PARALLEL VOLUME RENDERING OF IRREGULAR GRIDS

A Dissertation Presented

by

José Cláudio Teixeira e Silva Junior

to the graduate school in partial fulfillment

of the requirements for the Degree of

Doctor of Philosophy

in

Computer Science

State University of New York at Stony Brook

December 1996

State University of New York at Stony Brook

The Graduate School

José Cláudio Teixeira e Silva Junior

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

Arie E. Kaufman, Dissertation Diretor
Leading Professor, Computer Science

Theo Pavlidis, Committee Chair
Leading Professor, Computer Science

Joseph S. B. Mitchell
Associate Professor, Applied Mathematics

Amitabh Varshney
Assistant Professor, Computer Science

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract of the Dissertation

Parallel Volume Rendering of Irregular Grids

by

José Cláudio Teixeira e Silva Junior

Doctor of Philosophy

in

Computer Science

State University of New York at Stony Brook

1996

This dissertation contains our contributions in methods to speed up volume rendering, an important subfield of scientific visualization. We develop a framework composed of a system and a set of algorithms for handling large datasets of various forms (e.g., regular and irregular volumetric grids). A special emphasis of our framework is on the development of practical parallel algorithms for visualization.

For the regular grid case, where research of efficient techniques is fairly advanced, we propose a parallelization of a known rendering algorithm. Our major contributions in this case are the introduction of content-based load balancing and the pipelined compositing approach. We present the new algorithms and their implementation.

For irregular grids, we propose a fast rendering algorithm. Our method uses a sweep-plane approach to accelerate ray casting, and it can handle disconnected and nonconvex (even with holes) unstructured irregular grids with a rendering cost that decreases as the disconnectedness decreases. The algorithm is carefully tailored to exploit spatial coherence even if the image

resolution differs substantially from the object space resolution. We establish the practicality of our method through experimental results, and we also provide theoretical results, both lower and upper bounds, on the complexity of ray casting of irregular grids. Our work in irregular grids also includes a proposal for a parallelization of the method for distributed-memory machines.

Our final contribution is in the simplification of irregular grids. Because the size of the grids can sometimes be overwhelming, we discuss the simplification problem to approximate representations of irregular grids. We present a complete solution for the two-dimensional case (e.g., height-field terrains), and preliminary work on extensions to general polyhedral surfaces and three-dimensional irregular grids.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank the numerous people who helped me throughout my graduate studies. First of all, I would like to thank my advisor, Leading Professor Arie Kaufman, for an endless supply of ideas, suggestions, criticisms and state-of-the-art research facilities without which none of this work would have been possible. I would like to express my gratitude to Professor Joseph Mitchell, for being a mentor and friend; to the members of my committees, Professors Jieh Hsiang, Theo Pavlidis, Steve Skiena, Amitabh Varshney, for their time and useful criticisms; and the Stony Brook faculty, in particular, Professors Amit Bandopadhay, Tzi-cker Chiueh, Ker I. Ko, I. V. Ramakrishnan, David Warren, and Anita Wasilewska for their help and continuing support. Many thanks to the computer science staff, in particular, Kathy Germana, Betty Knittweis, Stella Mannino, Peggy Thomas, and Brian Tria for making my life at Stony Brook so much easier.

Thanks to Pat Crossno, George Davidson, Dino Pavlakos, and Brian Wylie, from Sandia National Labs, who were special friends as well as contributors to my work. To the members of the Visualization Lab, Rick Avila, Ingmar Bitter, Dirk Bartz, Baoquan Chen, Rui Chiou, Akio Doi, Jihad Ell-Sana, Taosong He, Lichan Hong, Shigeru Muraki, Hanspeter Pfister, Lisa Sobierajski, Ikuko

Takanashi, and Sid Wang, who helped me learn so much. Thanks also to Yi-Jen Chiang, Martin Held and Jim Klosowski, for helpful discussions.

I would like to thank the people who directly contributed code to my projects: Mauricio Fan Lok, for his work on PVR; Ashish Tiwari and Bernardo Piquet, for their work on irregular grid rendering; Dan Evans, Francine Evans, and Ken Gordon, for their work on surface decomposition into terrains and simplification; Professor Wm. Randolph Franklin, for making his triangulation code available; Professor David Dobkin, for this 2D arrangement code; and the members of the graphics community, who so often share their hard-developed code through the Internet.

Thanks to Ivan Almeida, Tito Autrey, Karen Bernstein, Mauricio Cortes, Steve Dawson, Patricia Gomez, Owen Kaiser, Daren Krebsbach, Gregório Pacelli, Maria da Paz, C. R. Ramakhrisnan, Jairo Rocha, Lori and Toni Scarlatos, Pedro Souto, Michael Vernick, Michael Wynblatt who made my life at Stony Brook quite enjoyable. Many friends from Brazil helped to keep this work in focus, in particular, Júlio Martins, Professor José Evangelista, Professor Jonas de Miranda, Professor Luquésio Petrola, Professor Antônio Oliveira, and Professor Luis Velho; and my friends from GCCG, Rui Bastos, João Comba, Alexandre Cunha, Marcelo Walter and Marcelo Zuffo.

I thank Eduardo Prado and Renata Grumberg for being very special friends. I would also like to thank my family for support throughout these long years. Special thanks to my brother, Patriolino, my sister Mariana, and my grandmother Maria de Lourdes. My mother, Maria das Graças, who has been a great source of inspiration and support throughout my life, without her it would have been impossible to finish this work. Most of all, I thank my wife,

Juliana, she helped all throughout my graduate studies, in both technical and non-technical matters.

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, Intel and Hewlett Packard.

A Juliana e Mamãe.

Chapter 1

Introduction

This dissertation contains contributions primarily in volume rendering, a subfield of scientific visualization. The major focus of this work is in speeding up volume rendering methods. Faster algorithms can be used in better user interfaces and larger datasets, hence more effective visualization tools.

Scientific visualization has been formally defined only recently [74]. In just a few years, it has grown to span a very large field of research with several important distinct subfields. The reason for the rapid development of visualization research lies in the intrinsic limitations of our perceptual systems. Not only can the human mind make sense of only a very limited amount of information, but also our perceptual systems are well suited for different types of data [43]. Research in scientific visualization encompasses methods that transform raw data into different representations where it is easier for us to understand. This transformation is usually into images, sound or touch (see [52] for details on the use of virtual environments, including haptic feedback). The use of dynamic displays and other immersive technology can help the

human mind process information that normally would be completely outside of the capacity of our senses.

The best way to define scientific visualization is by example. One use of visualization techniques, is to study the interaction between different cells of our immune system. For instance, in [80] a collaboration between biologists and visualization experts lead to the better understanding of the 3D structure of the bonding between Helper T-cell and B-cells. Advanced visualization techniques make it possible for us to look at T-cell receptor densities on the surface of a T-cell/B-cell interaction. With this information, the biologists are able to understand in more detail how our immune system works at the microscopic level, what hopefully can lead to better drugs and treatment of diseases. Without these visualization techniques it is virtually impossible to infer the spatial relationship of the structures. Another application in medicine is in computer-assisted radiation treatment. With the aid of 3D visualization, it is much easier to calculate the correct amount of radiation and its target with the necessary precision. Other applications include surgery planning, prosthesis simulation, etc.

Another completely different set of applications comes from engineering. Over the last century, several techniques for simulating physical processes have been created (e.g., computational fluid dynamics, finite element analysis). Some typical applications are weather prediction, engine simulations, etc. Most of these applications generate large volumes of scalar and vector fields on the order of several gigabytes of data. It is clearly impossible for any human being to inspect this data in raw format (usually matrices of real number). The development of useful and insightful representations of such

data is a very hard research problem in scientific visualization (see [105] for an excellent text on visualization techniques).

Our particular focus in this work is on volume visualization [54], which is concerned with the representation, modeling, manipulation and rendering of volumetric objects. See Chapter 2 for a short introduction to volume visualization techniques. In general, volumes are three-dimensional objects defined over some closed domain, where, at every point in space, a scalar or vector field is defined. Unlike polygon representations, one important property of volumetric representations is that it also contains information from the interior of regions, not only their boundary.

Volume rendering is a general solution to address the complex problem of generating meaningful pictures of volumetric data. The goal is to create insightful graphical representations for the engineer, scientist or physician and to help them understand collections of numerical and physical data. Volume rendering consists of color mapping the properties of the volumetric data in an intuitive and consistent way.

Even though volume rendering applications are being used by some scientists, there are still several obstacles to its effective use. Among these obstacles, the most visible is the response time. Generating volume rendered images is slow, even for relatively simple optical models [70] (e.g., no global illumination) and small datasets. The slowness of volume rendering applications and lack of interactivity prevents it from being widely used, which in turn is the main reason for the widespread use of (lower quality) polygonal isosurface rendering techniques. The latter, even though less accurate and flexible, can be sped up by graphics workstations enabling interactivity between the model

and the scientist. At times, scientists just use the isosurface models [63] for interactivity and experimentation, and perform the final rendering using more costly volume rendering techniques. But this defies one of the real advantages of volume rendering.

There are several reasons for requiring faster volume rendering, among them is the fact that post-mortem visualization can be improved with the addition of computational steering. That is, for long running scientific simulations the ability to steer the computation would be very useful, especially in environments where computations take days to run to completion, and resources are precious (large machines usually have a relatively large batch job queue). Interactive visualization techniques could help the user change simulation parameters and see the results in real-time (see [92] for a real-time volume visualization architecture for regular grid datasets). Virtual reality environments also have a need for real-time volume visualization.

Many of the important scientific visualization applications are actually volume rendering applications. These include the previously discussed medical and biological applications, as well as scientific data visualization. Regardless of the specific application, a common ground to all of them is the large computational requirements by today's standards. For instance, a high resolution computer tomography of the human body is over half a gigabyte in size, not only taking up a lot of storage, but rendering it can take several hours on the most powerful workstations. With technological advances in medical imaging, even larger datasets are expected.

The grand challenge problems are another source of extremely large datasets.

In these problems, scientists are trying to simulate real phenomena for several purposes, including the design of more efficient, less pollutant and crash-resistant cars, studying the earth atmosphere, weather prediction, and even military goals, such as the simulation of nuclear explosions. A major difference between these problems and medical imaging (that is hard to notice for non-experts) is the fact that for these latter problems the datasets are given in irregular format and thus are much harder to render.

In this dissertation we propose methods to speed up volume rendering. We develop a framework composed of a system and a set of algorithms for handling large datasets of various forms (e.g., regular and irregular volumetric grids). A special emphasis of our framework is on the development of practical parallel algorithms for visualization.

For the regular grid case, where research of efficient techniques is fairly advanced, we propose a parallelization of a known rendering algorithm. Our major contributions in this case are the introduction of content-based load balancing and the pipelined compositing approach. We present the new algorithms, their implementation, and performance results.

We also propose a fast algorithm for rendering general irregular grids. Our method uses a sweep-plane approach to accelerate ray casting, and can handle disconnected and nonconvex (even with holes) unstructured irregular grids with a rendering cost that decreases as the disconnectedness decreases. The algorithm is carefully tailored to exploit spatial coherence even if the image resolution differs substantially from the object space resolution. We establish the practicality of our method through experimental results, and we also provide theoretical results, both lower and upper bounds, on the complexity

of ray casting of irregular grids. Our work in irregular grids also includes a proposal for a parallelization of our method for distributed-memory machines.

Our final contribution is in the simplification of irregular grids. Because the size of the grids can sometimes be overwhelming, we discuss the simplification problem to approximate representations of irregular grids. We present a complete solution for the two-dimensional case (e.g., height-field terrains), and preliminary work on extensions to general polyhedral surfaces and three-dimensional irregular grids.

This dissertation is organized as follows:

In Chapter 2 we review basic issues of volume visualization which is needed in later chapters. In Chapter 3 we discuss our work on parallelizing regular grid volume rendering. In particular, we describe *content-based load balancing*, a technique for keeping all the processors of a parallel machine busy during rendering. We also discuss our approach to achieve more efficient overall rendering implementation with *pipelined volume rendering*. A preliminary version of this work appeared in [110, 113].

