EFFICIENT RENDERING OF VOLUMETRIC IRREGULAR GRIDS DATA

A Dissertation Presented

by

Ricardo Farias

TO THE GRADUATE SCHOOL IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

APPLIED MATHEMATICS AND STATISTICS

State University of New York at Stony Brook

June 2001

State University of New York at Stony Brook

The Graduate School

Ricardo Farias

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Joseph S. B. Mitchell, Dissertation Advisor
Professor, Applied Mathematics and Statistics

Cláudio T. Silva, Dissertation Co-Advisor
Adjunct Assistant Professor, Applied Mathematics and Statistics

Esther M. Arkin, Committee Chair
Professor, Applied Mathematics and Statistics

Klaus Mueller
Assistant Professor, Computer Science

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract of the Dissertation

Efficient Rendering of Volumetric Irregular Grids Data

by

Ricardo Farias

Doctor of Philosophy

in

Applied Mathematics and Statistics

State University of New York at Stony Brook

2001

In this dissertation, we show the results of our research on the field of volumetric rendering of unstructured grid data sets. We present and explain in detail our most important results and contributions.

Volumetric rendering is a highly computational intense process. Images generated by this process show informations about the interior of the data, not only about the surface, by considering the data composed by semi-transparent materials.

A technique to be used as a tool for real-time analysis, is required to achieve a minimum rate of 10 frames per second (ideally 30 fps). Even the fastest algorithm for volumetric rendering takes about 3.5 seconds to generate an image of a data set composed by half million cells in a computer with a fast processor, nowadays.

In the first part of our research, we optimized the fastest algorithm (at the time), developed by Bunyk et al. [6], and put together some approximation techniques to speed up the image generation. In a time-critical fashion, approximate images are delivered until the system is given more time, when it generates more and more accurate images up to the exact one.

In the second part, we present a novel, simple, fast, memory efficient, exact and robust volume rendering algorithm based on the sweep paradigm, called **ZSweep**. This algorithm and the further work we developed on it, became the most important contribution of our research. Extensions developed on this algorithm are its adaptation to run on shared memory architectures, results got on SGI machines, and time and memory requirements of an **out-of-core** implementation. Also another simple out of core volume rendering algorithm is proposed.

To my wife Vânia, my son Renato, my daughter Letícia

and

My Parents Roberto Hudson and Myrian

Contents

List of Figures

List of Tables

Acknowledgments

First I would like to express my deep gratitude to my advisor, Professor Joe Mitchell and my Co-Advisor Cláudio Silva, for their support, friendship, encouragement and guidance over the past five years.

I thank also the members of my committee, Professors Esther M. Arkin and Klaus Mueller for their time and useful criticism. Many thanks to the AMS Department staff, in particular, Scott Connelly, Claire Dugan, Nancy Policastro, Loretta Budd, and Laura Colucci for their help and patience, making this hard period of my life much easier.

Thanks to Brian Wylie, Dino Pavlakos and Pat Crossno from Sandia National Labs, for their contributions to my work. To all of our Computational Geometry group students, Jim Klosowski, Changkil Lee, Xinyu Xiang, Tsung-Chin Ho, Nenad Jovanovic, Petr Konecny, Saurabh Setia and Pyiush Kumar for helpful discussions, mutual help and friendship.

I would like to thank Paul Bunyk and Cláudio Silva for making their codes available. Also thanks to the the graphics community, who so often share through the internet their hard-developed code. Thanks to Peter Williams and Will Schroeder for providing some interesting datasets, NASA for making available the Blunt Fin, Liquid Oxigen Post and Delta Wing datasets.

Thanks to many friends from Brazil who helped me not only with motivation, but with their friendship during the dark period of adaptation and qualifying exams: Professor Antônio Oliveira, Professor Alex Motta Borges; my friends from LCG, Luiz Marcos Garcia Gon calves, Marcelo Eugênio Kallmann, José Luis de Souza Pio, Fernando Wagner da Silva, Antônio Lopes Apolinário Junior, Antônio Alberto Silva Junior, Victor Toso, Gilson Antonio Giraldi, José Augusto Pereira Brito and José Maria Ribeiro Neves. Thanks also to Murilo and Clarice Camargo for the short but high quality time we spent together.

I thank my family for supporting me during all these years. My nephews Anderson and Heverson whose sporadic visits always comforted me a lot. Special thanks go to my dad Roberto Hudson and my mom Myrian, who are also my friends and have always supported my crazy dreams, and without whom nothing would have been possible. Thanks to my sister Jane, brothers Hudson and Edson who completed my life helping me to become who I am today. Thanks go also to my inlaws Evando and Maria Madureira. Most of all, I thank my wife Vãnia, for her friendship and support during my graduate studies, in all aspects, my son Renato and daughter Letícia who have always been the reason for my persistence and determination. I finally thank *God* for putting so many wonderful people in my life.

Thank you all.

Funding for this research has come from a variety of sources. I received direct (e.g., stipend) funding from CNPq – Brazil (Ph.D. fellowship), Sandia

National Labs and the Office of Applied Mathematics of the U.S. Department of Energy, and the National Science Foundation. Money for equipment and other purposes have been provided by grants and donations from the National Science Foundation, the State University of New York at Stony Brook, Sandia National Labs and Compaq.

Chapter 1

Introduction

Volume rendering is the sub-field of the *visualization* with the goal of generating 2D images from 3D volumetric data sets.

The importance of volume rendering comes from the fact that data generated from computer simulations in sciences such as fluid dynamics, finite element analysis, aerodynamics, etc, and also data acquired from satellites for weather forecast, terrain mapping, etc, result usually in large data files of giga-bytes in size. While all five human senses are very limited, our vision sense has the important characteristic of allowing us to capture information at a rate on the order of 100 million bits per second, since such information is presented to us in the appropriate format: images.

To be able to analyze such data sets, we need to "see" them, and the images generated must be consistent with our common sense. For instance, if the data represents clouds, then the more dense the cloud, the darker it should be displayed on the screen. Also, the hotter a region in space is, the brighter in red color it should be shown. This coherence between the data, the type of experiment and the appearance of the image generated, makes it possible for us to visually analyse this huge amount of data in a reasonable amount of time.

Besides *color coherence*, just mentioned, another image characteristic crucial for its correct analysis, is the definition. It will dictate how clearly details of the data (either experiment or simulation) will be shown, making it easier and faster to precisely analyze of the data. It is usual to use approximate rendering algorithms to interactively find a position and angle of observation, and then apply a precise and more expensive rendering algorithm to generate the exact image. For example, iso-surface extraction can make use of graphics hardware to deliver images at interactive frame rates. In areas like surgery, in medicine, it is desirable that the image generated even during the *fly through*, is as precise as possible, to avoid mistakes.

To achieve this goal, computer scientists are actively investigating new algorithms and techniques to speed up the volume rendering process. As will be explained in Chapter 2, data can be represented as regular or irregular grids, while a more general representation would be unstructured, where no connectivity is considered. In the last decade, fast and exact algorithms were developed and optimized for regular grid data (which utilizes implicit grid information), achieving the desired real-time frame rate (about 30 frames per second). It made this kind of representation very attractive for scientists in various areas of research. But it was noticed that data density is not homogeneously distributed in space, and that a regular grid representation becomes quickly prohibitive, as the amount of information grows, due to the high level of redundancy.

A more compact and efficient representation for volumetric data is an irregular grid. The grids can be refined in regions of high density of information and sparse in low density regions. A drawback of such efficiency in space representation is that more adjacency information must be taken into consideration, making it much harder to perform rendering. This is the focus of our research. In this dissertation we present our results in this field.

Dissertation Outline

This dissertation is organized as follows:

In Chapter 2 we present a brief review of volume rendering and the most important efficiency issues related to the parallelization of rendering algorithms.

In Chapter 2.3.3 we describe the development of a tool for time critical scientific visualization. We integrated a fast ray-tracing algorithm with some approximation methods, in both image and object spaces, that allowed us to trade accuracy for speed in the image generation process. Both the original and the simplified meshes are kept in memory, and the system delivers approximated images, showing more accurate ones when more time is available.

In Chapter 4 we propose a novel volume rendering algorithm we call *ZSweep*. The algorithm is based on the sweep paradigm; see [56]. By sweeping all points in the data in increasing z direction the algorithm projects the faces of the cells incident on the point. The intersections for each pixel are kept in an ordered list that is used in a future step of compositing. The efficiency arises from the fact that the algorithm exploits the implicit (approximate) global ordering that the z-ordering of the vertices induces on the cells that are incident on them. Simplicity, speed, memory efficiency, and robustness were the contributions of this algorithm is to the field of volume rendering.

In Chapter 5 we present the first parallelization results of the *ZSweep* algorithm. Details about modifications necessary to avoid costly overheads and to achieve a good load balancing are discussed. Also some cache coherence analysis is given.

In Chapter 6 we propose two *out-of-core* volume rendering algorithms. The first is a simple scheme of simple implementation that presents the desirable characteristic of using a constant amount of memory, independent of the size of the image required, we called memory insensitive rendering (MIR). The second algorithm is more involved and is an extension of the ZSweep algorithm, which brings down the memory requirement for the original algorithm implementation.

Finally, Chapter 7 summarizes our work and gives some future directions.

Chapter 2

Review

In this chapter we review the basic concepts of volume rendering and issues related to its parallelization. We start with a brief overview about data representation.

2.1 Data Representation

Volume rendering is a subfield of visualization, which is an important field of study in computer graphics. The goal of volume rendering is to create (two-dimensional) images of (three-dimensional) volumetric data.

A volumetric dataset consists of information at sample locations in space. The information may represent a *scalar* field, such as density in a computed tomography (CT), or a vector field, such as velocity in a flow field, or even a combination between these two, such as, energy, density, and momentum in a computational fluid dynamics simulation.

Volumetric datasets are often represented in rectilinear grids, as a $3D$ grid of volume elements, called *voxels.* Each voxel is a unit of the volume and has associated with it some property of the object or phenomenon being studied. If all voxels are identical cubes the dataset is said to be *regular.* Examples are found in medical data, simulations of fluid dynamics experiments, and finite element models. A variation on regular grids is *curvilinear grids*, which can be thought of as the result of a "warping" of a regular grid. *Unstructured grids* consist of arbitrarily shaped polyhedral cells, with no particular relation to rectilinear grids. In this work, we focus on presenting efficient solutions for visualizing volumetric datasets given as unstructured grids.

The most common techniques of volume rendering are *ray-casting*, *splatting*, *shear-warp* and *texture mapping* based rendering. All of these techniques can be applied to regular datasets, but only ray-casting and texture mapping (by means of 3D texture mapping), can be applied to unstructured grids datasets.

Since we deal only with unstructured grids, we point to [50] as a reference giving a survey and comparison among the techniques applied to regular grid data.

We start by defining some important terms and concepts. We are given a set V of n points in 3D space ($\Re^3$). If no connection between the points is given, such set is said to represent a *point cloud.* In unstructured grids, connections are given between the points, which serve to organize the points into a polyhedral decomposition of $\Re^3$ into *vertices*, *edges*, *faces*, and *cells*:

- Vertices are the input points V; vertices are 0-dimensional.

- Edges are line segments connecting two vertices; edges are 1-dimensional.

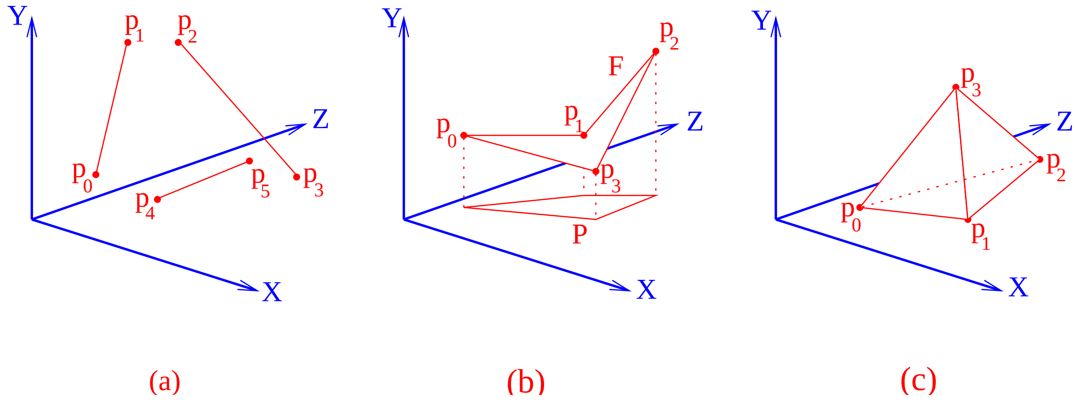


Figure 1: Basic geometrical connectivities. (a) Edges connect pairs of points. (b) Points p_0, p_1, p_2 and p_3 are connected in a cycle that forms the boundary of the face F. To give a better 3D impression, we show the projection, P, of F on the (x, y)-plane. Note that faces that are determined by more than 3 points may be non-planar. (c) A tetrahedral cell is defined by four triangular faces.

See Figure 1.a.

- Faces are 2-dimensional polygons represented by a cycle of edges. See Figure 1.b.
- Cells are 3-dimensional connected regions bounded by the faces. See Figure 1.c.

Faces can be defined by three or more points. An important issue is that when a face is determined by more then three points, it cannot be guaranteed that the points are coplanar. This can lead to problems in the interpolation step of the rendering procedure, as we discuss later.

We emphasize the distinction between a surface embedded in the 3D space, bounding a 3D object, and an actual solid model of a 3D object. See Figure 2.

3D objects are represented by (3-dimensional) cells, while a 2D surface is represented only by a union of triangles or polygons, with no explicit data representing the points in 3D that comprise the body of the object surrounded by the surface. When generating an image using only the boundary faces, as polygon rendering techniques do, no information about the interior is taken into account.

If all cells of an unstructured grid are tetrahedral, we say that such a dataset is represented by a *tetrahedral grid*. If the cells are given as hexahedra (each having six faces), the dataset is said to be a *hexahedral grid* dataset. We explicitly mention these two types of cells, since these are the most common irregular grid representations found in the literature.

In Section 2.3.3 we describe the ray-casting algorithm, proposed by Bunyk et al [6]; it uses cell connectivity to figure out the list of cells intersected by the ray cast through each pixel of the image. In Section 4 we discuss the previous works based on the *sweep* paradigm, including the *ZSweep* algorithm that we propose in this thesis. In Section 5 and Section 6 we present the extensions we have made to *ZSweep*, including the implementation and analysis of its parallelization for shared-distributed memory architectures. We also describe its adaptation to the *out-of-core* framework, in which the image is generated in parts, and only the minimum data information necessary to generate each part of the image is read in to memory at a time, allowing the visualization of much larger datasets.

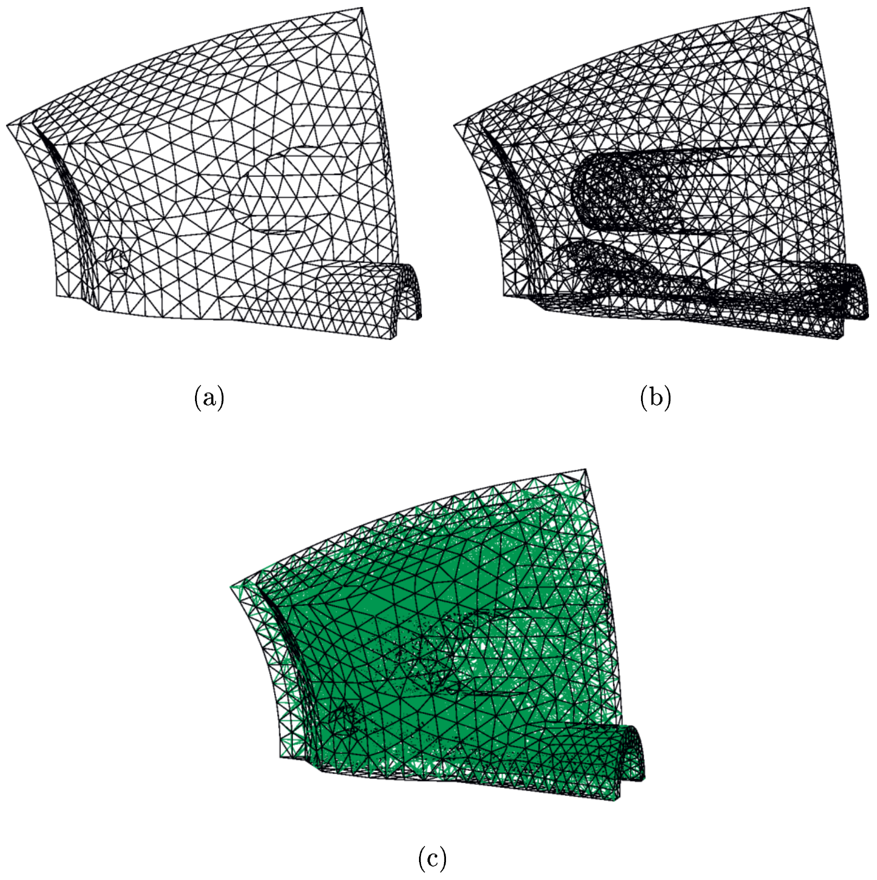


Figure 2: As an example of real 3D object we use the tetrahedral mesh for the dataset SPX. (a) Frontal boundary faces. (b) All boundary faces. (c) The entire mesh, where the edges of the interior faces are shown in green. Note that the mesh shown in (b) is a 2D surface, while the mesh shown in (c) is a real 3D mesh since information about its interior is considered by taking into account interior faces.

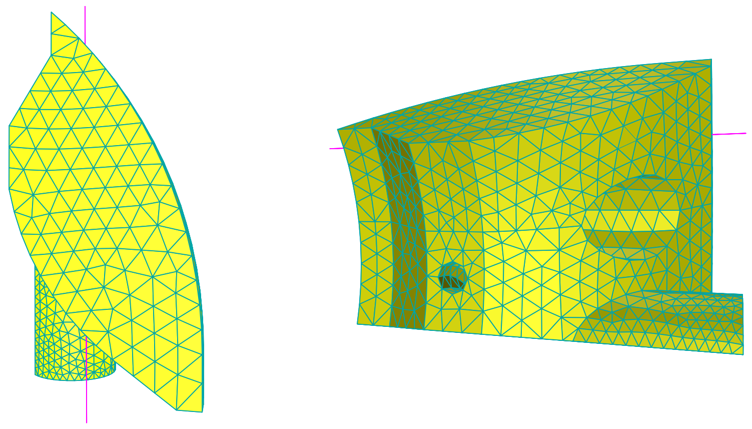
2.2 Volume Rendering

We now review some of the basic ideas and concepts of volume rendering.

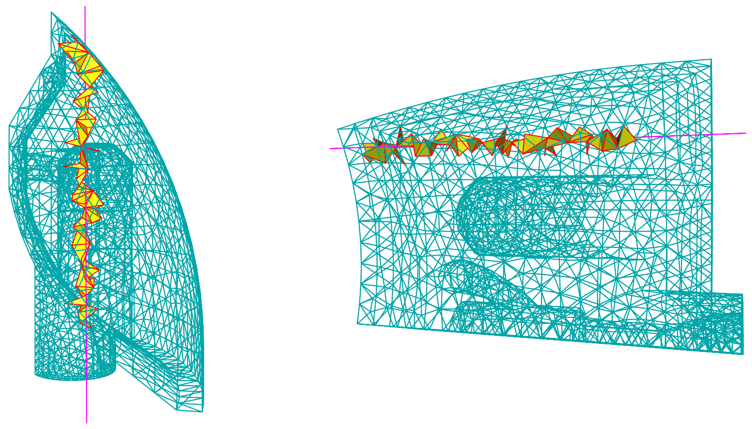
The rendering process is a very computational intensive and complex process. To control each step of the process, while making optimizations possible, it is usual to break the whole process into a sequence of small steps. Such a sequence is called the *rendering pipeline* and is discussed in the next section. To make our discussion more general, we comment on the general rendering pipeline, which applies to both polygon rendering and volume rendering.

In polygon rendering, each pixel is assigned the characteristics of the closest intersection among all intersections between the ray cast through the pixel, and all polygons in the scene. In contrast, in volume rendering, the final color of a pixel comes from the sum of the contributions from each cell intersected by the ray. Even though at first it may seem that in both cases we would have to find all intersections between the ray for each pixel and all faces in the scene, in polygon rendering one is often able to eliminate many of the faces before starting the more expensive steps of the pipeline. For each step discussed below, we note if it is applicable only to polygon rendering or also to volume rendering.

Finally, we refer to Figure 3, where we show from two different points of views the results of applying a ray tracing (Figure 3 (a)), where only information from the boundary faces can be taken into account to generate the final image, and of applying ray casting (Figure 3 (b)), where the ray penetrates the volume, instead of stopping at its surface. In volume rendering the faces are considered to be semi-transparent, allowing the ray of light to pass from



(a) **Surface Information Only**



(b) **Interior Information Considered**

Figure 3: This is the data set SPX, seen from two different view points. (a) Only surface information is shown for both viewing directions. (b) In addition to the external edges, we show all faces intersected by the ray cast through the pixel (160,230) when generating a 400-by-400 image.

cell to cell. Ray casting determines for each pixel a list of intersections that will be used to compute the *lighting integral*, resulting in the pixel's final color.

Each step of the *Volume Rendering Pipeline* groups a set of operations necessary to perform a specific part of the whole process. The pipeline subdivision depends on the particular implementation requirements. We will adopt the following subdivision; see Figure 4. For convenience we split the pipeline into two stages, the geometric stage and the rasterization stage.

Here we adopt the term *model* or *object* to indicate a connected portion of the dataset; a single file may can contain many (disconnected) models.

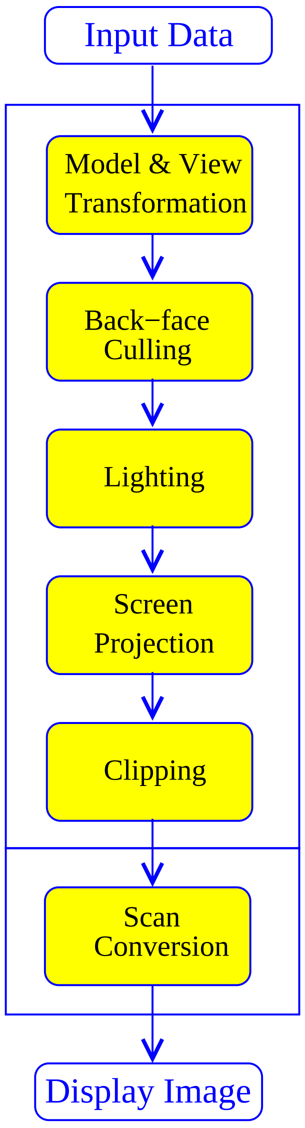
2.2.1 Volume Rendering Pipeline: Geometric Stage

Transformations and computations in the object space are performed at this stage.

Modeling and Viewing Transformation

When first read into memory, an object resides in its on *object space* coordinates. Each object has associated with it a sequence of transformations to achieve the desired position and orientation relative to the world coordinate system. The world space is unique, allowing all objects to be treated uniquely throughout the pipeline.

