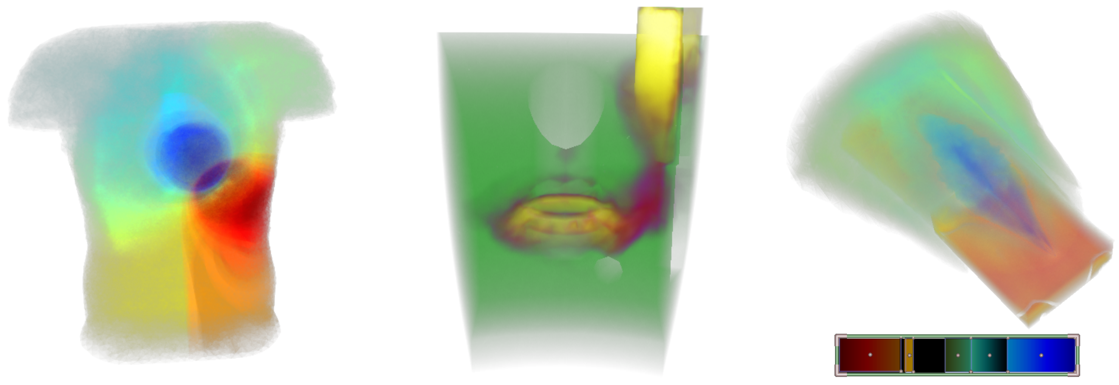
(a) (b) (c)

Figure 7: Results of rendering the (a) torso (b) spx and (c) kew datasets with the HAVS algorithm.

7 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 datasets. 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. Finally, we thought that the GPU would be the main bottleneck during rendering. Hence, in our prototype implementation, we did not spend much time optimizing the CPU sorting algorithm. This was a mistake, because the ATI Radeon 9800 has been able to render over six million cells per second when sorting on the CPU is not taken into account.

There are several issues that we have not studied in depth. The most important goal is to develop techniques that can refine datasets 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 datasets 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 dataset. 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.* [24] and de Berg *et al.* [7].

8 Conclusions and Future Work

In this paper we presented a novel algorithm for volume rendering unstructured data. 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 non-convex meshes with very low memory overhead and requires only minimal and completely automatic preprocessing of the data. Maximum data size is bounded by the available main memory of the system. Another contribution of our work is an alternative technique for computing the 3D pre-integrated transfer function lookup table needed for accurate rendering entirely on the GPU. Coupled with our rendering algorithm, this allows users to interactively design transfer functions with the user interface.

The bottleneck of the Shirley-Tuchman PT algorithm [30] is the time required to compute the polygonal decomposition necessary for rendering the tetrahedra. Our optimized software implementation of their algorithm can process 300-700 Ktets/sec. Wylie *et al.* [38] describes a GPU implementation of the PT algorithm. They report rendering rates of up to 940 Ktets/sec when using a constant color per cell, and 495 Ktets/sec when using linear variation. Our rendering rates are between five to ten times faster, while producing higher quality images through the use of a 3D pre-integrated transfer function table. Another recent technique is the GPU-based ray casting of Weiler *et al.* [35]. The fastest rendering rates reported in their work are 764 Ktets/sec for a 148K Sphere dataset. The technique of Weiler has certain limitations on the size and shape (convexity) on the datasets that make it less general than cell-projection techniques. Both of these approaches require a substantial amount of connectivity information for rendering, resulting in a higher memory overhead than our work.

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. We thank Ricardo Farias for insightful discussions that helped shape many ideas presented in this paper. The Teem toolkit [17] proved very useful for processing our datasets and results. We thank Mark Segal from ATI for his prompt answers to our questions. We 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 dataset, Bruno Notrosso (Electricite de France) for the spx dataset, Hung and Buning for the blunt fin dataset, and Neely and Batina for the fighter dataset. Steven P. Callahan is supported by the Department of Energy (DOE) under the VIEWS program. Milan Ikits is supported by NSF grant ACI-9978063 and the DOE Advanced Visualization Technology Center. Cláudio T. Silva is partially supported by the DOE under the VIEWS program and the MICS office, and the National Science Foundation under grants CCF-0401498 and EIA-0323604. The work of João L. D. Comba is supported by a CNPq grant 540414/01-8 and FAPERGS grant 01/0547.3).