Rendering of irregular grids is described in Chapter 4, where we propose a new algorithm, as well as practical and theoretical results. The importance of this method lies in the fact that previous software-based techniques were just too slow or inaccurate for practical use for large datasets. This chapter is based on work published in [112]. In Chapter 5, we describe a parallelization scheme of our algorithm to render irregular grids.

Our preliminary results on the simplification of irregular grids is presented in Chapter 6 [111], where we introduce the *Greedy-Cuts* triangulation technique and describe its implementation for the case of height-field terrains.

The PVR system, described in Chapter 7, serves as our testbed for most of our work. PVR is a state-of-the-art parallel rendering system and it is currently being used in a number of research institutions for interactive visualization of very large datasets. Further information can be found in [80, 113].

Finally, Chapter 8 summarizes our work and sets forth some directions of research for the future.

Chapter 2

Overview of Volume Rendering

Volume rendering [54] is a powerful computer graphics technique for the visualization of large quantities of 3D data. It is specially well suited for three dimensional scalar [59, 29, 121, 97] and vector fields [19, 71]. Fundamentally, it works by mapping quantities in the dataset (such as color, transparency) to properties of a cloud-like material. Images are generated by modeling the interaction of light with the cloudy materials [130, 73, 72]. Because of the type of data being rendered and the complexity of the lighting models, the accuracy of the volume representation and of the calculation of the volume rendering integrals [9, 51, 50] are of major concern and have received considerable interest from researchers in the field.

A popular alternative method to (direct) volume rendering is isosurface extraction, where given a certain value of interest $\lambda \in \mathcal{R}$, and some scalar function $f : \mathcal{R}^3 \rightarrow \mathcal{R}$, a polygonal representation for the implicit surface $f(x, y, z) = \lambda$ is generated. There are several methods to generate isosurfaces [63, 75, 82, 86], the most popular being the marching cubes method [63].

Isosurfaces have a clear advantage over volume rendering when it comes to interactivity. Once the models have been polygonized (and simplified [106] – marching cubes usually generate lots of redundant triangles), hardware supported graphics workstation can be used to speed up the rendering. Isosurfaces have several disadvantages: they lack fine detail, they lack flexibility during rendering (especially for handling multiple transparent surfaces), they do not represent the interior information, they are not good for amorphous data, and the binary decision process, where surfaces are either inside or outside a given voxel, tends to create artifacts in the data. (There is also an *ambiguity* problem, that has been addressed by later papers such as [86]).

2.1 Volumetric Data

Volumetric data comes in a variety of formats, the most common being cartesian or regular data. (We are using the taxonomy introduced in [118].) Cartesian data is typically a 3D matrix composed of voxels. A *voxel* can be defined in two different ways, either as the datum in the intersection of each three coordinate aligned lines, or as the small cube, either definition is correct as long as used consistently. While the regular data has the same representation but can also have a scaling matrix associated with it.

Irregular data comes in a large variety, including curvilinear data, that is data defined in a *warped* regular grid, or in general, one can be given scattered (or unstructured) data, where no explicit connectivity is defined. In general, scattered data can be composed of tetrahedra, hexahedra, prisms, etc. An important special case is tetrahedral grids. They have several advantages,

including easy interpolation, simple representation (specially for connectivity information), and the fact that any other grid can be interpolated to a tetrahedral one (with the possible introduction of Steiner points). Among their disadvantages is the fact that the size of the datasets tend to grow as cells are decomposed into tetrahedra. In the case of curvilinear grids, an accurate decomposition makes the cell complex contain five times as many cells. More details on irregular grids are postponed until Chapter 4.

2.2 Interpolation Issues

In order to generate the cloud-like properties from the volumetric data, one has to make some assumptions about the underlying data. This is necessary because the rendering methods typically assume the ability to compute values as a continuous function, and (for methods that use normal-based shading) at times, even derivatives of such functions anywhere in space. On the other hand, data is given only at discrete locations in space usually with no explicit derivatives. In order to correctly interpolate the data, for the case of regular sampled data, it is generally assumed the original data has been sampled at a high enough frequency (or has been low-pass filtered) to avoid aliasing artifacts [40]. Several interpolation filters can be used, the most common by far is to compute the value of a function $f(x, y, z)$ by trilinearly interpolating the eight closest points. Higher order interpolation methods have also been studied [12, 69], but the computational cost is too high for practical use.

In the case of irregular grids, the interpolation is more complicated. Even finding the cell that contains the sample point is not as simple or efficient as in

the regular case [83, 94]. Also, interpolation becomes much more complicated for cells that are not tetrahedra (for tetrahedra a single linear function can be made to *fit* on the four vertices). For curvilinear grids, trilinear interpolation becomes dependent on the underlying coordinate frame and even on the cell orientation [127, 38]. Wilhelms et al. [127] proposes using inverse distance weighted interpolation as a solution to this problem. Another solution would be to use higher order interpolation. In general, it is wise to ask the creator of the dataset for a suitable fitting function.

2.3 Optical Models for Volume Rendering

Volume rendering works by modeling volume as cloud cells composed of semi-transparent material which emits its own light, partially transmits light from other cells and absorbs some incoming light [128, 70, 72]. Because of the importance of a clear understanding of such a model to rendering both, regular and irregular grids, the actual inner workings of one such mechanism is studied here. Our discussion closely follows the one in [128].

We assume each volume cells (differentially) emits light of a certain color $E_\lambda(x, y, z)$, for each color channel λ (red, green and blue), and absorbs some light that comes from behind (we are assuming no multiple scattering of light by particles – our model is the simplest "useful" model – for a more complete treatment see [70]).

Correctly defining opacity for cells of general size is slightly tricky. We define the *differential opacity* at some depth z to be $\Omega(z)$. Computing $T(z)$, the fraction of light transmitted through depth 0 to z (assuming no emission

of light inside the material), is simple, we need to notice that the amount of transmitted light at $z + \Delta z$ is just the amount of light at z minus the attenuation $\Omega(z)$ over a distance of Δz:

$$T(z + \Delta z) = T(z) - \Omega(z)T(z)\Delta z \tag{1}$$

what (after making a division by Δz and taking limits) implies

$$\frac{dT(z + \Delta z)}{dz} = -\Omega(z)T(z) \tag{2}$$

The solution to this linear equation of the first order [17] with boundary condition $T(0) = 1$ is:

$$T(z) = e^{-\int_0^z \Omega(u)du} \tag{3}$$

The accumulated opacity over a ray from front-to-back inside a cell of depth d is $(1 - T(d))$. An important special case is when the cell has constant differential opacity Ω, in this case $T(z) = e^{-\Omega z}$. Before we continue, we can now solve the question of defining *differential opacity* Ω from the *unity* opacity (usually user defined and saved in a transfer function table). A simple formula can express Ω in terms of O:

$$\Omega = \log(\frac{1}{1 - O}) \tag{4}$$

If the model allows for the emission of light inside the material, a similar calculation can be used to calculate the intensity I_λ for each color channel inside a cell. In this case using an initial intensity $I_\lambda(0) = 0$, the final system and solutions are as follows:

$$\frac{dI_\lambda(z)}{dz} = -\Omega(z)I_\lambda(z) + E_\lambda(z) \tag{5}$$

$$I_\lambda(z) = T(z)\int_0^z \frac{E_\lambda(v)}{T(v)}dv \tag{6}$$

Specializing the solution for constant color and opacity cells (as done above) we get the simple solution:

$$I_\lambda(z) = \frac{E}{\Omega}(1 - e^{-\Omega z}) \tag{7}$$

Usually, for computational efficiency, the exponential in the previous equation is approximated by its first terms in the Taylor series. [128, 70, 72] describe in detail analytical solutions under different assumptions about the behavior of the opacity and emitted colors inside the cells, extensions to more complex light behavior and the several tradeoffs of approximating the exponentials with linear functions.

The previous equations show how to calculate the continuous color and opacity intensity, usually this calculation is done once for every cell, and the results from each cell are *composited* in a later step. Compositing operators were first introduced in [93], and are widely used. The most used operator in volume visualization is the **over** operator, its operation is basically to add the brightness of the current cell to the attenuated brightness of the one behind, and *in the case of front-to-back compositing* update the opacities of the cells. The equations for the **over** operator are:

$$C_o = C_a + C_b(1 - O_a) \tag{8}$$

$$O_o = O_a + O_b(1 - O_a) \tag{9}$$

It is important to note, that in these equations the colors are saved pre-multiplied by the opacities (i.e., the actual color is C_o/O_o), this saves one multiplication per compositing operation.

2.4 Ray Tracing

A popular method to generate images from volume data is to use *ray tracing* or *ray casting* [39, 59]. Ray casting works by casting (at least) one ray per image pixel into volume space, point sampling the scene with some lighting model (like the one just presented) and compositing the samples as described in the previous section. This method is very flexible and extremely easy to implement. There are several extensions of basic ray casting to include higher order illumination effects, like discrete ray tracing [131], and volumetric ray tracing [117]. Both of these techniques take into account global illumination effects incorporating more accurate approximations of the more general rendering equation [50].

Because of the large size of volumes, volumetric ray casting (and ray tracing) is very expensive. Several optimizations have been applied to ray tracing [60, 61, 20]. One of the most effective optimizations are the *presence acceleration* techniques, that exploit the fact volumetric data is relatively sparse [60, 61, 20, 134, 133]. Levoy [60] introduced the idea by using an octree [99] to skip over empty space. His idea was further optimized by Danskin and

Hanrahan [20] to not only skip over empty space, but also to speed up sampling calculations over uniform regions of the volume. Another important acceleration techniques include *adaptive image sampling* and *early ray termination*. Recently, Lacroute and Levoy [55] have introduced a hybrid method that combines some of the previous optimizations in a very efficient class of volume rendering algorithms.

Avila, Sobierajski and Kaufman [8, 116] introduced the idea of exploiting the graphics hardware on workstations to speed up volume rendering. First, they introduce PARC (Polygon Assisted Ray Casting) [8], a technique that uses the Z-buffer [31] to find the closest and farthest possibly contributing cells. Later, a revised technique [116] is proposed that (still using the Z-buffer) can produce a better approximation of the set of contributing cells.

Their algorithm consists of first creating a polygonal representation of the set of contributing cells (based on axis aligned quadrilaterals) from a *coarse* volume (see Figure 1). The coarse volume is calculated by grouping neighboring voxels together, creating *supervoxels*. Each supervoxel is then tested for the presence of *interesting* voxels (i.e., voxels that belong to the range of voxels mapped to non-zero intensities and opacities by the transfer functions). All six external faces of supervoxels are then marked based on its possible visibility (the second method seems to need to project all the faces).

In order to perform the actual rendering, in the first method (called *Depth Buffer PARC*), all the visible quadrilaterals are transformed and scan-converted twice. Once for finding the first non-empty front voxel, and again to determine the final integration location. In the second method (called *Color Buffer PARC*), a sweep along the closest major axis is generated by coloring the

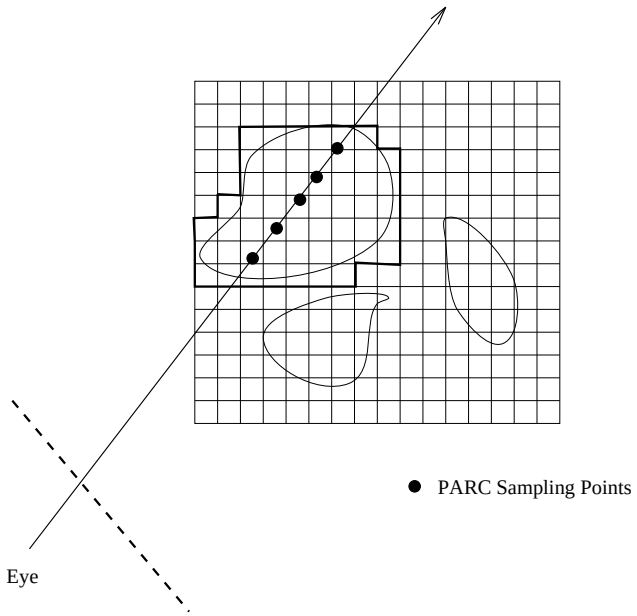


Figure 1: *Polygon Assisted Ray Casting.*

PARC cubes with power of two numbers (so they do not interfere with each other), what leaves a footprint of the intervals (t_i, t_{i+1}) that can be used to better sample the regions having interesting voxels. This can be quite a savings, given that volumes are quite sparse (most of the time, only 5-10% of a volume contains any lighting and shading information for a given set of transfer functions).