To decrease the amount of work to be done, only the *visible* part of the data has to be rendered. To determine the vertices and cells that must be taken into account, a *view transformation* is performed. This transformation places the camera (or observer) in the origin of the world, looking in a specific direction,



(a)

Figure 4: The *Volume Rendering Pipeline.*

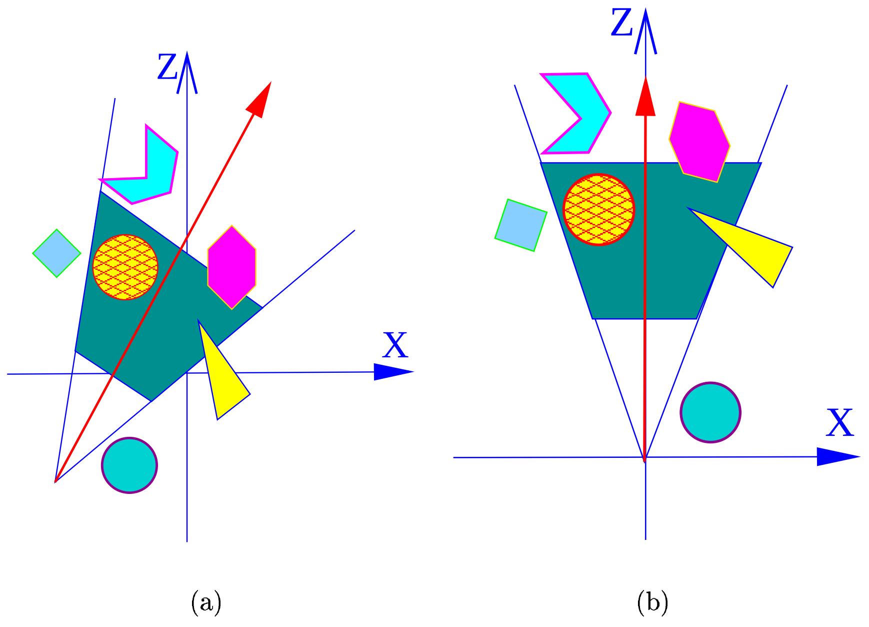


Figure 5: On the left, the camera is positioned as the user specified. The viewing transformation translates the view point to the origin, looking toward the negative z-axis direction. It makes clipping and projection operations simpler and faster. This procedure applies to both perspective and orthogonal projections.

usually the negative z-axis. After applying these two transformations to all data, the objects are now said to lie in the *eye space*. See Figure 5.

Back-Face Culling

This step, applicable only to polygon rendering, eliminates all faces whose normals make angles smaller than $\pi/2$ degrees with the viewing direction. In the case of volume rendering, where the data is defined be interior faces as well, back-face culling is not well defined, since the back of one cell is most likely the front of another cell. If the scene contains both types of models (polygonal and volumetric), the application must be prepared to handle both models, differently.

Lighting

This is the step of the *Volume Rendering* pipeline in which the color for each pixel of the image is computed. Various physical models can be used. The more elaborate the model, the more realistic the images that are generated and the more expensive and slower this step of the pipeline becomes.

Optical models used in volume rendering are based on physical models of interaction between light and matter. The mathematical equation for this purpose is known as the *volume rendering integral* (VRI) [4, 36, 47, 33]. The model we analyze here is the one we have been using throughout our implementations, where the particles that comprise the matter of the object being visualized are assumed to absorb and to emit light.

For the geometrical considerations used in obtaining the expression for the optical model, we refer to Figure 6. In order to derive the VRI, we are going to analyze the amount of light both absorbed and emitted by a cylindrical cross section of the data with volume given by $V = A\Delta s$, where A is area of the base of the cylinder and Δs is its height. See Figure 6(a). For simplicity the particles that comprise the object are considered to be identical spheres, with radius r, resulting in a projected area of $S = \pi r^2$ on the frontal area of the cross section; see Figure 6(b). Given ρ as the particle density in the volume, there are $N = \rho A\Delta s$ particles in the cross section. For small Δs, the total projected area can be approximated by $N\pi r^2 = \rho A\pi r^2\Delta s$ or $\rho\pi r^2\Delta s$ per unit

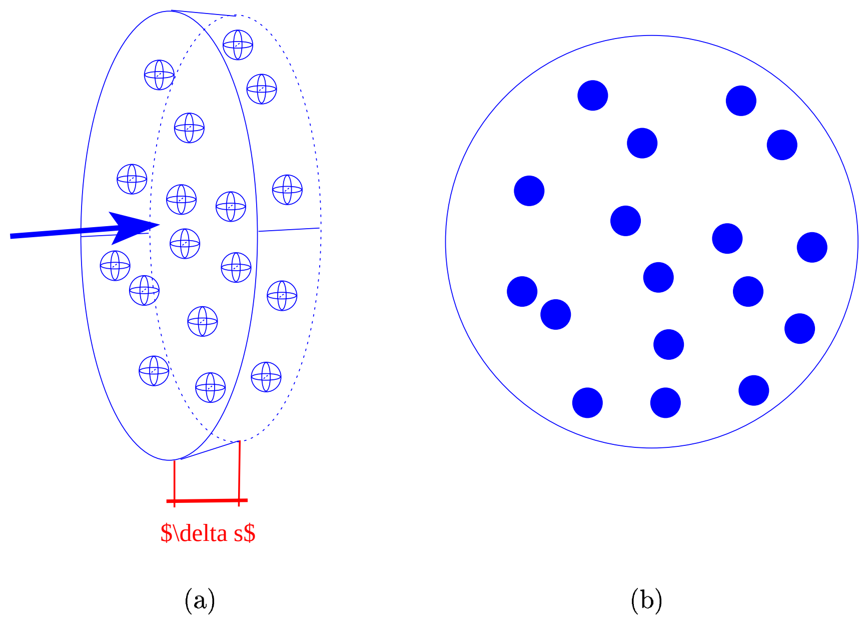


Figure 6: (a) In this cross section of an object, the small spheres represent the particles that comprise the object. The arrow indicates the viewing direction. (b) This is the frontal view of the cross section; the small circles are the projections of the particles on the front face.

area of the cross section. In the limit of $\Delta s \to 0$, no overlap occurs and the exact expression is given by the following differential equation:

$$\frac{dI}{ds} = \rho(s)\pi r^2 \mathrm{I}_e(s) - \rho(s)\pi r^2 \mathrm{I}_a(s) \tag{1}$$

Here the terms $I_e(s)$ and $I_a(s)$ are respectively the light intensities emitted and absorbed by the differential cross section. It is usual to express $\tau(s) = \rho(s)\pi r^2$.

$$\frac{dI}{ds} = \mathrm{E}(s) - \tau(s)\mathrm{I}_a(s) \tag{2}$$

.

$E(s) = \tau(s)I_e(s)$ is called the *emission term* while $\tau(s)$ is called the *absorption coefficient* in the second term on the right-hand side of Equation 2.

This equation can be solved by passing the second term on the right-hand side to the left-hand side, multiplying the whole equation by the term $exp\,(\int_0^s \tau(s)dt)$, and performing the integration with respect to s, from 0 to z, on both sides of the equation. The result is:

$$\mathrm{I} = \mathrm{I}_0\mathrm{T}(z) + \int_0^z \mathrm{E}(z)\mathrm{d}T(z)\mathrm{d}z \tag{3}$$

The term $\mathrm{T}(z)$ can be understood as the "transparency" of the object. The transparency and the *opacity*, $\mathrm{O}(z)$, are related as follows: $\mathrm{T}(z) = 1 - \mathrm{O}(z)$. Considering the intensities I as the intensities in terms of color, absorbed or emitted by the particles in the object, we can rewrite equation 3 as

$$\mathrm{C}(z) = \mathrm{C}_0 + \int_0^z c(z)\,(1 - \mathrm{O}(z))\,\mathrm{d}z, \tag{4}$$

and the opacity $\mathrm{O}(z)$ is obtained from

$$\mathrm{O}(z) = \mathrm{O}_0 + \int_0^z \mathrm{o}(z)\mathrm{d}z. \tag{5}$$

This can be approximated as

$$\mathrm{O}_n = \mathrm{O}_c + \frac{1}{2}\left(\mathrm{o}_c + \mathrm{o}_n\right)\Delta z. \tag{6}$$

Finally, we obtain an analytical expression for the solution to Equation 4, considered up to its second degree term:

$$\mathrm{C}_n = \mathrm{C}_c - \frac{1}{2}\left(\mathrm{c}_c + \mathrm{c}_n\right)\left(\mathrm{O}_c - 1\right)\Delta z - \frac{1}{24}\left(3\mathrm{c}_c\mathrm{o}_c + 5\mathrm{c}_n\mathrm{o}_c + \mathrm{c}_c\mathrm{o}_n + 3\mathrm{c}_n\mathrm{o}_n\right)\Delta \mathrm{z}^2. \tag{7}$$

Note that we changed the limits of integration from the range $(0, z)$ to the range (z_c, z_n), in order to correspond to the integration from the *current* z to the *next* z.

Projection

As we reach this point in the pipeline, all data is ready to be normalized to fit on the screen. Remember that the data lies in the *view space*, usually the unit-radius cube, with extreme points at $(-1, -1, -1)$ and $(1, 1, 1)$. Depending on the application's needs, one can decide to use either *orthographic* projection (also called *parallel* projection) or *perspective* projection. Both transformations can be represented by a $4x4$ matrix and are well covered in the literature, so we omit explicit review here.

Clipping

Only the primitives completely or partially inside the view volume need to be passed to the next step of the rendering pipeline, which then draws them on

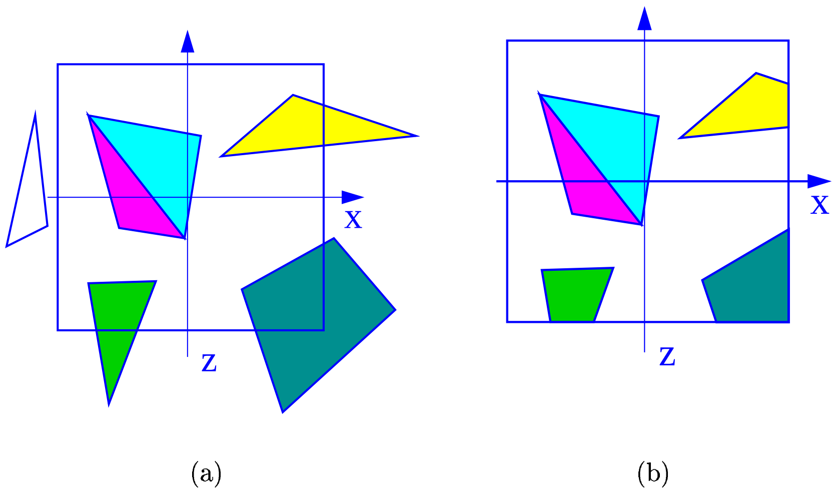


Figure 7: Clipping a set of triangles.

the screen. Among such primitives, those that lie partially inside the viewing volume require clipping; see Figure 7. Clipping is applicable in both polygon and volume rendering algorithms.

2.2.2 Volume Rendering Pipeline: Rasterization Stage

After all transformations have been performed in the object space, we have the data ready to generate the final $2D$ image to be displayed on the screen. This process is the *rasterization* or *scan conversion*. A very simple way to think about this step of the pipeline is to consider a triangular face of the data, compute its bounding box, and make a double loop in X and Y coordinates, and for each pixel inside the bounding box, compute its z coordinate. If scalar field values (which are given at the vertices of the input) is also to be taken into account, one can compute them by means of a bilinear interpolation of the scalar values defined on each of the three vertices. We use this information to compute the contribution for the three color components, *(r,g,b)*, for each pixel, and also the opacity. Modern video cards have support for triangular face scan conversion, which accelerates this process.

The color computed for each pixel is in the format (r, g, b, o), where the o stands for the opacity; these values are kept in a buffer called the *color buffer*. It is usual to use two buffers at the same time. All drawing is performed in one buffer and it is only sent to the screen when all of the primitives have been scan converted. At this time, we start drawing in the second buffer and when it is completed, it is flushed, and so on. The double buffer technique avoids undesired effects, such as flicking, while the program is drawing each primitive.

There is another buffer called the *Z-Buffer*, which keeps for each pixel the z coordinate of the face closest to the observer. By comparing the z value for each new intersection for each pixel, it is possible to decide which value must be kept. The *Z-Buffer* resolves occlusion problems in hardware very inexpensively. Note that the *Z-Buffer* is applicable only to opaque objects in the scene. In scientific visualization, where all data is assumed to be semi-transparent, this use of the *Z-buffer* is not applicable.

There is another buffer designed to handle transparency between two faces – the *stencil buffer*. To use this buffer for volume rendering, extra care must be taken, since the faces must be sent to it in the correct depth order, which requires an ordering on the cells (often obtained from a topological (partial) ordering of the cells). Furthermore, if the faces in the data contain "cycles" in the visibility (partial) ordering, it is impossible to obtain a correct order, unless some of the faces involved in the loop are cut, in order to break cycles.

2.3 Parallel Rendering Basics

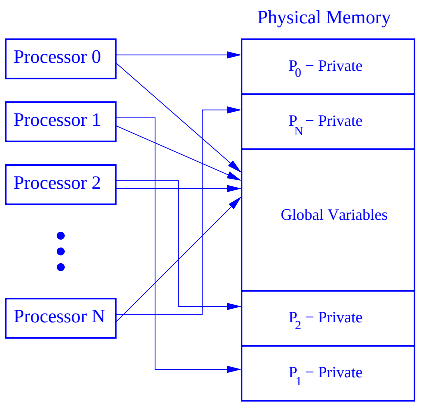
For a more thorough discussion of parallel rendering background, we refer the reader to [16]; the discussion we give in this section contains only the minimum information necessary for the reader to grasp the basic ideas about parallel processing applied to volume rendering.

An algorithm is said to be *embarrassingly parallel* if it can be easily parallelized, with an implementation that performs little or no inter-process communication, and a very small overhead is introduced due to parallel issues.

2.3.1 Parallel Architectures

Parallel architectures is an extensive subject. Here we give just a brief idea about the two most important classes of parallel architectures: *shared memory* and *distributed memory* architectures. For further background on this topic, see [17].

For *shared memory* machines, all of the memory available in the machine can be accessed by any one of the processors. When executing a parallel code, the operating system creates the first *thread*, called the *parent thread*. The parent thread is responsible for creating all of the *global variables*, which will be accessible by any process in the parallel part of the code. After reading all necessary data and allocating memory space for all global variables, it reaches the part of the code where the parallel commands will order the operating



(a)

Figure 8: Parallel implementation considerations. Each processor has its on private set of variables, while all processors are able to access a common memory area for the global variables. Two implementation issues must be carefully considered for the sake of efficiency: modifying global variables only when necessary, and memory locality for cache coherence.

system to create the desired number of threads or processes. Note that the number of threads has no connection to the number of processors in the machine. One can create, say, five threads in a one-processor machine, and the processor time will be shared between the processes, managed by the operating system. If the number of threads one creates is less than or equal to the number of processors in the machine, the operating system assigns one thread for each available processor. See Figure 8.

When the operating system creates all threads, any variable created by one thread can be accessed only by itself (private variables), but the global variables can still be accessed by all threads. This means that global data can be manipulated by any of the processors in a transparent way, making the access the same as usually experienced in a serial program.

Parallelizing a serial algorithm into this type of architecture is often straightforward, but, in order to achieve efficiency, care must be taken when it comes to global variables access. All processes can *read* the value of the same variable at the same time without problem. But if the access modifies the content of a global variable, the processor must make sure that no other process will try to modify this same variable at the same time by *locking* its global access permission during the modification. On the other hand, the processor must *unlock* the variable global access as soon as possible to avoid other processes having to wait idle for the variable to have its access freed.

In this type of architecture, *processors* access RAM memory using the *data bus*. The memory access speed is a function of the *data bus band-width*. This dependency imposes a limitation on the scalability of the *shared memory* architecture, or the number of processors that can be added to the configuration.

Some smart schemes were introduced to speed up this architecture while also allowing a greater number of processors to be used. One such scheme is non-uniform memory access, where each processor has a *local* memory module; accessing its own memory is faster than accessing other processors' *remote* memory. In this scheme the *bus band-width* is not as restrictive, as in the previous scheme, allowing a much better scalability. But the cost for *remote* memory access is higher, and the programmer is responsible for avoiding as much as possible its occurrence.

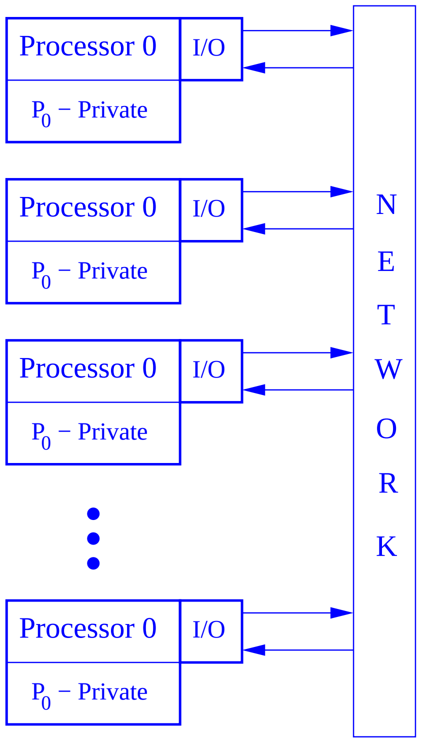
The other important class of architecture is the *distributed memory* architecture. Here, each processor can be seen as a stand-alone computer and communication between two different *CPU*'s must be carried out by means of networking communications. *Message passing* is the most common such technique used. See Figure 9.

In this architecture, access to remote memory becomes very expensive and a different approach for a parallel implementation must be considered. A more elaborate scheme is required to avoid communication as much as possible. For instance, in ray-casting algorithms if each process is assigned with the task of computing the final color of a set of k pixels of the image, only these k sets of (r, g, b) values will have to be sent to the process responsible for assembling the final image. On the other hand, this approach requires a lot of information to be duplicated on each processor. Another possible scheme is to divide the data among the processors, avoiding data duplication; however, this will require each processor to send more information around, increasing the communication cost.

The decision of which scheme to use is highly dependent upon the characteristics of the algorithm chosen, the type of data to be handled, and the specifications of the hardware available.

2.3.2 Types of Parallelism

The types of parallelism one can apply in the volume rendering include *functional parallelism*, *data parallelism*, and *temporal parallelism.* The method to be used depends upon the application and the rendering algorithm to be developed. These methods can also be combined into a hybrid method, which may be more efficient.



(a)

Figure 9: Distributed memory architecture representation. Each processor can be seen as a complete computer, with CPU, memory, and I/O. Communication between two processors is accomplished by means of low-level I/O operations that will send their request to the network layer of the operating system, and wait for its answer. Communication in this architecture is a lot more expensive than in shared memory architecture.

Functional Parallelism

When breaking the rendering process into a set of steps of the pipeline, each of which has a well-defined task or function, we can think of the pipeline as a serial solution for the rendering pipeline. Then, one can assign each of these steps to a different processor and have them compute each step in parallel. Since each step has to finish processing its task before it can send it forward to the next step, one can easily realize that the speed limitation for this type of parallelization is the number of steps in the pipeline, the number of processors involved in the computation, and also the speed of the slowest step involved in the process. See Figure fig:VRPipeline.

Data Parallelism

Another approach is to perform the parallelization on the data itself. The data is divided among the processors, each of which performs the serial rendering on its part of the data. The limitation here is the dependence on the number of processors. Inter-process communication costs must be taken into account and in some cases can result in a considerable overhead in the rendering cost. The choice of rendering algorithm will also depend on these characteristics of the network.

Two classes of data parallelism can be adopted: *object parallelism*, in which data is divided in the object space, and *image parallelism*, in which the image generation task is divided in the image space and each processor is assigned with one or more parts of the image.

Bottlenecks can be avoided by implementations that exploit both object and image parallelism. Achieving a good balance between them is a hard problem, since the workload involved in each level is highly dependent on factors such as scene complexity, average screen area of transformed geometric primitives, sampling ratio and image resolution.

In short, load balancing is a crucial factor for the algorithm efficiency. This issue will be discussed in more detail in Section 2.3.3.

Temporal Parallelism

In order to generate animations, it is necessary to create thousands of high-quality images. In this case, the time to create each image is not as important as the time to create the whole animation sequence. Thus, the whole animation sequence can be broken into smaller parts and each part can be assigned to a different processor.

2.3.3 Parallel Programming Efficiency

Implementing a parallel version of an algorithm gives rise to some additional costs not present in serial programming. Such costs are called "parallel overheads," which may be caused by some or all of the following reasons:

- load imbalance,
- inter-process communication,
- additional or duplicated computations, and
- additional memory requirements.

To understand why these overheads appear in parallel implementations of rendering algorithms, we have to analyze some further concepts; some are common to most parallel programming, while others are specific to the parallel rendering field.

Dividing work among processors

Recall that data can be divided in *object space* or in *image space*. While *object space* usually achieves a more uniform division of primitives among the processors, making it a better option for the *distributed memory* architecture, it tends also to require more communication between the processors. Another problem with this type of division is that having the same number of primitives for each process does not guarantee the same amount of work for each of them.

Division in *image space* can result in a very uneven division of work, but since each processor will compute the final color of all pixel assigned to it, minimum communication between processors is necessary. However, a bad aspect of *image space* data division is that primitives can be mapped into several regions of the screen, resulting in redundancy (more memory requirement) and loss of spatial coherence on the boundary of the regions, which are non-existent costs in a serial implementation.

Load Balancing

The efficiency of a parallel system is directly related to how evenly the workload is distributed among the processors. In parallel rendering, there are many issues to be taken into account. In polygon rendering, for instance, the primitives may vary in size, number of vertices, illumination requirements, geometric transformations, etc. Furthermore, the back-face culling and subsequent culling steps of the pipeline can cause more imbalance.

A much more serious source of load imbalance is caused by the non-uniform distribution of the primitives in the image space. Processors responsible for rasterizing dense regions of the image will do significantly more work, while some processors may end up with empty regions. Also the mapping of objects onto the image is view-dependent, changing from frame to frame.

Research in parallel computing has suggested some smart schemes for work distribution, which we now briefly discuss.

Static Data Distribution

In *static distribution* the data is subdivided in a preprocessing step. The work is then assigned to each processor. The process finishes when all processors have finished their tasks. Figure 10 shows several different strategies of image partitioning, each resulting in different load balancing characteristics. Dividing the data in large blocks (Figure 10(a)) usually results in poor load balancing, unless the data is very uniformly distributed in space. Fine-grained schemes (Figure 10(c,d)) result in better work load but may incur more overhead. Results in [71, 70] indicate that square regions (as in Figure 10(b)) tend to minimize the loss of coherence since they have the smallest perimeter-to-area ratio of any other rectangular subdivision scheme.