REFERENCES

[1] T. Aila, V. Miettinen, and P. Nordlund. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics*, 22(3):792–800, July 2003.

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

[3] ATI. Personal Communication, 2004.

[4] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics (Proc. SIGGRAPH 84)*, volume 18, pages 103–108, July 1984.

[5] 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*, 18(3):369–376, Sept. 1999.

[6] R. Cook, N. Max, C. Silva, and P. Williams. Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data. *IEEE Transactions on Visualization and Computer Graphics*, 2004. (to appear).

[7] M. de Berg, M. J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic Input Models for Geometric Algorithms. In *Proc. Annual Symposium on Computational Geometry*, pages 294–303, 1997.

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

[9] C. Everitt. Interactive Order-Independent Transparency. Technical report, NVIDIA, 2001. `http://developer.nvidia.com`.

[10] R. Farias, J. Mitchell, and C. Silva. ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering. In *Proc. IEEE Volume Visualization and Graphics Symposium*, pages 91–99, 2000.

[11] R. Farias and C. T. Silva. Out-Of-Core Rendering of Large, Unstructured Grids. *IEEE Computer Graphics and Applications*, 21(4):42–51, 2001.

[12] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Computer Graphics (Proc. SIGGRAPH 80)*, volume 14, pages 124–133, July 1980.

[13] L. Guibas. Computational Geometry and Visualization: Problems at the Interface. In N.M.Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 45–59. Springer-Verlag, 1991.

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

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

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

[17] G. L. Kindlmann. Teem, 2003. `http://teem.sourceforge.net`.

[18] J. M. Kniss, S. Premože, C. D. Hansen, P. Shirley, and A. McPherson. A Model for Volume Lighting and Modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.

[19] M. Kraus and T. Ertl. Cell-Projection of Cyclic Meshes. In *Proc. IEEE Visualization*, pages 215–222, Oct. 2001.

[20] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proc. IEEE Visualization*, pages 287–292, 2003.

[21] A. Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Computer Graphics and Applications*, 9:43–55, July 1984.

[22] N. L. Max. Sorting for Polyhedron Compositing. In *Focus on Scientific Visualization*, pages 259–268. Springer-Verlag, 1993.

[23] N. L. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[24] J. S. B. Mitchell, D. M. Mount, and S. Suri. Query-Sensitive Ray Shooting. *International Journal of Computational Geometry and Applications*, 7(4):317–347, Aug. 1997.

[25] M. Newell, R. Newell, and T. Sancha. A Solution to the Hidden Surface Problem. In *Proc. ACM Annual Conference*, pages 443–450, 1972.

[26] S. Roettger and T. Ertl. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. In *Proc. IEEE Volume Visualization and Graphics Symposium*, pages 23–28, 2002.

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

[28] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[29] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast Shadows and Lighting Effects Using Texture Mapping. In *Proc. ACM SIGGRAPH*, pages 249–252, July 1992.

[30] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Proc. San Diego Workshop on Volume Visualization*, 24(5):63–70, Nov. 1990.

[31] C. T. Silva, J. S. Mitchell, and P. L. Williams. An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. In *Proc. IEEE Symposium on Volume Visualization*, pages 87–94, Oct. 1998.

[32] C. Stein, B. Becker, and N. Max. Sorting and Hardware Assisted Rendering for Volume Visualization. In *Proc. IEEE Symposium on Volume Visualization*, pages 83–89, Oct. 1994.

[33] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys*, 6(1):1–55, Mar. 1974.

[34] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proc. IEEE Visualization*, pages 333–340, Oct. 2003.

[35] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based View-Independent Cell Projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, 2003.

[36] P. L. Williams. Visibility-Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, Apr. 1992.

[37] C. Wittenbrink. R-Buffer: A Pointerless A-Buffer Hardware Architecture. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 73–80, 2001.

[38] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral Projection using Vertex Shaders. In *Proc. IEEE/ACM Symposium on Volume Graphics and Visualization*, pages 7–12, 2002.