2.5 Projection

Ray casting, described in Section 2.4, works from the image space to the object space (volume dataset). Another way of achieving volume rendering is to reconstruct the image from the object space to the image space, by computing

for every element in the dataset its contribution to the image. Several such techniques have been developed [29, 126].

Westover's PhD dissertation [125] describes the *Splatting* technique. In splatting, the final image is generated by computing for each voxel in the volume dataset its contribution to the final image. The algorithm works by virtually "throwing" the voxels onto the image plane. In this process every voxel in the object space leaves a *footprint* in the image space that will represent the object. The computation is processed by virtually "peeling" the object space in slices, and by accumulating the result in the image plane.

Formally the process consists of reconstructing the signal that represents the original object, sampling it and computing the image from the resampled signal. This reconstruction is done in steps, one voxel at a time. For each voxel, the algorithm calculates its contribution to the final image, its footprint, and then it accumulates that footprint in the image plane buffer. The computation can take place in back-to-front or front-to-back order. The footprint is in fact the reconstruction kernel and its computation is key to the accuracy of the algorithm.

Westover [126] proves that the footprint does not depend on the spatial position of voxel itself (for parallel projections), thus he is able to use a lookup table to approximate the footprint. During computation the algorithm just need to multiply the footprint with the color of the voxel, instead of having to perform a more expensive operation.

Although projection methods have been used for both regular and irregular grids, they are more popular for irregular grids. In this case, projection can be sped up by using the graphics hardware (Z-buffer and texture mapping) [108].

We leave a complete discussion of these methods for Chapter 4.

2.6 Summary

Volume rendering is a powerful computer graphics technique for the visualization of large quantities of 3D data, especially well suited for three dimensional scalar and vector fields. It works by modeling the volume as cloud cells composed of semi-transparent material which emits its own light, partially transmits light from other cells and absorbs some incoming light. The most common input data type is a *regular (Cartesian) grid* of *voxels*. Ray tracing is the most popular rendering technique for regular datasets, while projection is a popular method for irregular grids.

Chapter 3

Parallel Rendering of Regular Grids

In this chapter we present our research in developing a high performance parallel volume rendering engine for our Parallel Volume Rendering system (described in Chapter 7). Our research has introduced two contributions to parallel volume rendering: *content-based load balancing* and *pipelined compositing*. Content-based load balancing (Section 3.2) introduces a method to achieve better load balancing in distributed memory MIMD machines. Pipelined compositing (Section 3.3) proposes a new component dataflow for implementing the *Parallel Ray Casting* pipeline.

The major goal of the research presented in this chapter is to develop and implement algorithms for volume rendering extremely large datasets at reasonable speed with an aim on achieving real-time rendering on the next generation of high-performance parallel hardware. The sizes of volumetric

data we are primarily interested in are in the approximate range of 512-by-512-by-512 to 2048-by-2048-by-2048 voxels. Our primary hardware focus is on distributed-memory MIMD machines, such as the Intel Paragon and the Thinking Machines CM-5.

A large number of parallel algorithms for volume rendering have been proposed. Schroeder and Salem [104] have proposed a shear based technique for the CM-2 that could render 128^3 volumes at multiple frames a second, using a low quality filter. The main drawback of their technique is low image quality. Their algorithm had to redistribute and resample the dataset for each view change. Montani et al. [81] developed a distributed memory ray tracer for the nCUBE, that used a hybrid image-based load balancing and context sensitive volume distribution. An interesting feature of their algorithm is the use of clusters to generate higher drawing rates at the expense of data replication. However, their rendering times are well over interactive times. Using a different volume distribution strategy but still a static data distribution, Ma et al. [64] have achieved better frame rates on a CM-5. In their approach the dataset is distributed in a K-d tree fashion and the compositing is done in a tree structure. Others [48, 11, 84] have used similar load balancing schemes using static data distribution, for either image compositing or ray dataflow compositing. Nieh and Levoy [85] have parallelized an efficient volume ray caster [58] and achieved very impressive performance on a shared memory DASH machine.

3.1 Performance Considerations

In analyzing the performance of parallel algorithms, there are many considerations related to the machine limitations, like for instance, communication network latency and throughput [84]. *Latency* can be measured as the time it takes a message to leave the source processor and be received at the destination end. *Throughput* is the amount of data that can be sent on the connection per unit time. These numbers are particularly important for algorithms in distributed memory architectures. They can change the behavior of a given algorithm enough to make it completely impractical.

Throughput is not a big issue for methods based on volume ray casting that perform static data distribution with ray dataflow as most of the communication is amortized over time [81, 48, 11]. On the other hand, methods that perform compositing at the end of the rendering or that have communication scheduled as an implicit synchronization phase have a higher chance of experiencing throughput problems. The reason for this is that communication is scheduled all at the same time, usually exceeding the machines architectural limits. One should try to avoid synchronized phases as much as possible.

Latency is always a major concern, any algorithm that requires communication pays a price for using the network. The start up time for message communication is usually long compared to CPU speeds. For instance, in the iPSC/860 it takes at least 200μs to complete a round trip message between two processors. Latency hiding is an important issue in most algorithms, if an algorithm often blocks waiting for data on other processors to continue its execution, it is very likely this algorithm will perform badly. The classic ways

to hide latency is to use pipelining or pre-fetching [46].

Even though latency and throughput are very important issues in the design and implementation of a parallel algorithm, the most important issue by far is *load balancing*. No parallel algorithm can perform well without a good load balancing scheme.

Again, it is extremely important that the algorithm has as few inherently sequential parts as possible if at all. Amadahl's law [46] shows how speed up depends on the parallelism available in your particular algorithm and that *any*, however small, sequential part will eventually limit the speed up of your algorithm.

Given all the constraints above, it is clear that to obtain good load balancing one wants an algorithm that:

- Needs low throughput and spreads communication well over the course of execution.
- Hides the latency, possibly by pipelining the operations and working on more than one image over time.
- Never causes processors to be idle and/or wait for others without doing *useful work*.

A subtle point in our requirements is in the last phrase, how do we classify *useful work* ? We define useful work as the number of instructions I_{opt} executed by the best sequential algorithm available to volume render a dataset. Thus, when a given parallel implementation uses a suboptimal algorithm, it ends up using a much larger number of instructions than theoretically necessary as

each processor executes more instructions than $\frac{I_{opt}}{P}$ (P denotes the number of processors). Clearly, one needs to compare with the best sequential algorithm as this is the actual speed up the user gets by using the parallel algorithm instead of the sequential one.

The last point on useful work is usually neglected in papers on parallel volume rendering and we believe this is a serious flaw in some previous approaches to the problem. In particular, it is widely known that given a transfer function and some segmentation bounds, the amount of useful information in a volume is only a fraction of its total size. Based on this fact, we can claim that algorithms that use static data distribution based only on spatial considerations are presenting "efficiency" numbers that can be inaccurate, maybe by a large margin.

To avoid the pitfalls of normal static data distribution, we present in the next section a new way to achieve realistic load balancing. Our load balancing scheme, does not scale linearly, but achieves very fast rendering times while minimizing the "work" done by the processors.

3.2 Content-Based Load Balancing

This section explains our new approach to load balancing, which is able to achieve accurate load balancing even when using presence acceleration optimizations. The original idea of our load balancing technique came from the PARC [8] acceleration technique. We notice that the amount of "work" performed by a presence accelerated ray caster is roughly directly proportional to the number of *full supervoxels* contained in the volume.

We use the number of full supervoxels a given processor is assigned as the measure of how much work is performed by that particular processor. Let P denote the number of processors, and c_i the number of full supervoxels processor i has. In order to achieve a good load balancing (by our metric) we need a scheme that *minimizes* the following function for a partition $X = (c_1, c_2, \ldots)$:

$$f(X) = \max_{i \neq j} |c_i - c_j|, \forall i, j \leq P \tag{10}$$

Equation 10 is very general and applies to any partition of the dataset D into disjoint pieces D_i. In our work we have tried to solve this optimization problem in a very restricted context. We have assumed that each D_i is convex. (We show later that this assumption makes it possible to create a *fixed* depth sorting network for the partial rays independently calculated each disjoint region.) Furthermore, we only work with two very simple subdivisions: slabs and a special case of a BSP-tree.

Before we go any further, it is interesting to study the behavior of our load balancing scheme in the very simple case of a slab subdivision of the volume D. Slabs (see Figure 2) are consecutive slices of the dataset aligned on two major axes. Given a volume D, with s *superslices* and p processors with the restriction that each processor gets contiguous slices, the problem of calculating the "best" load balancing partition for p processors consists of enumerating all the $(s-1)(s-2)\ldots(s-p+1)$ ways of partitioning D, and choosing the one that *minimizes* Equation 10.

The problem of computing the optimal (as defined by our heuristic choice)

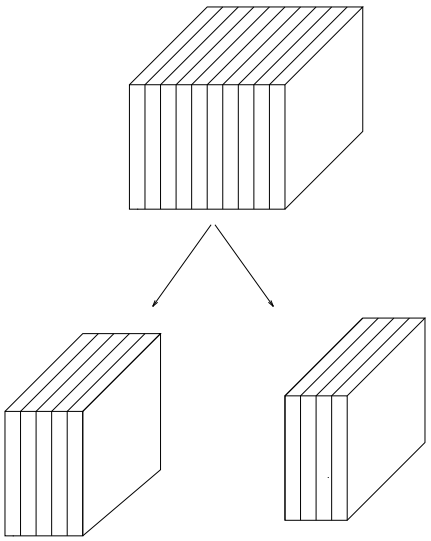


Figure 2: *During slab-based load balancing, each processor gets a range of continuous data set slabs. The number of full supervoxels determines the exact partition ratio.*

load balance partition indices can be solved naively as follows. We can compute all the possible partitions of the integer n, where n is the number of slabs, into P numbers, where P is the number of processors (it is actually a bit different, as we need to consider addition non-associative). For example, if $n = 5$, and $P = 3$, then $1 + 1 + 3$ represents the solution that gives the first slab to the first processor, the second slab to the second processor and the remaining three slabs to the third processor. Enumerating all possible partitioning to get the optimal one is a feasible solution but can be very computationally expensive for large n and P. We use a slightly different algorithm for the computations that follows, we choose the permutation with the smallest square difference from the average.

In order to show how our approach works in practice, let us work out the

example of using our load balancing example to divide the *neghip* dataset (the negative potential of a high-potential iron protein of 66^3 resolution) for four processors. Here we assume the number of superslices to be 16, and the number of supervoxels to be 64 (equivalent to a level 4 PARC decomposition). Using a voxel threshold of 10-200 (out of a range up to 255), we get the following 16 supervoxel count for each slab, out of the 1570 total full supervoxels:

12, 28, 61, 138, 149, 154, 139, 104, 106, 139, 156, 151, 129, 62, 29, 13

A naive approach to load balancing (such as the ones used in other parallel volume renderers) would assign regions of equal volume to each processor resulting in the following partition:

$$12 + 28 + 61 + 138 = 239$$
$$149 + 154 + 139 + 104 = 546$$
$$106 + 139 + 156 + 151 = 552$$
$$129 + 62 + 29 + 13 = 233$$

Here processors 2 and 3 have twice as much "work" as processors 1 and 4. Using our metric, we get:

$$12 + 28 + 61 + 138 + 149 = 388$$
$$154 + 139 + 104 = 397$$
$$106 + 139 + 156 = 401$$
$$151 + 129 + 62 + 29 + 13 = 384$$

One can see that some configurations will yield better load balancing than others but this is a limitation of the particular space subdivision one chooses

to implement, the more complex the subdivision one allows, the better load balancing but the harder it is to implement a suitable load balancing scheme and the associated ray caster. Figure 3 plots the examples just described for the naive approach. Figure 4 shows how well our load balancing scheme works for a broader set of processor arrangements.