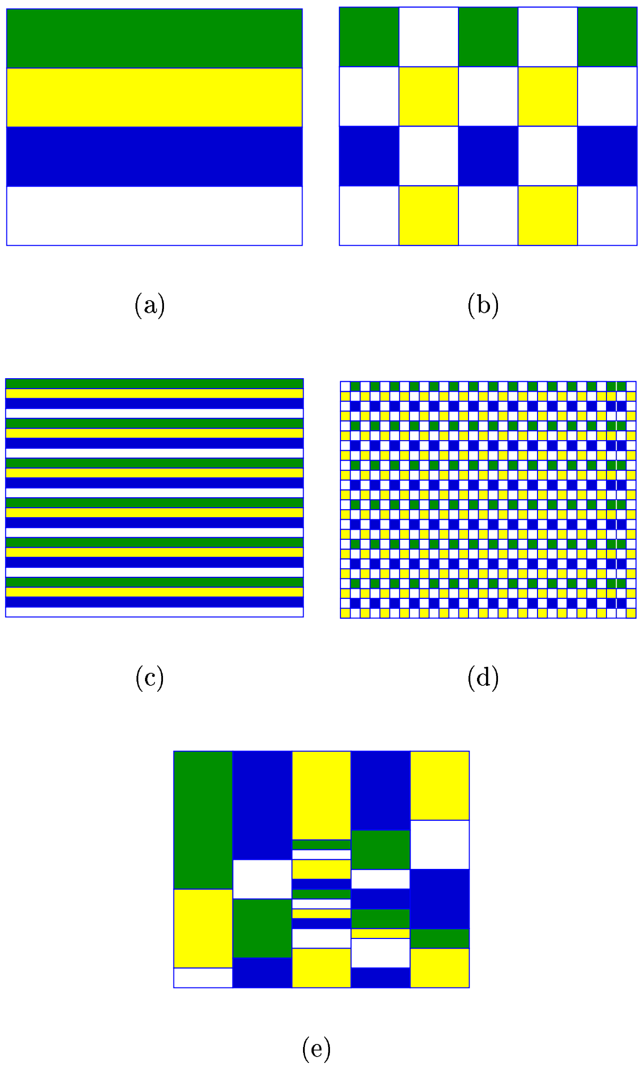


Figure 10: Example of five different data division schemes. (a) Blocks of lines of pixels. (b) Rectangular regions. (c) Interleaved lines of pixels. (d) Interleaved lines and columns of pixels. (e) Adaptive schemes. (Based on a picture from [72]).

Dynamic Data Distribution

Two strategies are typical in this type of data distribution – *demand-driven* and *adaptive*. In demand-driven distribution, the problem is decomposed into a large number of independent tasks. Each idle processor gets the next task and processes. This scheme continues until all of the tasks are completed. One way to avoid poor load balancing when using this strategy is to process more expensive tasks first. Another problem, though, is to estimate the cost of the tasks, which must be accomplished heuristically in a preprocessing step, introducing further computational overhead. Another way to minimize load imbalance is to use a large number of *fine-grained* tasks. But this can imply more overhead due to loss of coherence and task assignment. The *adaptive* strategy (Figure 10(e)) performs a first subdivision of the data and assignment to the processors. Then, whenever a processor becomes idle by finishing its initial task, it checks with the non-idle processors and requests tasks from them. A good example is the *stealing* scheme used by Whitman. The algorithm divides the data in image space in a relatively small number of coarse tasks and assigns them to processors using an on-demand scheme. When a processor becomes idle and no more tasks are available from the initial pool, it searches for the processor with the largest workload and "steals" part of its work. Its main overhead is due to having to access non-local information.

Dynamic schemes tend to result in better load balancing than static schemes, but they perform better on shared memory architecture machines, which present high band-width for remote memory access. They do not work well in message-passing architectures, which present high communication costs.

Granularity

Granularity refers to the sizes of the tasks to be performed by each processor. The term *fine-grained* means small tasks while *coarse-grained* indicates substantial average amount of work per processor. It should be clear that the finer the granularity, in general, the higher the overhead that is incurred for task scheduling and communication and the better the load balancing. One must carefully consider this issue, depending on the target architecture and the overhead of the algorithm to be parallelized.

Scalability

The *scalability* of a parallel system refers to its efficiency in handling bigger datasets (*data-scalability*), or its speedup (*performance-scalability*), when more processors are made available to it. Both the hardware architecture and the design of the software for rendering must be taken into consideration. For instance, adding processors to a shared-memory machine, does not increase the memory bandwidth; at some point, the memory bus becomes saturated and the performance stalls. This is why distributed memory architectures, which do not rely so much on memory bandwidth, tend to scale better to much larger (hundreds, even thousands) numbers of processors.

Coherence

In computer graphics, the term *coherence* describes the tendency for “nearby” pixels tend to have similar colors [64]. It is usual to consider three different types of coherence.

- *Frame coherence* (also known as *temporal coherence*): comparing two frames in a sequence, one notes that an object in one frame tends to have almost the same position in space and the same shape. Pixels showing this object will have its colors only slightly modified from frame to frame.

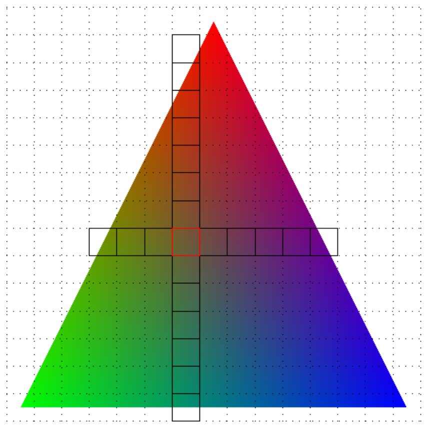
- *Span coherence*: Neighbor pixels in a line of the screen tend to have similar colors.

- *Scanline coherence*: Neighbor pixels in a column of the screen tend to have similar colors.

Both *span coherence* and *scanline coherence* are types of *spatial coherence*, which is used by rasterization algorithms to achieve high efficiency. *Frame coherence* is heavily used in programs that create animation sequences to improve performance and also by image compression algorithms (e.g., jpeg).

When adopting image partitioning parallelization, the coherence is lost at the boundaries of the partitions, resulting in computational overheads. The probability that a primitive will intersect a partition boundary depends on the size, shape and number of image partitions [51, 71]; these issues must be carefully considered in the implementation of a parallel rendering program [19].

In ray-casting algorithms the tendency of rays cast through neighboring pixels to intersect the same set of cells (in the same order) is called *ray coherence*. Ray coherence has been exploited in conjunction with data caching to reduce communication loads in parallel volume rendering and ray-tracing algorithms [46, 2].



(a)

Figure 11: *Spatial coherence* in image space. Pixels tend to have approximately the same value compared to immediate neighbors. Scan conversion can be highly optimized by exploiting this type of coherence.

Chapter 3

Time-Critical Rendering

[1] Many papers have presented rendering techniques and simplification ideas with the objective of speeding up image generation for irregular grid data sets. For large data sets, however, even the current fastest algorithms are known to require seconds to generate each image, making real-time analysis of such data sets very difficult, or even impossible, unless one has access to powerful and expensive computer hardware. In order to synthesize a system for handling very large data sets analysis, we have assembled algorithms for rendering, simplification and triangulation, and added to them some optimizations. We have made some improvements on one of the best current algorithms for rendering irregular grids, and added to it some approximation methods in both image and object space, resulting in a system that achieves high frame rates, even on slow computers without any specific graphic hardware. The algorithm

[1]This Chapter is based on work published: Time-Critical Rendering of Irregular Grids. R. Farias, J. Mitchell, C. Silva and B. Wylie, In Proceedings of the XIII SIBGRAPI – International Conference, Gramado - RS, Brazil. October 2000 IEEE.

adapts itself to the time budget it has available for each image generation, using hierarchical representations of the mesh for faster delivery of images when transformations are imposed to the data. When given additional time, the algorithm generates finer images, obtaining the precise final image if given sufficient time. We were able to obtain frame rates of the order of 5Hz for medium-sized data sets, which is about 20 times faster than previous rendering algorithms. With a trade-off between image accuracy and speed, similar frame rates can be achieved on different computers.

3.1 Introduction

Direct volume rendering methods are very useful tools in the visualization of scalar and vector fields. Techniques for volume rendering work primarily by modeling the volume as cloud-like cells composed of semi-transparent material that emits its own light, partially transmits light from other cells, and absorbs some incoming light [47]. In this thesis, we address the problem of rendering (non-curvilinear) *irregular grids* (or *unstructured meshes*), having no implicit connectivity. Such structures are effective at representing *disparate* field data. Irregular grid data comes in several different formats; see, e.g., [75]. The introduction of new methods for generating high-quality adaptive meshes has made the general unstructured irregular grids a most important data type to be visualized.

Several papers have presented efficient methods to render irregular grids, including re-sampling techniques, ray-casting techniques [23, 67], sweep-based algorithms [81, 61], and projective methods [76, 14]. This large body of work, mostly done in the past decade, has dramatically increased the efficiency with which we can render irregular grids. In studying the computational complexity of these techniques, one finds a wide range of tradeoffs. Consider an irregular grid composed of n cells (b of the cells being in the boundary), and a given screen (image) of size k-by-k pixels. Projective techniques work by projecting, in visibility order, the polyhedral cells that comprise the mesh onto the image plane, and incrementally compositing the cell's color and opacity into the final image. Regardless of the screen resolution, an image is only *complete* once each of its n cells have been correctly depth-sorted and projected onto the screen. This does not take into consideration the fact that some of the cells may be too small to make a significant contribution by themselves. In contrast, in ray casting techniques, for each pixel potentially only the cells that actually intersect a ray cast through that pixel need to be touched. This effectively is the case for the technique proposed in [6], which, if r is the average number of cells intersecting a given ray, takes time $O(b + rk^2)$.

Our focus in the work presented in this Chapter is on achieving real-time exploration, while possibly trading accuracy for speed. For real-time exploration, there are usually hard bounds on the overall rendering time T; e.g., for 30Hz, $T = \frac{1}{30}$ sec. Here, we explore tradeoffs necessary to design such *time-critical* irregular grid rendering systems. We utilize algorithms, based on extensions to existing techniques, which are scalable, allowing them to run on a wide range of machines. Our goal is to provide essentially the same level of interactivity, regardless of the machine speed, while trading accuracy for speed in a consistent way.

3.2 The Rendering Algorithm

At the core of our time-critical volume rendering system is a variation of the ray-casting algorithm proposed in [6]. The algorithm is outlined as follows:

(1) We transform each of the n cells into screen space.

(2) For each pixel, we compute a (sorted) list containing the *boundary* cells that intersect the ray through the pixel center.

(3) For each pixel, we perform ray casting incrementally by computing cell intersections, one cell at a time, in front-to-back order along the ray, using a traversal of the cell adjacency information (similar to [23]).

This algorithm is very simple to implement, and quite fast in practice. Step *(1)* takes $O(n)$ time. Step *(2)* takes $O(\sum_{i,j} b_{i,j} \log b_{i,j})$ time, where $b_{i,j}$ is the number of boundary faces that project onto pixel (i, j). Step *(3)* is an *output-sensitive* step, depending linearly on the total number of ray-cell intersections. We note that a straightforward method for obtaining a time-critical performance is simply to sample a regular subimage (performing l-by-l ray casts instead of k-by-k, for $l < k$), then rescale to the full-size image (which can be efficiently performed in hardware using OpenGL). Another feature of this ray-casting method is the fact that the computation is "embarrassingly parallel" ([67]) in the shared-memory model, allowing for a readily implemented parallel version. In order to understand better how the rendering algorithm works and why our optimizations were necessary, we now discuss in more details steps *(2)* and *(3)*.

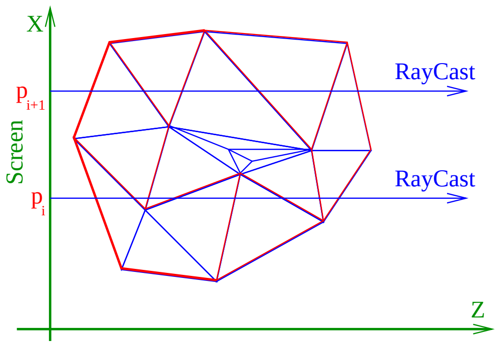


Figure 12: In this cross section of the volume to be visualized, the indices p_i and p_{i+1} represent two neighboring pixels through which two rays are cast. The visible boundary faces are highlighted, while the intersected faces are represented in red. Notice that in the middle of the mesh there is a set of cells, not stabbed by any of the rays, which do not need to be transformed.

Boundary Projection. The algorithm projects each "visible" boundary face onto the screen, creating for each pixel in the projection a list with the intersected "visible" faces. (*Visible* faces are the ones whose outward normal makes an angle greater than 90 deg with the viewing direction.) Assuming that the boundary is generally not highly erratic, these lists should be short; in practice, we expect that the maximum boundary-list complexity ($\max_{i,j} b_{i,j}$) to be constant (i.e., $O(1)$). Thus, while we could sort the lists (each in time $O(b_{i,j} \log b_{i,j})$) as we create them, we do not bother to do so in practice; instead, each time we need to know the next visible boundary face that occurs along a ray, we simply step through the short list to find the one remaining with lowest z-coordinate.

Ray Casting. The *current face* is initialized to be the first boundary face intersected along the ray through a given pixel. We then compute where the ray exits the cell (on the *next face*) it just entered, and we compute the scalar value at both the entry point and exit point of the cell (using bilinear interpolation). We then compute the contribution of the current cell to the pixel's color and opacity, adding this to the running sum that represents the integration. If the *next face* is a boundary face, the computation continues only if the remaining list of visible boundary faces is nonempty; the *current face* is then advanced to the next visible boundary face in the list, and we continue along the ray. If the *next face* is an interior face, we determine the neighbor cell on the other side of the *next face*, set the *current face* to the *next face*, and compute the new *next face* based on where the ray exits the neighbor cell. (Each face has pointers to its neighboring cells. A boundary face has only one neighboring cell.) This "walking" along rays is simple, in principle; however, we note that special care must be taken for the degenerate situations when the ray hits an edge or a vertex of the mesh.

Optimizations. In order to use this algorithm in a time-critical setting, we modify step *(1)* to have running time dependent on image-quality. Instead of transforming all the vertices and faces in the mesh, we transform only the ones on the boundary. Then we project them on the screen, and from then on, we *incrementally* transform the interior faces (and their defining vertices) that are intersected by the rays cast. Once transformed, we tag them as such to avoid duplicate transformations. Depending on the image resolution, which determines the number of rays to be cast, and on the viewing position, only a fraction of the data is actually touched. See Fig. 12. While there is some overhead in testing whether a primitive (vertex or cell) has already been transformed, we have found, in all data sets tested, that this optimization decreases the rendering time.

(Below, we discuss a parallelization of step *(2)*.)

3.3 Time-Critical Algorithm

Our goal is to achieve the highest frame rate possible by using a highly optimized rendering algorithm (discussed in the previous section), together with both image-space and object-space approximations. Beyond improvements in speed, these techniques will allow us to have greater control over the rendering procedure, giving us the flexibility to trade off between the image generation time and its accuracy. Such a trade-off will heavily depend on the machine's speed and the maximum acceptable simplification, to be controlled by the user.

In this section we discuss the image-space and object-space approximations we employ in our system. An alternative technique for the simplification of irregular grids is presented in [24].

Multi-Resolution Images

To generate multi-resolution images, we choose the simplest image-space simplification algorithm possible, to avoid spending time with both expensive computations and boundary constraints. We render the exact color for one pixel and duplicate it over a p-by-p pixels square, for a small value of p. (In our tests, we allow p to range from 1 to 9.) As will be shown later, the multi-resolution approximation has a narrow limit of its effectiveness for both speed-up and inversely for its error. For a 3-by-3 resolution, the gain in the rendering speed is high, while the visual impact is acceptable, still allowing the user to distinguish small details in the approximated image. Depending on the data set, larger values of p will only decrease the render time by a small amount, while resulting in very crude images.

Mesh Simplification Algorithm

To further improve the rendering time, we made use of object-space levels of detail, creating simplified meshes that are cheaper to render. We employ a method that is relatively simple, based on ignoring mesh connectivity, simplifying the point data (scalar values at mesh vertices), and then reconstructing an approximating mesh by re-triangulating the simplified point data. Special care is given to approximating the mesh boundary, while ensuring that the retriangulation of the interior point data does not induce artifacts from concavities in the boundary. We now elaborate on the steps of the algorithm: (1) interior mesh simplification; (2) boundary mesh simplification; (3) preserving concave boundary regions; (4) re-triangulating the simplified mesh; and elimination of transparent cells.

Interior Mesh Simplification. We propose a simple and very fast algorithm for volumetric data simplification based on the use of a *kd-tree.* The criteria for internally arranging the vertices inside the kd-tree is as follows. A vertex inserted into the kd-tree will lie inside an existing region if its distance to the *center* of a region is smaller than a given value, called the *radius* of the region. (The radius is obtained from the user-specified simplification rate; see below.) If the current vertex does not lie inside any existing region, it will define a new region and its coordinates will determine the center. After inserting all vertices into the kd-tree, the new (simplified) mesh will have one vertex corresponding to each region of the final kd-tree. These vertices will have coordinates and scalar values equal to the averaged coordinates and scalar values of all vertices inserted into the region. See Fig. 13.

One way around the problem of loss of boundary information if the kd-tree is used to simplify the entire mesh is to send only the interior points to the kd-tree and at the end, merge the simplified set of points with the boundary points. See Fig. 14.

The algorithm computes the radius for the regions in the following way. It computes the number of vertices equivalent to the percentage of simplification input by the user (say S). Now it remains to be found the radius for the regions that will result in a number of regions approximately equal to the number S. The algorithm starts by computing the diagonal (D) of the data set and then generates the *kd-tree* with half this value, or $R = D/2$. If the resulting number is greater than S the algorithm updates R as $R = R + R/2$; otherwise, if the number of regions is less than S, the algorithm updates R as $R = R - R/2$ and regenerates the *kd-tree* for the new region radius. This procedure is repeated until the desired number S of regions is obtained. Note that this search for the radius is a binary search, requiring at most $O(\log(n))$ steps. Once we find a mesh with the desired number of points, the algorithm proceeds to simplify the boundary.

This algorithm can be used to simplify the whole mesh, but it can cause undesired loss of boundary information. We choose alternatively to use a surface simplification algorithm to obtain the approximation of the boundary of the data set. This is discussed in the next section.

Boundary Mesh Simplification. In our first approach, we did not consider surface simplifications. However, we noticed that the surfaces of some data sets can contain a significant fraction of the total number of the data set vertices, so we devised a way to make it available as an option. This flexibility is necessary because some data sets in our tests presented problems with the surface simplification even if the rate of simplification was very small, of the order of 10%.

As we mentioned above, the simplification of the boundary requires a different approach because care must be taken to avoid destroying the form of the data set and still preserve its topology. Changes in the shape will be noticed more immediately then errors introduced to the color of the data set. In our current system we use the popular algorithm presented in [22], which allows not only surface simplification taking into account an expected error, but also allows the simplification to be performed over surface meshes whose vertices possess colors[2].

In the preprocessing phase, the algorithm tags all faces and vertices belonging to the boundary and make sure that each face has its vertices in counterclockwise order with respect to the exterior of the data set. (It is mandatory that the face normals are pointing outwards.) We then send these faces and

[2]Remember that each vertex has a scalar value associated with it, requiring the simplification algorithm to be able to handle colored surfaces.

vertices, with the desired rate of simplification, to *QSlim*[3].

All vertices of the new simplified mesh will be retrieved from the *kd-tree* and from *QSlim*. We expect that the vertices retrieved from the *kd-tree* will still be interior vertices. If any vertex retrieved from the *kd-tree* becomes exterior due to the boundary simplification, it will be eliminated in a future step; we note that this can cause *holes* in the simplified data set.

Preserving Concave Boundary Regions. After retrieving all vertices from the *kd-tree* and from *QSlim*, we have to rebuild the mesh connectivity, which will be explained in the next section. We use a Delaunay triangulation for rebuilding the mesh connectivity.

This scheme works fine for convex data sets; however, for data sets that possess concave regions, further precautions must be taken. When the vertices are sent to the *qhull*[4] code, to generate the Delaunay triangulation, all concavities in the boundary disappear, since we obtain a triangulation of the convex hull of the points. In order to avoid losing this important information, in the preprocessing phase we identify all vertices belonging to boundary concavities. This identification is done by (a) marking all points belonging to the boundary; (b) using qhull to create the convex hull of the boundary; and (c) tagging the points that belong to the boundary but are not on the convex hull.

Each concave point will have an associated *ghost* vertex, very close to it, but just outside the boundary (in the direction of the averaged outwards normal). These ghost vertices will have a special associated scalar value that

[3]QSlim code is available at Michael Garland's Home Page: graphics.cs.uiuc.edu/~garland/

[4]Qhull code is available at www.geom.umn.edu/software/download/qhull.html

Data Set	Vertices	Faces	Tetrahedra	Memory
SPX	2,896	27,252	12,936	11M
Blunt Fin	40,960	381,548	187,395	75M
Combustion	47,025	437,888	215,040	86M
Post	109,744	1,040,588	513,375	191M
Delta	211,680	2,032,084	1,005,675	370M

Table 1: Information about the data sets used in our experiments. The memory usage is reported to render an image at resolution of 128^2.

indicates to our ray casting function its transparency. See Fig. 15.

The distance between the *ghost* vertex and its related concave vertex is another parameter that can be controlled by the user (by default, we use 3% of the length of the data set's diagonal).

Re-triangulating the Simplified Mesh. After the simplification, the new set of points (which may contain up to $2b$ points in addition to the vertices in the simplified set) is sent to *qhull* [5], which returns a (Delaunay) tetrahedralization. The problem now is that any face that contains (at least) one ghost vertex must be considered to be transparent; there can be a significant number of such faces.

Eliminating Transparent Cells. At first, our code treated transparent faces individually, disregarding the contribution for any pair of faces in which at least one of them was transparent. This, however, was very inefficient and slowed the rendering function. Instead of thinking about transparent faces, one can think about transparent cells. Any tetrahedron which contains at least one transparent face, can be considered transparent and can be completely

Data Set	Resolution	Bunyk et al.	Optimized
	128^2	2s	1s
Blunt Fin	256^2	8s	4s
	512^2	27s	13s
	1024^2	104s	50s
	128^2	4s	2s
Combustion	256^2	10s	5s
	512^2	37s	14s
	1024^2	141s	52s
	128^2	5s	3s
Oxygen Post	256^2	19s	8s
	512^2	72s	27s
	1024^2	271s	100s
	128^2	4s	2s
Delta Wing	256^2	13s	6s
	512^2	43s	23s
	1024^2	157s	72s

Table 2: Optimization time results. Our optimized version is consistently faster than the previous implementation.

eliminated from the tetrahedra set. This criterion enormously reduces the number of tetrahedra in the resulting simplified mesh, restoring the shape of the original mesh very accurately, while simplifying our rendering code, since faces no longer require special treatment for been transparent.

