THE *K*-BUFFER AND ITS APPLICATIONS TO VOLUME RENDERING

by

Steven Paul Callahan

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computational Engineering and Science

School of Computing

The University of Utah

August 2005

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Steven Paul Callahan

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Cláudio T. Silva

Robert M. Kirby II

Peter Shirley

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of ____Steven Paul Callahan____ in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

______________________	______________________________
Date	Cláudio T. Silva Chair: Supervisory Committee

Approved for the Major Department

____

Krzysztof Sikorski
Chair/Director

Approved for the Graduate Council

____

David S. Chapman
Dean of The Graduate School

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. Although for regular grids texture-based techniques are well suited for current Graphics Processing Units (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 Central Processing Unit (CPU) 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 three-dimensional (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 the *k*-buffer, 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.

To further increase the interactivity of unstructured volume rendering, we describe a new dynamic level-of-detail (LOD) technique that allows real-time rendering of large tetrahedral meshes. Unlike approaches that require hierarchies of tetrahedra, our approach uses a subset of the faces that compose the mesh. No connectivity is used for these faces so our technique eliminates the need for topological information and hierarchical data structures. By operating on a simple set of triangular faces, our algorithm allows a robust and straightforward graphics hardware implementation. Because the subset of faces processed can be constrained to arbitrary size, interactive rendering is possible for a wide range of data sets and hardware configurations.

To Kristy, thanks for all the support and understanding.

CONTENTS

ABSTRACT . . . . **iv**

LIST OF FIGURES . . . . **viii**

LIST OF TABLES . . . . **x**

ACKNOWLEDGMENTS . . . . **xi**

CHAPTERS

1. INTRODUCTION **1**

2. PREVIOUS WORK **6**

3. *K*-BUFFER **10**

3.1 Nearly-Sorted Object-Space Visibility Ordering 10
3.2 The *k*-Buffer 11

4. HARDWARE-ASSISTED VISIBILITY SORTING **13**

4.1 Volume Rendering Algorithm 13
4.2 K-Buffer Hardware Implementation 14
4.3 Experimental Results 18
4.3.1 CPU Sorting 18
4.3.2 *k*-Buffer Analysis 20
4.3.3 Render Performance 21
4.3.4 Comparison 24
4.4 Discussion 25

5. DYNAMIC LEVEL-OF-DETAIL **27**

5.1 A Sample-Based LOD Framework 27
5.1.1 Face Subsampling 29
5.2 Sampling Strategies 30
5.2.1 Topology Sampling 31
5.2.2 View-Aligned Sampling 31
5.2.3 Field Sampling 32
5.2.4 Area Sampling 33
5.3 Implementation 33
5.4 Results 35
5.5 Discussion 36

6. CONCLUSION **41**

APPENDICES

A. GPU CODE **43**

B. OTHER APPLICATIONS OF THE *K*-BUFFER **48**

REFERENCES **52**

LIST OF FIGURES

1.1 Results of volume rendering the (a) fighter (b) blunt fin with the HAVS algorithm. 2

4.1 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. 14

4.2 Psuedo-code for the partial sort performed on the CPU. 14

4.3 Example of the *k*-buffer with $k = 3$ (see also Section 4.2). (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 pre-integrated 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. 15

4.4 Psuedo-code for a GPU fragment sorter using the *k*-buffer 16

4.5 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. 17

4.6 Rendering artifacts resulting from the fragment level race condition when simultaneously reading and writing the same buffer. In our experience, it has been quite hard to notice these artifacts. 17

4.7 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). 21

4.8 Results of rendering the (a) Torso (b) Spx (c) Kew and (d) Heart data sets with the HAVS algorithm. 23

5.1 Classification of LOD simplification techniques in 2D represented by a mesh and the function occurring at a ray **r** through the mesh. Undefined areas of the volume are expressed as dashed lines. (a) The original mesh showing the function $g(t)$ that ray **r** passes through. (b) The mesh after sample-based simplification where the function approximation $\bar{g_1}(t)$ is computed by removing samples from the original function $g(t)$. (c) The mesh after domain-based simplification, where the approximating function $\bar{g_2}(t)$ is computed by resampling the original domain. 29

5.2 A 2D example of sampling strategies for choosing internal faces. (a) A topology sampling which calculates layers by peeling boundary tetrahedra. (b) A view-dependent sampling that selects the faces most relevant to the current viewpoint. (c) A field sampling which uses stratified sampling on a histogram of the scalar values. (d) An area sampling which selects the faces by size. 30

5.3 Pseudocode for extracting the topology layers of a tetrahedral mesh. 32

5.4 Overview of the dynamic LOD algorithm. (a) The LOD algorithm samples the faces and dynamically adjusts the number of faces to be drawn based on previous frame rate. (b) The HAVS volume rendering algorithm sorts the faces on the CPU and GPU and composites them into a final image. 34

5.5 Plot of the render time for the Spx2 (blue), Torso (red), and Fighter (black) at different LODs. Approximately 3% LOD is the boundary only and 100% LOD is a full quality image. 36

5.6 Error measurements of the different sampling strategies for 14 fixed viewpoints on the Spx2, Torso, and Fighter data sets. Root mean squared error is used to show the difference between the full quality rendering and the LOD rendering at 10 fps. 37

5.7 Direct comparison of the different sampling strategies with a full quality rendering of the Spx2 data set (800 K tetrahedra). Full quality is shown at 2.5 fps and LOD methods are shown at 10 fps (3% LOD for area sampling and 10% LOD on all other strategies). 38

5.8 The Fighter data set (1.4 million tetrahedra) shown in multiple views at full quality on top (1.3 fps), 15% LOD (4.5 fps) in the middle, and at 5% LOD (10.0 fps) on the bottom. The LOD visualizations use area-based sampling. . . . 40

B.1 2D lookup table for determining if two fragments contain an isosurface between them at contour value *c*. 49

B.2 The Spx data set with an isosurface generated using the *k*-buffer. The front faces of the isosurface are shown in green and back faces of the isosurface are shown in blue. 50

LIST OF TABLES

4.1 Analysis of sorting algorithms 19

4.2 *k*-buffer analysis 20

4.3 Performance of the GPU sorting and drawing 22

4.4 Total performance of HAVS 22

4.5 Time comparison in milliseconds with other algorithms 25

5.1 Preprocessing time in seconds of the different sampling strategies 35

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Cláudio Silva for his time, effort, and ideas without which this thesis would not have been possible. I would like to thank the committee members who helped organize and compose this thesis: Cláudio Silva (chair), Mike Kirby, and Peter Shirley. I also acknowledge those who co-authored the papers that compose this thesis, namely Milan Ikits, João Comba, Peter Shirley and Cláudio Silva. Carlos Scheidegger, Huy Vo, and Louis Bavoil provided invaluable code contributions. In particular, Vo wrote the fast radix sort used in our system. I thank Fábio Bernardon for the use of his HW Ray Caster code and Ricardo Farias for the ZSWEEP code. I thank Mark Segal from ATI for his prompt answers to our questions and donated hardware. I am grateful to Patricia Crossno, Shachar Fleishman, Nelson Max, and John Schreiner for helpful suggestions and criticism. I 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, Neely and Batina (NASA) for the Fighter data set, and the Scientific Computing and Imaging Institute (University of Utah) for the Torso data set. Steven P. Callahan is supported by the Department of Energy (DOE) under the VIEWS program. This work was also partially supported by the National Science Foundation under granst CCF-0401498, EIA-0323604, and OISE-0405402. Portions of Chapters 1–4 were published under the title Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering [4] by IEEE Computer Society with the Log Number TVCG-0131-1004.

CHAPTER 1

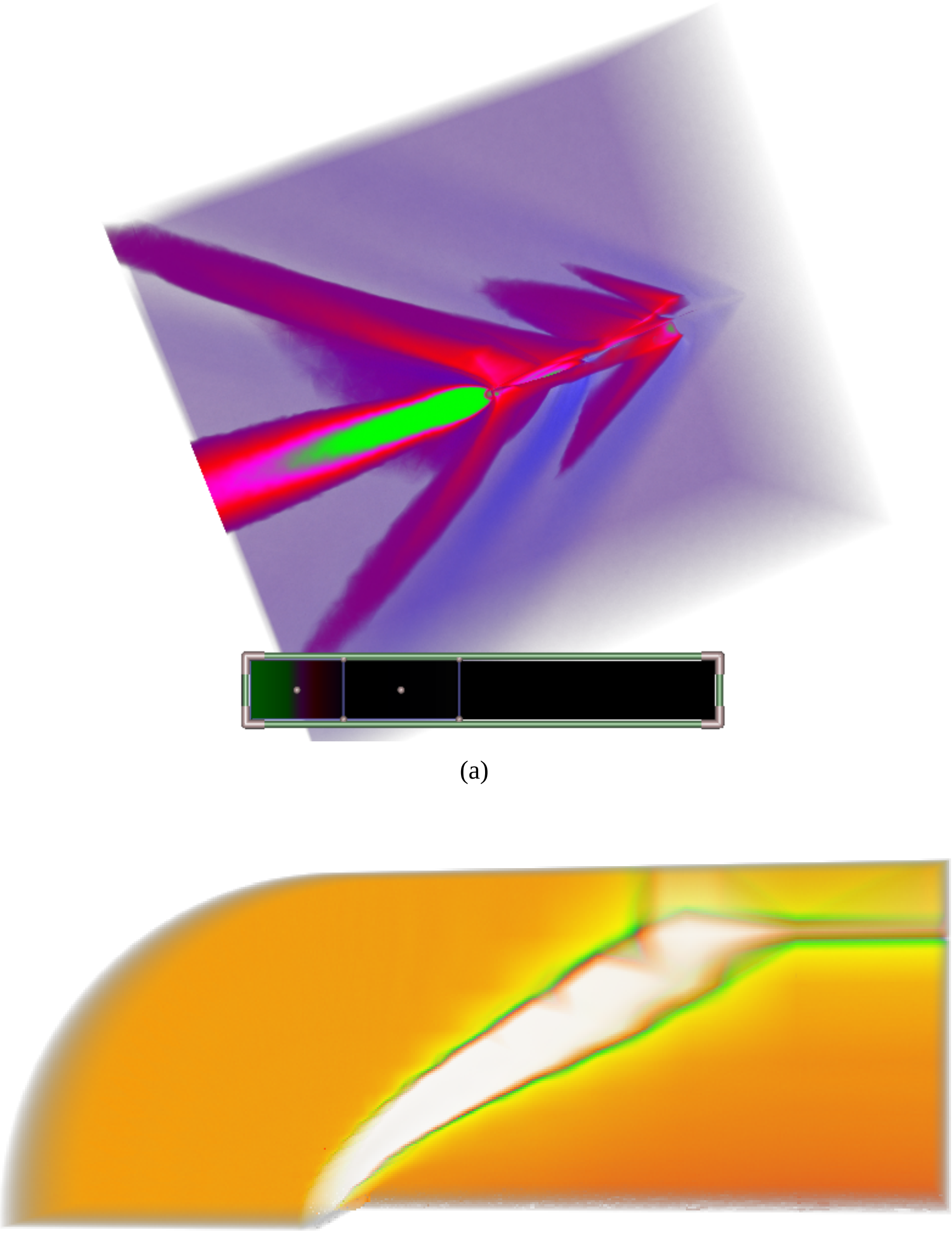
INTRODUCTION

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 thesis 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 [50]. To create an image, the equations for the optical model must be integrated along the viewing ray for each pixel (see Figure 1.1). 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.

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 [59, 32]. 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



(a)

(b)

Figure 1.1. Results of volume rendering the (a) fighter (b) blunt fin with the HAVS algorithm.

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 [33].

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 [63] 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 Meshed Polyhedra Visibility Ordering (MPVO) algorithm [76]. 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 [9, 65, 37, 10].

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 [10, 71, 73].

These approaches are typically too slow for interactivity when full resolution meshes are rendered. Thus, level-of-detail (LOD) techniques are an attractive way to trade off rendering quality for rendering speed. However, unlike for rectilinear volumes, it is not well studied how LOD techniques should be applied to unstructured meshes.

There have been two basic approaches to LOD for unstructured meshes, typically for the special case of tetrahedral meshes. The first has been employed in ray casting where the scalar field along the ray is sparsely sampled to improve speed. The second is to simplify the mesh into a hierarchy of new meshes, each with fewer tetrahedra than its parent mesh. The ray tracing approach has the advantage of simplicity and robustness, and the mesh simplification approach has the advantage that it can be easily used in a graphics hardware (GPU) implementation.

This thesis describes an LOD technique that attempts to capture the simplicity of the ray sampling method while still allowing a natural GPU implementation. This is accomplished by sparsely sampling the faces between tetrahedra in the mesh without creating an explicit hierarchy of LODs. The GPU is used to perform ray integration between these sparsely sampled faces without the need for connectivity information. This avoids the complexity of the mesh simplification methods, but still allows a GPU implementation, which makes a fast and robust LOD rendering system possible.