Figures 5 and 6 show the rendering times on the Intel Paragon, showing the correlation between the number of supervoxels (or *cubes*) a processor has and the amount of work it has to perform. By comparing these graphs and those in Figures 3 and 4, one can observe that our load balancing is effective, compared to the naive approach of equally subdividing the dataset. If one was calculating a single image, the total rendering time of the image subparts would be the maximum of every processor plus the compositing time.

So far we have shown that our load balancing metric (Equation 10) contains an accurate metric for parallel load balancing for the case of slab subdivisions. Unfortunately, as a domain subdivision scheme, slabs have a few limitations:

- Slabs can have a huge difference in projected screen area due to its unbounded aspect ratio.
- Slabs induce a linear compositing tree, what introduces a large amount of latency (that grows linearly with the number of processors). (Details in Section 3.3).

These shortcomings of slabs led us to an alternative space decomposition structure previously used by Ma et al. [64, 65], the *Binary Space Partition* (BSP) tree, originally introduced by Fuchs et al. [35].

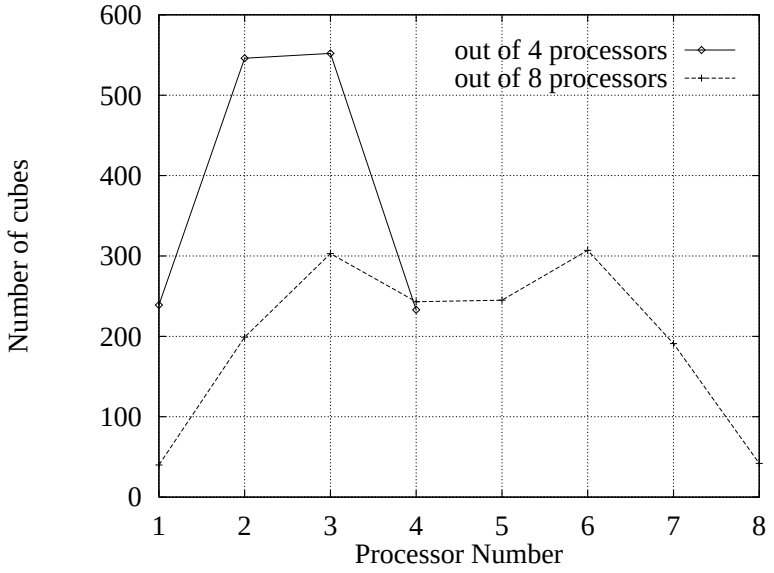


Figure 3: *The graph shows the number of cubes per processor under naive load balancing.*

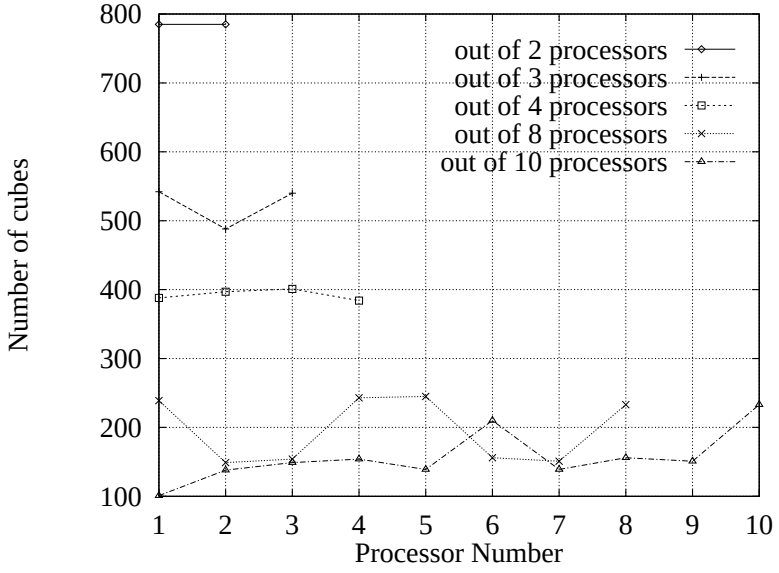


Figure 4: *Load balancing measures for our algorithm. The graph shows the number of cubes the processor receives in our algorithm.*

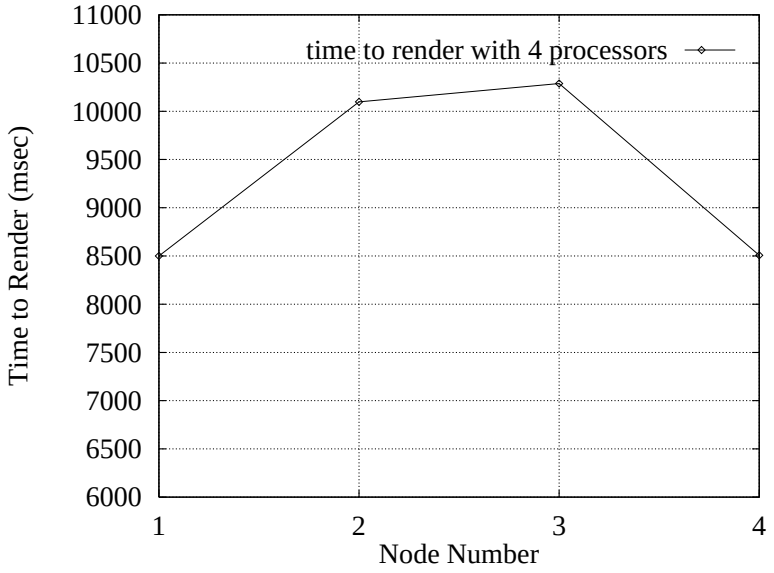


Figure 5: *Naive load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using the naive load balancing.*

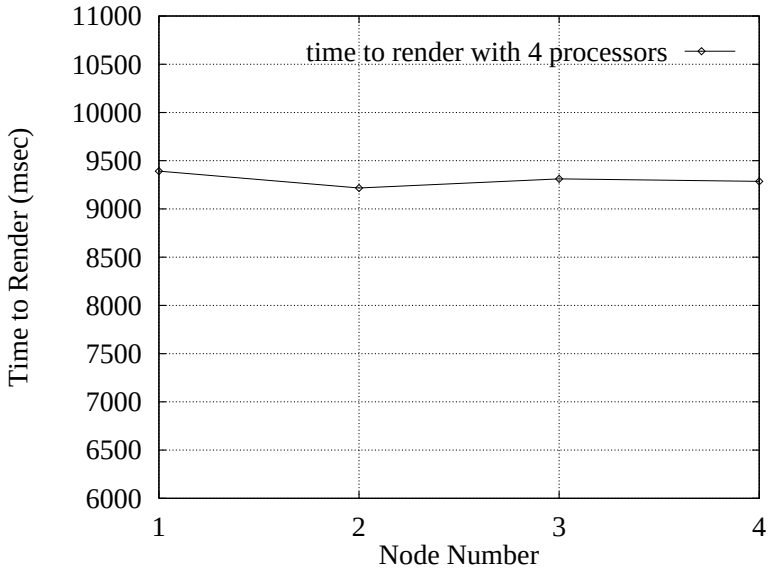


Figure 6: *Our load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using our load balancing.*

Our implementation of BSP trees is quite different from Ma et al. [64, 65], even without considering the load balancing scheme (another major difference is explained in Section 3.3). There, partition cuts are made along each direction, while in our case we perform cuts always along the axis that has the *longest span*. If the cut is always performed on the middle of the longest span axis, it is easy to prove we get a bounded aspect ratio on the the final subdivision (assuming that *enough* cuts are made). In our case, because the load balancing scheme makes the cuts non-uniform (e.g., not in the middle), we can not guarantee a bounded aspect ratio.

Our decomposition method works by performing axis aligned cuts on the rectangular volume dataset, using the following two rules:

- Always generate a cut subdividing the volume along the largest axis. The intuition is to minimize variations of the sizes of the volumes, that is, try to generate partitions as close to cubic as possible. The reason to generate partitions as close to a cube as possible is so that the projection of these partitions from any angle tend to be relatively of the same size, what aids in distributing rendering and compositing load across all the processors. (See Figure 7.)

- Instead of performing a geometric based partition only (like in the middle), always choose the coordinate along the axis of partition that subdivides the (remaining) volume into relatively equal number of full (subsampled) cubes. (See Figure 8.) This guarantees a good load balancing across all the processors.

The cuts are performed recursively m times until we have divided the

original volume D into 2^m regions. Usually m processors will be used for rendering.

We should note here that for the slab case, we were trying all the possible partition combinations in order to optimize the load balancing. This was only possible due to the small number of processors and to the simplicity of the decomposition. In our BSP-tree based optimization, we simply use the recursive decomposition exampled above, not trying to find global minimum, but simply using the binary search strategy. Section 3.4 and Section 3.5 describe implementation and performance issues of our rendering engine in detail.

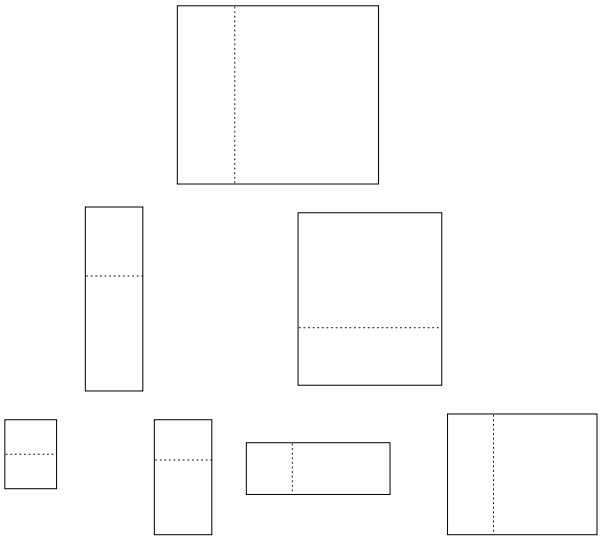


Figure 7: *An example of the partition scheme we used for load balancing. The bottom represents a possible decomposition for 8 nodes. Notice that a cut can be made several times over the same axis to optimize the shape of the decomposition.*

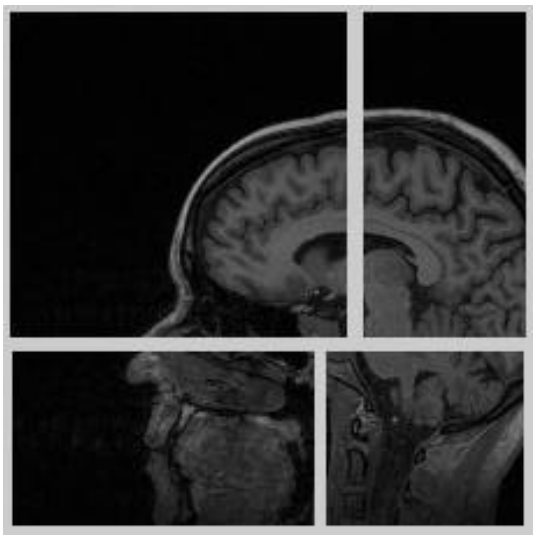


Figure 8: *A cut through the partition accomplished using our load balancing scheme on an MRI head. It is easy to see that if a regular partition scheme were used instead, as the number of processors increase, large number of processors would get just empty voxels to render.*

3.3 The *Parallel Ray Casting* Rendering Pipeline

The *Parallel Ray Casting* rendering pipeline (we are using the same terminology as [31]) is the way the different pieces of a parallel ray caster are put together. In our research we introduce a new volume rendering pipeline, that separates rendering nodes and compositing nodes.

In most other algorithms, the same processors are used for both sampling and shading the volume, as well as for compositing images (in some cases sub-images) without regard to the fact compositing is of a different computation nature than rendering (in much the same way geometric transformations are different from scan-conversion). Compositing two images is a relatively inexpensive operation, but when performed in parallel (because of the dataset

partitioning) it uses a considerable number of messages and is also highly synchronized. Given the fact that messages are slow as compared to processors speed (e.g., a modern CPU can execute hundreds of thousands of instructions for the cost of a single message), it is a waste of processing power to use the same number of rendering processors for compositing , and to have a phase of computation for compositing (like in [64, 65]).