3.4 Experimental Results

We report our results on an SGI machine (with a single 300MHZ MIPS R12000 processor and 512 Mbytes of memory). Table 1 lists the data sets we used in our experiments and measurements and all its relevant information, such as

Data Set	Resolution	Vertices Transformed	Cells Transformed
	128^2	25K	160K
Blunt Fin	256^2	30K	215K
	512^2	35K	270K
	1024^2	39K	319K
	128^2	47K	433K
Combustion	256^2	47K	437K
	512^2	47K	437K
	1024^2	47K	437K
	128^2	53K	315K
Oxygen Post	256^2	66K	431K
	512^2	83K	588K
	1024^2	96K	770K
	128^2	83K	450K
Delta Wing	256^2	114K	708K
	512^2	148K	1040K
	1024^2	169K	1398K

Table 3: Speed up from lazy transformations. Compare the number of vertices and cells transformed for each resolution; it becomes clear the source of the speed-up.

number of vertices, faces and cells. In Table 2 we compare the times obtained by the original algorithm [6] with the times we obtained with our optimized version. Our optimized version of Bunyk et al's algorithm, is, by itself, a factor of two improvement. Our changes to step *(1)*, utilizing a lazy transformation of the vertices, is shown to be very effective. See table 3.

The first approximation we use is to render the data set at a lower resolution and use *OpenGL* efficient interpolation to show the image in a larger resolution. In Fig. 16 we show the exact image of *Liquid Oxygen Post* at the resolution of 300^2 (Fig. 16(a)) and the image interpolated (Fig. 16(b)) from 128^2 to 300^2.

Pixel size	1^2	2^2	3^2	5^2	7^2	9^2
Blunt Fin	2.8s	1.6s	1.2s	0.7s	0.5s	0.4s
Combustion	5.1s	2.9s	2.0s	1.2s	0.9s	0.8s
Oxygen Post	11.9s	5.1s	4.1s	3.0s	1.9s	1.7s
Delta Wing	34s	13s	14s	14s	11s	4.4s

Table 4: Down sampling time speed up. The times are all in seconds. The top row has the effective pixel size used.

The error, measured as the mean difference for all three color components of the RGB equal $(3.53\%, 2.43\%, 23.38\%)$. Note that even for a difference of 23% on the blue component, the interpolated image looks just a little bit brighter than the exact one.

A two-time speed-up in running time is not enough, our goal is to achieve much faster frame-rates in a scalable framework. We basically explored two different ways to achieve this goal: multi-resolution image generation and hierarchical mesh simplification.

Multi-resolution Image Generation

Table 4 summarizes the rendering times to generate images for the four bigger data sets, for different image resolutions, obtained by running the code on a PC computer. We can see that the running time continues to drop as we effective increase the pixel size from 1-by-1 to 9-by-9. Unfortunately, the larger the pixel size, the smaller the speed up, and the improvement is negligible after 7-by-7. As the pixel size increases, the image quality decreases accordingly. In Fig. 17, we show a typical set of images computed under these different approximations.

Mesh Simplification

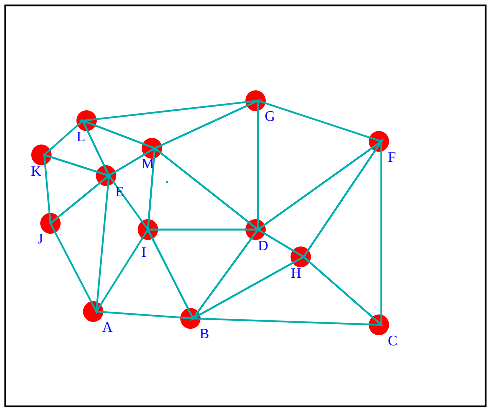
By introducing mesh simplification in our algorithm, we ended up with a large number of possible combinations between all these approximation algorithms; we include only a sample of the results here. Taking into account the optimization we made to the original Bunyk et al's algorithm and by combining all approximations we introduced in this work, we were able to raise the frame rate from 0.35 Hz (original data set at full resolution) to 3.5 Hz (with 90% of mesh simplification at 9^2 pixel resolution. A speed up factor of about 20 (remember that our optimized version of the render algorithm is twice as fast as Bunyk's original algorithm). To conclude this section we include in Fig. 18 some pictures of the *Oxygen Post* data set in full resolution and mesh simplification of 0%, 25%, 50%, 75% and 90%. Each row of image shows the image generated using each of these simplification. On the right column the image was generated on full pixel resolution, while on the right column we show the image for 9×9 pixels approximation. The error noticed on the edges the image is due to losses of detail of the boundary of the original data set. This can be avoided if one trades the simplification of the boundary that is performed separately from the interior simplification. For instance, see Fig. 19. The loss of information on the edges is due to the fact that for larger simplification rates the surface simplification algorithm generates meshes that contain some faces whose normals point inwards. Thus, depending on the data set characteristics and the desired rate of compression, one must trade between between opting or not for boundary simplification. Leaving the boundary unsimplified, even for 75% simplification of the interior points, the image looks almost the same. The errors measured between the image of *Oxygen Post* with simplification in (Fig. 18(a)) and its image for 75% of simplification only for the interior points in (Fig. 19(b)) led to an error, for *(r,g,b)* colors, respectively equal to (1.86%,1.18%,9.06%). Note that the error for the *blue* color is the only one to present a considerable value. But as the predominant color in the image depends on *red* and *green*, the error introduced by the simplification, led just to a slightly brighter image.

3.5 Conclusion

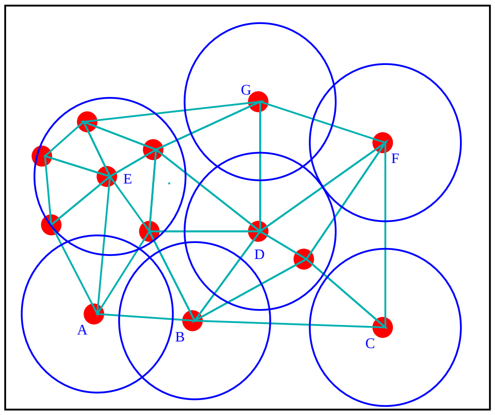
In this Chapter, we started exploring time-critical techniques for rendering irregular grids. We are primarily interested in developing techniques that are scalable, in the sense of being able to trade accuracy for rendering time, while achieving acceptable image quality in most reasonable cases. We have proposed a variation of the algorithm of [6]. Our technique differs from his in that it performs lazy transformations, it is able to generate images at multiple resolutions, and it works in parallel, using both an image-space and object-space technique. Our results are preliminary and expected to continue to improve. We have developed a simple GUI for our rendering code which allows the user to rotate moderately large data sets, including the Blunt Fin and Combustion Chamber, and even the Delta Wing. See Fig. 20.

Exploiting pixel coherence, we were able to obtain a frame rate of 3.5 Hz; this is to be compared with 3.5 seconds per frame to generate the exact image. Even though this result is far from the desirable 10-30 frames per second, it already allows us to rotate the data sets while keeping a decent amount of detail.

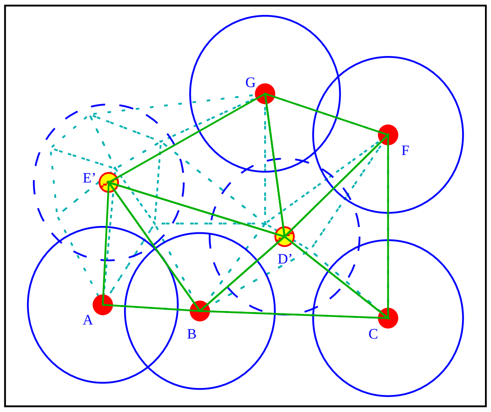
Much work remains, particularly on the parallelization. We are working to apply coherence to accelerate the ray casting step, noting that neighboring pixels are likely to be similar. Hence, if there is little time to compute a ray (say, during a fast rotation), it is reasonable to assume a filtered down-sampled version of the image might be a visually accurate representation. We are currently exploiting extensions of these ideas further, in particular as it relates to time coherence. For instance, in an environment where the image is being computed at a lowered resolution, it might not be necessary to perform step 2 (boundary face projection) all the time. The set of visible boundary cells computed in the frame i may still be visible in frame $i + 1$, albeit they might not intersect the ray emanating from the middle of the pixel from which it was originally visible. But, if the resulting image will be filtered anyway, simply a reprojection of those faces, and the ray integration from them might give a reasonably accurate picture. In fact, it might even be possible to reuse the intersection calculations.



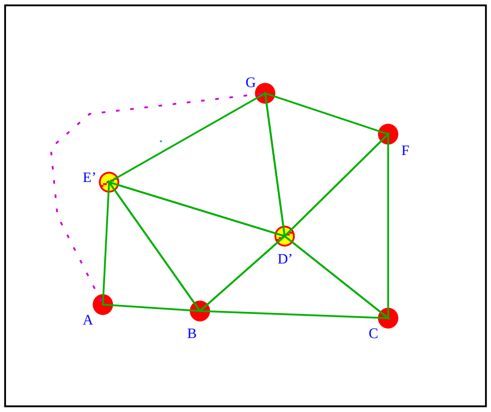
(a) **Original Mesh**



(b) **KD-Tree**

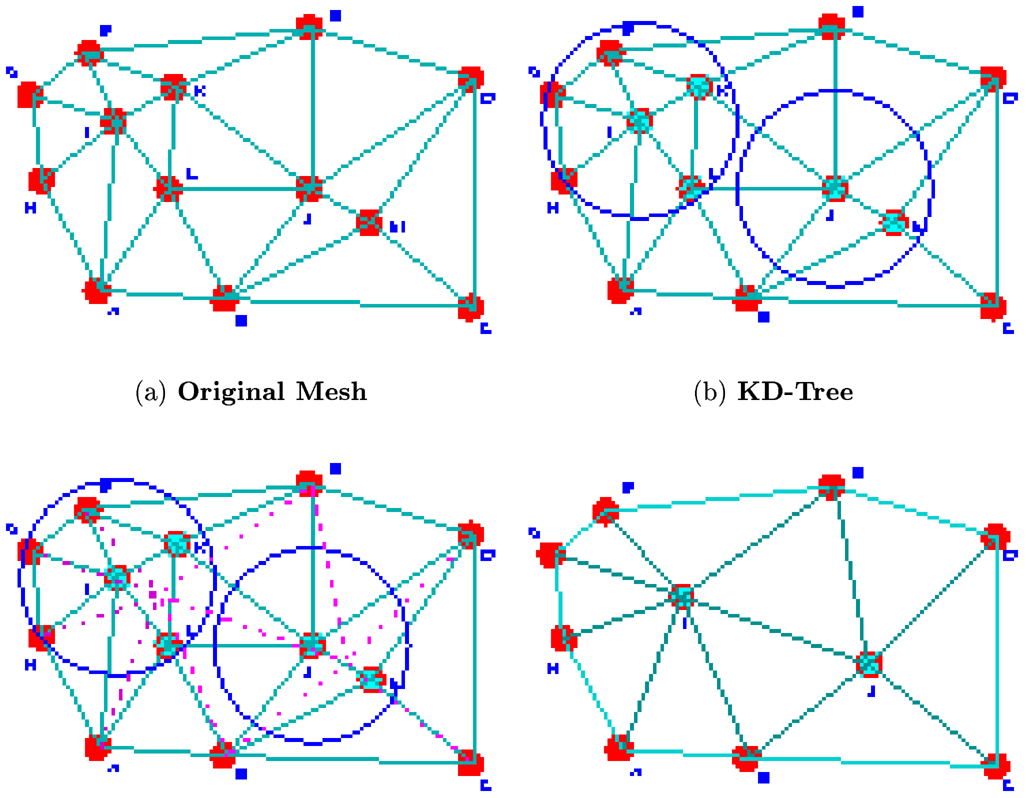


(c) **Remaining Vertices**



(d) **Bad Resulting Boundary**

Figure 13: Loss of boundary information. It happens if the kd-tree is used to simplify the entire mesh. (a) Original mesh. (b) All points are sent to the kd-tree. (c) The averaged center for each region, or the result points for each region. (d) The final simplified mesh, where the dotted line represents the original contour detail that was lost.



(a) **Original Mesh** (b) **KD-Tree**

(c) **Remaining Vertices** (d) **Correctly Resulting Boundary**

Figure 14: Interior points simplification scheme. (a) The original mesh. (b) Only interior points are inserted into the kd-tree. Note that the sequence in which the points are sent to the kd-tree is the same regardless of whether or not we choose to preserve the boundary. That is true once the code scans the points and just skips the points labeled as boundary. (c) The averaged center for each region, or the result points for each region. (d) The final simplified mesh.

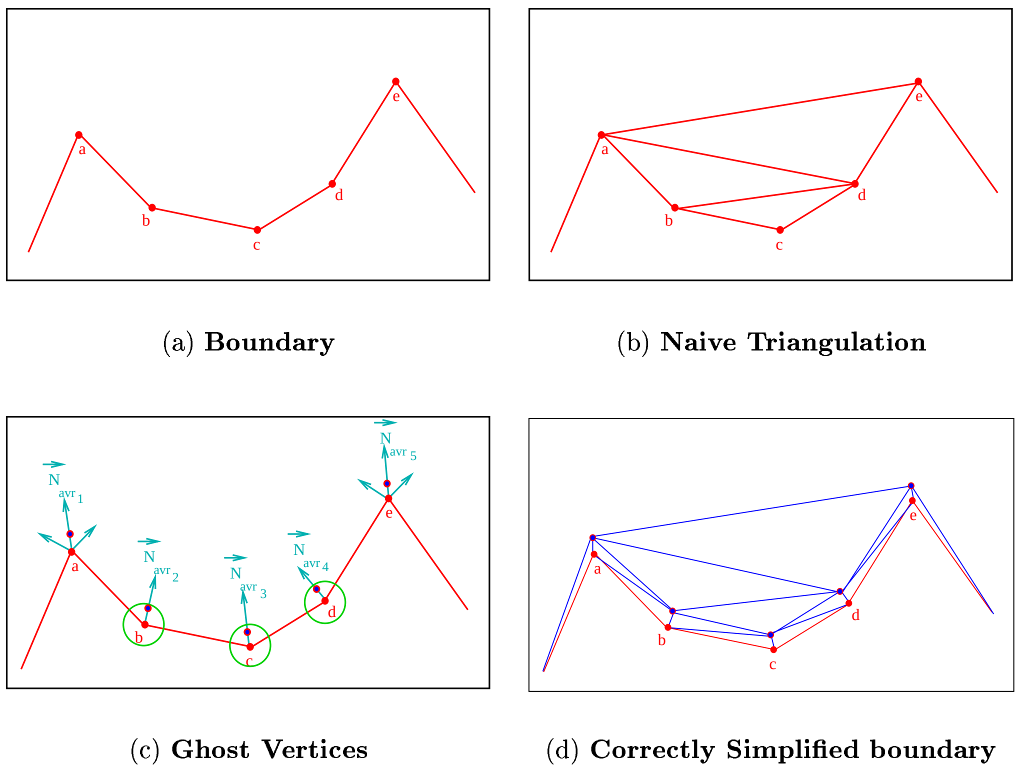
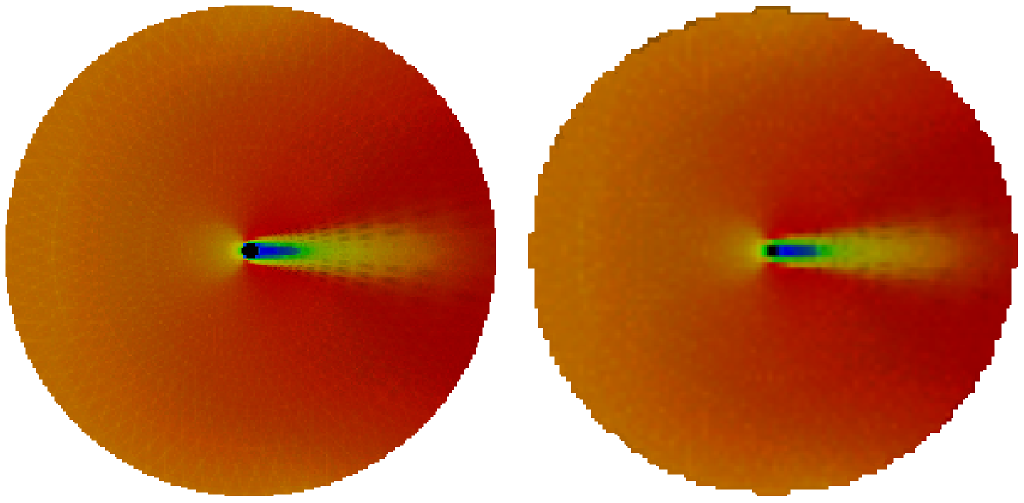


Figure 15: Using ghost points to preserve boundary information. (a) The vertices from a to e are tagged as belonging to a concave region of the boundary. (b) This is the final triangulation if we naively triangulate the data set without taking precautions to preserve concave regions. (c) Ghost vertices are inserted in the direction of the averaged normal for each concave vertex. (d) After the triangulation we can retrieve the concavity of such regions by eliminating any face that contains at least one ghost vertex.



(a) **Exact Image** (b) **Interpolated Image**

Figure 16: Speeding up by scaling the resulting images. (a) Exact image of Liquid Oxygen Post rendered at 300^2 pixel. (b) Image rendered at 128^2 and interpolated to 300^2 pixels.

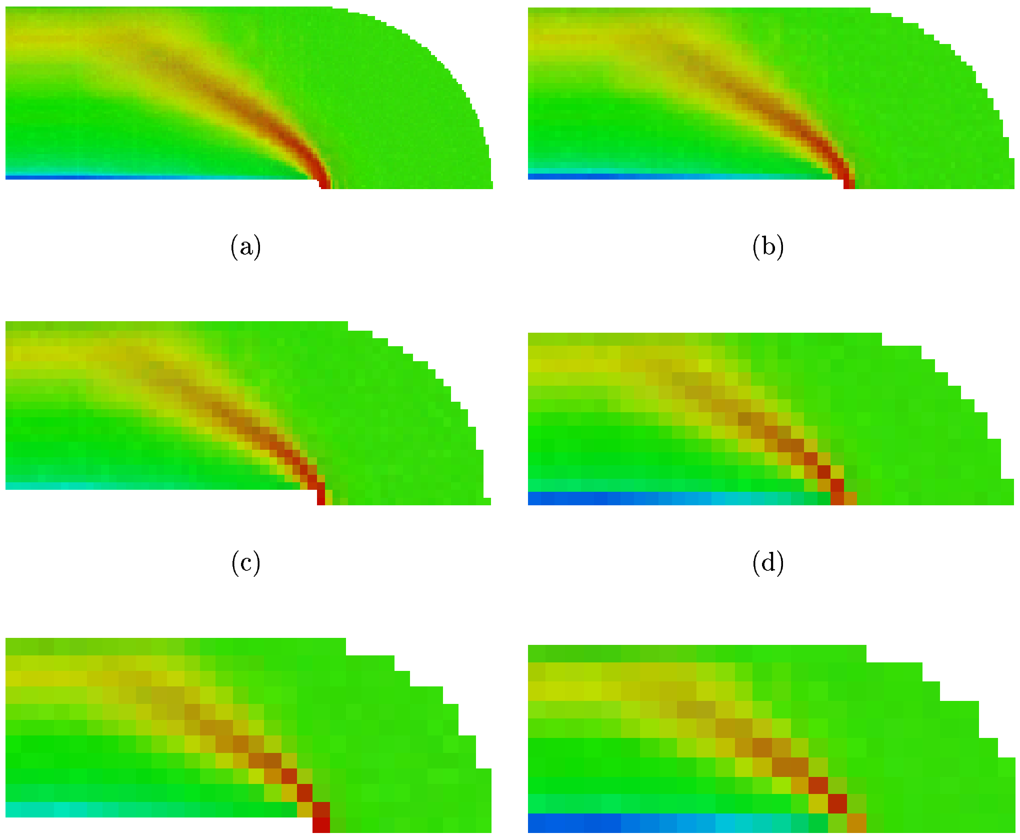
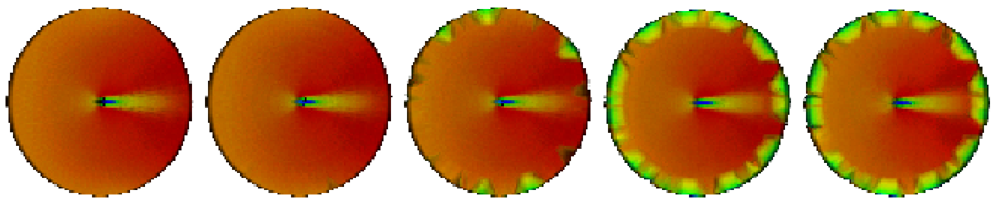
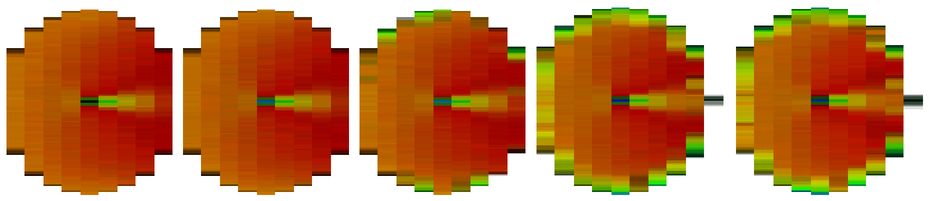


Figure 17: Image down-sampling on the Blunt Fin. (a) exact image (1-by-1 pixel size); (b) 2-by-2; (c) 3-by-3; (d) 5-by-5; (e) 7-by-7; (f) 9-by-9.



(a)



(b)

Figure 18: Applying all simplification schemes. (a) One pixel (full resolution) images for mesh simplification of 0%, 25%, 50%, 75% and 90%. (b) Same mesh simplification for 9×9 pixels image resolution.

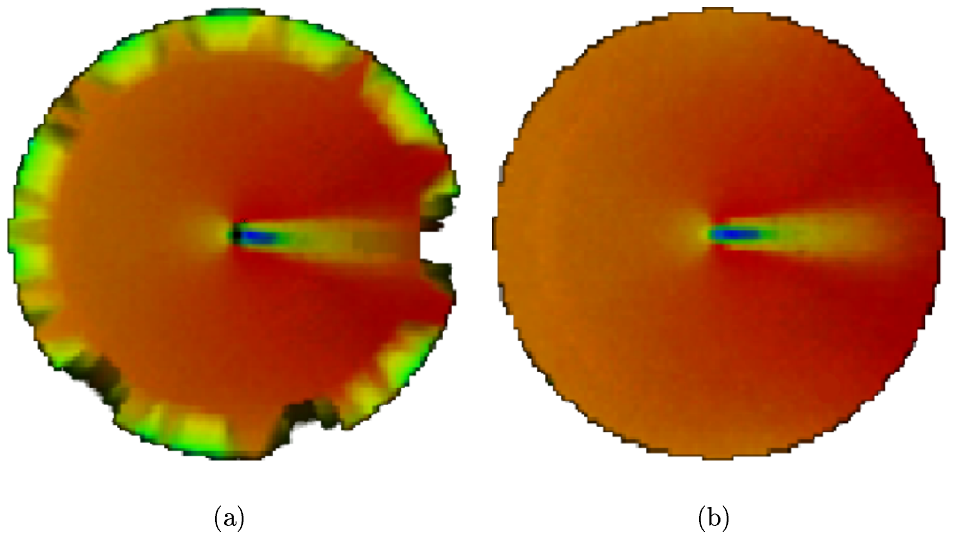
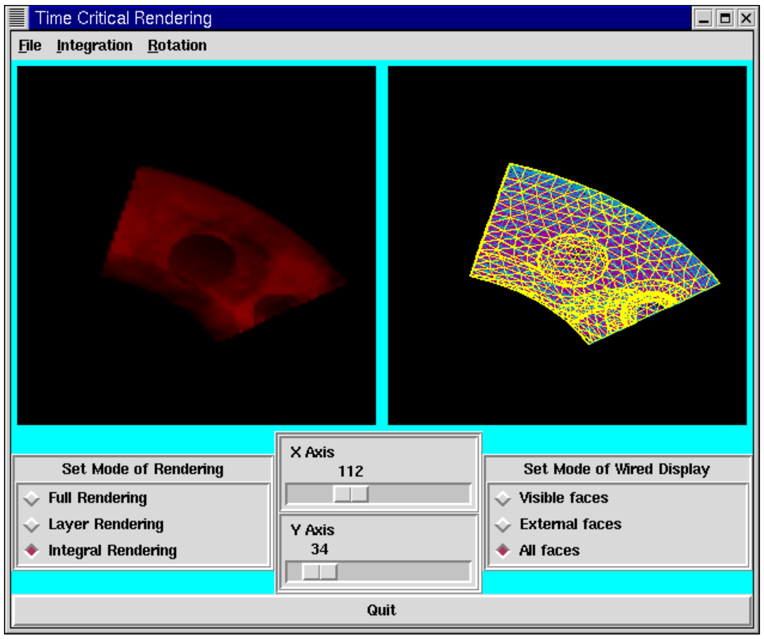


Figure 19: *Liquid Oxygen Post* simplification. (a) Both interior and boundary points were simplified resulting in 26K points and 328K faces. (b) Only interior points were simplified, resulting in 37K points and 494K faces.