In this thesis we build on the previous work of Farias et al. [22], Carpenter [5], and Danskin and Hanrahan [15], and propose a new volume rendering algorithm with dynamic LOD. Our main contributions are:

- We introduce the *k*-buffer, a fixed depth A-buffer for sorting in image space.
- 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 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 has 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.
- We propose a new sample-based LOD framework for rendering unstructured grids that can be implemented by simply using a subset of mesh faces.
- We examine several strategies for deciding which faces to draw when using dyanamic LOD, and discuss their relative merits.
- We provide an efficient implementation of our LOD sampling techniques, and show that they can be applied to generate high-quality renderings of large unstructured grids.

The remainder of this thesis is organized as follows. We summarize related work in Chapter 2. In Chapter 3, we describe the *k*-buffer, define *k*-nearly sorted sequences, and provide further details on the functionality of the *k*-buffer. In Chapter 4, we show how to efficiently implement the *k*-buffer using the programmable features of current GPU hardware. Chapter 5 describes our dynamic LOD algorithm and introduces our sample-based simplification technique. Finally, in Chapter 6, we provide concluding remarks and directions for future work.

CHAPTER 2

PREVIOUS 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 [31, 49, 10, 33, 23]. We limit our coverage to the most directly related work in the area of visibility ordering using both software and hardware techniques and level-of-detail (LOD) approaches for unstructured grids.

In computer graphics, work on visibility ordering was pioneered by Schumacker et al. and is later reviewed in [68]. An early solution to computing a visibility order given by Newell, Newell, and Sancha (NNS) [55] continues to be the basis for more recent techniques [67]. 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 [27] 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 nonleaf 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. [27] 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 methods presented above operate in *object-space*, i.e., they operate on the primitives before rasterization by the graphics hardware [2]. Carpenter [5] 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 [77] is a pointerless A-buffer hardware architecture that implements a method similar to a software algorithm described in [47] 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 [20, 38]. 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 [34] is an alternative design that 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 that shift much of the computation to the GPU for increased performance. Kipfer et al. [35] 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 [58] 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 our algorithm without the need for sorting. Recently, Wylie et al. have shown how to implement the Shirley-Tuchman tetrahedron projection directly on the GPU [78]. 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 by Weiler et al. for convex meshes [71] and more recently for nonconvex meshes [73].

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 [22] 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 [1], 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.

Because the size of the unstructured grids continues to increase faster than our visualization techniques can handle them, other research has focused on approximately rendering unstructured grids to maintain interactivity [24, 8, 42, 53]. The two main techniques have been to use LOD representations of the data [66, 6, 7, 29], or to sparsely sample viewing rays.

The approximate rendering was first done by sampling sparsely along viewing rays [15]. This idea can work for unstructured meshes as well, provided there is a mechanism for skipping cells entirely (e.g., [56]). Alternatively, a multiresolution approach is commonly used to increase the rendering speed of triangle meshes [45]. They often work by dynamically selecting a set of triangles to approximate the surfaces to within some error bound, or to meet some target frame rate [28, 44].

For structured grids, computing and dynamically rendering multiple LODs is relatively straightforward. This can be accomplished by using hardware-accelerated techniques that involve slicing the volume with view-aligned texture hierarchies [74, 41]. Because the data are structured, the view-aligned slices can be computed quickly and compositing can be accomplished by blending the fragments into the framebuffer in the order in which they are generated.

For unstructured meshes the problem is not as well studied. One way to speed up the rendering is to statically simplify the mesh in a preprocessing step to a mesh small enough for the volume renderer to handle [7, 29, 6]. However, this approach only provides a static approximation of the original mesh and does not allow for dynamic changes to the level of detail. This way, the user cannot easily refine features of interest, or dynamically adapt the LOD to the capabilities of the hardware being used. Dynamic LOD approaches are preferable and have been shown to be useful by Museth and Lombeyda [53] even if the pictures generated are of a more limited type than full-blown volume renderings. To our knowledge, the only two approaches that describe dynamic LOD volume rendering for unstructured meshes are the recent works by Cignoni et al. [8] and Leven et al. [42].

Cignoni et al. [8] propose a technique based on creating a progressive hierarchy of tetrahedra that is stored in a multitriangulation data structure [25] that is dynamically updated to achieve interactive results. Their algorithm is quite clever (and involved), as an efficient implementation requires the use of compression of the topology and hierarchical information to be practical. Leven et al. [42] convert the unstructured grid into a hierarchy of regular grids that are stored in an octree, and can be rendered using LOD technique for regular grids. Their experiments show that the resulting LOD hierarchy is over two orders of magnitude larger than the original data. Both of these techniques require some form of hierarchical data structures, fairly involved preprocessing, and relatively complex implementations.

Many methods have simplified triangle meshes into smaller sets of textured polygons (e.g., [46, 17]). Another approach is to use compression schemes to minimize the bandwidth of the geometry [18]. Of these, our method is most similar in spirit to the randomized Z-buffer [69], where a subset of all the geometry is drawn. It is also related to the sampling work of Danskin and Hanrahan [15]. Some of our sampling strategies are reminiscent of Monte Carlo rendering (e.g., [14, 11]).

CHAPTER 3

K-BUFFER

3.1 Nearly-Sorted Object-Space Visibility Ordering

Visibility ordering algorithms (e.g., Extended Meshed Polyhedra Visibility Ordering [65]) 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 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 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 [60]. 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 [9]. 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 [22]. 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 [22]. In our approach, we apply a more *aggressive* technique for managing the A-buffer.

3.2 The *k*-Buffer

The original A-buffer [5] 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 4.2).

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 pre-integrated table lookup. In practice, the hardware implementation is simplified by keeping the *k*-buffer entries unsorted.

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 4.3, even simple and inexpensive object-space ordering leads to fragments that are almost completely sorted.

CHAPTER 4

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 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 4.1). 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 4.2, 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.

4.1 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 Figure 4.1). The complete pseudo-code for our algorithm is given in Figure 4.2.

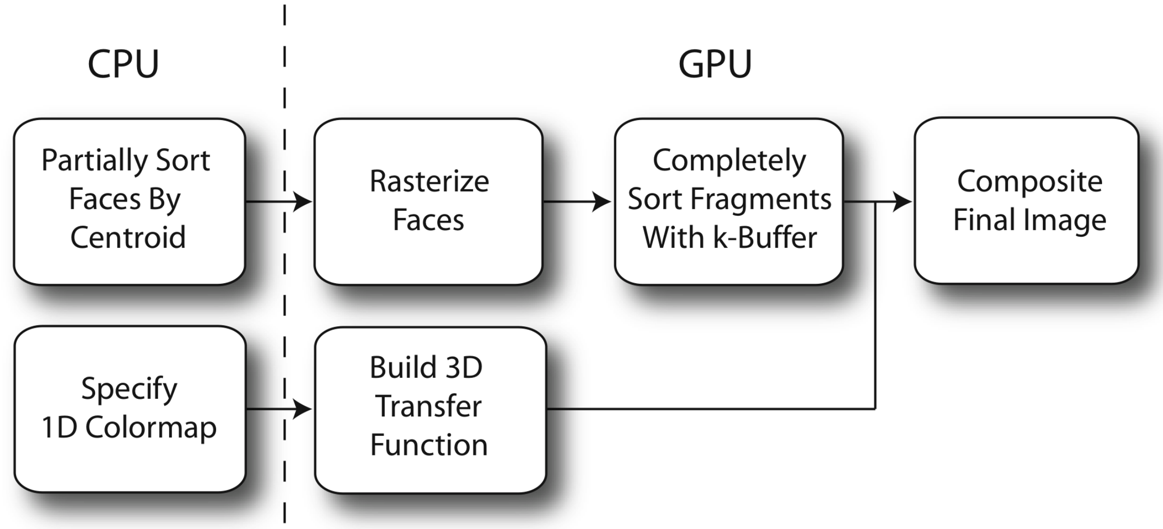

CPU
GPU
Partially Sort Faces By Centroid
Rasterize Faces
Completely Sort Fragments With k-Buffer
Composite Final Image
Specify 1D Colormap
Build 3D Transfer Function

Figure 4.1. 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.

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

Figure 4.2. Psuedo-code for the partial sort performed on the CPU.

4.2 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 comprises three stages (see Figure 4.3 and the source code in Appendix A). First, the program reads the accumulated color and opacity from the frame-

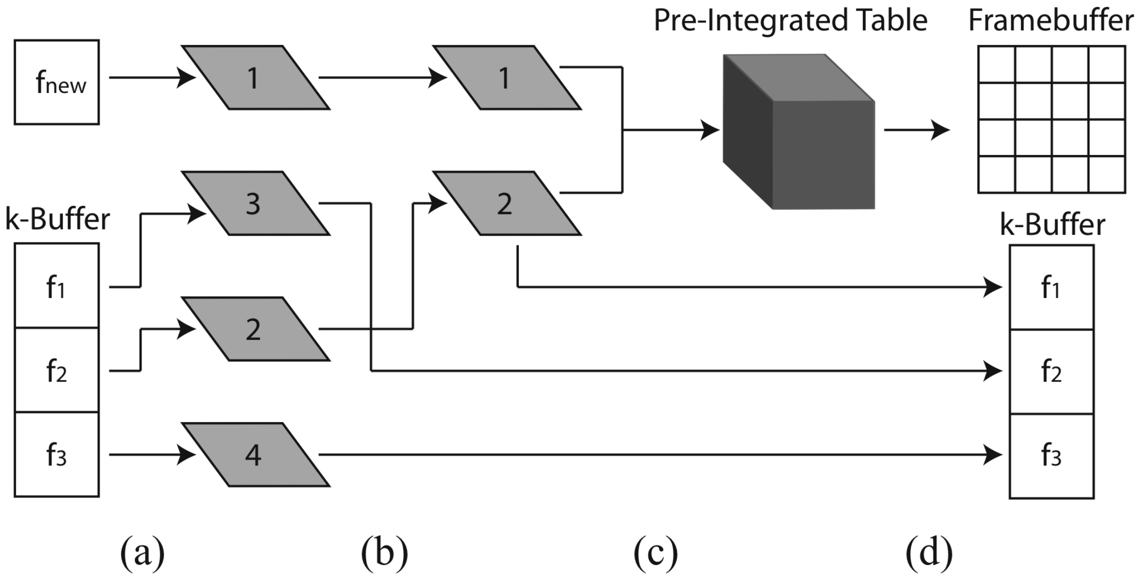


Figure 4.3. Example of the k-buffer with $k = 3$ (see also Section 4.2). (a) We start with the incoming fragment and the current k-buffer entries and (b) find the two entries closest to the viewpoint. (c) Next, we use the scalar values (v_1, v_2) and view distances (d_1, d_2) of the two closest entries to look up the corresponding color and opacity in the pre-integrated 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.

buffer. 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 pre-integrated 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 4.3). The complete pseudo-code for our GPU fragment sorter ($k = 3$) is given in Figure 4.4

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 [36, 39]. This approach produces incorrect results, unless the

```
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 pre-integrated table
            using $v_1$, $v_2$, and $\Delta d$
        Composite $c_1$ and $c_2$ into framebuffer
        Write $r_1$, $r_2$, and $r_3$ back into $k$-buffer
```

Figure 4.4. Psuedo-code for a GPU fragment sorter using the *k*-buffer

primitive is aligned with the screen, which happens only when view-aligned slicing is used to sample the volume (see Figure 4.5). 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 [62]. 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 [33]. Unfortunately, compositing triangles of varying sizes can yield artifacts, as shown by Figure 4.6. One way to remedy this problem is to draw triangles

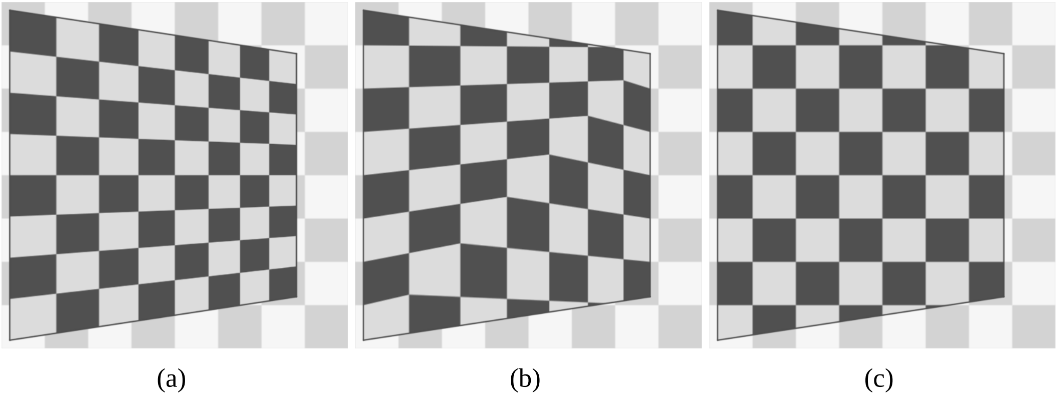


Figure 4.5. Screen-space interpolation of texture coordinates. (a) The rasterizer interpolates vertex attributes in perspective space, which is typically used to map a 2D texture onto the faces of a 3D object. (b) Using the projected vertices of a primitive as texture coordinates to perform a lookup 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.