Instead, it is more efficient to overlap compositing and rendering. In our first attempt at this (presented in [110]), we tried to use spare time from the rendering nodes to composite the images. This scheme seems to work well when the number of nodes is small, but it introduces a very high memory overhead (due to buffering of partial results and the linear compositing tree).

An interesting aspect of separating the compositing and rendering nodes, is that one can allocate different number of nodes depending on the expected performance. With all of this in mind, we introduce the idea of having a rendering pipeline composed of clusters. In Montani et al. [81], this idea is proposed in the context of replicating the dataset multiple times and allocating disjoint scan-lines for each cluster. Our cluster scheme is similar but more general. Here, clusters represent a "computational unit of some sort", not necessarily only for rendering, but each cluster represents a phase of the computation being performed. For instance, in a given pipeline configuration, we may have not only rendering and compositing, but also a caching cluster, whose purpose is to buffer copies of the dataset for later viewing, or a computational cluster, where scientific code runs concurrently with the visualization process. An interesting aspect of the PVR system is actually to allow for a flexible and intuitive configuration of the pipeline and cluster configuration.

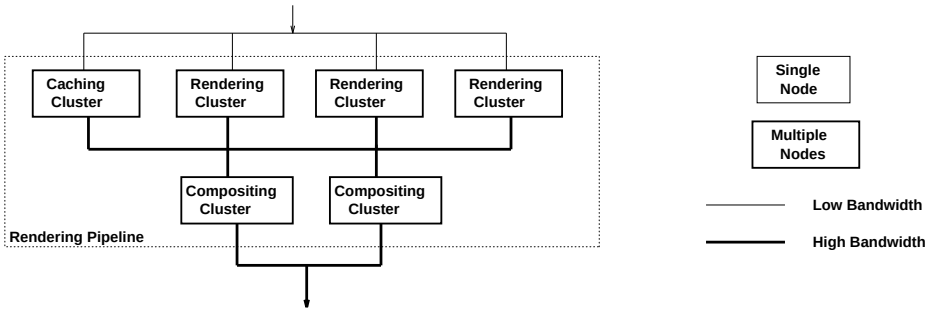


Figure 9: *Representation of the rendering pipeline. Commands come from the top into the respective clusters. Each cluster is considered as a different functional unit of the rendering process and hides its internal structure from the other clusters. Flexible configuration of the nodes into clusters, and their respective functions are provided by the PVR system.*

Figure 9 shows a typical configuration during rendering. Each cluster function needs to be specified for a complete understanding of the rendering process.

Caching Cluster

The caching cluster has the simplest function of all. Its purpose is just to serve as an I/O cache for data that is to be sent for rendering. Its primary use lies in the fact that for computing the necessary information for load balancing (number and location of full supervoxels), it is necessary to have access to the full dataset, and at least two passes of the complete dataset would be needed (one for computing the information, another for sending the data to the corresponding nodes). Because it is very slow to read data of the disks, we choose to "waste" a few nodes as cache only. Once one has put away these nodes, they can be used for other purposes, such as data preparation, or

caching for time-dependent data. This way, the during a large computation, the compute nodes can offload raw data into the caching nodes for preparation for further rendering. This is an useful property in large parallel machines as it is very slow to write data to disk, specially when there is a limited amount of external disk space. Also, this allows for fast data preparation, and change in parameters without increasing the amount of necessary rendering nodes.

An important aspect of the data distribution performed by the caching nodes, is that distributed data has to be moved to each cluster that needs that data in an efficient manner.

Rendering Cluster

The rendering nodes are responsible for sampling and compositing their part of a ray. In order to avoid global communication, each sub-volume region assigned to a rendering node is convex, and actually belongs to a global BSP-tree that makes compositing much simpler (see Figure 10). The data is distributed to each node according to the load balancing criteria described earlier.

Currently, each node has a corresponding screen it computes (a subset of the scanlines can be specified, to lower the computational burden on a given rendering cluster, or on groups thereof – see the Chapter 7 for more details). As each rendering cluster computes its images, it waits for an available compositing cluster to receive its output in the form of n images of the predetermined size. Because rendering clusters may be grouped (as shown later), the output of the images might have to be synchronized across clusters, but we chose a method of synchronization that is local and minimizes the synchronization overhead. The basic idea is to keep "access tokens", that are kept by the

compositing nodes, and passed on (in a lazy fashion) from compositing to the rendering nodes during computation. A rendering node just sends its image down the pipeline if it has a token, otherwise it just waits for the token to arrive. As long as there is enough "compositing capacity" to handle the amount of rendering messages, the rendering nodes never wait.

The structure of the rendering nodes makes each of its nodes completely synchronization and communication free during rendering (amongst themselves). The only requirement of rendering nodes relative to the compositing nodes is that the later accept a set of n messages with n images, every time these are computed. With this in mind, it is possible to abstract the exact number of nodes in each cluster, in other to exploit other types of load balancing. For instance, nothing prevents a set of n rendering nodes, from partitioning the volume into $4n$ pieces, and sending $4n$ images down the pipeline. In such a way, one can incorporate the load balancing of Karia [53] in our system. Memory is the major bottleneck in this kind of organization.

Compositing Cluster

The compositing nodes are responsible for regrouping all the sub-rays back together in a consistent manner, in order to keep image correctness. This is only possible because composition is an associative operation, so if we have to sub-ray samples where one ends where the other starts, it is possible to combine their samples into one sub-ray recursively until we have a value that constitutes the full ray contribution to a pixel.

Ma et al. [65] use a different approach to compositing, where instead of

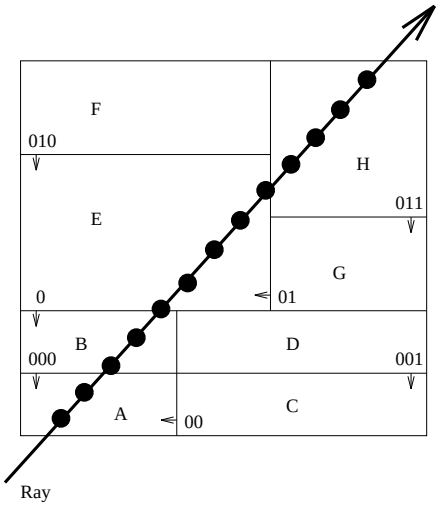


Figure 10: *Data partitioning shown in two dimensions. The dataset is partitioned into 8 pieces (marked A . . . H) in a canonical hierarchical manner by the 7 lines (planes in 3D) represented by binary numbers. Once such a decomposition is performed, it is relatively easy to see how the samples get composited back into a single value.*

having separate compositing nodes, the rendering nodes switch between rendering and compositing. Our method is more efficient (in his latest paper, Ma [66] adopts a similar decomposition of the nodes into two classes) because we can use the special structure of the sub-ray composition to yield a high performance pipeline, where multiple nodes are used to implement the complete pipeline (see Figure 11). Also, the structure of compositing requires synchronized operation (e.g., there is an explicit structure to the composition, that needs to be guaranteed for correctness purposes), and light weight computation, making it much less attractive for parallelization over a large number of processors, specially on machines with slow communication compared to CPU speeds (almost all current machines).

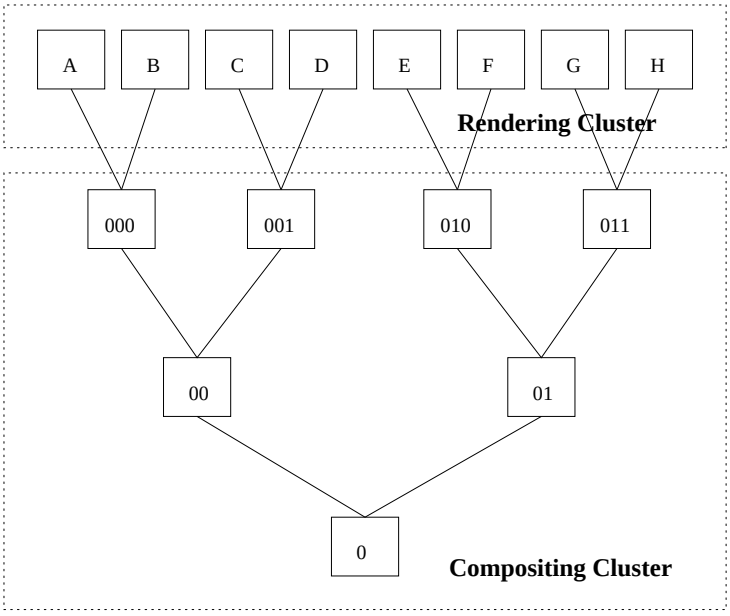


Figure 11: *The internal structure of one compositing cluster, one rendering cluster and their interconnection is shown. In PVR, the communication between the compositing and the rendering clusters is very flexible, with several rendering clusters being able to work together in the same image. This is accomplished by using a set of tokens that are handled by the first level of the compositing tree in order to guarantee consistency. Because of its tree structure, one properly synchronized compositing cluster can work on several images at once, depending on its depth. The compositing cluster shown is relative to the decomposition shown in Figure 10.*

It is easy to see that compositing has a very different structure than rendering. Here, nodes need to synchronize at every step of the computation, making the depth of the compositing tree a hard limit on the speed of the rendering. That is, if one uses 2^m nodes for compositing, and it takes t_c time to composite two images, even without any synchronization or communication factor in, it takes at least mt_c time to get a completely composited image.

Fortunately, most of this latency can be hidden by pipelining the computation. Here, instead of sending one image at a time, we can send images continuously into the compositing cluster, and as long as we send images at a rate lower than one for every t_c worth of time, the compositing cluster is able to composite those at full speed, and after mt_c times, the latency is fully hidden from the computation. As can be seen for our discussion, this latency hiding process is very sensitive to the rate of images coming in the pipeline. One needs to try to avoid "stalls" as much as possible. Also, one can not pipe more than the overall capacity of the pipeline.

Several implications for real-time rendering can be extracted from this simple model. Even though the latency is hidden from the computation, it is not hidden from the user, at least not totally. The main reason is the overall time that an image takes to be computed. Without network overhead, if an image takes t_r time to be rendered by the rendering cluster, the first image of a sequence takes (at least) time $t_r + mt_c$ to be received by the user. Given that people can notice even very small latencies, our latency budget for real-time volume rendering is extremely low and will definitely have to wait for the next generation of machines to be build. We present a detailed account of the timings later in this chapter.

Virtualization

Going back to the previous discussion, we see that as long as t_r is larger than t_c we don't have anything to worry about with respect to creating a bottleneck in the compositing end. As it turns out, t_r is much larger than t_c, even for relatively small datasets. With this in mind, an interesting question is how to allocate the compositing nodes, with respect to size and topology.

The topology is actually fixed by the corresponding BSP-tree, that is, if the first level of the tree has $n = 2^h$ images (if one image per rendering node, than n would be the number of rendering nodes), than potentially the number of compositing nodes required might be as high as $2^h - 1$. There are several reasons not to use that many compositing nodes. First, it is a waste of processors. Seconds, the first-image latency grows with the number of processors in the compositing tree.

Fortunately, we can lower the number of nodes required in the compositing tree by a process known as virtualization. Here, each actual node simulates a group of nodes of the topology tree. There are a couple of interesting caveats in this "simulation". First, the number of image messages every compositing node has to execute increases from two, to some higher power of two, thus potentially increasing the amount of memory required in an exponential fashion. This can be easily solved by performing the compositing computation in a breadth-first fashion, including the distribution of tokens. This increases the latency and synchronization, but dramatically decreases the number of nodes necessary for compositing. Second, the virtualization process is not arbitrary, and nodes have to be allocated in some power of the original number of nodes in the tree.

Types of Parallelism

Due to the fact that each rendering node gets a portion of the dataset, this type of parallelism is called "object-space parallelism". The structure of our rendering pipeline makes it possible to exploit other types of parallelism. For instance, by using more than a single rendering cluster to compute an image, we are making use of "image-space parallelism" (in PVR, it is possible to specify that each cluster compute disjoint scanlines of the same image; see [114] for the issues related to image-space parallelism). The clustering approach coupled with the inherent pipeline parallelism available in the compositing process (because of its recursive structure) gives rise to a third parallelism type, namely "time-space parallelism" or "temporal parallelism". In the latter, we can exploit multiple clusters by concurrently calculating sub-rays for several images at once, that can be sent down the compositing pipeline concurrently. Here, it is important for the correctness of the images, that each composition step be done in lockstep, in order to avoid mixing of images. It should be clear by now that there are several advantages to our separation of nodes into our two types.