Set Mode of Rendering
Full Rendering
Layer Rendering
[✓ Integral Rendering]

X Axis 112

Y Axis 34

Set Mode of Wired Display
Visible faces
External faces
[✓ All faces]

Figure 20: A screen shot of the time-critical system GUI. On the left window is shown the rendered image and on the right window is shown the grid of the data set. The user can rotate up, down, left and right, and zoom in and out.

Chapter 4

ZSweep Algorithm

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

[1]This Chapter is based on work published: ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering. R. Farias, J. Mitchell, and C. Silva. In *2000 Volume Visualization Symposium*, pages 91–99. October 2000.

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

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

4.1 Introduction

We study the problem of rendering unstructured grid volumetric data. In this Chapter, our focus is on direct volume rendering, a term used to define a particular set of rendering techniques which avoids generating intermediary (surface) representations of the volume data. Instead, the scalar field is generally modeled as a cloud-like material, and rendered by computing a set of lighting equations. In general, while evaluating the volume rendering equations [47], it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells intersected by the ray, so that the overall integral in the rendering equation can be evaluated.

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

Related Work

Early work in adapting ray tracing techniques for rendering unstructured grids is described in Garrity [23] and Uselton [67]. These techniques are "exact", in the sense that in principle (i.e., without accounting for degeneracies), for each pixel, a correct cell stabbing order is computed. Unfortunately, these techniques tend to be relatively slow, despite the optimizations proposed in the respective articles.

Shirley and Tuchman [59] show how to exploit polygon-based graphics hardware in the volume rendering equations for one tetrahedron. By using the MPVO technique of Williams [76] to visibility sort the cells in back-to-front order, they propose a "projective" method for rendering unstructured grids. This particular projective technique had several limitations, including the fact that the MPVO technique of Williams is only able to generate "correct" visibility order for certain types of datasets, and the actual approximation proposed in [59] generates visual artifacts. Improving on the Shirley and Tuchman technique, Stein et al. [63] propose techniques to explore texture mapping to improve the visual quality, and an $O(n^2)$ sorting algorithm which is able to compute correct visibility order for general acyclic unstructured grids.

Their work is further improved by Williams et al [78], Silva et al [62], and Comba et al [14], leading to almost linear-time (in practice) “exact” visibility sorting techniques. Max et al[48] proposed a different sorting technique based on “power” sorting. This technique is more restricted than the MPVO-sorted grids, in fact, it is only guaranteed to produce correct sorting results for acyclic complexes (triangulations). Despite its shortcomings, this technique is quite useful, and has been used extensively in practice, leading to excellent rendering times (see Cignoni et al [13, 11], and Wittenbrink [80]). Recently, Cignoni and De Floriani [12] have proposed a more general extension of power sorting, but provide little experimental results.

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

The plane sweep paradigm, which is based on processing geometric entities in an order determined by passing a line or a plane over the data, has been used widely in computational geometry for the design of efficient algorithms; see [56]. It has also been used in devising efficient volume rendering algorithms.

Giersten [25] pioneered the use of sweep algorithms in volume rendering.

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

Following the same basic idea of sweeping the data, Yagel presented a different approach to rendering unstructured grid data, which also allowed some further speed-up using hardware support, as he shows in [81]. His sweep algorithm is based (as is ours) on a sweep with a plane parallel to the viewing plane. Just like Giertsen's algorithm, Yagel's does not need to compute and keep explicit cell adjacency information, allowing it to be memory-efficient in its basic data structures. Graphics hardware can be used to accumulate the contributions of each slice to the final image. The main drawback of this algorithm is its memory consumption, which can be substantial, since it must store the polygons resulting from each slice. Also, its accelerated version requires graphics hardware support.

The Lazy Sweep algorithm [61] is the most recent algorithm based on the sweeping paradigm. It was shown to be faster and more memory efficient than its predecessors. Besides the array of vertices and the array of cells, the only other adjacency information used is a list (the "vertex use set"), for each vertex, that keeps the indices of all cells that are incident on the vertex.

Two other papers by Westermann and Ertl [68, 69], describe techniques to exploit the graphics hardware to achieve fast rendering, also based on the sweep paradigm.

We also briefly discuss the ray-casting algorithm of Bunyk et al. [6], which we use in our experimental comparisons. This is a fast algorithm, but it requires a lot memory. It's basic idea is as follows:

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

determine the entry and exit points of the ray as it passes into and out of the volume.

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

Recently, Hong and Kaufman [29, 28] proposed a very fast ray casting technique for curvilinear grids. Their work is similar in some ways to [6], but optimized for curvilinear grids, which makes it faster and use far less memory than [6]. It's restrictions is that it is applicable only for structured datasets.

Finally, we mention that the new algorithm presented in this paper, is also related in some ways with the *A-buffer* [7] approach, optimized with the use of the order of the vertices to assure a *quasi-order* projection of the faces. The work the *A-buffer* has to perform to order the intersections between the ray cast and the faces projected is very small.

4.2 The ZSweep Algorithm

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

The algorithm is a simple sweep with a plane, Π, parallel to the viewing plane, in order of increasing z-coordinate. (This is the only similarity with

Yagel et al's algorithm.) *Events* occur when Π encounters a vertex v, at which point we project the faces of cells that are incident to v and lie beyond v (in z-coordinate).

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

- The *vertex use set*, $U(v)$, associated with vertex v is a list of all cells that are incident on v.
- We say that vertex v is a *swept vertex* if it has already been swept over by Π (its z-coordinate, v_z, is less that the current sweep value, z.
- A face f is a *swept face* if at least one vertex of f is a swept vertex.
- A cell c is a *swept cell* if all of its faces are swept. Since our algorithm maintains the swept status explicitly only for vertices, we use the following observations to determine the swept status of a cell: A tetrahedral cell is swept if and only if (at least) two of its vertices have been swept; a hexahedral cell is swept if and only if (at least) five of its vertices have been swept.

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

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

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

The main loop of the algorithm is the sweep in the z-direction, which is performed simply by stepping through the event list. When the ith vertex, v_i, of the event list is encountered, we *project* [2] each face f that is incident on v_i for which v_i is the vertex having minimal z-coordinate. (Necessarily, such faces f have not been swept.) The faces to project are readily determined by examining the use set of v_i. Refer to Figure 21 for an illustration in two dimensions.

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

To guarantee accuracy in the rendering algorithm, it is important to make certain that the projection of the faces is done in a correct order for each pixel. The order in which faces are projected in our ZSweep is according to

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

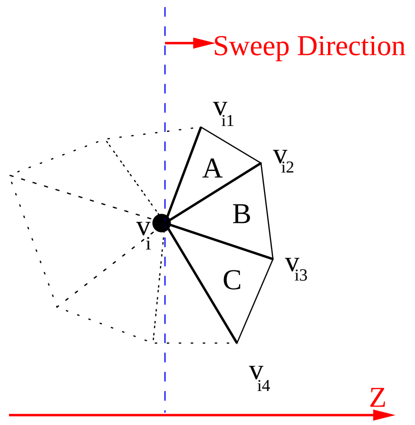


Figure 21: When the sweep plane Π encounters vertex v_i, the cells A, B, and C are first encountered, so the (highlighted) faces (v_i, v_{i_1}), (v_i, v_{i_2}), (v_i, v_{i_3}), and (v_i, v_{i_4}) are projected.

the z-coordinate of the first vertex encountered of the face. This order is *not*, however, sufficient to guarantee that faces are projected automatically in correct depth order for every pixel. For example, in Figure 21, faces (v_i, v_{i_1}) and (v_i, v_{i_2}) are each projected when we encounter v_i. While a local analysis of the faces at v_i would permit us to project (v_i, v_{i_1}) *before* (v_i, v_{i_2}), we would have to project also face (v_{i_1}, v_{i_2}) before (v_i, v_{i_2}) in order to have those pixels in the projection of (v_{i_1}, v_{i_2}) have the correct ordering of projected faces. We do not, however, project face (v_{i_1}, v_{i_2}) until Π reaches the vertex v_{i_1}. While in two dimensions it is possible to project faces (edges of triangles) in an order that is consistent in z, in three dimensions it is well known that the precedence relation induced by depth ordering can have cycles. (Even three triangles in space can form a cycle.)

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

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

Initially, we define the z-target to be the maximum z-coordinate among the vertices adjacent to the first vertex, v_0, encountered by Π. When the sweep reaches the z-target (say, corresponding to vertex v), we compose, in order, the entries of the pixel lists into the accumulated value being kept for each pixel, starting from the last z-coordinate where composition left off for that pixel, and ending when we reach the depth of the z-target. (Thus, we may not compose *all* entries of the pixel list; those corresponding to z-coordinates beyond the z-target are not composed yet, as there is a chance that there are faces not yet projected whose z-coordinates will precede them). This incremental composition is done for each pixel whose pixel list has more than one entry. We remove from the pixel lists all of the entries that we compose, except for the last one (since it will be needed in order to continue the composition later). After composing the values at all of the relevant pixels, we reset the z-target to be the maximum z-coordinate of the vertices adjacent to v, and continue the sweep. In the example of Figure 21, if v_i was the previous z-target, then, when it is encountered, the new z-target is set to the z-coordinate of v_{i_3}.

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

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

4.3 Implementation Details

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

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

4.3.1 Preprocessing and Basic Structures

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

4.3.2 Sweep

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

Now we consider how the sweep function identifies which face must be sent to the projection function. If vertex v_i is the vertex with smallest z-coordinate (there can be many of them in degenerate cases), we know that this vertex does not have any cell in its **use set** that has already been projected. But this case is a special case of the general case, so we will only discuss the general case, as depicted in Figure 21. Suppose vertex v_i has just been obtained from the heap. We scan v_i's use set to visit all of its touching cells. Notice that all cells represented by dashed lines in the figure are considered *swept* cells (by our definition). Suppose that we find the cell B; we can project both faces, between cells $A - B$ and between cells $B - C$. Then suppose we now find cell A. To avoid projecting twice all the interior faces of the dataset, we make use of a very small hash table. (Below, we discuss a further optimization ("sparsification") that we perform in order to minimize, but not eliminate, such occurrences.) In practice, we have observed that the hash table holds at most 15 faces per vertex, for data sets up to half a million cells.

If the current cell is hexagonal the only extra care that must be taken is to create two triangular faces and send them to the *hash table.* (The projection function will not even know if the face came from a tetrahedral or hexahedral cell.) We have used some connection information to avoid generating intersecting triangular faces; see Figure 22. One problem that will happen if we have intersecting faces is that once the hash key is built based on the indices for the vertices, 4 different faces will be included into the hash table. This will not cause the code to fail, but it can cause undesirable artifacts in the final image if the four vertices are not coplanar, which is true in general.

To address this issue, we use the connection between the global index and relative indices for the points. The point p_i in the global array of points can appear as any of the L vertices of a given L-vertex cell. Given a point we can easily find out the faces of the cell that use it. Each internal face will be found

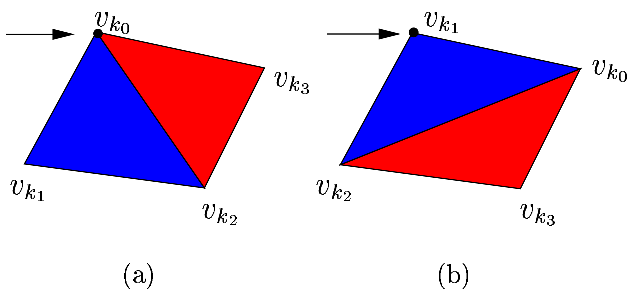


Figure 22: (a) The two triangular faces created when the current vertex v_{k_1} is the local vertex 0 for this face. (b) A case in which the current vertex is the second vertex for this same face, when found for the other cell that shares this face. This will happen if one always creates the faces starting from the local vertex 0 for all faces. Remember also that the hash key is generated based on the indices values, and that in this case four different faces will be included into the hash table.

twice by the algorithm. We use the unique global index as the identification for the vertices. Consider the face shown in Figure 22. Assume that the cells C_k and C_p are the ones that share this face. Suppose that for cell C_k, the current vertex appears as its first local vertex (v_0). This will lead us to find the two triangular faces shown in Figure 22(a). When we find the same face on another cell, suppose that the current vertex appears as its second local vertex. To find the same combination for the three vertices independent of their relative local position for both neighbor cells, we used a “wrapping” computation to find the global index of each vertex, starting with the current vertex. So if the current vertex is the first vertex in the hexahedral face of the cell C_k, the indices for the two triangular faces are given by:

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

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

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

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

4.3.3 Projection

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

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

4.3.4 Delayed Compositing

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

4.3.5 Optimizations

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

Sparse Data Representation

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

In particular, we do the following. We keep all cells that have faces on the boundary, since they are essential for those boundary faces. We then

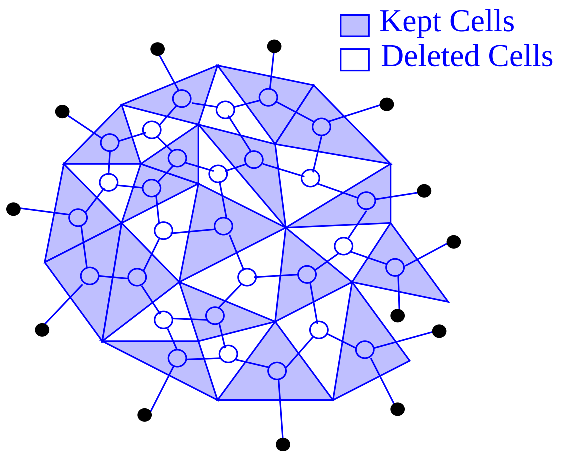


Figure 23: Over the mesh, it is shown the equivalent graph with a edge covering. In the graph the nodes represent the cells and the edges represent the face between the cells. A ghost node must be included for each boundary face, to make it possible their representation in the graph.

iteratively mark cells for deletion. Each time we delete a cell, we mark its neighboring cells as "essential" (they are not permitted to be deleted. We continue until all cells are either deleted or marked "essential."

Our "sparsification" technique is related to the "chess-boarding" technique of Cignoni et al [11]. In [11], they save memory by avoiding the duplication of the "edges" of a regular grid dataset during isosurfacing.

Use of Previous Heap Result

This optimization is only important if one wants to use the code to generate a sequence of images. Once the first image has been rendered, the heap class is able to keep a integral copy of itself. So supposing that the dataset is rotated by a small angle, it is usually true that the vertices are likely to have almost

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

Table 5: This table shows the number of cells (tetrahedra/hexahedra), the total number of vertices and faces, as well as the number of boundary vertices and faces for all datasets. The rightmost column shows the number of cells for each dataset after sparsification, which always results in at most 56% of the cells being kept.

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

Table 6: The first three columns show the total memory required by ZSweep to render each dataset in different resolutions. The fourth and fifth columns show the preprocessing times, on the K7-PC, for the original data sets and its sparse representation. The last column shows the preprocessing time on the SGI platform measured for the original data sets.

the same order than in the previous order. If instead of sending the vertices every time from the original global array, they are inserted into the heap in the order they had in the previous sweep, then the next ordering will be performed in linear time (in practice), since the vertices will be almost in order already. (A similar optimization is used in [80].)

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

4.4 Experimental Results

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

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

We ran our experiments on several popular datasets available from NASA, including *Blunt Fin*, *Combustion Chamber*, *Liquid Oxygen Post*, and *Delta*

Wing. We use the tetrahedralized versions of these datasets, since our algorithm is intended to visualize unstructured grids. (For structured datasets one should opt for algorithms designed specifically to exploit the implicit representation of the grid, which allows for fast and highly memory-efficient algorithms; e.g., see [28].) We also perform our experiments on two other datasets, selected in order to verify the functionality of our implementation in the case of holes and hexahedral cells: *SPX*, which is a small tetrahedral dataset having *holes*, and *Hexahedral*, which is a small hexahedral dataset.

Table 5 gives basic information about all six datasets used in our experiments. In the rightmost column is shown the size of the data after sparsification, which eliminates, on average, about 53% of the cells. This savings allows the algorithm not only to reduce its total memory consumption, but also to reduce considerably the reading and preprocessing time.

4.4.1 ZSweep Performance

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

Table 6 shows *ZSweep*'s preprocessing times, which include reading and generating the *use set* for all vertices and memory usage required to create different image sizes. When larger images are required to be generated, more memory is necessary, since for each pixel the algorithm has to keep an ordered list (the pixel list) of intersected faces. The required memory grows sublinearly, however, since for an image 16 times larger, the memory goes up by less than

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

Table 7: Render time (in seconds) and the number of pixels processed for each dataset and each image size.

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

Table 8: Render time (in seconds) and the number of pixels processed for each dataset and each image size.

a factor of 2.

Table 7 shows *ZSweep* rendering times on the SGI platform. The resolutions were chosen to allow us to compare our results with previous works. The compilation was performed in 32 bits with highest possible optimization ("-O3"). Table 7 shows the equivalent tests performed on the K7-PC platform.

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

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

Table 9: While ZSweep uses about twice the memory that *Lazy Sweep* requires, it is 4.4 times faster. As the image size grows, *Bunyk et al.* becomes slightly faster than ZSweep, but at the cost of much higher memory consumption.

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

Table 10: In this case ZSweep is about 4.75 times faster than *Lazy Sweep*, while again using about twice the memory. For this dataset, *ZSweep* was faster than the *Bunyk et al.* method for all image sizes considered, while using substantially less memory.

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

Table 11: To create an image of size 300^2 *ZSweep* requires 60% more memory than *Lazy Sweep*. But, while *Lazy Sweep* maintains its memory requirements essentially the same even for larger images, ZSweep needs to allocate more and more memory, because of the pixel lists. Again ZSweep is comparable to *Bunyk et al.* in speed, but much more memory efficient.

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

Table 12: Delta Wing is a medium-size dataset with over a million tetrahedra. For this dataset it becomes clear that the pixel lists are slowing down the algorithm. But it is still 2.4 times faster than *Lazy Sweep*. And even though *ZSweep* became slower than *Bunyk et al.* algorithm, the memory this last one needs is still a big problem nowadays.

datasets.

4.4.2 Comparison with Other Methods

We compare our results with two fastest and most recent algorithms available for unstructured grids, Lazy Sweep [61] and the ray-casting algorithm of [6]. (We do not compare here with hardware-accelerated algorithms, as we are studying the performance here of pure software implementations.) We compare the render costs, both in rendering time and total memory consumed for each of the three methods, for all four NASA datasets (those on which the other two methods apply); see Tables 9–12.

One last note we make is that *ZSweep*, just like *Bunyk et al.*, was implemented using a lighting model that, although simple, is computationally more expensive than the model used on the lazy sweep work. So even in the cases where *Lazy Sweep* compares in speed with *ZSweep*, keep in mind that the final image generated by *ZSweep* will be more accurate in terms of the lighting. [3]

[3]Our lighting model is the same as that used in *Bunyk et al.*, based on integration of

4.5 Conclusion

The unstructured grid volume rendering algorithm (ZSweep) we presented in this Chapter has proven to be a very competitive option for both general and

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

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

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

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

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

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

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

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

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

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

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

As a comparison, the lighting model used in Lazy Sweep amounts to a table lookup for each color channel, and multiplication by the transparency. It is possible to make the lighting model considerably more accurate and complex. A good example is the one used in the HIAC system, described in Williams et al [78]. Max [47] gives a good survey of optical models for volume rendering.

specific applications due to its relatively low memory requirements, high speed, accuracy and simplicity. It is considerably simpler and faster than the previous sweep-based rendering algorithms (without hardware assistance). As with the previous algorithms of [61] and [6], the accuracy of the final image does not depend on the characteristics of the dataset grid.

Also, as with the Lazy Sweep method of [61], ZSweep is memory efficient, even though a highly anomalous dataset could cause the pixel lists, maintained for each pixel of the screen, to become lengthy, making it necessary for further precautions (e.g., partitioning of the viewing plane into subimages) to be taken to avoid having these lists consume too much memory. We note, however, that for all tests on all datasets mentioned on Table 5, we did not notice an unexpected increase of memory usage. We did expect the memory allocation to increase for larger images, since as the image size increases, each face projected will insert intersection units into more and more pixel lists. While ZSweep is more than twice as fast as [61], it uses from 20% to 60% more memory, which is not enough even to slow down the reading/preprocessing step compared to the Lazy Sweep method. We have methods of reducing the memory requirements that we are exploring in our continuing investigations.

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

On the other hand, even though ZSweep is slower than [6] in some cases, it uses considerably less memory and ZSweep does not have the difficulty that arises from having ray casts that hit degenerate points at vertices or edges of the grid.

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

Besides parallelization, we are also exploring other improvements on ZSweep, including (1) further reducing the memory requirements by partitioning the image space and running the algorithm separately on subimages; (2) adapting ZSweep for walkthrough applications; currently, we assume that views are from outside the datasets, and clipping the outside can be performed efficiently within our framework; (3) exploring the development of a hardware-assisted version of the ZSweep; (4) add other cell formats (including nonconvex cells).

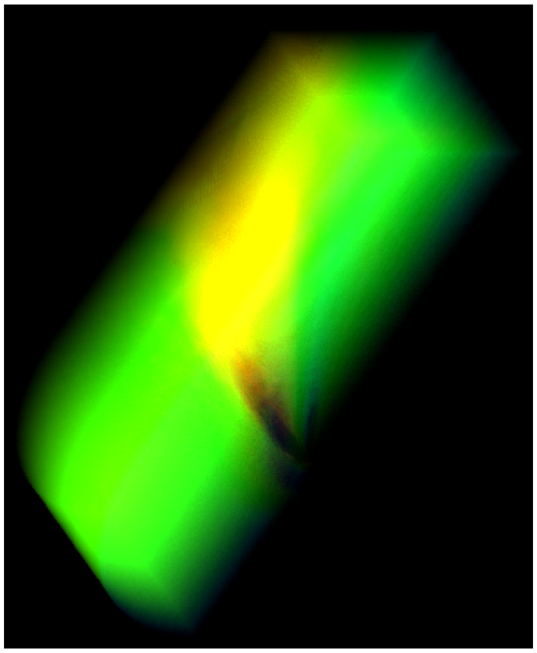


Figure 24: Image of Blunt Fin created in $512x512$.