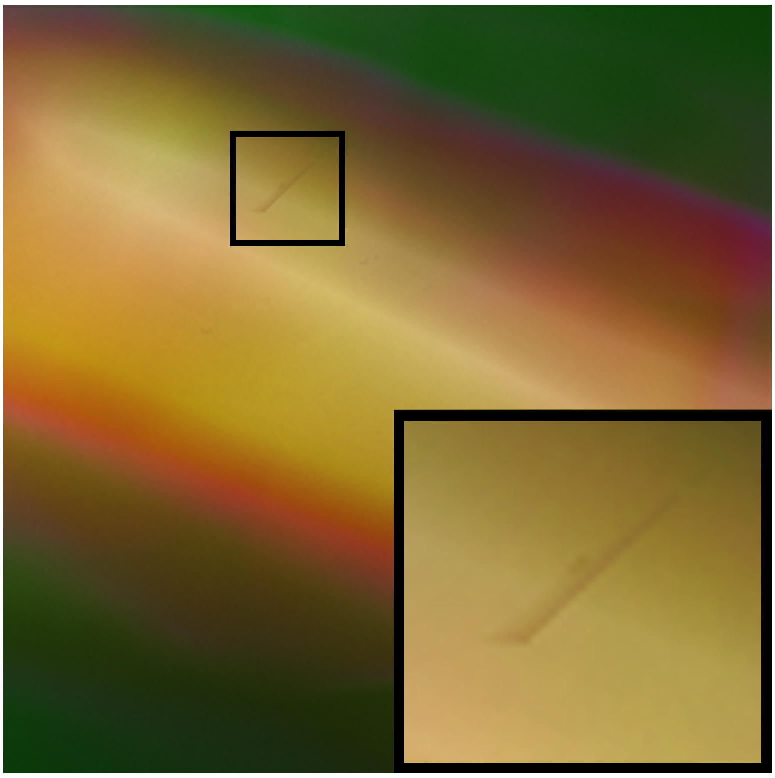


Figure 4.6. Rendering artifacts resulting from the fragment level race condition when simultaneously reading and writing the same buffer. In our experience, it has been quite hard to notice these artifacts.

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 pre-integrated 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. Since the algorithm described above can handle holes properly, complete handling of holes will be added once next generation hardware becomes available.

4.3 Experimental Results

Our implementation was tested on a PC with a 3.2 GHz Pentium 4 processor and 2048 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.

4.3.1 CPU Sorting

We tested several commonly used sorting algorithms described in [12, 61] for generating nearly sorted sequences. Table 4.1 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 re-sorting previously

Table 4.1. Analysis of sorting algorithms

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

sorted sequences (e.g., mergesort). We found, however, that re-sorting 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 [61], 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 [70]. 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 [61] 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 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 upwards of 15 million faces per second.

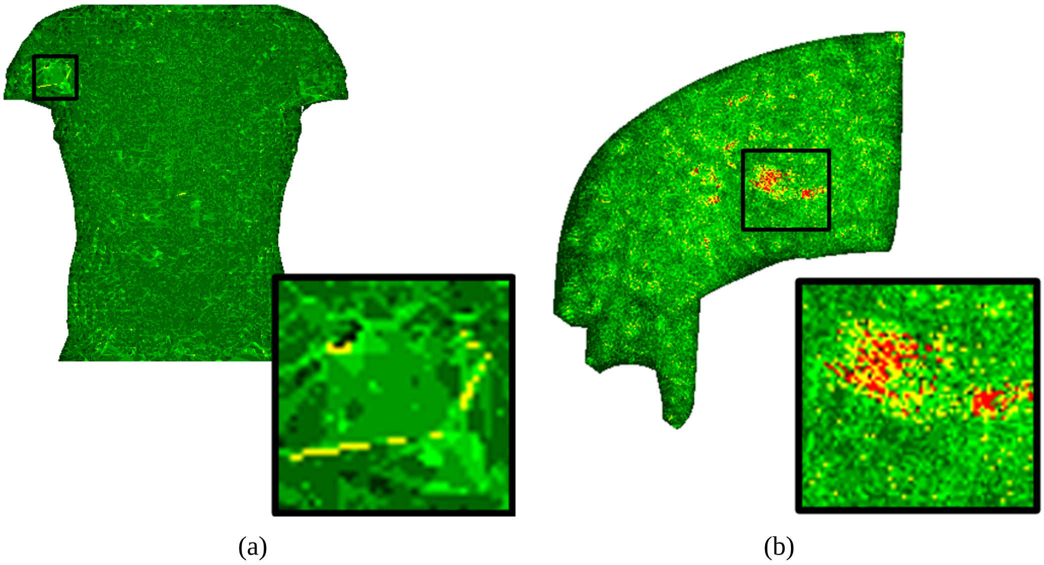
4.3.2 *k*-Buffer Analysis

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 4.2. 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. Figure 4.7 contains a sample of these images. This analysis shows that the problematic areas are usually caused by sliver cells, those that 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 [19, Chapter 5], we tried to isolate these bad cells by analyzing how much they "differ" locally from a DT

Table 4.2. *k*-buffer analysis

Data set	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



(a) (b)

Figure 4.7. 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).

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 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 [26] 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 a transparent transfer function is used. Also, even in our worst example (Torso), only 0.6% of the pixels *could* be incorrect. For a pixel to be truly 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.

4.3.3 Render Performance

Table 4.3 and Table 4.4 show the performance of our hardware-assisted visibility sorting algorithm on several data sets using the average values of 14 fixed viewpoints. Table 4.3 shows

Table 4.3. Performance of the GPU sorting and drawing

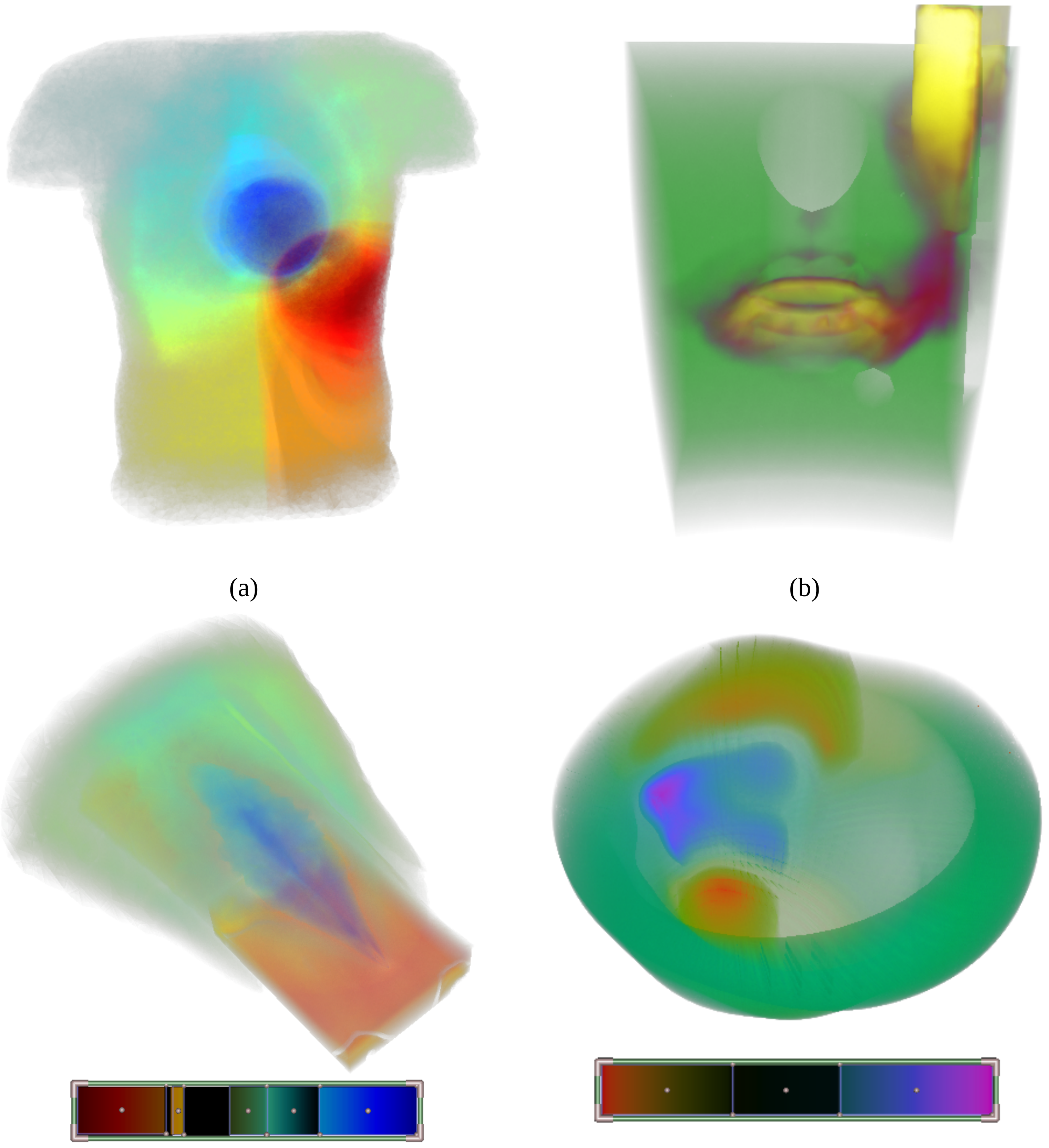
		$k=2$		$k=6$	
Data set	Cells	Fps	Tets/sec	Fps	Tets/sec
F117	240,122	9.71	2331 K	4.42	1062 K
Kew	416,926	5.45	2267 K	3.75	1561 K
Spx2	827,904	2.07	1712 K	1.70	1407 K
Torso	1,082,723	3.13	3390 K	1.86	1977 K
Fighter	1,403,504	2.41	3387 K	1.56	2190 K

Table 4.4. Total performance of HAVS

Data set	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

only the GPU 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 4.4 includes the time required to sort the faces on the CPU as well as the GPU with $k=2$. This represents the rendering rates 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 pre-integrated 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.

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 4.3). 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 Figure 4.8 and Figure 1.1). Modifying the opacity or loading a new colormap triggers a pre-integrated table update, which renders the data set using the GPU



(a)

(b)

(c)

(d)

Figure 4.8. Results of rendering the (a) Torso (b) Spx (c) Kew and (d) Heart data sets with the HAVS algorithm.

sort only. We found that in general, a 128^3 pre-integrated table is sufficient for high quality rendering.

4.3.4 Comparison

Table 4.5 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 [63] implemented by Nelson Max, Peter Williams and Cláudio Silva that uses the MPVO with nonconvexities (MPVONC) algorithm for visibility ordering[76]. 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 store 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. [78] 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 then these PT methods, while producing higher quality images through the use of a 3D pre-integrated table. Another recent technique is the GPU-based ray casting of Weiler et al. [72, 73]. 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. [3], which is approximately twice as fast as the original technique reported by Weiler. The ZSWEEP method [22] 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

Table 4.5. 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 RC	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

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.

4.4 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. Finally, we thought that the GPU would be the main bottleneck during rendering. Hence, in our prototype implementation, we did not spend as much time optimizing the CPU sorting algorithm.

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. [52] and de Berg et al. [16].

CHAPTER 5

DYNAMIC LEVEL-OF-DETAIL

5.1 A Sample-Based LOD Framework

In scientific computing, it is common to represent a scalar function $f : D \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$ as sampled data by defining it over a domain D, which is represented by a tetrahedral mesh. For visualization purposes, we define the function f as linear inside each tetrahedron of the mesh. In this case, the function is completely defined by assigning values at each vertex $v_i(x,y,z)$, and is piecewise-linear over the whole domain. The domain D becomes a 3D simplicial complex defined by a collection of simplices c_i. It is important to distinguish the domain D from the scalar field f. The purpose of visualization techniques, such as isosurface generation [43] and direct volume rendering [48] are to study intrinsic properties of the scalar field f. The time and space complexity of these techniques are heavily dependent on the size and shape of the domain D.

For large data sets, it is not possible to achieve interactive visualization without using an approximation. In these cases, it is often useful to generate a reduced-resolution scalar field $\bar{f} : \bar{D} \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$, such that:

- the new scalar field $\bar{f}$ approximates f in some natural way, i.e., $|\bar{f} - f| \leq \varepsilon$;
- the new domain $\bar{D}$ is smaller than D.