3.4 Implementation Issues

We have implemented a parallel volume renderer within the PVR system. The current implementation was developed for the Intel Paragon running SUNMOS (Sandia-University of New Mexico Operating System) and uses a subset of the Intel's NX communication library (it also runs under Intel OSF/1). The code was written to be portable to other architectures and an MPI port is currently

planned. The parallel renderer consists of four main parts, some of which are based on earlier code described in [110].

Communication Structure

SUNMOS implementation of NX, does not allow for interrupt-driven receives (NX `hrecv`), that is, it is not possible to specify functions to be called upon receiving certain types of messages. The first version of our code (originally developed for the Intel iPSC/860) required this capability. In order to avoid this, the implementation is loosely based on RPC calls. Basically, when a node wants to enter a certain protocol exchange with some other node, it sends in an RPC call specifying what function the other node is to perform, the other node "jumps" to the specified function and the communication is performed in lockstep until the end of the function (the synchronization actually can be performed only in the end). This can also be used with global operations.

For instance, the code a compositing node sends to a compositing node to send it a token is:

```
tmp_f = &NodeToken;
csend(PVR_MSG_DISPATCHER, &tmp_f, sizeof(int (*)()), node, 0);
```

At the receiving end, upon receiving a dispatcher message, it just calls the function, which in turn increments the token flag. It processor do not have the same address space, the same behavior is easy to achieve with function tables, but requires the modification of the main function table file every time a function is added to the program.

Memory Requirements

In a distributed-memory environment, memory is a very important resource, because unused memory on one node, can not be used by another node. Furthermore, the machine we have implemented our results on, the Intel Paragon installed at Sandia National Labs in Albuquerque, New Mexico, has only 16MB of memory on the majority of its nodes, and only 512 nodes with 32MB of memory. (The smaller Paragon installed at Stony Brook has 32MB in each node.)

In order to see the limitation, we have to note that the internal representation of every pixel requires 4 floating point numbers, one for each color channel (Red, Green and Blue) and another one for opacity. The major reason for using floating point numbers is the pressing need for speed. With this representation, each 512-by-512 image needs 4MB of memory by itself, not counting the memory necessary for performing the Z-buffer scan-conversion that our PARC-based ray caster needs. In the compositing nodes, who need to save at least two images at a time, the memory requirements for images is over 8MB, plus the communication buffer space necessary to receive asynchronous messages (required by SUNMOS). This means between 12–16MB of image memory is necessary to composite 512-by-512 images.

When saving images to disk, or sending them over the network we use a different representation. The floating point images are transformed into 8-bit per channel images, requiring considerable less space. When transmission over the network is necessary (mostly for interactive viewing purposes), a simple lossless compression is performed (RLE based).

3.5 Performance Analysis

In this section we present a few performance figures of our algorithm that demonstrate our approach is sound and fast. The main points that we are discussing are: the effectiveness of PARC load balancing, the communication overhead of the compositing scheme, and overhead as compared to a sequential implementation. The effectiveness of our PARC load balancing was studied extensively in a previous section, but to complete our choice of using PARC as our ray casting algorithm, it is interesting to compare its advantages to a more naive ray casting approach where no presence accelerations are adopted.

Effectiveness of PARC Optimization

A conventional ray caster where the rays are cast from start to end by calculating intersections with the bounding box of the object is only slightly different from a PARC ray caster. A PARC ray caster actually does more work than a naive one, as it needs to scan convert and to find t_0 and t_1 from the Z buffer. The place that a PARC ray caster really gains performance is in the fact that it better approximates the volume bounds. It should be clear that the higher the cost of the shading function per step, the more advantageous it is to calculate these bounds well. In Figure 12, we can see how a PARC based ray caster performs against a naive ray caster under different shading functions. For our purposes we consider "light" shading a method that uses 5-10 instructions per sample, "medium" a method that uses 50 instructions per sample, and "heavy" shading functions require about 300 instructions per sample. Nieh and Levoy [85] have reported that trilinear interpolating a ray

sample takes 320 instructions. One can see from Figure 12 that not only times but also the rate of increase of cost decreases as one computes more samples.

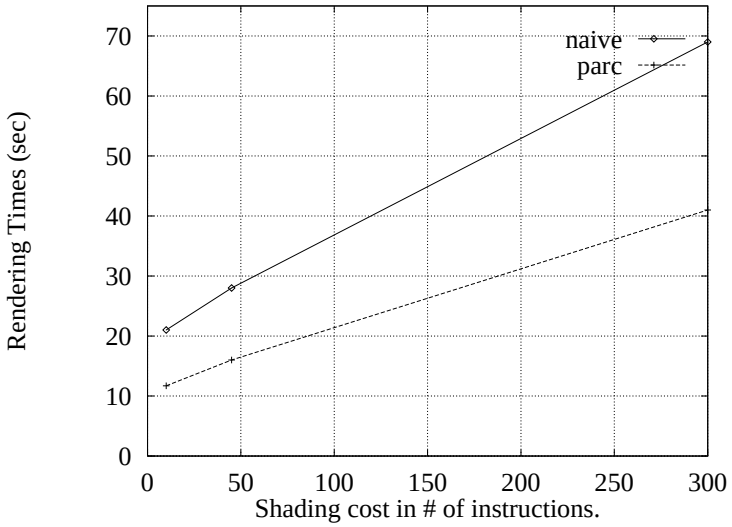


Figure 12: *PARC versus naive ray casting. Times were calculated on a Sparc1000.*

The work performed during rendering each ray can be broken into I_r, the initialization work, and W_r, the work performed to calculate and shade the samples along the ray. If perfect load balancing is achieved for every ray, each processor will perform $\frac{W_r}{P} + I_r$ work per ray, that is, the initialization time is replicated for every ray. If $W_r \gg I_r$, then we can achieve very high scalability with the algorithm, otherwise, as the number of processors increases the amount of work done on the initialization by all the processors PI_r gets larger than W_r, thus limiting the performance. This makes optimization of the initialization time critical to the performance of the algorithm.

Load Balancing Performance and Overall Scalability

Figure 13 shows the rendering times for each frame of a 72-frame animation sequence of the visible human dataset. This is a full 360-degree rotation along the y-axis. The times are wall-clock times calculated at the collector node as it receives the images and saves them to a local disk. Each image is 400-by-400, with three color channels. For rendering, the images are represented as an array of pixels, each of which is represented as four floating point numbers (what amounts to 16 bytes per pixel). At 400-by-400, each image is over 2.5MB. Images are transmitted from the rendering nodes to the compositing nodes, until they reach the root node of the compositing tree. There, images are converted to RGB format, with one byte per color channel and transmitted to the collector node. The final images (with 480,000 bytes) are saved to disk by the collector. Computing the complete animation takes 129.23 seconds, or 1.79 seconds per frame, resulting in 32MB of data being saved to disk. There are noticeable sparks in the image generation rate and these deserve further study. We hypothesize the source of the *stalls* in the pipeline are due to load imbalance and also contention in writing the images to the disk (the collector node stalls the pipeline whenever an image is received before the previous image is saved). One can see the first image takes considerable longer than the others — this is the pipeline initialization cost.

Our next step is to extend the system to render the full RGB visible human (14GB) with high temporal resolution (a 72-frame rotation uses 5 degree increments. Smaller increments are highly desirable, but a 0.5 degree increment would make the animation files huge, at over 300MB). This will require the use of parallel I/O, a capability that currently we do not have, and dedicated use

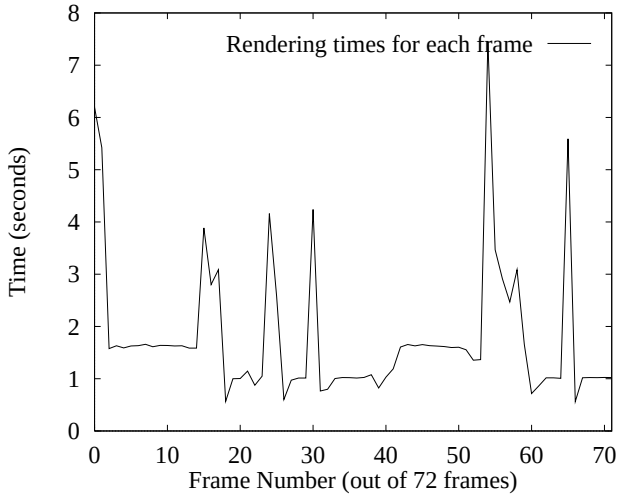
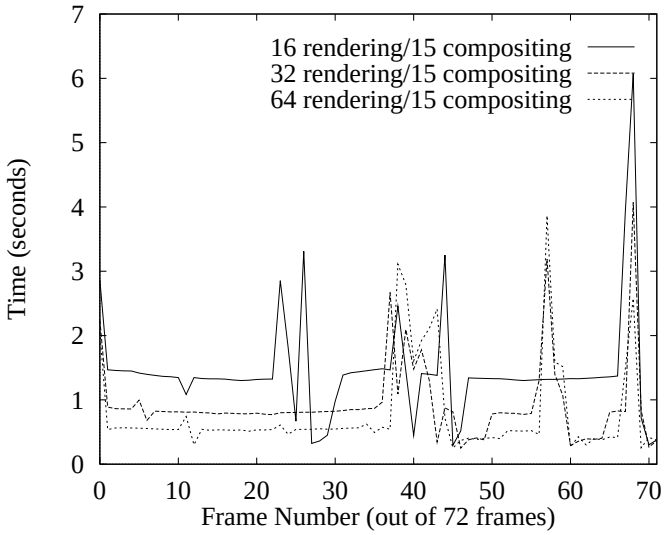


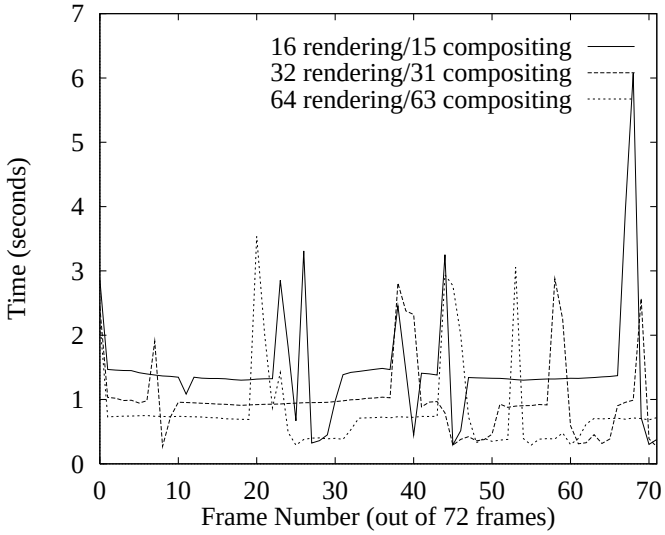
Figure 13: *Rendering times for a 72-frame animation sequence of the 512-by-512-by-1877 visible human dataset. Each image is 400-by-400.*

of a very large parallel machine, such as the entire 1840-node Intel Paragon at Sandia.

In order to show PVR's scalability, we use a 256-by-256-by-937 version of the visible human dataset. Figure 14 shows the rendering times for 5 different configurations, varying the number of rendering and compositing nodes. The five configurations are: 16 rendering nodes (RN) and 15 compositing nodes (CN) (total rendering time – TRT – is 104.10 seconds, or 1.44 seconds per frame); 32 RN (2 clusters of 16) and 15 CN (TRT is 67.24 seconds, or 0.93 seconds per frame); 64 RN (4 clusters of 16) and 15 CN (TRT is 56.73 seconds, or 0.78 seconds per frame); 32 RN (1 cluster) and 31 CN (TRT is 71.42 seconds and 0.99 seconds per frame); 64 RN (1 cluster) and 63 CN (TRT is 58.79 seconds, or 0.81 seconds per frame). A simple conclusion that can be drawn from this data is that it is not cost effective to increase the size of the compositing cluster for relatively small images.