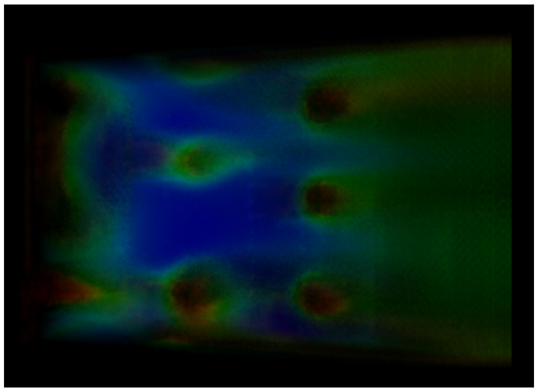


Figure 25: Image of Combustion Chamber created in $512x512$.

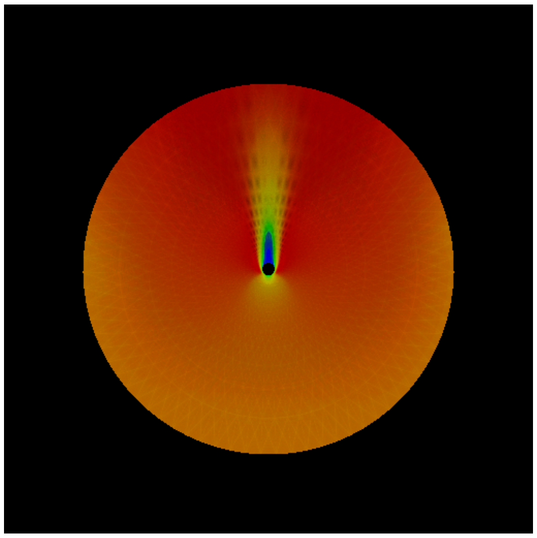


Figure 26: Image of Liquid Oxygen Post created in 512x512.

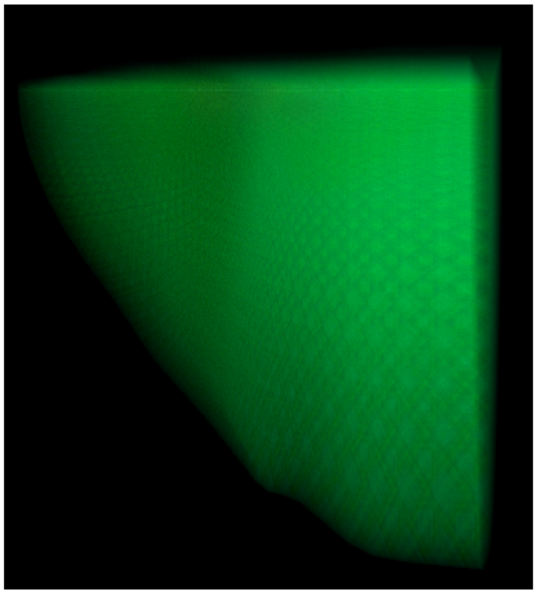


Figure 27: Image of Delta Wing created in 512x512.

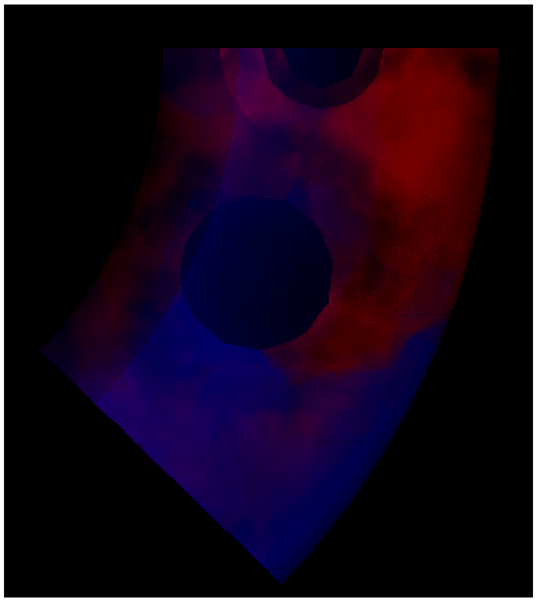


Figure 28: Not as famous, this is spx.off. It contains holes what makes it more challenging to visualize. Image created in $512x512$.

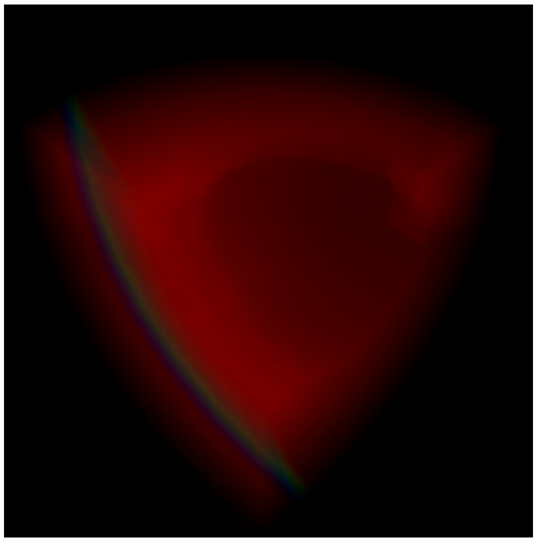


Figure 29: This file is a very small hexahedral grid data set that we used to test the part of our implementation that handles hexahedral cells data sets. Image created in $512x512$.

Chapter 5

Parallelizing ZSweep

[1] In this Chapter we describe a simple parallelization of the ZSWEEP algorithm for rendering unstructured volumetric grids on distributed-shared memory machines, and study its performance on three generations of SGI multiprocessors, including the new Origin 3000 series.

The main idea of the ZSWEEP algorithm is very simple; it is based on sweeping the data with a plane parallel to the viewing plane, in order of increasing z, projecting the faces of cells that are incident to vertices as they are encountered by the sweep plane. Our parallel extension of the basic algorithm makes use of an image-based task partitioning scheme. Essentially, the screen is divided in more tiles than the number of processors, then each processor performs the sweep independently on the next available tile, until no more tiles are available to render.

[1]This Chapter is based on work to appear: Parallelizing the ZSWEEP algorithm for distributed-shared memory architectures. R. Farias and C. Silva. To appear in the International Workshop on Volume Graphics, VG01, June 2001.

Here, we detail the modifications necessary to efficiently extend the sequential algorithm to work on shared-memory machines. We report on the performance of our implementation, and show that the *tile-based* ZSWEEP is naturally cache friendly, achieves fast rendering times, and substantial speedups on all the machines we used for testing. On one processor of our Origin 3000, we measure the L2 data cache hit rate of the *tile-based* ZSWEEP to be over 99%; a parallel efficiency of 83% on 16 processors; and rendering rates of about 300 thousand tetrahedra per second for a 1024×1024 image.

5.1 Introduction

In this Chapter, we describe a parallel extension of our ZSWEEP [20] algorithm for rendering unstructured grids on distributed shared-memory machines. Despite the substantial progress on the state-of-the-art in rendering of irregular grids, high-quality renderings of very large grids still take a substantial amount of time. Our goal is to explore the availability of small and mid-size parallel machines for rendering (and also to provide a path for exploring much larger machines). We focus on distributed-shared memory hardware, since these capabilities are quite common in servers sold by major vendors, including SGI, SUN, and IBM.

Although the programming model for shared-memory parallelization is quite trivial, achieving good performance on actual machines is usually hard. Even *embarrassingly* parallel algorithms, such as ray casting irregular grids [31] usually do not scale well beyond a few processors. Several issues such as proper load balancing need to be taken into account for good performance.

Quite possibly, the hardest issue to deal with in distributed-shared memory machines is memory coherence and related issues. The problem comes from the fact that access to memory is non-uniform, since often the data one processor needs actually resides in physical memory that belongs to another processor. Hardware designers have developed intricate techniques for optimizing memory access (such as the deployment of large caches and aggressive memory prefetching strategies) but still software has to be carefully developed to collaborate with the hardware, and avoid performance killers such as unnecessary sharing of data. In general, one needs algorithms with a high degree of cache coherence to perform well on distributed shared-memory machines.

Direct volume rendering is a term used to define a particular set of rendering techniques which avoids generating intermediary (surface) representations of the volume data. Instead, the scalar field is generally modeled as a cloud-like material, and rendered by computing a set of lighting equations. In general, while evaluating the volume rendering equations [47], it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells intersected by the ray, so that the overall integral in the rendering equation can be evaluated.

ZSWEEP [20] is an algorithm for the computation of the sorted order of the cells intersected by all the rays in a given image. The main idea of the ZSWEEP algorithm is very simple; it is based on sweeping the data with a plane parallel to the viewing plane (shown in blue on Fig. 30), in order of increasing z, *projecting* the faces of cells that are incident to vertices as they are encountered by the sweep plane. ZSWEEP's face projection is different from the ones used in projective methods, e.g. [59]. During face projection,

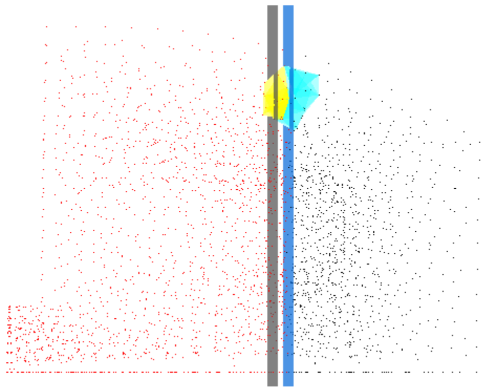


Figure 30: The plane sweep is shown in blue while the plane determined by the *target Z* is shown in light-gray. The sweeping direction is from the right to the left and the swept vertices are shown in black while the still untouched vertices are shown in red. Faces in the *use set* of the current vertex are identified and shown as previously projected faces (light-blue) and faces to be projected (yellow), the ones that lie ahead of the plane sweep.

we simply compute the intersection of the ray emanating from each pixel, and store their z-value, and other auxiliary information, in *sorted* order in a list of intersections for the given pixel. The actual lighting calculations [47] are deferred to a later phase (b). Compositing is performed as the "target Z" plane (shown in gray on Fig. 30) is reached. The efficiency arises from: (1) the fact that the algorithm exploits the implicit (approximate) global ordering that the z-ordering of the vertices induces on the cells that are incident on them, thus leading to only a very small number of ray intersection are done out of order; (2) the use of early compositing which makes the memory footprint of the algorithm quite small. The key properties for the efficiency of ZSWEEP is the fact that given a mesh with v vertices and c cells, the amount of sorting ZSWEEP does is $O(v \log v)$ (in practice), i.e., depending on the number of ray intersections, this is substantially lower than the amount of sorting necessary to sort all the intersections for each pixel.

Contributions:

- We propose a simple parallel extension of the basic algorithm using an image-based (i.e, tiling) task partitioning scheme. Following Nieh and Levoy [54], our algorithm is based on an adaptive image-based task scheduling scheme. Basically, we divide the screen into tiles, which are dynamically assigned to the processors.

- We describe the changes that need to be performed to the original algorithm to efficiently implement a *tile-based* ZSWEEP.

- We perform a detailed analysis of the memory characteristics of the *tile-based* ZSWEEP. In particular, we show that the image tiling strategy improves the memory coherency of ZSWEEP, and can lead to the whole set of ray intersections fitting in the secondary level (L2) cache. On the Origin 3000 this leads to better than 99% hit rate and greatly improved rendering rates.

  Even on single-processor machines the *tile-based* ZSWEEP is considerably more efficient than the original algorithm.

- Finally, we study load balancing and efficiency of the parallel ZSWEEP on three generations of SGI multiprocessors, including the new Origin 3000 series.

The Chapter is organized as follows. In Sec. 5.2, we briefly describe related work. In Sec. 5.3, we present the parallel algorithm. Then in Sec. 5.4, we present our experimental results on three different kinds of SGI multiprocessors. Sec. 5.5 ends the Chapter with final remarks, and future work.

5.2 Related Work

We keep our related work section short and focus on parallel rendering algorithms for irregular grids and other work directly relevant to our work. The Chapter on ZSWEEP 4[20] contains references to previous work in volume rendering of irregular grids. For a discussion of computational complexity issues in rendering of irregular grids, we point the reader to [61].

As we said before, for evaluating the volume rendering equations, it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells intersected by the ray, so that the overall integral in the rendering equation can be evaluated.

One solution to this problem is to compute the intersections of rays with each cell in the mesh independently, then sort each list of intersections before compositing is performed. This is essentially the approach proposed by Ma and Crockett [44]. In more detail, their technique distributes the cells among processors in a round-robin fashion. For each viewpoint, each processor independently computes the ray intersections, which are later composited in a second phase of the algorithm. One of the potential shortcomings of this technique is that it requires the storage of a very large number of ray intersections. Ma and Crockett cleverly avoid this potentially crippling shortcoming by scheduling the computation using a k-d tree. As shown on [44, 45], their algorithm has been shown to be very scalable on message-passing machines, including the IBM SP-2 and the Cray T3D. Recently, Hofsetz and Ma [30] have developed an efficient shared-memory version of this algorithm, which they demonstrate on a 16-processor SGI Origin 2000. They showed that a naive port of the original algorithm lead to poor performance, but with substantial changes to the original implementation, very good performance was achieved.

One of the advantages of the Ma and Crockett technique is that no mesh connectivity is necessary. At the same time, by completely ignoring connectivity, this algorithm does not exploit a lot of the coherence intrinsic in the mesh, which both raises its memory requirements, and forces it into having to sort potentially very large lists. Most other algorithms for rendering irregular grids actually attempt to use mesh coherence (in the form of connectivity among cells), and try to get the sorting cost as close to linear as possible.

Hong and Kaufman [31] proposes a very efficient ray-casting based rendering algorithm for curvilinear grids. Their work is similar in some ways to [6], but optimized for curvilinear grids, which makes it faster and use far less memory than [6]. Our interest in their work is the fact that they parallelized their fast ray caster on a 16-processor SGI machine using an image-based task scheduling scheme similar to the one we use in this Chapter. The speedups achieved were on the order of 11.88 on 16 processors, or 74% efficiency. The parallelization of a ray casting technique has also been studied by Uselton [67]

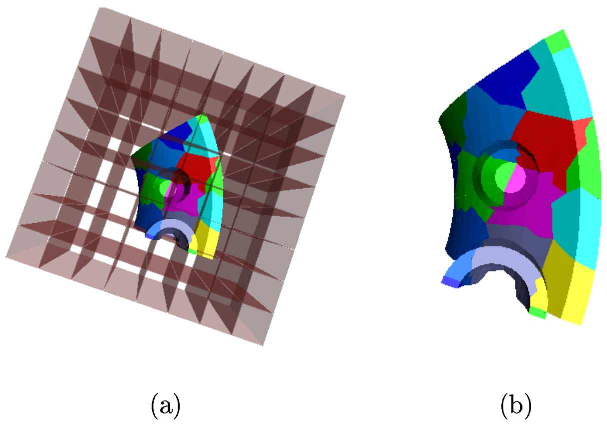


Figure 31: In (a) we show the shafts generated by each tiling region. In (b), we give a close up of the decomposition on the dataset. Each region was computed by intersecting the octree with the shafts shown in (a).

with very good results.

Challinger [8] and Wilhelms et al [74] propose similar scanline rendering algorithms (similar in several respects to [61]). Both papers report on parallelizations, which is the main focus of [8]. Challinger also uses an image tiling scheme for parallelization with very good results, which are reported separate for different phases of the algorithm, and when taken all into account, amount to impressive speedups of a little over 70 on 100 processors of a BBN TC2000.

Still on shared-memory machines, Williams [77] reports parallelizing his rendering algorithm for an 8-processor SGI 4D/VGX. Other notable papers (which focus on rendering regular grids) include Nieh and Levoy [54] and Lacroute [37, 38]. We would like to note that irregular grid rendering algorithms tend to be hard to parallelize with screen-space parallelism because of their object-space disparate resolution. That is, possibly, a large number of cells project into a small area of the screen.

5.3 The Parallel ZSWEEP Algorithm

In this section we describe our parallelization of the ZSWEEP algorithm. The sequential algorithm is highly efficient, and uses little extra memory on top of the original dataset. It is based on computing ray intersections with the faces of the cells, which are "roughly" pre-sorted in depth by using a sort of the vertices of the cells. Each time a vertex is found during a z-sweep, the faces incident on it are marked, and the ray intersections for the pixels that overlap with the faces are computed, and inserted on intersection lists. In order to avoid having the lists get arbitrarily large, ZSWEEP employs a scheme for early compositing. See [20] for full details.

Following previous works, including Nieh and Levoy [54] and Hong and Kaufman [31], our parallelization is based on breaking the screen into tiles. Then placing the tiles into a work queue which processors compete for work. Each processor continously fetches a tile from the work queue, and computes the subimage corresponding to that tile until all the tiles have been rendered. In order for a given processor to compute the image for a tile using ZSWEEP, we must determine all the vertices from any face that intersects the "shaft" emanating from that tile. This is similar to the parallel view sort of Challinger [8], and is primarily the main difference between the sequential and parallel ZSWEEP, since in the sequential algorithm the vertices are known apriori (that is, all of them are sorted in depth).

For efficiency purposes, we made a small data structure change. While in the sequential ZSWEEP implementation the **use set** of a vertex is the list of cells incident on it, in the parallel version we decided to break the cells into its faces and keep them in the use set of the vertices. The reason for the modification comes from the fact that the projection of a face requires a somewhat expensive setup (see [6] for details). Since faces might intersect multiple tiles of the screen, the setup time would be replicated multiple times. By actually having a list of the faces, we are able to parallelize these computations as a first phase in the parallel render.

We separate the computation of the vertices that belong to an image tile into a view independent phase which is performed only once when the data is first loaded, and a view dependent phase which is performed by each processor when rendering a given tile. The view independent phase consists of (1) constructing an octree of the vertices of the mesh; (2) computing for each octree "leaf" the faces which intersect that leaf (in the implementation we use the bounding box of the faces, which is a conservative estimate); (3) record for each leaf the list of faces which have non void intersection. From such list we are able to determine the vertices that must be considered in the sweeping phase, for each leaf. The view dependent phase uses the octree to find which leaves intersect the shaft corresponding to the tile, then uses the union of all the vertices assigned to those leaves as the input for the rendering routine. (See Fig. 31.)

The actual rendering algorithm is fairly simple. Given p processors and f faces, each processor transforms $\frac{f}{p}$ faces. The image is divided into tiles, and

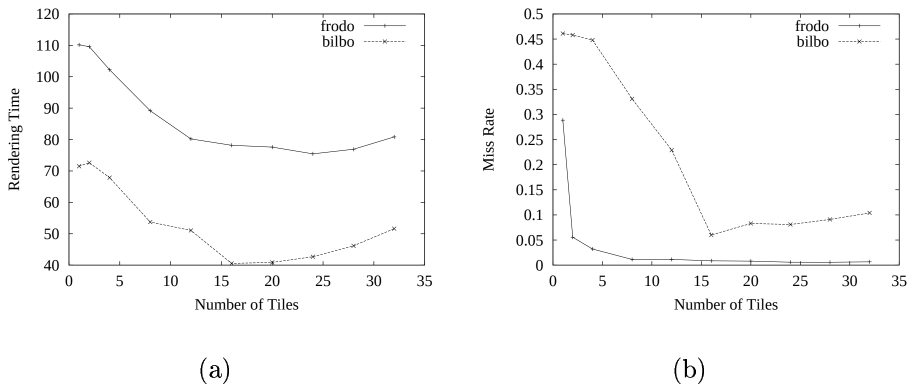


Figure 32: Rendering the small SPX dataset at 1024×1024 resolution under different tiling. In (a), we see the rendering times. In (b), the L2 data cache miss rate. By using tiling, the miss rate drops considerably to under 1% on **frodo**.

each processor will incrementally grab a tile, and render the subimage corresponding to that tile. Rendering a tile is performed by (a) finding the leaves of the octree which project inside the particular tile, (b) computing vertices of all the faces which intersect any of the leaves found, and (c) projecting the faces in order (that is, the last phase is simply the sequential ZSWEEP applied to the subset of the vertices which have faces projecting inside the tile). In our implementation, we are careful to clip the projection of the faces to within the tile being computed.

As shown in Sec. 5.4, we achieve very good load balancing with this simple scheme. The cost of rendering a tile is dependent both on its area, and the number of points which project into it. Experimentally, we have found that the area cost is considerably larger than the cost associated with the number of points. The number of points can vary as much as by a factor of five, and have little impact on the running time of the region.

5.4 Experimental Results

In this section we summarize our findings about the performance of our algorithm. We ran our experiments on three difference machines, all manufactured by SGI:

- **bilbo**: 12-processor SGI Onyx. The processors are 194Mhz MIPS R10000. Eight of them have 1 MB of secondary cache, and the other four have 2 MB of secondary cache. Bilbo has 2 GB of memory. This machine is a snoop-based multiprocessor [17, Chapter 6], a design which is popular in small parallel machines, but does not scale well.

- **smaug**: 24-processor SGI Origin 2000. It is equipped with sixteen 250 Mhz MIPS R10000 and eight 300 Mhz MIPS R12000. Each R10000 has a 4MB secondary level cache, while each R12000 has a 8 MB secondary level cache. Smaug has 14 GB of memory. This machine is based on a scalable shared-memory system, and it uses directory-based cache coherence [17, Chapter 8].

- **frodo**: 16-processor SGI Origin 3000. It is equipped with sixteen 400 Mhz MIPS R12000 and it has 12 GB of memory. This machine is a faster and more scalable directory-based distributed shared-memory system. In particular, each parallel "node" has four processors (compared to two for the O2K), and higher memory bandwidth, and much lower latency.

5.4.1 Sequential Tile-Based ZSWEEP

An interesting fact is that the tile-based ZSWEEP is faster by almost 50% than the original. This is somewhat counter intuitive, since it actually does more work: it needs to sort vertices multiple times (the actual number depends on tiling and resolution of octree), and it definitely touches faces multiple times, although the actual pixel calculations are quite similar. A potential advantage of the tile-based approach is that the "target Z" used for early compositing is likely to be more accurate. But when we first noticed this speedup from tiling, we suspected that these performance gains actually arise from better memory coherency.

We used `perfex`, an SGI IRIX tool which is able to configure and retrieve the MIPS R10K hardware counters, to validate our hypothesis. In Fig. 32, we show some of our findings. In particular, we can see that the L2 data cache hit rate [2] is greatly improved with caching, and there is a corresponding improvement in rendering times. On **frodo**, we get better than 99% hit rates, and on **bilbo** they were improved from just a bit over 50% to over 90%. Another interesting statistics is the number of TLB misses which changes by a factor of 300 on some of the runs, thus indicating the considerable better data locality of the tile-based approach.