LOD techniques attempt to compute multiple approximations $\bar{f}_i$ from f at decreasing resolutions for interactive visualization. Recently, techniques have been proposed that hierarchically simplify the tetrahedral mesh by using edge collapses (see [8]). These techniques work similarly to triangle based simplification and use connectivity information to incrementally cull simplices c_i from the domain D, i.e., when a 1-simplex is collapsed, several 2- and 3-simplices become degenerate and can be removed from the mesh. Most techniques order the collapses using some type of error criterion, stopping when the size of the domain $|\bar{D}|$ reaches the desired LOD. $\bar{f}$ computed in this way for LOD can can be considered as a *domain-based simplification* of f because the domain D is being resampled with fewer vertices.

An alternative approach for computing $\overline{f}$ is *sample-based simplification*. If we consider a ray $\mathbf{r}$ that leaves the view point and passes through screen-space pixel (x,y) then enters the scalar field f, a continuous function $g(t)$ is formed for $\mathbf{r}$ where $g(t) = f(\mathbf{r}_0 + t\mathbf{r}_d)$. In the domain D, represented by a tetrahedral mesh, this function $g(t)$ is piecewise-linear and defined by the set of points $P = \{\mathbf{p_i}^{x,y}\}$. An approximation $\overline{g}(t)$ can be created by using a subset $\overline{P}$ of P. In other words, by removing some of the samples that define $g(t)$, we obtain an approximating function $\overline{g}(t)$. This subsampling can occur on the tetrahedral mesh using any of the 2- or 3-simplices.

The key difference between domain and sample-based simplification is that they approximate the domain D in different ways with respect to the volume integral. If you consider the ray $\mathbf{r}$ passing through a medium represented by D, the volume rendering integral is computed at each 2-simplex intersection within D (see Max [50]). Domain-based simplification approximates the domain D, then computes an exact volume integral over the approximate geometry. Sample-based simplification uses the original geometry to represent D, then computes an approximate volume integral over the original geometry. Figure 5.1 shows a 2D example of the function $g(t)$ and approximate functions $\overline{g_1}(t)$ and $\overline{g_2}(t)$ using these two approaches as the ray $\mathbf{r}$ passes through D. It is important to emphasize that sample-based simplification provides different, though not necessarily better, results than domain-based simplification. The advantage of this approach is the simplicity of the simplification method and data structures necessary to perform dynamic LOD.

Because we want $\overline{f}$ to be a good approximation of f, when using sample-based simplification it is necessary to ensure that each ray $\mathbf{r}$ passing through $\overline{f}$ encounters at least two samples to avoid holes in the resulting image. Furthermore, by removing geometry without constraint in a non-convex mesh, we could possibly be computing the volume integral over undefined regions outside of D. This problem can easily be resolved by always guaranteeing that the boundary $\overline{B}$ of $\overline{f}$ is always sampled (see Figure 5.1).

This problem is similar to importance sampling where the integral can be approximated using probabilistic sampling or Monte Carlo techniques [14, 64]. However, this is a much more difficult problem when considering the entire space of functions that occur because of the infinite number of viewpoints. Therefore, sampling strategies that attempt to optimize the coverage of the functions from all viewpoints become necessary.

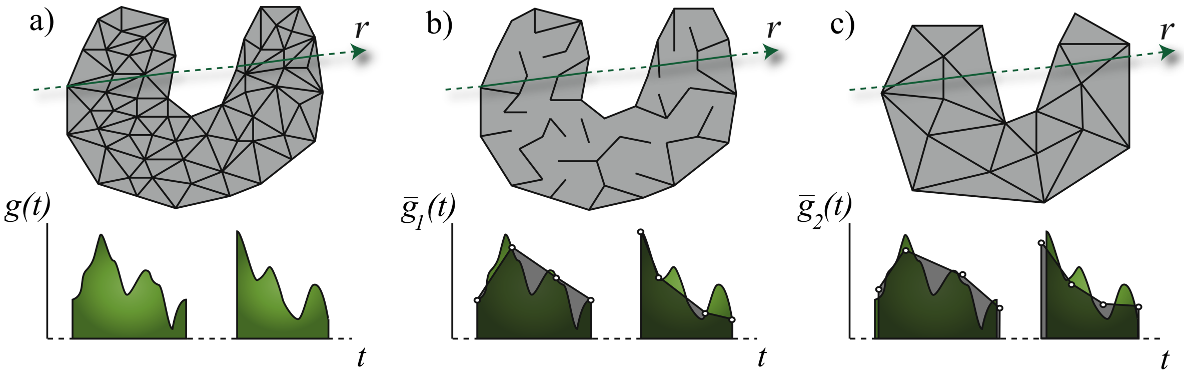


Figure 5.1. Classification of LOD simplification techniques in 2D represented by a mesh and the function occurring at a ray **r** through the mesh. Undefined areas of the volume are expressed as dashed lines. (a) The original mesh showing the function $g(t)$ that ray **r** passes through. (b) The mesh after sample-based simplification where the function approximation $\bar{g_1}(t)$ is computed by removing samples from the original function $g(t)$. (c) The mesh after domain-based simplification, where the approximating function $\bar{g_2}(t)$ is computed by resampling the original domain.

5.1.1 Face Subsampling

The sample-based simplification strategy described above operates on the existing geometric representation of the mesh. In principle, the sampling could be done on any of the simplices that compose the mesh. We choose to sample by the faces (triangles) that make up the tetrahedra. This is due to the flexibility and speed of the sampling that it allows.

If we consider the topology of the mesh as a collection of triangles (2-simplices) embedded in $\mathbb{R}^3$, $\bar{f_i}$ can be computed at each LOD i by selectively sampling a portion of the faces. Thus, by removing one triangle, we are effectively removing one sample of the function $g(t)$ that represents f along the ray **r**. The advantage of this technique is that it provides a natural representation for traversing the different LODs.

Given a set of unique faces F in a tetrahedral mesh, boundary faces B and internal faces I can be extracted such that $B \cup I = F$. Since B gives a minimum set of faces that bound the domain D of the mesh, B should remain constant. By adding and removing faces from I, we get an approximation of the mesh where $B \cup I \subseteq F$. This leads to a simple formula for determining the number of faces in I that need to be selected in each pass:

$$|I| = \frac{|I| \times TargetTime}{RenderTime},$$

where $TargetTime$ is the amount of time you would like each frame to take for interactive viewing, i.e., 0.01 seconds, and $RenderTime$ is the time that the previous frame required to render. This allows the system to adapt the LOD to the current view and render complexity.

It also allows an easy return to the full quality mesh by selecting all the internal faces to be drawn with the boundary faces.

5.2 Sampling Strategies

To minimize visual error the faces should be chosen while accounting for both transfer function and viewpoint. However, to maximize visual smoothness when the viewpoint or the transfer function change, the faces should be based only on mesh geometry and scalar field values. This conflict between accuracy and ability to change viewpoint and transfer function easily indicates that the best sampling strategy will depend on user goals and properties of the data. For this reason we provide a variety of sampling strategies.

We described the steps that are required to achieve interactive rates given an internal face list. However, the heuristics that are incorporated to assign importance to the faces are just as important. We describe four methods that operate on different properties of the mesh: topology sampling, view-dependent sampling, field sampling, and area sampling strategies (Figure 5.2). The first two strategies are deterministic. The second two strategies are randomized, each rewarding different data attributes. We also implemented a naive randomized

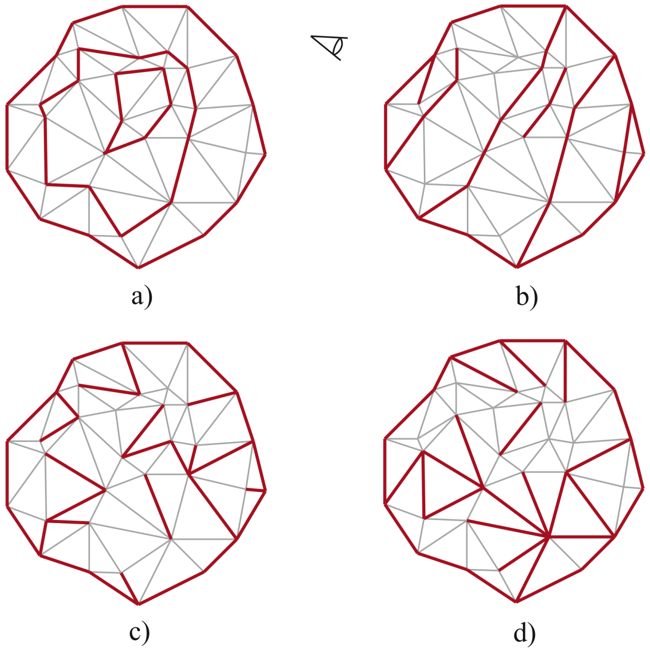


Figure 5.2. A 2D example of sampling strategies for choosing internal faces. (a) A topology sampling which calculates layers by peeling boundary tetrahedra. (b) A view-dependent sampling that selects the faces most relevant to the current viewpoint. (c) A field sampling which uses stratified sampling on a histogram of the scalar values. (d) An area sampling which selects the faces by size.

algorithm that selected a uniformly random subset of faces, but these results were poor for the same reason unstratified sampling is poor when sampling pixels: clumping and holes [51].

5.2.1 Topology Sampling

The first sampling method that we employ is based on the topology of the initial tetrahedral mesh. This approach assigns layers to some of the faces in an attempt to select faces that are connected, resulting in an image that looks more continuous. Similar work has been done for determining a sequence of nonconvex hulls on a point set [21]. This approach requires a preprocessing step in which we extract the boundary, remove the tetrahedra on the boundary, and repeat until all the tetrahedra have been *peeled* from the mesh. A new layer value is assigned to the faces on the boundary at each step in the algorithm and separated into the boundary face list (layer zero) and the internal face list (all other faces). The preprocessing layer extraction algorithm is given in Figure 5.3.

This algorithm assigns layers to some of the faces, but not all of them. In practice, it uses enough of the faces for a good image. However, when all of the layer faces cannot be drawn because they exceed the limit of internal faces allowed for interactive rendering, a subset of the layers are rendered. This is done by picking an even distribution of the layers throughout the mesh until the target internal face count is reached.

5.2.2 View-Aligned Sampling

The second sampling strategy that we use is view-dependent sampling. The intuition to this approach is that the faces perpendicular to the viewing direction are the most visible ones. Therefore, by selecting the internal faces that are more closely aligned to the current view, we can optimize the screen-space coverage. A simple approach would be to perform a dot product between each internal face normal and the viewing direction to order the faces in the internal face list. However, this approach is costly because it requires processing on every face independent of the LOD. Instead, we use a simple technique based on the work of Kumar et al. [40] and Zhang and Hoff [79] for back-face culling. In a preprocessing step, the faces are clustered by their normals. The clusters are computed by fitting a bounding box to a unit sphere and subdividing the faces of the box into regions. This grid is then projected onto the sphere, and the internal faces are clustered based on their normal's location in the distribution. Clearly, this is not a uniform distribution, but in practice it produces good results. We used a 4x4 grid for each of the faces of the bounding box, which results in 96 normal clusters. This

```
EXTRACT_LAYERS
    $CurrentLayer \leftarrow 0$
    for each tetrahedra $t$
        $Peeled[t] \leftarrow$ false
    while there exists $t$ such that $Peeled[t]$ = false
        for each face $f$
            $f \leftarrow$ External
        Set $s = \emptyset$
        for each tetrahedra $t$ such that $Peeled[t]$ = false
            for each face $f$ in $t$
                if $f$ is already in $s$
                    $s(f) \leftarrow$ Internal
                else insert $f$ into $s$
        for each face $f$ such that $f$ = External
            $f \leftarrow CurrentLayer$
            for each tetrahedra $t$ such that $f$ and $t$ share a vertex
                $Peeled[t]$ = true
        $CurrentLayer \leftarrow CurrentLayer + 1$
```

Figure 5.3. Pseudocode for extracting the topology layers of a tetrahedral mesh.

allows us to detect view-aligned faces based on their cluster instead of individually, which significantly increases the performance of the algorithm.

5.2.3 Field Sampling

Our third approach for sampling internal faces is based on the field values of the tetrahedral mesh. In most cases this is represented as a scalar at each vertex in the mesh. This technique assigns importance to a face based on the average scalar value at its vertices. In a preprocessing step, a histogram is built using the normalized scalar values, which shows the distribution of the field. Using an approach similar to stratified sampling in Monte Carlo integration, we divide this histogram into evenly spaced intervals (we use 128), then randomly pick the same number of faces from each interval to be rendered. Unlike Monte Carlo integration, which provides the underlying theory for randomly sampling to approximate a function, we are randomly sampling over the entire space of functions. The internal face list is filled with these random samples, so that no extra computation is required for each viewpoint or change in LOD.