(a) Scaling the number of clusters in a rendering group.



(b) Scaling both the rendering and compositing nodes.

Figure 14: *Rendering times for a 72-frame animation sequence of a 256-by-256-by-937 version of the visible human dataset. Each image is 250-by-250.*

3.6 Conclusions and Future Work

We have shown that using PARC cubes for measuring useful work generates an intuitive way to load balance volume ray casting on distributed memory parallel machines. This not only generates a method that is theoretically sound but its implementation seems to present a method that is both efficient and scalable. We believe our method is simple, fast, uses coherency and achieves high resource utilization on a given machine. As we use PARC, we achieve a high utilization of the compute processors and thus a very fast rendering time on every processor. Because of our pipelined compositing scheme, we achieve a much higher network utilization than other methods.

There are several directions of research to be taken. First of all, in order to eliminate the stalls on the compositing pipeline, one might consider using feedback synchronization techniques, based on previous work by Van Jacobson [49], who designed a set of techniques to avoid congestion in TCP/IP networks. A variation of his *slow-start* and *round-trip-time estimation* technique, where the host slowly sends requests and adaptively changes the rate of requests with the feedback it receives from the network, could be used for that purpose. This can be implemented by having the master keep the number of outstanding image render requests, and setting a maximum on this number based on the number of processors and the amount of memory each has (depending on the *compositing capacity*). At the start of the computation the master begins sending image requests to the processors, and for every image received it sends two requests to the processors until the maximum is achieved. Also the master might keep a running average of the time taken to compute an image, computed

as $T_f = \alpha T_i + (1 - \alpha)M$, where T_f is the new estimate, T_i was the initial estimate, M is the time measured in the last image computation, and α is an amortization constant. By changing α, it is possible to make the master more or less responsive to changes in rendering times. By using this procedure, when this time increases the master can adaptively decrease the rate of requests or increase the rate if the processors begin computing images faster.

Another possible improvement is in the load balancing. The current load balancing works well for some datasets, but in others, sometimes to spread the load, the size of volume allocated to a particular node is larger than the amount of available memory. We currently avoid this problem by calculating sub-optimal partitions. A possible solution would be to incorporate scattered decompositions, where the volume is initially partitioned in sub-pieces, each of which is assigned to nodes depending on the number of full supervoxels it contains.

Finally, because the final bottleneck is always the sequential algorithm rendering speed (running in the rendering clusters), experimenting with other faster renderers, such as Lacroute's shear-warp renderer, might lead to big performance improvements.

Chapter 4

Rendering of Irregular Grids

We propose a fast algorithm for rendering general irregular grids (also in [112]). Our method uses a sweep-plane approach to accelerate ray casting, and can handle disconnected and nonconvex (even with holes) unstructured irregular grids with a rendering cost that decreases as the "disconnectedness" decreases. The algorithm is carefully tailored to exploit spatial coherence even if the image resolution differs substantially from the object space resolution.

In this chapter, we establish the practicality of our method through experimental results based on our implementation, and we also provide theoretical results, both lower and upper bounds, on the complexity of ray casting of irregular grids.

4.1 Introduction

Volume rendering methods are used to visualize scalar and vector fields by modeling a 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 [128, 72]. The most common input data type is a *regular (Cartesian) grid* of *voxels*. Given a general scalar field in $\Re^3$, one can use a regular grid of voxels to represent the field by regularly sampling the function at grid points $(\lambda i, \lambda j, \lambda k)$, for integers i, j, k, and some scale factor $\lambda \in \Re$, thereby creating a regular grid of voxels. However, a serious drawback of this approach arises when the scalar field is *disparate*, having nonuniform resolution with some large regions of space having very little field variation, and other very small regions of space having very high field variation. In such cases, which often arise in computational fluid dynamics and partial differential equation solvers, the use of a regular grid is infeasible since the voxel size must be small enough to model the smallest "features" in the field. Instead, *irregular grids* (or *meshes*), having cells that are not necessarily uniform cubes, have been proposed as an effective means of representing disparate field data.

Irregular grid data comes in several different formats [118]. One very common format has been *curvilinear grids*, which are *structured* grids in computational space that have been "warped" in physical space, while preserving the same topological structure (connectivity) of a regular grid. However, with the introduction of new methods for generating higher quality adaptive meshes, it is becoming increasingly common to consider more general *unstructured* (non-curvilinear) irregular grids, in which there is no implicit connectivity

information. Furthermore, in some applications *disconnected* grids arise.

In this chapter, we present and analyze a new method for rendering general meshes, which include unstructured, possibly disconnected, irregular grids. Our method is based on ray casting and employs a sweep-plane approach, as proposed by Giertsen [38], but introduces several new ideas that permit a faster execution, both in theory and in practice.

Definitions and Terminology

A *polyhedron* is a closed subset of $\Re^3$ whose boundary consists of a finite collection of convex polygons (*2-faces*, or *facets*) whose union is a connected 2-manifold. The *edges* (*1-faces*) and *vertices* (*0-faces*) of a polyhedron are simply the edges and vertices of the polygonal facets. A convex polyhedron is called a *polytope*. A polytope having exactly four vertices (and four triangular facets) is called a *simplex* (*tetrahedron*). A finite set S of polyhedra forms a *mesh* (or an *unstructured, irregular grid*) if the intersection of any two polyhedra from S is either empty, a single common edge, a single common vertex, or a single common facet of the two polyhedra; such a set S is said to form a *cell complex*. The polyhedra of a mesh are referred to as the *cells* (or *3-faces*). If the boundary of a mesh S is also the boundary of the convex hull of S, then S is called a *convex* mesh; otherwise, it is called a *nonconvex* mesh. If the cells are all simplices, then we say that the mesh is *simplicial*.

We are given a mesh S. We let c denote the number of connected components of S. If $c = 1$, we say that the mesh is *connected*; otherwise, the mesh is *disconnected*. We let n denote the total number of edges of all polyhedral cells in the mesh. Then, there are $O(n)$ vertices, edges, facets, and cells.

The image space consists of a screen of N-by-N pixels. We let $\rho_{i,j}$ denote the ray from the eye of the camera through the center of the pixel indexed by (i, j). We let $k_{i,j}$ denote the number of facets of S that are intersected by $\rho_{i,j}$. (Then, the number of cells intersected by $\rho_{i,j}$ is between $k_{i,j}/2$ and $k_{i,j}$.) Finally, we let $k = \sum_{i,j} k_{i,j}$ be the total complexity of all ray casts for the image. We refer to k as the *output complexity*.

Related Work

A simple approach for handling irregular grids is to resample them, thereby creating a regular grid approximation that can be rendered by conventional methods [127]. In order to achieve high accuracy it may be necessary to sample at a very high rate, which not only requires substantial computation time, but may well make the resulting grid far too large for storage and visualization purposes. Several rendering methods have been optimized for the case of curvilinear grids; in particular, Frühauf [34] has developed a method in which rays are "bent" to match the grid deformation. Also, by exploiting the simple structure of curvilinear grids, Mao et al. [68] have shown that these grids can be efficiently resampled with spheres and ellipsoids that can be presorted along the three major directions and used for splatting.

A direct approach to rendering irregular grids is to compute the depth sorting of cells of the mesh along each ray emanating from a screen pixel. For irregular grids, and especially for disconnected grids, this is a nontrivial computation to do efficiently. One can always take a naive approach, and for each of the N^2 rays, compute the $O(n)$ intersections with cell boundary 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.

Garrity [36] has proposed a preprocessing step that identifies the boundary facets of the mesh. When processing a ray as it passes through interior cells of the mesh, connectivity information is used to move from cell to cell in constant time (assuming that cells are convex and of constant complexity). But every time that a ray exits the mesh through a boundary facet, it is necessary to perform a "FirstCell" operation to identify the point at which the ray first reenters the mesh. Garrity does this by using a simple spatial indexing scheme based on laying down a regular grid of voxels (cubes) on top of the space, and recording each facet with each of the voxels that it intersects. By casting a ray in the regular grid, one can search for intersections only among those facets stored with each voxel that is stabbed by the ray.

The FirstCell operation is in fact a "ray shooting query", for which the field of computational geometry provides some data structures: One can either answer queries in time $O(\log n)$, at a cost of $O(n^{4+\epsilon})$ preprocessing and storage [2, 4, 24, 91], or answer queries in worst-case time $O(n^{3/4})$, using a data structure that is essentially linear in n [3, 107]. In terms of worst-case complexity, there are reasons to believe that these tradeoffs between query time and storage space are essentially the best possible. Unfortunately, these algorithms are rather complicated, with high constants, and have not been implemented or shown to be practical. (Certainly, data structures with superlinear storage costs are not practical in volume rendering.) This motivated

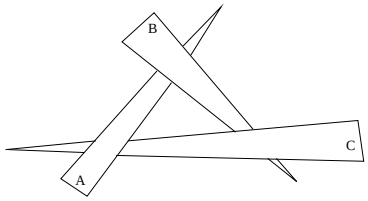


Figure 15: *3 triangles that have no depth ordering.*

Mitchell et al. [77] to devise methods of ray shooting that are "query sensitive" — the worst-case complexity of answering the query depends on a notion of local combinatorial complexity associated with a reasonable estimate of the "difficulty" of the query, so that "easy" queries take provably less time than "hard" queries. Their data structure is based on octrees (as in [98]), but with extra care needed to keep the space complexity low, while achieving the provably good query time.

Uselton [122] proposed the use a Z-buffer to speed up FirstCell; Ramamoorthy and Wilhelms [96] point out that this approach is only effective 95% of the time. They also point out that 35% of the time is spent checking for exit cells and 10% for entry cells. Ma [66] describes a parallelization of Garrity's method. One of the disadvantages of these ray casting approaches is that they do not exploit coherence between nearby rays that may cross the same set of cells.

Another approach for rendering irregular grids is the use of projection ("feed-forward") methods [72, 129, 108, 119], in which the cells are projected onto the screen, one-by-one, in a *visibility ordering*, incrementally accumulating their contributions to the final image. One advantage of these methods is the ability to use graphics hardware to compute the volumetric lighting

models in order to speed up rendering. Another advantage of this approach is that it works in object space, allowing coherence to be exploited directly: By projecting cells onto the image plane, we may end up with large regions of pixels that correspond to rays having the same depth ordering, and this is discovered without explicitly casting these rays. However, in order for the projection to be possible a depth ordering of the cells has to be computed, which is, of course, not always possible; even a set of three triangles can have a cyclic overlap, as shown in Figure 15. Computing and verifying depth orders is possible in $O(n^{4/3+\epsilon})$ time [1, 23, 25]. In case no depth ordering exists, it is an important problem to find a small number of "cuts" that break the objects in such a way that a depth ordering does exist; see [23, 14]. BSP trees have been used to obtain such a decomposition, but can result in a quadratic number of pieces [35, 89]. However, for some important classes of meshes (e.g., rectilinear meshes and Delaunay meshes [30]), it is known that a depth ordering always exists, with respect to any viewpoint. This structure has been exploited by Max et al. [72]. Williams [129] has obtained a linear-time algorithm for visibility ordering convex (connected) acyclic meshes whose union of (convex) cells is itself convex, assuming a visibility ordering exists. Williams also suggests heuristics that can be applied in case there is no visibility ordering or in the case of nonconvex meshes, (e.g., by tetrahedralizing the nonconvexities which, unfortunately, may result in a quadratic number of cells). In [123], techniques are presented where no depth ordering is strictly necessary, and in some cases calculated approximately. Very fast rendering is achieved by using graphics hardware to project the partially sorted faces.

Two important references on rendering irregular grids have not yet been

discussed here — Giertsen [38] and Yagel et al. [132]. We discuss Giertsen's method in the next section. For details on [132], we refer to the reader their paper.

In summary, projection methods are potentially faster than ray casting methods, since they exploit spatial coherence. However, projection methods give inaccurate renderings if there is no visibility ordering, and methods to break cycles are either heuristic in nature or potentially costly in terms of space and time.