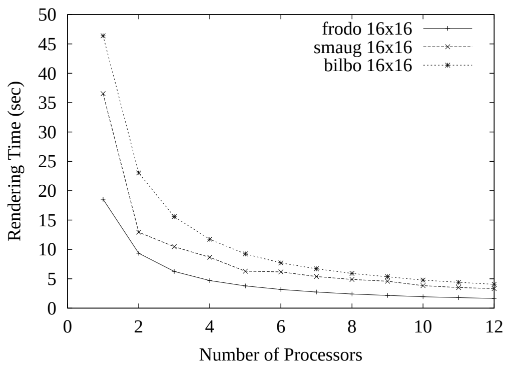


Figure 33: Running times on up to 12 processors for the Post dataset. Images of size 512×512 with 16-by-16 tiling.

5.4.2 Load Balancing

We ran a battery of tests for studying the scalability of our algorithm on all these machines. We tried to use the machines when they were free, although this was virtually impossible for smaug which is used for heavy batch processing of data. We ran jobs on smaug at times of lightest load. Unfortunately, given the heterogenous nature of the CPUs, it is really not possible to make very accurate measurements on that machine. The other two machines were used at idle times. We generated 512×512 images under different conditions, and changing the tiling granularity. We use the term X-by-Y tiling decomposition to mean that the image was subdivided into X times Y regions. That is, an 8-by-8 tiling decomposition means that the image was divided into 64 tiles.

[2] L2 (secondary) data cache hit rate is the fraction of data accesses that are satisfied from a cache line already resident in the secondary data cache. It is calculated as 1.0 - (secondary data cache misses divided by primary data cache misses). This is the exact definition from the `perfex` man page.

Fig. 33 shows the running times for the Post dataset on the different machines on up to 12 processors. As can be seen from the picture, the rendering times are quite fast, and improve as the number of processors increase. As expected, frodo is considerably faster than the other two machines, and the parallel efficiency is about 93% with 12 processors (11.2 speedup). It is interesting to note that even on the **bilbo**, which has considerably inferior memory system, our parallel algorithm is able to scale quite nicely. Part of the credit might go to the fact that ZWEEP tends to minimize data movement.

The tiling granurality has an impact on the performance. We use Ma's load imbalance metric to study the impact on load balancing of different tiling sizes. Given a set of processors where the average rendering time is t_{avg} and the maximum rendering time is t_{max}, Ma [43] defines the imbalance to be:

$$1 - \frac{t_{avg}}{t_{max}};$$

basically, his metric measures the spread of the running times among the different processors around the mean. In Fig. 34, we plot the imbalance. As can be seen in the picture, **bilbo** and **frodo** behave almost exactly the same, while smaug, due to its different speed processors, exhibits more load imbalance. The worst load imbalance happens for 8-by-8 tiling decomposition, and can be as high as 30%. Part of the problem is that because the dataset is not uniform, some parts of the screen might have a very large number of faces, that need to be rendered. With a 16-by-16 tiling decomposition, things get substantially better, and the load imbalance is lower than 5%.

On **frodo**, for the Post, using 16-by-16 tiling decomposition, the speedups

Rendering Times				
Image Dimension 512 × 512				
	SPX	SPX1	SPX2	SPX3
1	4.51	9.95	38.10	186.05
2	2.32	5.09	19.65	97.96
4	1.18	2.62	10.37	50.55
8	0.63	1.39	5.60	26.63
16	1.38	1.93	10.30	27.56
Image Dimension 1024 × 1024				
	SPX	SPX1	SPX2	SPX3
1	15.39	27.86	73.40	267.46
2	7.78	13.97	36.85	135.13
4	3.99	7.08	18.47	68.12
8	2.11	3.71	9.71	36.27
16	1.28	2.17	5.61	21.72
Image Dimension 2048 × 2048				
	SPX	SPX1	SPX2	SPX3
1	82.89	145.74	298.78	731.99
2	41.85	73.08	150.23	375.3
4	21.17	36.71	75.65	188.3
8	11.04	19.07	39.20	98.09
16	6.50	10.96	22.07	56.27

Table 13: On **f**rodo for **S**PX and its subdivisions, for increasing image size. Each subdivison contains 8 times more tetrahedra than its previous representation.

Datasets Information		
Dataset	# of vertices	# of cells
Oxygen Post	109K	513K
SPX	2.9K	13K
SPX1	20K	103K
SPX2	150K	830K
SPX3	1150K	6620K

Table 14: The first four are tetrahedralized versions of the well-known NASA datasets. SPX is an unstructured grid composed of tetrahedra. We have subdivided each tetrahedron into 8, for each version of the last three, that is, SPX3 is 512 times larger than SPX. The number of vertices and tetrahedra are listed in thousands.

are 12.3 for a 512×512 image, and 13.5 for a 1024×1024 image, or approximately 84% efficiency. The best rendering times for the Post are 1.5 seconds for a 512×512 image, and 4.44 seconds for a 1024×1024 image. In general, it is possible to improve the load balancing by simply incresing the tiling resolution. In fact, we were able to get efficiencies of almost 90% by tweaking the parameters. A better solution would be to have an adaptive technique which automatically fine tunes the load balancing. We have actually implemented such a scheme, but were not able to make it work consistently yet.

5.4.3 Data and Image Scalability

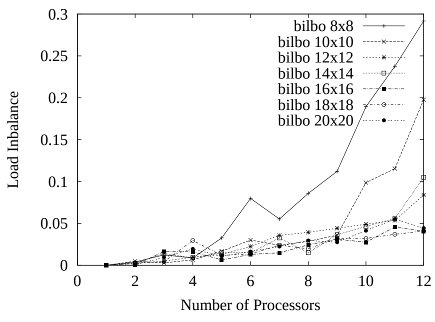
Finally, we present some results related to the data and image scalability of our parallel code. We took the SPX dataset and subdivided it multiple times (by breaking each tetrahedra into eight). For each version of the dataset, we rendered it ten times along a uniform rotation of the y-axis. The images were computed at different resolutions, and the full results are reported in Table 13, and some subset are plotted in Fig. 35.

5.5 Conclusion

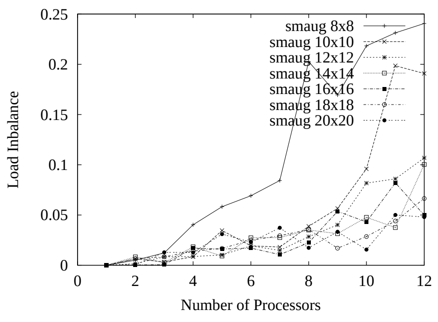
In this Chapter we present a simple parallelization of the ZSWEEP algorithm for distributed-shared memory machines. Other than changes to the actual code to make it more modular, and to isolate shared variables, we only had to perform one major architectural change to the algorithm to make it parallel: the introduction of an octree for the vertices so we can efficiently find which faces project into a given tile. In this work, we were able to keep all the nice features of ZSWEEP, i.e., the fact that it is very simple to implement, robust, and memory efficient.

We were able to achieve a parallel efficiency of 84% on 16 processors on an SGI Origin 3000 machine. The complexity of rendering a tile is dependent both on the number of primitives which project on the tile, and the area of the tile. In order to further speed up the code for more processors, we believe we might need a more fine grain load balancing scheme which is able to dynamically partition regions when we discover that we have too many primitives that project in it.

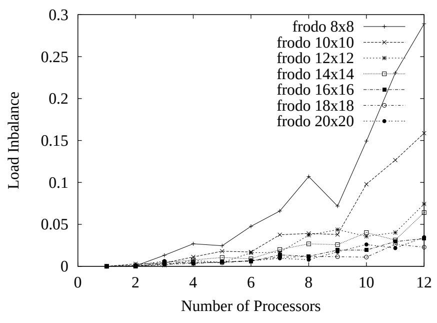
It would be useful to run our code on larger SMP machines. The reported results are for a version of the code parallelized with the **m_fork** calls of SGI IRIX. We have ported this code to POSIX Pthreads, which runs quite well on Linux, but we have not performed detail analysis of the Pthread version performance yet.



(a) **bilbo**



(b) **smaug**



(c) **frodo**

Figure 34: Load imbalance with different tiling parameters. Post dataset for images of size 512×512.

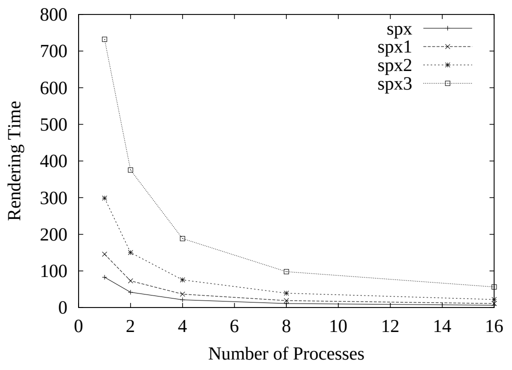


Figure 35: See data from Table 13 for the 2048 × 2048 image dimension.

Chapter 6

I/O Volume Rendering

[1] We address the problem of rendering large unstructured volumetric grids on machines with limited memory. This problem is particularly interesting because such datasets are likely to come from computations generated on supercomputers, that is, machines with superior resources than even the most powerful workstations.

Here, we present a set of techniques which can be used to render arbitrarily large datasets on machines with very little memory. In particular, we present two techniques which vary in rendering speed, disk and memory usage, ease of implementation, and preprocessing costs. The first technique is completely disk-based, and requires a small amount (actually, constant) main memory. It works by performing one scan over the file containing the unstructured grid (assuming this file has been *normalized* as a pre-processing step), one external-memory sort, and a final accumulation scan which computes the image. The

[1]This Chapter is based on work to appear: Out of Core Rendering of Large Unstructured Grids. R. Farias and C. Silva, To appear in the Special Issue of CG&A – Large-Scale Data Visualization, July/August 2001.

second technique is based on our ZSWEEP algorithm, and it is more involved both in its preprocessing, implementation, and main memory requirements, but it is substantially faster, in some cases up to an order of magnitude faster.

We have implemented both techniques, and we present results on rendering a few large datasets under different conditions (image resolutions, main memory configurations, etc), and discuss the tradeoffs of using the techniques presented in this Chapter.

6.1 Introduction

The need to visualize unstructured volumetric data arises in a broad spectrum of applications including structural dynamics, structural mechanics, thermodynamics, fluid mechanics, and shock physics. One of the most powerful visualization techniques is direct volume rendering, a term used to define a particular set of rendering techniques which avoids generating intermediary (surface) representations of the volume data. Instead, the scalar field is generally modeled as a cloud-like material, and rendered by computing a set of lighting equations [47].

In this Chapter, we address the problem of direct volume rendering of large unstructured volumetric grids on machines with limited memory. This problem is particularly interesting because such datasets are likely to come from computations generated on supercomputers, that is, machines with superior resources to even the most powerful workstations.

Our work nicely complements the recent trend of developing efficient out-of-core scientific visualization techniques. Given a large unstructured grids, we currently have a number of external memory visualization tools, (e.g., streamline computation [66], isosurface computation [10], surface simplification [41]), which enable scientists to visualize their large datasets on machines with limited memory. For instance, by coupling the techniques of [10] and [41] isosurfaces of arbitrarily large datasets can be computed and simplified, effectively making it possible to visualize such large datasets on any machine with enough disk. Our work is adding direct volume rendering algorithms to this already considerably powerful toolbox.

We present two techniques which vary in rendering speed, disk and memory usage, ease of implementation, and preprocessing costs. The first technique is completely disk-based, and requires a small amount (actually, constant) main memory. The second technique is based on our ZSWEEP algorithm, and it is more involved both in its preprocessing, implementation, and main memory requirements, but it can be substantially faster.

6.2 Related Work

In this section we cover the work related to both fields of rendering of unstructured grids data sets, and out-of-core visualization.

6.2.1 Unstructured Grid Volume Rendering

Here, we consider existing unstructured-grid volume rendering techniques from a memory-usage point of view, their applicability to render very large grids, and potential extensions for out-of-core rendering. The memory usage of current techniques vary widely, and it is not straightforward to classify the different techniques. Among the various characteristics that generally affect the memory usage of existing techniques, are:

- size of the dataset, in terms of its number (and type) of cells and vertices[2];
- screen resolution (and image-space "depth" of the dataset) [3];
- use of mesh connectivity information, some techniques explicitly use connectivity information, while others use different means of inferring it (such as discrete buffers used for determining depth information), or completely avoid using any kind of connectivity;
- underlying data structures used for efficiency or accuracy, for instance, some techniques cache extra information per cell, or per face, of the dataset for efficiency purposes.

A number of efficient algorithms for rendering irregular grids have been developed. One class of algorithms is based on adapting ray tracing techniques for rendering unstructured grids, such as in the works of Garrity [23], Uselton [67], Bunyk et al [6]. In general, these techniques require random access

[2]Given a mesh with t tetrahedra and n vertices, the "bare essential" amount of memory necessary to hold it is $16(t+n)$ bytes.

[3]In image space the memory costs depends on the screen resolution, and on the "thickness" of the dataset along the z direction. Some techniques compute "slices" along z computed by intersecting discrete buffers of the same resolution as the screen with the unstructured grid. Assuming a byte per color channel, for computing an image of size N-by-N, with s slices, one would need $4sN^2$ bytes. We note that s should vary with the resolution of the dataset in z, that is, if there exists a ray which intersects the dataset in s_{max} cells, the closest s gets to s_{max}, the more accurate the image we (can) obtain.

to the cells, connectivity information, and in some cases, e.g., [6], extra memory to optimize the computation of intersections of rays with faces of the cell complex. In [82], an optimization for [6] is proposed which attempts to reduce the memory requirements by compositing samples as early as possible, but the proposed (view-independent) traversal is not able to limit the overall memory use. (The work of Hong and Kaufman [31] although similar to [6] is optimized for curvilinear grids, and uses considerably less memory since it uses the grid structure, and does not explicitly store cell or connectivity information.)

Other techniques have been developed which use scan-line algorithms, which sweep the data with a plane perpendicular to the image plane [74, 8]. Some of them, e.g. [61], are designed to be memory efficient, but still use the connectivity of the mesh. Others, such as those proposed by Giertsen [25] and Westermann and Ertl [68, 69] use discrete buffers to determine the order of compositing, and completely avoid the need for connectivity information. The use of discrete buffers in z have the potential to lower the accuracy of these techniques, and the buffers themselves can require a substantial amount of memory.

Some methods [81, 20] employ a different kind of sweep algorithm [56], and sweep planes in z. Yagel et al [81] samples the irregular grid with a fixed number (e.g., 50 or 100) of planes which are later composited together. Their technique does not use connectivity, but the space to keep the planes can be quite substantial, since it amounts to computing and caching a large number of images. Farias et al [20] developed ZSWEEP, also based on sweeping a plane in the z direction.

Another approach for rendering irregular grids is the use of projection ("feed-forward") methods [73, 48, 76, 59, 13] in which the cells are projected onto the screen, one-by-one. Most of these techniques exploit the graphics hardware to compute the volumetric lighting models [59], by first computing a *visibility ordering* [48, 76, 63, 14], and incrementally accumulating their contributions to the final image. With respect to memory usage, we can separate the visibility ordering algorithms into two classes: those that use connectivity to compute the ordering, e.g. [76, 14], and those that use some form of power sorting, e.g. [13]. The power sorting techniques only require an extra floating point number per cell, and they do not use connectivity information, but in general those techniques are not guaranteed to generate "correct" sorting results for a wide class of grids.

A simple approach (initially discussed in [60]) is to naively compute all intersections between each ray cast with all the cells, and perform a post-sorting to compute the image. That is, given an N-by-N image, and n cells, for each of the N^2 rays, compute the $O(n)$ intersections with cell facets in time $O(n)$, and then sort these crossing points (in $O(n \log n)$ time). However, this results in overall time $O(N^2 n \log n)$, and does not take advantage of coherence in the data: the sorted order of cells crossed by one ray is not used in any way to assist in the processing of nearby rays. Ma and Crockett [44] used this approach in the context of parallel architectures. Their technique distributes the cells among processors in a round-robin fashion. For each viewpoint, each processor independently computes the ray intersections, which are later composited in a second phase of the algorithm. To avoid the storage of a very large number of ray intersections, Ma and Crockett cleverly schedule the computation using a k-d tree.

6.2.2 Out-Of-Core Scientific Visualization

In this section, we briefly review the existing work on out-of-core scientific visualization techniques. For a general introduction to the theory and practice of external memory algorithms, we refer the interested reader to Abello and Vitter [1].

Cox and Ellsworth [15] propose a general framework for out-of-core scientific visualization systems based on application-controlled demand paging. Leutenegger and Ma [39] propose to use R-trees [27] to optimize searching operations on large unstructured datasets. Ueng et al [66] uses an octree partition to restructure unstructured grids to optimize the computation of streamlines. Shen et al [58] and Sutton and Hansen [65] have developed techniques for indexing time-varying datasets. Shen et al [58] apply their technique for volume rendering, while [65] focusses on isosurface computations.

Chiang and Silva [9] worked on I/O-optimal algorithms for isosurface generation. An interesting aspect of their work is that even the preprocessing is assumed to be performed completely on a machine with limited memory. Though their technique is quite fast in terms of actually computing the isosurfaces, the disk and preprocessing overhead of their technique is substantial. This lead to further research [10] on techniques which are able to trade disk overhead for time in the querying for the active cells. They developed a set of useful meta-cell preprocessing techniques.

Recently, external memory algorithms for surface simplification have been developed by Lindstrom [41] and El-Sana and Chiang [18]. The technique presented in [41] is able to simplify arbitrarily large datasets on machines with just enough memory to hold the output (i.e., the simplified) triangle mesh.

6.3 Out-Of-Core Rendering Algorithms for Unstructured Grids

In this section, we present two efficient direct volume rendering techniques for unstructured volumetric grids. The first technique is completely disk-based, and requires a small amount of main memory. The second technique is based on our ZSWEEP algorithm, and it is more involved both in its preprocessing, implementation, and main memory requirements, but it is substantially faster.

6.3.1 Memory-Insensitive Rendering

In developing efficient external memory algorithms one has to be aware of some of the characteristics of computer disks, and their difference to the (in-core) main memory system we are all accustomed to. The basic difference is that disks are not efficient for random access to locations because “seeks” require a large amount of mechanical movement (of the heads). For sequential access, disks are actually quite fast, with a raw bandwidth within a factor of 20 of the main memory system. Also, disk bandwidth can be increased quite inexpensively by using several disks in parallel. The appeal of hard drives is that the cost is much lower, on the order of 100 times cheaper than main memory. The need for sequential access when using disks has profound implications for external memory algorithms.

First of all, the file formats used for out-of-core algorithms have to be different, and generally more redundant. Indexed mesh formats are common for main memory techniques. For instance, it is common to save a list of the vertices represented with four floats: the position (x, y, z) and scalar field value; and the list of tetrahedra, referenced by four integers which refer to the vertices that define the given tetrahedron. Before such datasets can be used in our algorithm, they need to be "normalized", a process which dereferences the pointers to vertices. This process is thoroughly explained in [10]. For completeness, we briefly explain how to normalize such a file, with v vertices, and t tetrahedra. In an initial pass, we create two (binary) files, one with the list of vertices, and another with the list of tetrahedra. Next, in four passes, we dereference each index of tetrahedral file, and replace it with the actual position and scalar field values for the vertex. In order to do this efficiently, we first (externally) sort the current version of the tetrahedra file in the index we intend to dereference. This takes time $O(t \log t)$ using an (external memory) mergesort. Then, we perform a synchronous scan of both the vertex and (sorted) tetrahedra file, reading one record at a time, and appropriately outputting the deferenced value for the vertex. Note that this can be done efficiently in time $O(v + t)$ because all the references for vertices are sorted. When we are all done with all four passes, the tetrahedra file will contain t records with the "value" (not reference) of each of its four vertices.

We can now describe our first out-of-core rendering technique. The algorithm receives as input a transformation matrix, screen resolution, the normalized tetrahedron file, and associated transfer functions for lighting calculations.

(1) The first step in our algorithm is to read each cell (tetrahedron) from

the normalized file, transform it with the specified transformation matrix, and compute of all its ray intersections. For each pixel ρ_i, which intersects the cell in the interval (z_0, z_1), we output two records (ρ_i, z_0) and (ρ_i, z_1).

For lighting calculations, we also save an interpolated scalar field value. This allows for fast re-generation of images with different transfer functions, or (with some changes) the efficient rendering of time-varying datasets.

The amount of memory necessary to perform this step is minimal, just enough to hold the description of the cell, and temporary storage to compute one intersection, since they are written to disk one by one as they are computed. The amount of disk space required is proportional to the number of actual ray stabbings between rays and cells.

(2) The second (and generally, most time consuming) step in our algorithm consists of sorting the file with the ray intersections computed in the previous step, using an appropriate compare function. The compare function we use sorts primarily on the pixel id p_i, and secondarily on the depth of intersection z. That is, after the file is sorted, and the records for a particular pixel are together (i.e., they appear sequentially on the file), furthermore the records are ordered in increasing depth.

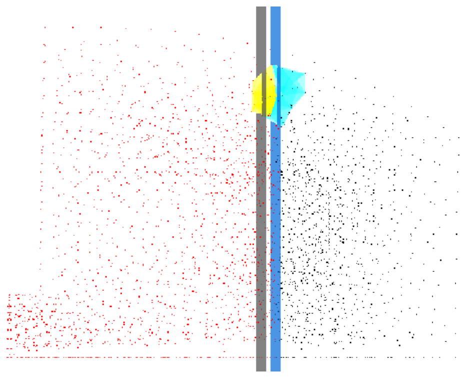
(3) The third, and final step in our simple scheme is to traverse the file generated in the previous step, use the transfer functions to light and composite the samples, which are already in the correct order.

Our simple algorithm is essentially an external memory version of the technique previously considered by [60, 44]. We would like to note that [60] discarded the technique as too inefficient because it did not use coherency between ray. In [44], this technique is used for its good load balancing characterists, and to make it practical, they had to optimize it to save space. But as an external memory technique, it is quite useful by itself, since it can render an arbitrarily large image of an arbitrarily large dataset if enough disk exists to save the intersection crossings, and it is extremely simple to implement. It does not use "any" random access to the dataset, and its implementation is extremely simple, only requiring an external sort routine, and code to perform ray-cell intersection.

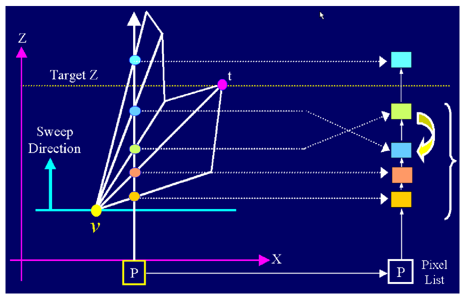
6.3.2 Out-Of-Core ZSWEEP

In this section, we describe a slightly more complex, but often more efficient out-of-core unstructured grid renderer, based on our ZSWEEP algorithm [20] (see Figure 36 for an overview).