5.2.4 Area Sampling

Our fourth strategy recognizes that if the removal of a triangle causes an error of a certain magnitude at a given pixel, then the total error magnitude for all pixels is proportional to the area of that triangle. Thus we could prioritize based on area. An easy way to do this while preserving the possibility of choosing small faces is to prioritize based on $A_i * \xi_i$ where A_i is the area in $\mathbb{R}^3$ of the ith face and ξ_i is a uniform random number. This randomizes the selected triangles, but still favors the larger ones being drawn. As with field sampling, this list does not need to be rebuilt for new viewpoints or number of faces to be drawn.

5.3 Implementation

The algorithm for dynamic LOD builds on the Hardware-Assisted Visibility Sorting volume rendering system (HAVS) proposed in Chapter 4. Figure 5.4 shows an overview of how the sampling interacts with the volume renderer.

In a preprocessing step, the boundary faces are separated from the internal faces and each is put in a list. Each internal face contains a neighboring face with the same vertices and scalar values. To avoid the redundancy of computing a zero-length interval, these duplicate faces are removed. The internal face list is reordered from most important to least important based on one of sampling strategies previously described. This allows us to dynamically adjust the number of faces that are being drawn by passing the first $|I|$ faces to the volume renderer along with the boundary faces. For full quality images, which are shown when the user is not interactively viewing the mesh, the entire internal face list is used.

Once the proper subset of faces has been selected, the HAVS algorithm prepares the faces for rasterization by sorting them by their centroids. This provides only a partial order of the faces in object-space since the mesh may contain faces of varying size or even visibility cycles. Upon rasterization, the fragments undergo an image-space sort via the k-buffer, which has been implemented using fragment shaders. The k-buffer keeps a fixed number of fragments (k) in each pixel of the framebuffer. As a new fragment is rasterized, it is compared with the other entries in the k-buffer, the two entries closest to the viewpoint (for front-to-back) are used to find the color and opacity for the fragment using a lookup table which contains the pre-integrated volume integral. The color and opacity are composited in the framebuffer, and the remaining fragments are written back to the k-buffer (see Chapter 4 for more detail).

For dynamic LOD, we are interested in the time that each frame requires to render so we can adjust accordingly. Therefore, we track the render time at each pass and use it to adjust the

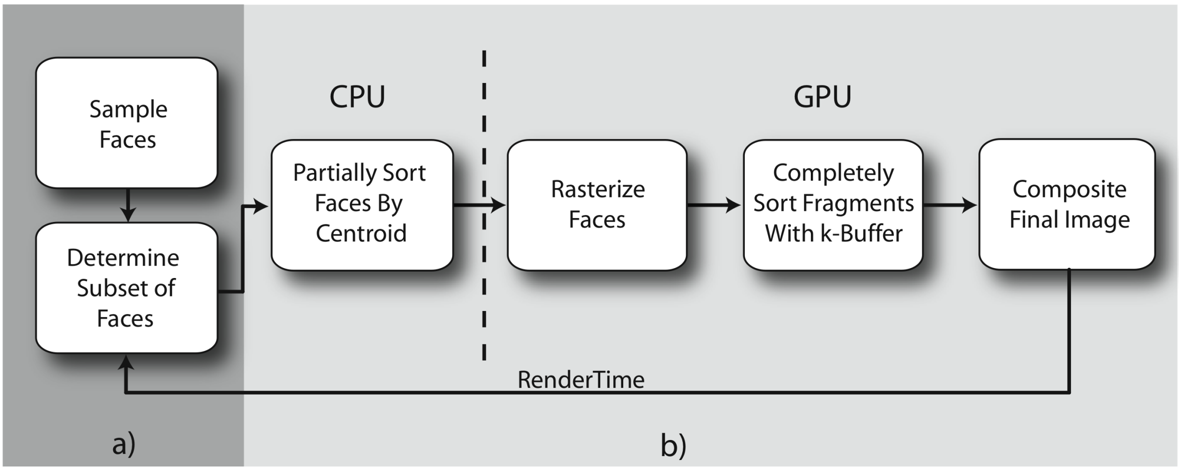

Sample Faces
Determine Subset of Faces
CPU
Partially Sort Faces By Centroid
GPU
Rasterize Faces
Completely Sort Fragments With k-Buffer
Composite Final Image
RenderTime
a)
b)

Figure 5.4. Overview of the dynamic LOD algorithm. (a) The LOD algorithm samples the faces and dynamically adjusts the number of faces to be drawn based on previous frame rate. (b) The HAVS volume rendering algorithm sorts the faces on the CPU and GPU and composites them into a final image.

number of internal faces that are sent to the rasterizer in the next step. This is very important when you are interacting with the system. Since the bottleneck of the volume renderer is rasterization, by zooming in or out, the frame rate can increase or decrease depending on the size of the faces being drawn. Dynamically adjusting the LOD ensures that frame rates remain constant. We use 10 frames per second as a target frame rate for a good balance in interactivity and quality.

As described above, HAVS requires a pre-integrated lookup table to composite the image-space fragments. This table is represented as a 3D texture that looks up a front fragment scalar, a back fragment scalar, and the distance between the fragments. Unfortunately, we remove samples, thereby introducing intervals that are larger than the maximum edge length from which the lookup table is built. In a software implementation, this problem could be resolved by repeatedly compositing the fragment until the gap has been filled similar to Danskin and Hanrahan [15]. However, this repetition does not map to current hardware. To solve this issue we create a separate lookup table, which is scaled to handle rays that span the entire bounding box of the mesh. This secondary lookup table is used during the dynamic LOD where the quality difference is not as noticeable and is replaced by the original table when rendering full quality images.

5.4 Results

Our experiments were run on a PC with a 3.2 GHz Pentium 4 processor and 2,048 MB RAM. The code was written in C++ with OpenGL and uses an ATI Radeon X800 Pro graphics processor with 256 MB RAM. The Spx2, Torso, and Fighter data sets were used to measure performance and to compare the sampling strategies.

An important consideration with LOD techniques is the preprocessing time to create the data structures necessary for interactive rendering. Table 5.1 shows the three data sets with their sizes and the time required to preprocess the original mesh into static LODs using the different sampling strategies. The results show that all strategies can be preprocessed quickly, which is important in real-world use.

Rendering a tetrahedral mesh using sample-based LOD dynamically is based on the fundamental assumption that the render time decreases proportionally to the number of faces that are sampled. The performance of our volume renderering system is based on the number of fragments that are processed at each frame. This roughly corresponds to the number of faces that are rasterized. In our experiments we computed the time required to render a data set with different sample increments. Figure 5.5 shows the performance of the volume rendering system as number of sampled faces increases. The almost linear scale factor provides a good estimate of the number of faces that would be required to render a data set at a specific frame rate.

Another important aspect of sample-based LOD is choosing a sampling strategy that gives the best approximation of your original mesh. To measure the effectiveness of our sampling strategies, we generated images of each data set at 14 fixed viewpoints using all the sampling strategies and compared the results with full quality results from the same viewpoints. Unfortunately there are not yet accepted methods for accurately predicting magnitude of the difference between two images (as opposed to whether images are perceived as identical where good methods do exist [54]). In the absense of such an ideal method, we compare images with root mean square error (RMSE). These measurements should only be used as a subjective tool

Table 5.1. Preprocessing time in seconds of the different sampling strategies

Data Set	Tetrahedra	Topology	View	Field	Area
Spx2	828 K	17.8	5.3	4.5	13.9
Torso	1,084 K	87.2	11.6	10.5	11.2
Fighter	1,403 K	75.6	15.3	13.9	15.3

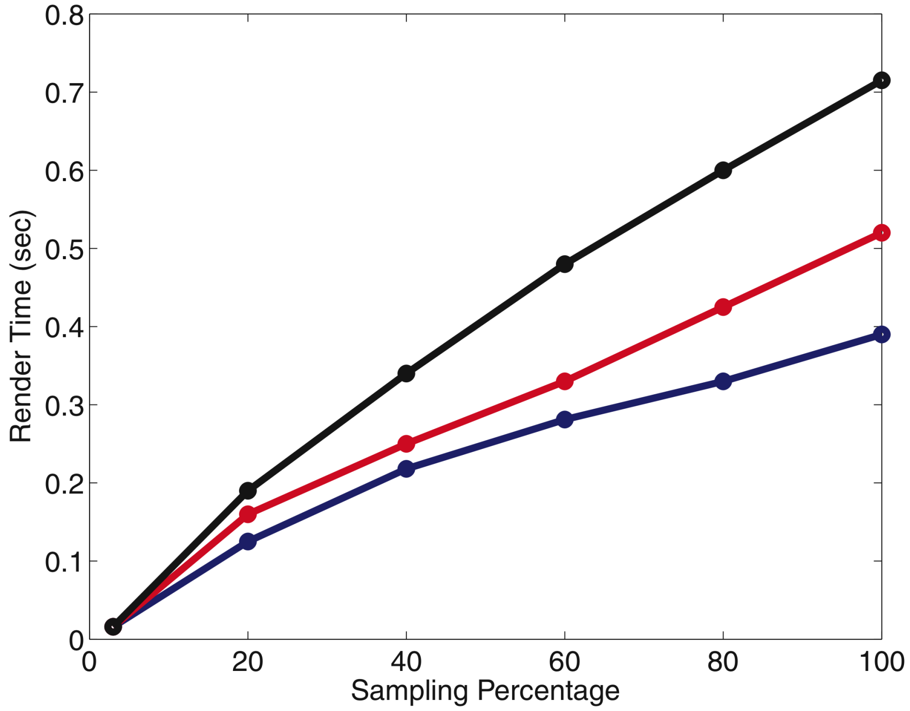


Figure 5.5. Plot of the render time for the Spx2 (blue), Torso (red), and Fighter (black) at different LODs. Approximately 3% LOD is the boundary only and 100% LOD is a full quality image.

for where the images differ as opposed to any quality ranking for LOD strategies. Figure 5.6 shows the three data sets at the different viewpoints and the error measured between each LOD sampling strategy and a full quality image. Notice that no sampling strategy is superior in all cases, but they all provide a good approximation.

To provide a more qualitative analysis of the sampling strategies, we also show images of the different techniques on the same data set. Figure 5.7 shows a direct comparison of the strategies with a full quality rendering.

5.5 Discussion

Our algorithm is designed to meet a speed goal while maintaining what visual quality is possible and in this sense it is oriented toward time-critical systems such as *TetSplat* [53]. Unlike TetSplat, our method aims to do direct volume rendering with transfer functions rather than surface rendering with castaways. Our method also has the advantage that it requires very little additional storage beyond the original mesh. Approaches that sample the unstructured

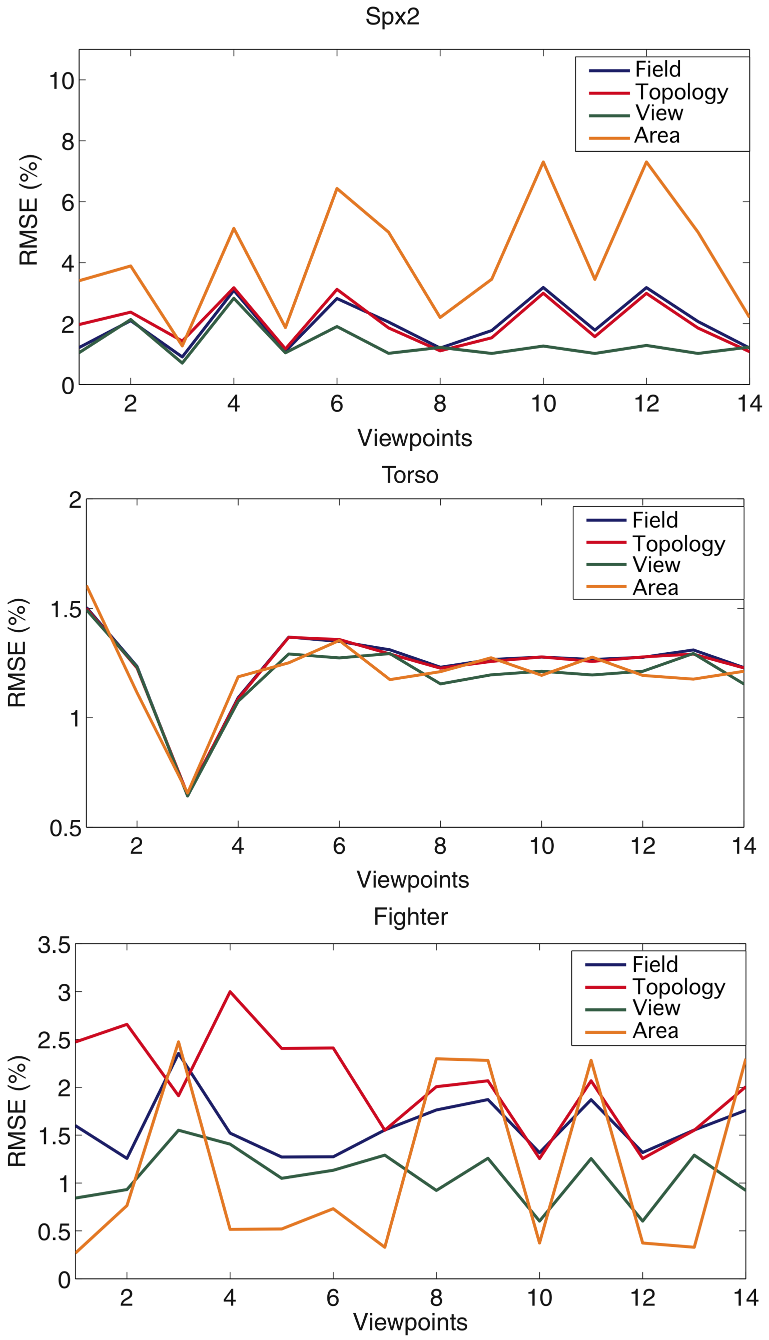
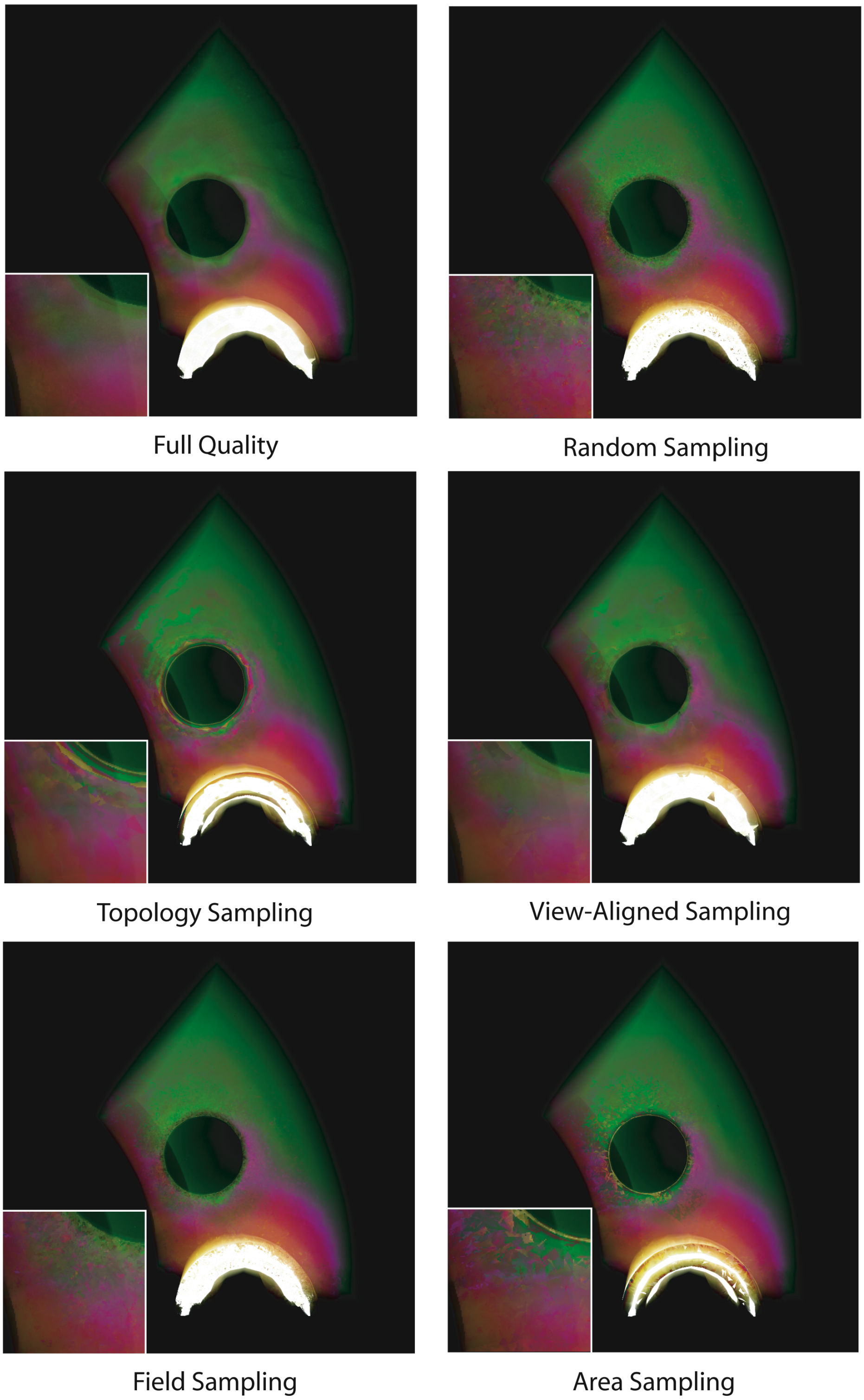


Figure 5.6. Error measurements of the different sampling strategies for 14 fixed viewpoints on the Spx2, Torso, and Fighter data sets. Root mean squared error is used to show the difference between the full quality rendering and the LOD rendering at 10 fps.


Full Quality

Random Sampling

Topology Sampling

View-Aligned Sampling

Field Sampling

Area Sampling

Figure 5.7. Direct comparison of the different sampling strategies with a full quality rendering of the Spx2 data set (800 K tetrahedra). Full quality is shown at 2.5 fps and LOD methods are shown at 10 fps (3% LOD for area sampling and 10% LOD on all other strategies).

mesh on a regular grid [42] increase the original data size substantially and require extensive preprocessing.

Our method is an alternative to rendering explicit hierarchical meshes [8]. While explicit hierarchies are elegant and provide excellent visual quality, they require more preprocessing than our technique, are difficult to implement, and they do not easily allow for continuous LOD. The main advantage of explicit hierarchies over our technique is that they can provide smoother imagery that may be a better visual match to an exact rendering. However, our subjective impression is that our approximate rendering is more than accurate enough for interaction with volume data, and what objective visual accuracy is needed for visualization tasks is an interesting and open question (see Figure 5.8).

A key characteristic of our algorithm is that it operates on mesh faces rather than mesh cells. This results in fewer rasterizations than methods that render cells by breaking them into polygons (e.g., [63, 75]). For example, given *n* tetrahedra, Projected tetrahedra algorithms render 3.4*n* triangles, while HAVS renders only 2*n*. Furthermore, the set of all possible mesh faces to be rendered do not change with viewpoint, so we can leave them in memory on the GPU for better efficiency. Another advantage of our system is that it works with perspective or parallel projections with no additional computation. Most importantly, a face-based method allows faces to be dropped without major errors in opacity because the HAVS method is aware of adjacent faces visible through a pixel, whereas dropping cells in cell-based methods leads to undesirable accumulation of empty space which causes the volume to gradually grow more and more transparent as cells are dropped. So for cells, unlike faces, some explicit simplification hierarchy is required.

Our four sampling strategies each select a subset on the set of mesh cell faces. View-aligned sampling attempts to choose a subset based on view directions, and this emphasizes the quality of individual frames at the expense of coherence between frames. The other three methods choose a subset independent of viewpoint and thus do not have flickering artifacts. Topology sampling builds a set of shells for its subsets and produces images without obvious holes, but does so at the expense of correlated regions in the image. Field sampling uses a randomized algorithm over binned densities to choose a subset. This is a robust strategy but because it completely ignores connectivity it has obvious discontinuities in intensity. Area sampling rewards larger faces and also achieves robustness via randomization. While rewarding larger faces does seem to lower visual error, it also lowers the number of triangles

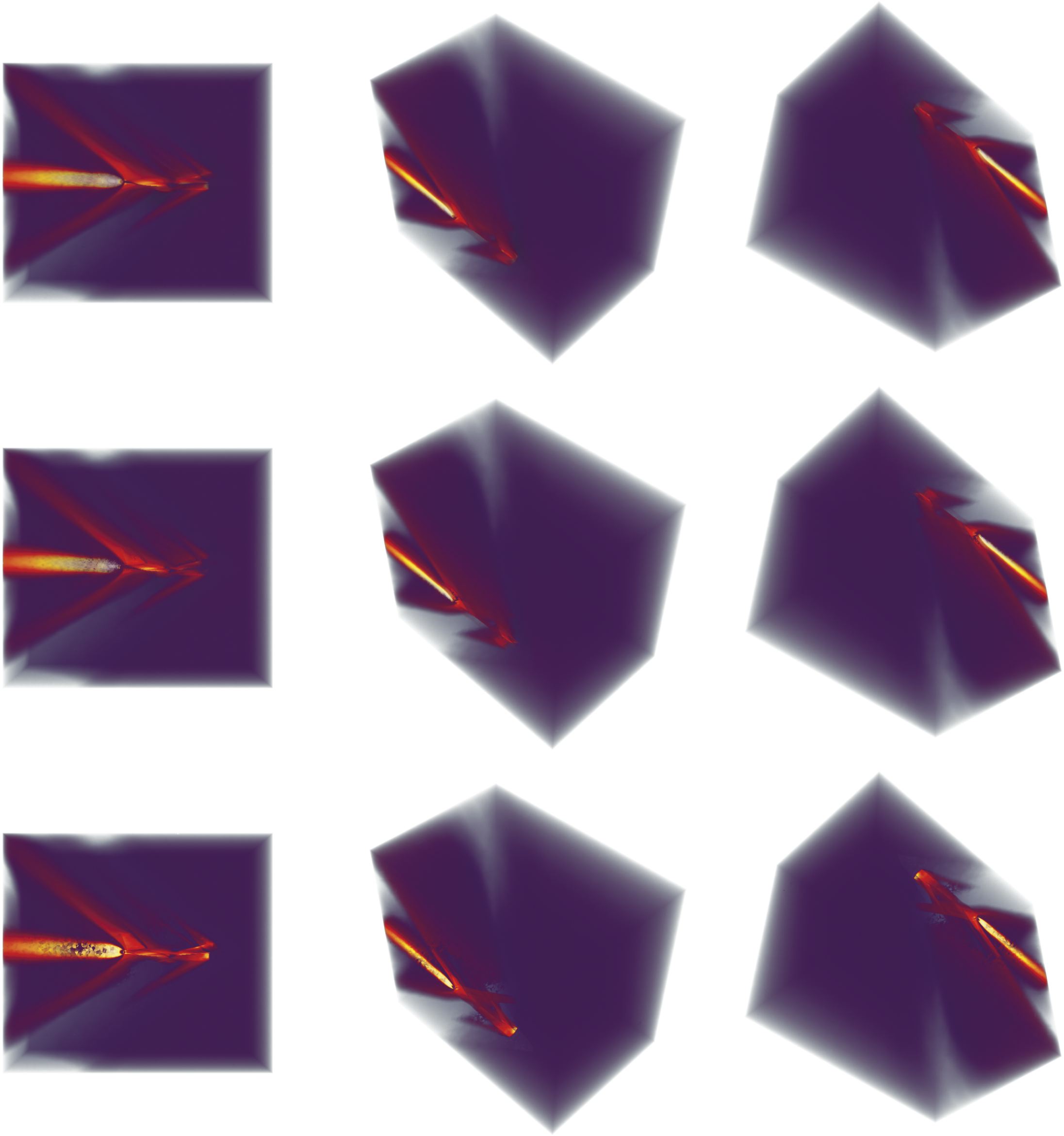


Figure 5.8. The Fighter data set (1.4 million tetrahedra) shown in multiple views at full quality on top (1.3 fps), 15% LOD (4.5 fps) in the middle, and at 5% LOD (10.0 fps) on the bottom. The LOD visualizations use area-based sampling.

that are rasterized because of the larger fill requirements. Overall, each method has its merits and we expect the best choice to be data and application dependent.

CHAPTER 6

CONCLUSION

In this thesis we presented a novel algorithm for volume rendering unstructured data sets with dynamic LOD. 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. We also show how this system can be extended to interactively render large data sets using dynamic LOD.

Our technique can handle arbitrary nonconvex meshes with very low memory overhead. Similar 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 by the available main memory of the system. In this thesis, 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.

Our dynamic LOD technique is based on an alternative solution to the subsampling problem needed for LOD rendering. Instead of using a *domain-based approach*, which maintains different versions of a valid unstructured mesh in a hierarchical data structure our *sample-based approach* is based on rendering with a subset of the mesh faces. These faces correspond to the intersection points of the underlying unstructured grid with rays going through the center of the screen-space pixels. In effect, our technique directly subsamples the volume rendering integral as it is computed. We have shown this to be effective in interactive visualization of unstructured meshes too large for approximation-free rendering. This effectiveness is largely because our technique is particularly well-suited to our volume rendering algorithm.