There are two sources of main memory usage in ZSWEEP: the pixel intersection lists, and the actual dataset. (The dataset storage requirements are our largest memory use, in fact besides the storage for the actual vertices and cells, we also need to keep the "use set" of each vertex, that is, the cells incident to each given vertex.) The basic idea in our out-of-core technique is to break the dataset into chunks of fixed size, which can be rendered independently without using more than a constant amount of memory. To further limit the amount of memory necessary, we subdivide the screen into tiles, and for each tile, we render the chunks that project into it in a front-to-back order, thus enabling

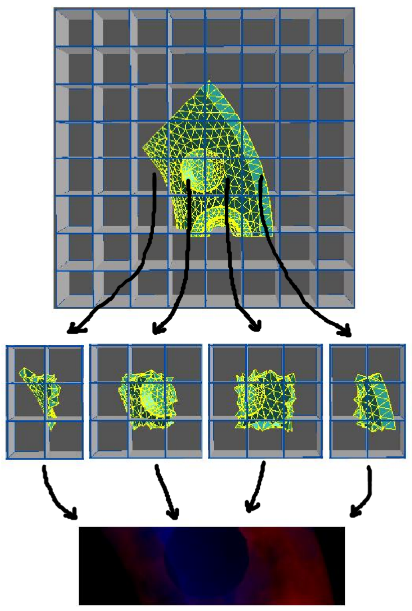


(a)

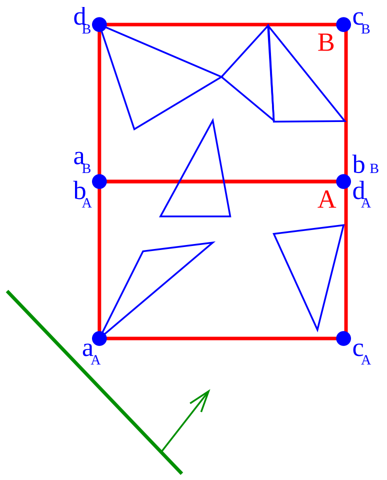


(b)

Figure 36: The main idea of the (in-core) ZSWEEP algorithm [20] is very simple; it is based on sweeping the data with a plane parallel to the viewing plane (shown in blue on (a)), in order of increasing z, *projecting* the faces of cells that are incident to vertices as they are encountered by the sweep plane. ZSWEEP's face projection is different from the ones used in projective methods, e.g. [59]. During face projection, we simply compute the intersection of the ray emanating from each pixel, and store their z-value, and other auxiliary information, in *sorted* order in a list of intersections for the given pixel. The actual lighting calculations [47] are deferred to a later phase (b). Compositing is performed as the "target Z" plane (shown in gray on (a)) is reached. The efficiency arises from: (1) the fact that the algorithm exploits the implicit (approximate) global ordering that the z-ordering of the vertices induces on the cells that are incident on them, thus leading to only a very small number of ray intersection are done out of order; (2) the use of early compositing which makes the memory footprint of the algorithm quite small. The key properties for the efficiency of ZSWEEP is the fact that given a mesh with v vertices and c cells, the amount of sorting ZSWEEP does is $O(v \log v)$ (in practice), i.e., depending on the number of ray intersections, this is substantially lower than the amount of sorting necessary to sort all the intersections for each pixel.



(a)



(b)

Figure 37: The rendering is performed in tiles, as shown in (a). Basically, for each tile, we find $\mathcal{M}$ the set of the meta-cells which project into it. Then, we sort the vertices of the bounding boxes of $\mathcal{M}$ in depth (front-to-back) order by inserting them on a queue $\mathcal{Q}$. The queue is used for sweeping the vertices, which have several marks, in particular, we tag vertices based on whether they are "bounding-box" or "dataset" vertices. When the first bounding-box vertex of a meta-cell m is touched, we retrieve all the vertices and cells of m from disk, transform the vertices, and insert them on $\mathcal{Q}$, tagging them as "dataset" vertices. The processing of out-of-core ZSWEEP is essentially the same as the in-core algorithm, but it performs operations lazily. As vertices are reached, faces are projected, and the overall operation is performed as shown in Figure 36. As bounding-box vertices are touched, we keep track of the number of bounding-box vertices of a given meta-cell we have seen so far. When this number is eight, we can safely deallocate the metacell, i.e., in (b), when we reach vertex d_a, we can free the memory from meta-cell a.

Dataset Information					
Dataset	# of vertices	# of cells	Octree	Leaves	Norm-file
Blunt Fin	41K	187K	40 KB	26 MB	12.7 MB
Comb. Chamber	47K	215K	40 KB	23 MB	14.6 MB
Oxygen Post	109K	513K	110 KB	82 MB	34 MB
Delta Wing	212K	1005K	254 KB	205 MB	68 MB
SPX	2.9K	13K	2.6 KB	1.2 MB	0.8 MB
SPX1	20K	103K	15 KB	12 MB	8 MB
SPX2	150K	830K	63 KB	110 MB	71 MB
SPX3	1150K	6620K	56 KB	706 MB	641 MB

Table 15: The first four are tetrahedralized versions of the well-known NASA datasets. SPX is an unstructured grid – see Figure 36(a) and 37(a) – composed of tetrahedra. We have subdivided each tetrahedron into 8, for each version of the last three, that is, SPX3 is 512 times larger than SPX. We list the number of vertices (in thousands), number of tetrahedra (in thousands), the size of the file that contains the octree information (in kilobytes), the size of the meta-cell data file that contains the information for each leaf (in megabytes), and the size of the normalized dataset (in megabytes).

the exact same optimizations which can be used with the in-core ZSWEEP algorithm. This idea of subdividing the screen into tiles, and the dataset into chunks which are rendered independently has successfully been applied to a parallelization of ZSWEEP.

Our algorithm is divided into two parts: a view independent preprocessing phase, which has to be performed only once and generates a data file on disk which can be used for all rendering requests; and a (view-depending) rendering algorithm. We described both of these phases next.

Preprocessing. Our preprocessing is simple, and quite similar to the meta-cell creation of Chiang et al [10]. Basically, we break the dataset file into several meta-cells[4] of small (roughly fixed) size. Given a "target" number of vertices per meta-cell, say m (our of v vertices total), we first externally sort all vertices by the x-values, and partition them into $\sqrt[3]{\frac{v}{m}}$ consecutive parts. Then, for each such chunk, we externally sort its vertices by the y-values, and partition them into $\sqrt[3]{\frac{v}{m}}$ parts. Finally, we repeat the process for each refined part, except that we externally sort the vertices by the z-values. We take the final parts as chunks. This is the main step in constructing the chunks, since it determines its shape and location in space. Observe that chunks may differ dramatically in their volumes, but their numbers of vertices are roughly the same. In general, the number of meta-cells is relatively small, can be safely assumed to fit in memory. In order to render a meta-cell, ZSWEEP needs to have all the cells that "spatially intersect" that metal-cell, and all the vertices that belong to those cells. These computations can be efficiently computed in external memory, for full details, we point the reader to Section 2.1 of [10]. The output of our preprocessing are two files, a small one with high level description of the meta-cells, including their bounding box, number of vertices, number of cells, and a pointer to the start of the "data" for the meta-cell in the main data file. The larger data file essentially has a list of the vertices and cells for each meta-cell. Note that several vertices and cells appear repeated (possibly multiple times) in this data file, since each meta-cell

[4]The meta-cells themselves, and their construction described in [10] are slightly different, since each cell belongs to a single meta-cell while in our case a cell belongs to as many meta-cells as it spatially intersects. This is not a substantial difference, and the normalization techniques described there still apply.

Rendering Times for the in-core ZSweep			
Dataset	512^2	1024^2	2048^2
SPX	7	26	118
SPX1	14	46	203
SPX2	29	93	383
SPX3	107	238	834

Table 16: Rendering times for the in-core ZSweep code running with one gigabyte of RAM.

is a self-contained unit.

Rendering Algorithm. Our rendering algorithm is quite simple. Basically, divide the screen in tiles and we render the image tile by tile. For each tile, we compute the meta-cells which intersect that tile, sort the meta-cells in a front-to-back order, and render it using the ZSWEEP algorithm. The details are shown in Figure 37.

6.4 Experimental Results

We report results for our two out-of-core rendering techniques proposed. We also include results for the in-core ZSWEEP algorithm. When not indicated, our results were obtained on a PC class machine equipped with an AMD K7 "Thunderbird" 1GHz processor, one IDE disk, and one gigabyte of main memory running Linux. In order to limit the amount of main memory available for testing purposes, we used the capability offered by the Linux kernel to indicate the amount of main memory to use by means of specifying the boot parameters directly into lilo, i.e., specifying "linux mem=32M" at the boot prompt. A similar methodology was used in [10]. Table 15 has information about the datasets used in our tests.

Memory Insensitive Rendering. We have generated several images of the benchmark datasets using our memory-insensitive irregular grid rendering MIR algorithm. Theoretically, MIR should have no dependency on the amount of main memory available. See Table 18. In all our experiments, our code never used more than 5 MB of main memory. It takes the normalized file as its input. Given a new point of view, it rotates the cells one by one, and projects their faces on the screen, by a scan conversion which is directly saved in a file, the projection file. The size of the projection file depends on the the dimension of the image, and also on the number of segments generated for each pixel. It can get quite large, but the algorithm works just the same. Note that the cost of the last step of the algorithm, the compositing, also depends on the average length of segments. Depending on the dataset and image size, MIR can use quite a lot of disk space, e.g., for the Delta, the projection file has 304 MB for a 512-by-512 image, 1.2 GB for a 1024-by-1024, and 4.8 GB for a 2048-by-2048.

Large Images. We have ran some tests with a large "cfd" dataset with roughly 1.5 million vertices, and 8.5 million cells. For generating a 5000-by-5000 image (which by itself takes up over 70 MB of disk) took MIR a total of 224 seconds on a SGI Origin 3000 equipped with R12K 400Mhz processors, and a fast SCSI disk array. The reason this is faster than in our other datasets

Out-Of-Core ZSweep Rendering Times						
Dataset	512^2		1024^2		2048^2	
SPX	8	615	34	2615	154	11846
SPX1	24	233	72	699	305	2961
SPX2	78	93	160	192	595	716
SPX3	289	43	418	63	1157	174

Table 17: Rendering times for the OOC-ZSweep using 128 MBytes of RAM. We show the time to generate the image and the cost per cell (in μs).

is that the number of actual ray intersections is quite small. We also generated a 10K-by-10K image from the same data set that took 824 seconds. In this case, the image occupies 300 megabytes of disk.

Out-Of-Core ZSWEEP. Tables 16 and 17 show some results with our OOC-ZSWEEP code. OOC-ZSweep has essentially constant memory usage per dataset irrespective of the size of the images being generated, being able to generate images which the original in-core ZSWEEP could not. For a large 2048-by-2048 image of the Delta, the in-core ZSWEEP would need over 380 MB of memory, while the OOC-ZSWEEP needs about 24 MB.

As we can see from our experiments, MIR and OOC-ZSweep are quite practical techniques which can be used under different conditions. OOC-ZSweep is usually more efficient than MIR, sometimes by a factor of 10 or more, but it requires that we preprocess the files with the meta-cell technique of [10] before renderings can be performed. Also, OOC-ZSweep definitely uses more memory than MIR. For generating a few high-resolution images of large datasets, MIR might be a very good choice. It is particularly simple to implement.

MIR Rendering Times with 32 MB RAM				
	Screen Resolution 512x512			
Dataset	Proj.	Order	Comp.	Total
Blunt Fin	45	213	44	302
Comb. Ch.	10	19	6	35
Oxygen Post	81	386	75	542
Delta Wing	103	412	79	594
	Screen Resolution 1024x1024			
Dataset	Proj.	Order	Comp.	Total
Blunt Fin	171	1030	180	1381
Comb. Ch.	24	82	26	132
Oxygen Post	291	1747	316	2354
Delta Wing	338	1965	322	2625
	Screen Resolution 2048x2048 *			
Dataset	Proj.	Order	Comp.	Total
Blunt Fin	254	589	233	1076
Comb. Ch.	52	190	55	297
Oxygen Post	435	922	422	1779
Delta Wing	496	1062	430	1988

Table 18: The table shows detailed timing information. The four columns for each image dimension show the time taken to project the cells on the screen, the time to order the projection file, the time to compose all intersections and the total render time. * The times for the 2048-by-2048 were obtained on a SGI R12K 400Mhz system, with a fast SCSI disk array. Although the processor is slower, the times are improved by the faster disks.

The MIR code is considerably slower, since it performs a lot more sorting[5] and disk I/O. We would like to point out that MIR might be particularly useful when trying to render a dataset from the same viewpoint with a different transfer funcion. Since, the lighting calculations are done during compositing (usually the least expensive pass), one can effectively generate images with different classifications very efficiently. Also, it would be efficient to render time-varying datasets.

[5]External sort algorithms are very important for the design and implementation of I/O-efficient algorithms. There are several issues in implementing external memory algorithms, and these issues can greatly affect the overall performance of a system. A particularly efficient external sort is **rsort** written by John Linderman at AT&T Research. We use **rsort** for the results presented in this Chapter.

Chapter 7

Conclusions

In this thesis we described our research on techniques for rendering large unstructured grid data. Our contributions include the development of the new algorithm, the *ZSweep*, its parallelization and also an *out-of-core* version of it. A simple out-of-core volume rendering is also presented not only for comparison purposes, but also as a simpler option.

First, we describe approximation methods to speed up a fast ray casting rendering algorithm for irregular grids data sets. Such approximation methods were put together to enable one to trade off between image approximations and image generation speed. Almost real-time frame rates are achieved for medium-sized data sets.

We also describe the novel ZSweep algorithm based on the sweep paradigm for rendering irregular grids data sets that is up to five orders of magnitude faster than previous algorithms based on the same paradigm. We discuss the practical and theoretical issues involved in rendering irregular grids.

A shared memory parallel implementation and an *out-of-core* implementation are presented for the *ZSweep* algorithm, showing the details of the implementation and the results obtained.

There are more possible optimizations and implementations to be done. We mention as future work an implementation of the algorithm for distributed memory architecture. The *ZSweep* algorithm should fit well in such architecture, since its simple mechanism allows a small overhead, as we noticed in the shared memory implementation, where a very good load balancing was achieved by means of a simple data distribution.

Our *ZSweep* code is being adapted by Kitware to be included into *VTK* to extend its ability to handle irregular grids data sets. In addition, we are working on an adaptation of the algorithm that will enable it to be incorporated into graphics hardware.

We consider as the greatest contribution of our research the introduction of a new rendering algorithm, **ZSweep**, for unstructured grids data sets, which brought more insight and new ideas to the field. For instance, our out-of-core approach evolved naturally from the data partitioning method in our parallel algorithm, which, in turn, was comparable to prior techniques of partitioning used in out-of-core mesh simplification and iso-surface extraction.

A future goal is to develop a distributed memory version of *ZSweep* and integrate it within a parallel volume rendering system.

Bibliography

[1] J. Abello and J. Vitter. *External Memory Algorithms and Visualization.* DIMACS Book Series, American Mathematical Society, 1998.

[2] D. Badouel, k. Boauatouch, and T. Priol. Distributing Data and Control for Ray Tracing in Parallel. *IEEE Computer Graphics and Applications*, pages 69–77, Vol. 14, Number 4. July 1994.

[3] C. Bajaj, V. Pascucci, D. Thompson, and X.Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium*, pages 97–104, 1999.

[4] J. F. Blinn. Light Reflection function for simulation of clouds and dusty surfaces. *In Proc. SIGGRAPH'82*, pages 21–29, 1982.

[5] C. Bradford Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

[6] P. Bunyk, A. Kaufman, and C. Silva. Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization, Proceedings of Dagstuhl '97*, pages 30–36, 2000.

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

[8] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 81–88, November 1993.

[9] Y. J. Chiang and C. Silva. I/O optimal isosurface extraction. *IEEE Visualization '97*, pages 293–300, November 1997.

[10] Y. J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. *IEEE Visualization '98*, pages 167–174, October 1998.

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

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

[13] P. Cignoni, C. Montani, and R. Scopigno. Tetrahedra based volume visualization. In H.-C. Hege and K. Polthier, editors, *Mathematical Visualization – Algorithms, Applications, and Numerics*, pages 3–18. Springer Verlag, 1998.

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

[15] M. B. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. *IEEE Visualization '97*, pages 235–244, November 1997.

[16] T. Crockett. ICASE : Parallel Rendering. NASA, ICASE Report No. 95-31, 1995.

[17] D. Culler, J. Singh, and A. Gupta. Parallel Computer Architecture, A Hardware-Software Approach. Morgan-Kaufmann, 1999.

[18] J. El-Sana and Y. J. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3), August 2000.

[19] D. Ellsworith. A New Algorithm for Interactive Graphics on Multicomputers. *IEEE Computer Graphics and Applications*, pages 33–40, Vol. 14, Number 4. July 1994.

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

[21] R. Farias and C. Silva. Parallelizing the ZSWEEP algorithm for distributed-shared memory architectures. *Submitted for publication*, 2001. Available from http://www.ams.sunysb.edu/~rfarias.

[22] M. Garland and P. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. *IEEE Visualization '98*, pages 263–270, October 1998.

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

[24] A. Van Gelder, V. Verma, and J. Wilhelms. Volume Decimation of Irregular Tetrahedral Grids. In Computer Graphics International, June 1999.

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

[26] R. Gregory. Eye and Brian. *Princeton University Press*, 1990.

[27] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf. Principles Database Systems*, pages 47–57, 1984.

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

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

[30] C. Hofsetz and K. L. Ma. Multi-threaded rendering unstructured-grid volume data on the sgi origin 2000. In *Third Eurographics Workshop on Parallel Graphics and Visualization*, 2000.

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

[32] J. Huang, R. Crawfis, and D. Stredney. Edge preservation in volume rendering using splatting. *1998 Symposium on Volume Visualization*, pages 63–69, 1998.

[33] J. T. Kajiya, B. P. Von Herzen. Ray tracing volume densities. *In Proc. SIGGRAPH'94*, pages 165–174, 1994.

[34] R. Kalawsky. The Science of Virtual Reality and Virtual Environments. *Addison-Wesley*, 1993.

[35] A. Kaufman. Volume Visualization. *IEEE Computer Society Press*, ISBN 908186-9020-8, 1990. Los Alamitos, CA.

[36] W. Krueger. The application of transport theory to the visualization of 3D scalar fields. *Computers in Physics 5*, pages 397–406, 1991.

[37] P. Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. *IEEE Parallel Rendering Symposium*, pages 15–22, October 1995.

[38] P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3), September 1996.

[39] S. Leutenegger and K. L. Ma. Fast retrieval of disk-resident unstructured volume data for visualization. In *External Memory Algorithms and Visualization, DIMACS Book Series, American Mathematical Society, vol. 50*, 1999.

[40] M. Levoy. Efficient ray tracing of volume data. In *ACM Trans. Comp. Graph.*, vol. 9, no. 3, pages 245–261, 1990.

[41] P. Lindstrom. Out-of-core simplification of large polygonal models. *Proceedings of SIGGRAPH 2000*, pages 259–262, July 2000.

[42] W. E. Lorensen, H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *ACM Computer Graphics*, pages 163–196, Vol. 21, Number 4. July 1987.

[43] K. L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. *IEEE Parallel Rendering Symposium*, pages 23–30, October 1995.

[44] K. L. Ma and T. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. *IEEE Parallel Rendering Symposium*, pages 95–104, November 1997.

[45] K. L. Ma and T. Crockett. Parallel visualization of large-scale aerodynamics calculations: A case study on the Cray T3E. *Symposium on Parallel Visualization and Graphics*, pages 15–20, October 1999.

[46] P. Mackerras, and B. Corrie. Exploiting Data Coherence to Improve Parallel Volume Rendering. *IEEE Parallel and Distributed Technology*, pages 8–16, Vol. 2, Number 2. Summer 1994.

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

[48] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):27–33, November 1990.

[49] M. McCormick, T. DeFanti and M. Brown. Visualization in Scientific Computing. *Report of the NSF Advisory Panel on Graphics, Image Processing and Workstations*, 1987.

[50] M. Meissner, J. Huang, D. Bartz. A Practical Evaluation of Popular Volume Rendering Algorithms. *2000 Volume Visualization and Graphics Symposium*, pages 81–90, October, 2000.

[51] S. Molnar, M. Cox, D. Ellsworth, and H. Fichs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, pages 23–32, Vol. 14, Number 4. July 1994.

[52] C. Monks, P. Crossno, G. Davidson, C. Pavlakos, A. Kupfer, C. Silva and B. Wylie. Three Dimensional Visualization of Proteins in Cellular Interactions. *IEEE Visualization '96*, pages = ??–??, 1996.

[53] K. Mueller, T. Moeller, and R. Crawfis. Splatting without the blur. *Proc. Visualization'99*, pages 363–371, 1999.

[54] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization Proceedings*, pages 17–24, October 1992.

[55] H. Pfister and A. Kaufman. Cube-4 – A Scalable Architecture for Real-Time Volume Rendering. *ACM/IEEE Volume Visualization 1996*, pages ??–??.

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

[57] W. Schroeder, K. Martin and B. Lorensen. The Visualization Toolkit. *Prentice-Hall*, 1996

[58] H. W. Shen, L. J. Chiang, and K. L. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. *IEEE Visualization '99*, pages 371–378, October 1999.

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

[60] C. Silva, J. S. B. Mitchell, and A. E. Kaufman. Fast rendering of irregular grids. *1996 Volume Visualization Symposium*, pages 15–22, October 1996.

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

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

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

[64] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *Computing Surveys*, pages 1–55, Vol. 6, Number 1. March 1974.

[65] P. M. Sutton and C. D. Hansen. Accelerated isosurface extraction in time-varying fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, April - June 2000.

[66] S. K. Ueng, C. Sikorski, and K. L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, October - December 1997.

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

[68] R. Westermann and T. Ertl. The VSbuffer: Visibility ordering of unstructured volume primitives by polygon drawing. *IEEE Visualization '97*, pages 35–42, November 1997.

[69] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. *Proceedings of SIGGRAPH 98*, pages 169–178, July 1998.

[70] D. S. Whelan. Animac: A Multiprocessor Architecture for Real Time Computer Animation *Ph.D. Thesis Dissertation*, California Institute of Technology, 1985.

[71] S. Whitman. Multiprocessor Methods for Computer Graphics Rendering. *Jones and Barttlett*, Boston, 1992.

[72] S. Whitman. Load Balancing for Parallel Polygon Rendering. *IEEE Computer Graphics and Applications*, pages 41–48, Vol. 14, No. 4, July 1994.

[73] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):275–284, July 1991.

[74] J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. *IEEE Visualization '96*, pages 57–64, October 1996.

[75] J. Wilhelms. Pursuing interactive visualization of irregular grids. In *Visual Computer, vol. 9, no. 8, 1993.*

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

[77] P. Williams. Parallel volume rendering finite element data. In *Proceedings of Computer Graphics International*, 1993.

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

[79] C. Wittenbrink, T. Malzbender, and M. Gross. Opacity-weighted color interpolation for volume sampling. *1998 Symposium on Volume Visualization*, pages 135–142, 1998.

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

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

[82] C. K. Yang, T. Mitra, and T. Chiueh. On-the-fly rendering of losslessly compressed irregular volume data. In *Proc. IEEE Visualization 2000*, 2000.