Last, we would like to re-emphasize 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. In the future it would be useful to do user studies to determine what types of visual error are detrimental to success in visualization tasks to validate the effectiveness of our LOD approach. Such studies could guide which sampling strategies are best. It would also be interesting to see whether our face sampling method would be useful in ray tracing volume renderers as the data reduction might benefit them as well. Another area of future work is to extend our system to handle data sets too large for main memory through the use of compression techniques similar to [18] or out-of-core storage of the data set as in [13]. We would also like to extend our algorithm to handle hexahedron, which should involve either using quadrilaterals directly or splitting them into triangles for efficiency. 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.

APPENDIX A

GPU CODE

A.1 Vertex Program for HAVS

```
!!ARBvp1.0
# --------------------------------------------------------------------------
# Vertex program for Transactions on Visualization and Computer Graphics:
# "Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering"
# (C) 2005 Steven Callahan, Milan Ikits, Joao Comba, Claudio Silva
# --------------------------------------------------------------------------

ATTRIB iPos = vertex.position;
ATTRIB iTex0 = vertex.texcoord[0];
PARAM mvp[4] = { state.matrix.mvp };
PARAM mv[4] = { state.matrix.modelview };
OUTPUT oPos = result.position;
OUTPUT oTex0 = result.texcoord[0];
OUTPUT oTex1 = result.texcoord[1];

# --------------------------------------------------------------------------
# transform vertex to clip coordinates
DP4 oPos.x, mvp[0], iPos;
DP4 oPos.y, mvp[1], iPos;
DP4 oPos.z, mvp[2], iPos;
DP4 oPos.w, mvp[3], iPos;

# --------------------------------------------------------------------------
# transform vertex to eye coordinates
DP4 oTex1.x, mv[0], iPos;
DP4 oTex1.y, mv[1], iPos;
DP4 oTex1.z, mv[2], iPos;

# --------------------------------------------------------------------------
# texcoord 0 contains the scalar data value
MOV oTex0, iTex0;

END
```

A.2 Fragment Program for HAVS

```
!!ARBfp1.0
# --------------------------------------------------------------------------
# Fragment program for Transactions on Visualization and Computer Graphics:
```

```
# "Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering"
# (C) 2005 Steven Callahan, Milan Ikits, Joao Comba, Claudio Silva
# -------------------------------------------------------------------------
#
# The program consists of the following steps:
#
# 1. Find the first and second entries in the fixed size k-buffer list
#    sorted by d (6+1 entries)
# 2. Perform a 3D pre-integrated transfer function lookup using front and
#    back scalar data values + the segment length computed from the
#    distance values of the first and second entries from the k-buffer.
# 3. Composite the color and opacity from the transfer funcion with the
#    color and opacity from the framebuffer. Discard winning k-buffer
#    entry, write the remaining k-buffer entries.
#
# The following textures are used:
#
#   Tex 0: framebuffer (pbuffer, 2D RGBA 16/32 bpp float)
#   Tex 1: k-buffer entry 1 and 2(same)
#   Tex 2: k-buffer entry 3 and 4(same)
#   Tex 3: k-buffer entry 5 and 6(same)
#   Tex 4: transfer function (regular, 3D RGBA 8/16 bpp int)
#

# -------------------------------------------------------------------------
# use the ATI_draw_buffers extension
OPTION ATI_draw_buffers;
# this does not matter now, but will matter on future hardware
OPTION ARB_precision_hint_nicest;

# -------------------------------------------------------------------------
# input and temporaries
ATTRIB p = fragment.position; # fragment position in screen space
ATTRIB v = fragment.texcoord[0]; # v.x = scalar value
ATTRIB e = fragment.texcoord[1]; # fragment position in eye space
PARAM sz = program.local[0]; # scale and bias parameters
                             # {1/pw, 1/ph, (1-1/z_size)/max_len,
                             #  1/(2*z_size)}
TEMP a0, a1, a2, a3, a4, a5, a6; # k-buffer entries
TEMP r0, r1, r2, r3, r4, r5, r6, r7; # sorted results
TEMP c, c0; # color and opacity
TEMP t; # temporary (boolean flag for min/max, dependent texture
        # coordinate, pbuffer texture coordinate, fragment to eye distance)

# -------------------------------------------------------------------------
# compute texture coordinates from window position so that it is not
# interpolated perspective correct.  Then look up the color and opacity
# from the framebuffer
MUL t, p, sz; # t.xy = p.xy * sz.xy, only x and y are used for tex lookup
TEX c0, t, texture[0], 2D; # framebuffer color
```

```
# ---------------------------------------------------------------------------
# Check opacity and kill fragment if it is greater than a const tolerance
SUB t.w, 0.99, c0.w;
KIL t.w;

# ---------------------------------------------------------------------------
# set up the k-buffer entries a0, a1, a2, a3, a4, a5, a6
# each k-buffer entry contains the scalar data value in x or z
# and the distance value in y or w
TEX a1, t, texture[1], 2D; # k-buffer entry 1
TEX a3, t, texture[2], 2D; # k-buffer entry 3
TEX a5, t, texture[3], 2D; # k-buffer entry 5
MOV a2, a1.zwzw; # k-buffer entry 2
MOV a4, a3.zwzw; # k-buffer entry 4
MOV a6, a5.zwzw; # k-buffer entry 6

# ---------------------------------------------------------------------------
# compute fragment to eye distance
DP3 t, e, e;
RSQ t.y, t.y;
MUL a0.y, t.x, t.y; # fragment to eye distance
MOV a0.x, v.x; # scalar data value

# ---------------------------------------------------------------------------
# find fragment with minimum d (r0), save the rest to r1, r2, r3, r4, r5

# r0 = min_z(a0.y, a1.y); r1 = max_z(a0.y, a1.y);
SUB t.w, a0.y, a1.y; # t.w < 0 iff a0.y < a1.y
CMP r1, t.w, a1, a0; # r1 = (a0.y < a1.y ? a1 : a0)
CMP r0, t.w, a0, a1; # r0 = (a0.y < a1.y ? a0 : a1)

# r0 = min_z(r0.y, a2.y); r2 = max_z(r0.y, a2.y)
SUB t.w, r0.y, a2.y; # t.w < 0 iff r0.y < a2.y
CMP r2, t.w, a2, r0; # r2 = (r0.y < a2.y ? a2 : r0);
CMP r0, t.w, r0, a2; # r0 = (r0.y < a2.y ? r0 : a2);

# r0 = min_z(r0.y, a3.y); r3 = max_z(r0.y, a3.y)
SUB t.w, r0.y, a3.y; # t.w < 0 iff r0.y < a3.y
CMP r3, t.w, a3, r0; # r3 = (r0.y < a3.y ? a3 : r0)
CMP r0, t.w, r0, a3; # r0 = (r0.y < a3.y ? r0 : a3);

# r0 = min_z(r0.y, a4.y); r4 = max_z(r0.y, a4.y)
SUB t.w, r0.y, a4.y; # t.w < 0 iff r0.y < a4.y
CMP r4, t.w, a4, r0; # r4 = (r0.y < a4.y ? a4 : r0);
CMP r0, t.w, r0, a4; # r0 = (r0.y < a4.y ? r0 : a4);

# r0 = min_z(r0.y, a5.y); r5 = max_z(r0.y, a5.y)
SUB t.w, r0.y, a5.y; # t.w < 0 iff r0.y < a5.y
CMP r5, t.w, a5, r0; # r5 = (r0.y < a5.y ? a5 : r0);
CMP r0, t.w, r0, a5; # r0 = (r0.y < a5.y ? r0 : a5);

# r0 = min_z(r0.y, a6.y); r6 = max_z(r0.y, a6.y)
SUB t.w, r0.y, a6.y; # t.w < 0 iff r0.y < a6.y
CMP r6, t.w, a6, r0; # r6 = (r0.y < a6.y ? a6 : r0);
CMP r0, t.w, r0, a6; # r0 = (r0.y < a6.y ? r0 : a6);

# -------------------------------------------------------------------------
# find fragment with minimum d (r7) from r1, r2

# r7 = min_z(r1.y, r2.y);
SUB t.w, r1.y, r2.y; # t.w < 0 iff r1.y < r2.y
CMP r7, t.w, r1, r2; # r7 = (r1.y < r2.y ? r1 : r2);

# r7 = min_z(r7.y, r3.y);
SUB t.w, r7.y, r3.y; # t.w < 0 iff r7.y < r3.y
CMP r7, t.w, r7, r3; # r7 = (r7.y < r3.y ? r7 : r3);

# r7 = min_z(r7.y, r4.y);
SUB t.w, r7.y, r4.y; # t.w < 0 iff r7.y < r4.y
CMP r7, t.w, r7, r4; # r7 = (r7.y < r4.y ? r7 : r4);

# r7 = min_z(r7.y, r5.y);
SUB t.w, r7.y, r5.y; # t.w < 0 iff r7.y < r5.y
CMP r7, t.w, r7, r5; # r7 = (r7.y < r5.y ? r7 : r5);

# r7 = min_z(r7.y, r6.y);
SUB t.w, r7.y, r6.y; # t.w < 0 iff r7.y < r6.y
CMP r7, t.w, r7, r6; # r7 = (r7.y < r6.y ? r7 : r6);

# -------------------------------------------------------------------------
# set up texture coordinates for transfer function lookup

MOV t.x, r0.x; # front scalar
MOV t.y, r7.x; # back scalar
SUB t.z, r7.y, r0.y; # distance between front and back fragment
MAD t.z, t.z, sz.z, sz.w; # normalization scale and bias

# -------------------------------------------------------------------------
# transfer function lookup

TEX c, t, texture[4], 3D; # look up pre-integrated color and opacity

# -------------------------------------------------------------------------
# nullify winning entry if the scalar value < 0

CMP c, r0.x, 0.0, c;

# -------------------------------------------------------------------------
# composite color with the color from the framebuffer !!!front to back!!!
```

```
SUB t.w, 1.0, c0.w;
MAD result.color[0], c, t.w, c0;

# ---------------------------------------------------------------------------
# write remaining k-buffer entries

MOV r1.zw, r2.xxxy;
MOV r3.zw, r4.xxxy;
MOV r5.zw, r6.xxxy;
MOV result.color[1], r1;
MOV result.color[2], r3;
MOV result.color[3], r5;

END
```

APPENDIX B

OTHER APPLICATIONS OF THE *K*-BUFFER

B.1 Isosurfaces

A common approach to visualizing unstructured grids is to use isosurfaces that represent the scalar field for a given contour value. Traditional methods for generating an isosurface involve marching through the tetrahedra in the volume and using linear interpolation on the scalar values at the vertices to create triangles [43]. Recently, Pascucci [57] proposed a hardware-accelerated technique for computing isosurfaces using programmable graphics hardware. This is done by extracting the isosurface in a vertex program. We use a similar approach, except that our isosurface computation occurs at the fragment level, which we show to be more flexible for rendering multiple isosurfaces.

The HAVS algorithm operates by roughly sorting geometry in objects space, then finalizing the sort in image space (i.e., on the fragments). This has been shown to be a simple, fast, and extensible approach to volume rendering (see Chapter 4). It also provides a simple framework for isosurface extraction. The general idea of the extracting isosurfaces is to use a 2D lookup table for the isosurface instead of the 3D pre-integrated volume integral table used for volume rendering. This isosurface lookup table is precomputed to represent the front scalar value (v_1) and the back scalar value (v_2) of two fragments and stored on the GPU in a texture (see [59]). The table is regenerated with every contour value change c to provide a simple boolean test, which returns a 1 if $v_1 \leq c \leq v_2$ or $v_1 \geq c \geq v_2$ and 0 otherwise. Thus for every fragment that is rasterized, v_1 and v_2 are determined from the k-buffer and used to test if there is an isosurface that passes between the fragments which is given by (v_1, v_2) in the lookup table (see Figure B.1).

As described, this algorithm provides a simple way to compute a flat shaded isosurface. However, it is often desirable to use direct lighting on the surface of the isosurface. This can be done by computing a normal for each vertex v in the mesh in a preprocessing step using the gradient of the volume. Our k entries are expanded to include a normal for each fragment by storing the normal's x and y components. The z component of the normal can be computed

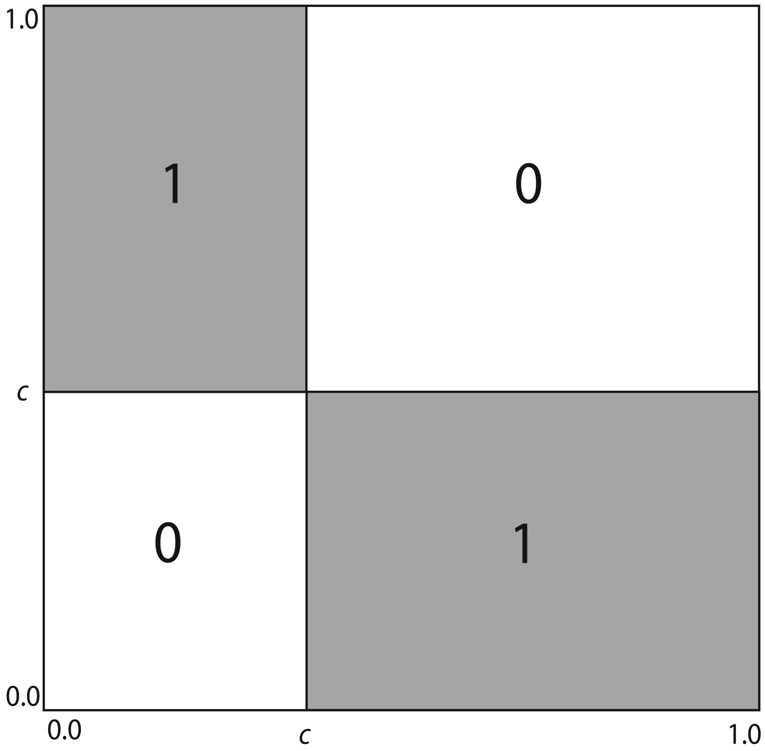


Figure B.1. 2D lookup table for determining if two fragments contain an isosurface between them at contour value *c*.

from *x* and *y*, thus one *k* entry can be condensed to four values (x,y,v,d). Lighting can then be performed by interpolating the normals from *k*-buffer fragment entries f_1 and f_2 and using the lighting equations on that normal. This representation allows a maximum of three *k*-buffer entries ($k = 3$), which is sufficient in the case of isosurfaces as the number of fragments that the contour value crosses are much more likely to be in visibility order than all the fragments. Figure B.2 shows the results of our *k*-buffer isosurfacing algorithm on the Spx data set.

The advantage of this algorithm over one that uses a vertex program is that with a simple extension, multiple isosurfaces can be visualized with transparency. Multiple isosurfaces are drawn by encoding the lookup table to use a different isosurface per *RBGA* channel. Transparency can easily be handled since the fragments are sorted in visibility order and can be composited correctly in the same manner that HAVS composites the volume integral.

B.2 Transparency

Rendering general transparent polygons has been the topic of much research due to the added complexity that compositing requires the geometry to be in visibility order. Everett [20] shows how this can be performed using multiple passes through the scene and *peeling* away

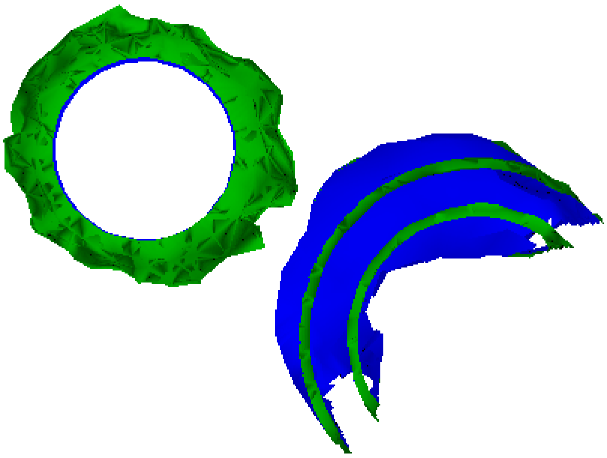


Figure B.2. The Spx data set with an isosurface generated using the *k*-buffer. The front faces of the isosurface are shown in green and back faces of the isosurface are shown in blue.

the front-most polygons at each pass. Krishnan et al. [38] use a similar approach at the fragment level. The Z^3 algorithm [34] uses a fixed amount of storage per pixel to composite the fragments correctly. More recently, Govindaraju et al. [30] describe an algorithm for sorting in image space using the GPU that requires several passes over the data.

Similarly, the *k*-buffer can be used as a means of compositing transparent geometry in the scene. This is accomplished in the same manner that HAVS sorts fragments for volume rendering. To render transparent geometry, each fragment is sorted using the *k*-buffer and composited into the framebuffer. The color and opacity for each fragment can be stored in a texture similar to the one used for the pre-integrated lookup table in HAVS. Since triangle meshes generally have a much lower depth complexity than tetrahedral meshes, performance improvements can be made to avoid redundant sorting. Since the *k*-buffer can successfully composite any *k* nearly-sorted sequence of the geometry, if we know the depth complexity of an object in the scene is less than *k*, then no sorting is required in object-space. Instead, the scene can be sorted in object-space by a collection of triangles instead of by each triangle.

Using the *k*-buffer for rendering transparent geometry requires only one pass and thus provides an efficient algorithm for handling this complex problem. We would like to explore this problem further because it introduces interesting challenges to try to optimize the sorting for complex transparent scenes.

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] 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 Institute, 2004.

[4] S. P. Callahan, M. Ikits, J. L. Comba, and C. T. Silva. Hardware-assisted visibility ordering for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[5] L. Carpenter. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, volume 18, pages 103–108, July 1984.

[6] Y.-J. Chiang and X. Lu. Progressive simplification of tetrahedral meshes preserving all isosurface topologies. *Computer Graphics Forum*, 22(3):493–504, 2003.

[7] P. Chopra and J. Meyer. Tetfusion: an algorithm for rapid tetrahedral mesh simplification. In *Proceedings of IEEE Visualization*, pages 133–140, 2002.

[8] P. Cignoni, L. D. Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):29–45, 2004.

[9] 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.

[10] R. Cook, N. Max, C. T. Silva, and P. Williams. Efficient, exact visibility ordering of unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):695–707, 2004.

[11] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction To Algorithms*, pages 40,127–173. McGraw–Hill, second edition, 2001.

[13] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. iWalk: Interactive out-of-core rendering of large models. Technical Report TR-653-02, Princeton University, 2002.

[14] B. Csebfalvi. Interactive transfer function control for monte carlo volume rendering. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics*, pages 33–38, 2004.

[15] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of Workshop on Volume Visualization*, pages 91–98, 1992.

[16] M. de Berg, M. J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. In *Proceedings of Annual Symposium on Computational Geometry*, pages 294–303, 1997.

[17] X. Décoret, F. Durand, F. Sillion, and J. Dorsey. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics*, 22(3):689–696, 2003.

[18] O. Devillers and P.-M. Gandoin. Geometric compression for interactive transmission. In *Proceedings of IEEE Visualization*, pages 319–326, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

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

[20] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA, 2001. `http://developer.nvidia.com`.

[21] M. J. Fadili, M. Melkemi, and A. ElMoataz. Non-convex onion-peeling using a shape hull algorithm. *Pattern Recognition Letters*, 25(14):1577–1585, 2004.

[22] R. Farias, J. Mitchell, and C. Silva. ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of IEEE Volume Visualization and Graphics Symposium*, pages 91–99, 2000.

[23] R. Farias and C. T. Silva. Out-of-core rendering of large, unstructured grids. *IEEE Computer Graphics and Applications*, 21(4):42–51, 2001.

[24] R. C. Farias, J. S. B. Mitchell, C. T. Silva, and B. Wylie. Time-critical rendering of irregular grids. In *Proceedings of the 13th Brazilian Symposium on Computer Graphics and Image Processing*, pages 243–250, 2000.

[25] L. D. Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings of IEEE Visualization*, pages 43–50, 1998.

[26] L. Freitag, P. Knupp, T. Munson, and S. Shontz. A comparison of optimization software for mesh shape-quality improvement problems. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 29–40, 2002.

[27] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, volume 14, pages 124–133, July 1980.

[28] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of ACM SIGGRAPH*, pages 247–254, 1993.

[29] M. Garland and Y. Zhou. Quadric-based simplification in any dimension. *ACM Transactions on Graphics*, 24(2), Apr. 2005.

[30] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *Proceedings of Symposium on Interactive 3D Graphics and Games*, pages 49–56, New York, NY, USA, 2005. ACM Press.

[31] 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.

[32] S. Guthe, S. Roettger, A. Schieber, W. Straßer, and T. Ertl. High-quality unstructured volume rendering on the pc platform. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 119–126, Sept. 2002.

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

[34] N. P. Jouppi and C.-F. Chang. Z3: an economical hardware technique for high-quality antialiasing and transparency. In *Procceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 85–93, Aug. 1999.

[35] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A GPU-based particle engine. In *Eurographics Symposium Proceedings Graphics Hardware*, pages 115–122, 2004.

[36] 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.

[37] M. Kraus and T. Ertl. Cell-projection of cyclic meshes. In *Proceedings of IEEE Visualization*, pages 215–222, Oct. 2001.

[38] S. Krishnan, C. T. Silva, and B. Wei. A hardware-assisted visibility-ordering algorithm with applications to volume rendering of unstructured grids, 2001.

[39] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization*, pages 287–292, 2003.

[40] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical back-face computation. In *Proceedings of Eurographics Workshop on Rendering techniques*, pages 235–ff., 1996.

[41] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interative texture-based volume visualization. In *Proceedings of IEEE Visualization*, 1999.

[42] J. Leven, J. Corso, J. D. Cohen, and S. Kumar. Interactive visualization of unstructured grids using hierarchical 3d textures. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics*, pages 37–44, 2002.

[43] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of ACM SIGGRAPH*, pages 163–169, 1987.

[44] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of ACM SIGGRAPH*, pages 199–208, Aug. 1997.

[45] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann Publishers, 2002.

[46] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Symposium on Interactive 3D Graphics*, pages 95–102, 1995.

[47] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9:43–55, July 1984.

[48] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 24(5):27–33, 1990.

[49] N. L. Max. Sorting for polyhedron compositing. In *Focus on Scientific Visualization*, pages 259–268. Springer-Verlag, 1993.

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

[51] D. P. Mitchell. Consequences of stratified sampling in graphics. In *Proceedings of ACM SIGGRAPH*, pages 277–280, 1996.

[52] 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.

[53] K. Museth and S. Lombeyda. Tetsplat: Real-time rendering and volume clipping of large unstructured tetrahedral meshes. In *Proceedings of IEEE Visualization*, pages 433–440, 2004.

[54] K. Myszkowski. The visible differences predictor: Applications to global illumination problems. In *Eurographics Rendering Workshop*, pages 223–236, 1998.

[55] M. Newell, R. Newell, and T. Sancha. A solution to the hidden surface problem. In *Proceedings of ACM Annual Conference*, pages 443–450, 1972.

[56] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[57] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Proceedings of IEEE TVCG Symposium on Visualization*, pages 293–300, 2004.

[58] S. Roettger and T. Ertl. Cell projection of convex polyhedra. In *Proceedings of Eurographics/IEEE TVCG Workshop on Volume Graphics 2003*, pages 103–107, 2003.

[59] S. Roettger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of IEEE Visualization*, pages 109–116, Oct. 2000.

[60] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[61] R. Sedgewick. *Algorithms In C*, pages 298–301,403–437. Addison-Wesley, third edition, 1998.

[62] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. In *Proceedings of ACM SIGGRAPH*, pages 249–252, July 1992.

[63] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Proceedings of San Diego Workshop on Volume Visualization*, 24(5):63–70, Nov. 1990.

[64] P. S. Shirley. Time complexity of monte carlo radiosity. In *Proceedings of Eurographics*, pages 459–465, 1991.

[65] C. T. Silva, J. S. Mitchell, and P. L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 87–94, Oct. 1998.

[66] O. G. Staadt and M. H. Gross. Progressive tetrahedralizations. In *Proceedings of IEEE Visualization*, pages 397–402, 1998.

[67] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 83–89, Oct. 1994.

[68] 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.

[69] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer. The randomized Z-buffer algorithm: Interactive rendering of highly complex scenes. In *Proceedings of ACM SIGGRAPH*, pages 361–370, 2001.

[70] H. Warren-Jr. *Hacker's Delight*, pages 261–265. Addison-Wesley, 2002.

[71] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of IEEE Visualization*, pages 333–340, Oct. 2003.

[72] 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.

[73] M. Weiler, P. N. Mallón, M. Kraus, and T. Ertl. Texture-Encoded Tetrahedral Strips. In *Proceedings of Symposium on Volume Visualization*, pages 71–78. IEEE, 2004.

[74] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-of-detail volume rendering view 3d textures. In *Proceedings of IEEE Volume Visualization*, 2000.

[75] J. Wilhelms and A. V. Gelder. A coherent projection approach for direct volume rendering. In *Proceedings of ACM SIGGRAPH*, pages 275–284, 1991.

[76] P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, Apr. 1992.

[77] C. Wittenbrink. R-Buffer: A pointerless a-buffer hardware architecture. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 73–80, 2001.

[78] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *Proceedings of IEEE/ACM Symposium on Volume Graphics and Visualization*, pages 7–12, 2002.

[79] H. Zhang and I. Kenneth E. Hoff. Fast backface culling using normal masks. In *Proceedings of Symposium on Interactive 3D Graphics*, pages 103–106, 1997.