Our Contribution

Our new algorithm for rendering irregular grids is based on a sweep-plane approach. Our method is similar to other ray casting methods in that it does not need to *transform* the grid; instead, it uses (as the projection methods) the adjacency information (when available) to determine ordering and to attempt to optimize the rendering. An interesting feature of our algorithm is that its running time and memory requirements are sensitive to the complexity of the rendering task. Furthermore, unlike the method by Giertsen [38], we conduct the ray casting within each "slice" of the sweep plane by a sweep-line method whose accuracy does not depend on the uniformity of feature sizes in the slice. Our method is able to handle the most general types of grids without the explicit transformation and sorting used in other methods, thereby saving memory and computation time while performing an accurate ray casting of the datasets. We establish the practicality of our method through experimental results based on our implementation. We also discuss theoretical lower and upper bounds on the complexity of ray casting in irregular grids.

4.2 Sweep-Plane Approaches

A standard algorithmic paradigm in computational geometry is the "sweep" paradigm [94]. Commonly, a *sweep-line* is swept across the plane, or a *sweep-plane* is swept across 3-space. A data structure, called the *sweep structure* (or *status*), is maintained during the simulation of the continuous sweep, and at certain discrete *events* (e.g., when the sweep-line hits one of a discrete set of points), the sweep structure is updated to reflect the change. The idea is to localize the problem to be solved, solving it within the lower dimensional space of the sweep-line or sweep-plane. By processing the problem according to the systematic sweeping of the space, sweep algorithms are able to exploit spatial coherence in the data.

Giertsen's Method

Giertsen's pioneering work [38] was the first step in optimizing ray casting by making use of coherency in order to speed up rendering. He performs a sweep of the mesh in 3-space, using a sweep-plane that is parallel to the x-z plane. Here, the viewing coordinate system is such that the viewing plane is the x-y plane, and the viewing direction is the z direction; see Figure 16. The algorithm consists of:

1. Transform all vertices of S to the viewing coordinate system.
2. Sort the (transformed) vertices of S by their y-coordinates; put the ordered vertices, as well as the y-coordinates of the scanlines for the viewing image, into an event priority queue, implemented in this case as an array, A.

3. Initialize the *Active Tetrahedra List* (ATL) to empty. The ATL represents the sweep status; it maintains a list of the tetrahedra currently intersected by the sweep-plane.

4. While A is not empty, do:

   (a). Pop the event queue: If the event corresponds to a vertex, v, then go to (b); otherwise, go to (c).

   (b). Update ATL: Insert/delete, as appropriate, the tetrahedra incident on v. (Giertsen assumed that the tetrahedra are disjoint, so each v belongs to a single tetrahedron.)

   (c). Render current scanline: Giertsen uses a memory hash buffer, based on a regular grid of squares in the sweep-plane, allowing a straightforward casting of the rays that lie on the current scanline.

By sweeping 3-space, Giertsen reduces the ray casting problem in 3-space to a 2-dimensional cell sorting problem.

Giertsen's method has several advantages over previous ray casting schemes. First, there is no need to maintain connectivity information between cells of the mesh. (In fact, he assumes the tetrahedral cells are all disjoint.) Second, the computationally expensive ray shooting in 3-space is replaced by a simple walk through regular grid cells in a plane. Finally, the method is able to take advantage of coherence from one scanline to the next.

However, there are some drawbacks to the method, including: (1) The original data is being coarsed into a finite resolution buffer (the memory hashing buffer) for rendering, basically limiting the resolution of the rendering, and possibly creating an aliasing effect. Also, his memory scheme cannot be easily

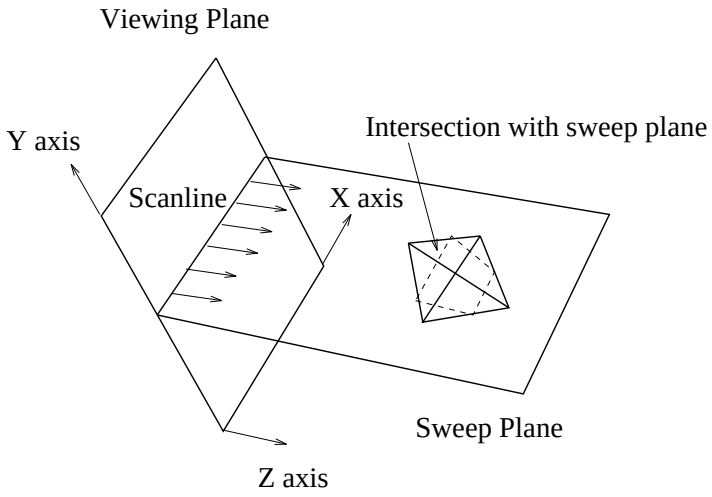


Figure 16: *A sweep-plane (perpendicular to the y-axis) used in sweeping 3-space.*

resolved by increasing the resolution of the buffer, as irregular grids have cell size variation of the order from 1:100,000, making it impractical to have a large enough buffer. In his paper, Giertsen mentions that when cells get mapped to the same location, this is considered a degenerate case, and the later cells are ignored; however, this form of collision resolution might lead to temporal aliasing when calculating multiple images. (2) Another disadvantage when comparing to other ray casting techniques is the need to have two copies of the dataset, as the transformation and sorting of the cells must be done before the sweeping can be started. (Note that this is also a feature of cell projection methods.) One cannot just keep re-transforming a single copy, since floating point errors could accumulate.

Yagel's Method

In [132], Yagel et al propose a method that uses a sweep-plane *parallel* to the viewing plane. Their method consists of slicing the grid, finding the collection of primitives in the intersection, and using the graphics hardware to scan-convert the current plane, that is then composited with the previous result. Basically, their method uses a sweep-plane to achieve what the projection method does.

Yagel's method can handle general polyhedral grids, and it seems to need a high amount of extra information besides the minimum necessary to keep the adjacency lists. Conceptually it can generate high quality images, but the shading is limited to those offered by the hardware. The simplicity of the method makes it very attractive. Using a RealityEngine^2 machine, Yagel and his co-workers achieved very impressive performance. One drawback of the method seems to be the amount of memory necessary for rendering, but this might just be a side effect of a very efficiency oriented implementation.

4.3 Our Algorithm

The design of our new method is based on two main goals: (1) the depth ordering of the cells should be correct along the rays corresponding to every pixel; and (2) the algorithm should be as efficient as possible, taking advantage of structure and coherence in the data.

With the first goal in mind, we chose to explore ray casting algorithms, as they have an inherent advantage for handling cycles among cells, a case causing difficulties for projection methods. To address the second goal, we use a sweep

approach, as did Giertsen, in order to exploit both *inter-scanline* and *inter-ray* coherence. Our algorithm has the following advantages over Giertsen's: (1) It avoids the explicit transformation and sorting phase, thereby avoiding the storage of an extra copy of the vertices; (2) It makes no requirements or assumptions about the level of connectivity or convexity among cells of the mesh; however, it does take advantage of structure in the mesh, running faster in cases that involve meshes having convex cells and convex components; (3) It avoids the use of a hash buffer plane, thereby allowing accurate rendering even for meshes whose cells greatly vary in size; (4) It is able to handle parallel and perspective projection within the same framework (e.g, no explicit transformations).

4.3.1 Performing the Sweep

Our sweep method, like Giertsen's, sweeps space with a sweep-plane that is orthogonal to the viewing plane (the x-y plane), and parallel to the scanlines (i.e., parallel to the x-z plane).

Events occur when the sweep-plane hits vertices of the mesh S. But, rather than sorting all of the vertices of S in advance, and placing them into an auxiliary data structure (thereby at least doubling the storage requirements), we maintain an event queue (priority queue) of an appropriate subset of the mesh vertices.

A vertex v is *locally extremal* (or simply *extremal*, for short) if all of the edges incident to it lie in the (closed) halfspace above or below it (in y-coordinate). A simple (linear-time) pass through the data readily identifies the extremal vertices.

We initialize the event queue with the extremal vertices, prioritized according to the magnitude of their inner product (dot product) with the vector representing the y-axis ("up") in the viewing coordinate system (i.e., according to their y-coordinates). We do *not* explicitly transform coordinates. Furthermore, at any given instant, the event queue only stores the set of extremal vertices not yet swept over, plus the vertices that are the upper endpoints of the edges currently intersected by the sweep-plane. In practice, the event queue is relatively small, usually accounting for a very small percentage of the total data size. As the sweep takes place, new vertices (non-extremal ones) will be inserted into and deleted from the event queue each time the sweep-plane hits a vertex of S.

The sweep algorithm proceeds in the usual way, processing events as they occur, as determined by the event queue and by the scanlines. We pop the event queue, obtaining the next vertex, v, to be hit, and we check whether or not the sweep-plane encounters v before it reaches the y-coordinate of the next scanline. If it does hit v first, we perform the appropriate insertions/deletions on the event queue; these are easily determined by checking the signs of the dot products of edge vectors out of v with the vector representing the y-axis. Otherwise, the sweep-plane has encountered a scanline. And at this point, we stop the sweep and drop into a two-dimensional ray casting procedure (also based on a sweep), as described below. The algorithm terminates once the last scanline is encountered.

We remark here that, instead of doing a sort (in y) of all vertices of S at once, the algorithm is able to take advantage of the partial order information that is encoded in the mesh data structure. (In particular, if each edge is

oriented in the $+y$ direction, the resulting directed graph is acyclic, defining a partial ordering of the vertices.) Further, by doing the sorting "on the fly", using the event queue, our algorithm can be run in a "lock step" mode that avoids having to sort and sweep over highly complex subdomains of the mesh. This is especially useful, as we see in the next section, if the slices that correspond to our actual scanlines are relatively simple, or the image resolution (pixel size) is large in comparison with some of the features of the dataset. (Such cases arise, for example, in some applications of scientific visualization on highly disparate datasets.)

4.3.2 Processing a Scanline

When the sweep-plane encounters a scanline, the current sweep status data structure gives us a "slice" through the mesh in which we must solve a two-dimensional ray casting problem. Let $\mathcal{S}$ denote the polygonal (planar) subdivision at the current scanline (i.e., $\mathcal{S}$ is the subdivision obtained by intersecting the sweep-plane with the mesh S.) In time linear in the size of $\mathcal{S}$, we can recover the subdivision $\mathcal{S}$ (both its geometry and its topology), just by stepping through the sweep status structure, and utilizing the local topology of the cells in the slice.

The two-dimensional problem is also solved using a sweep algorithm — now we sweep the plane with a sweep-line parallel to the z axis. Events now correspond to vertices of the planar subdivision $\mathcal{S}$. At the time that we construct $\mathcal{S}$, we identify those vertices in the slice that are locally extremal in $\mathcal{S}$ (i.e., those vertices that have edges only leftward in x or rightward incident on them.) These are inserted in the initial event queue. The *sweep-line status* is

an ordered list, stored and maintained in a binary tree, of the edges of $\mathcal{S}$ crossed by the sweep-line. The sweep-line status is initially empty. Then, as we pass the sweep-line over $\mathcal{S}$, we update the sweep-line status and the event queue at each event when the sweep-line hits an extremal vertex, making insertions and deletions in the standard way. This is analogous to the Bentley-Ottmann sweep that is used for computing line segment intersections in the plane [94]. We also stop the sweep at each of the x-coordinates that correspond to the rays that we are casting (i.e., at the pixel coordinates along the current scanline), and output to the rendering model the sorted ordering (depth ordering) given by the current sweep-line status (binary tree).

4.4 Analysis: Upper and Lower Bounds

We proceed now to give a theoretical analysis of the time required to render irregular grids. We begin with "negative" results that establish lower bounds on the worst-case running time:

Theorem 1 (Lower Bounds) *Let S be a mesh having c connected components and n edges. Even if all cells of S are convex, $\Omega(k + n \log n)$ is a lower bound on the worst-case complexity of ray casting. If all cells of S are convex* and, *for each connected component of S, the union of cells in the component is convex, then $\Omega(k + c \log c)$ is a lower bound. Here, k is the total number of facets crossed by all N^2 rays that are cast through the mesh (one per pixel of the image plane).*

Proof. It is clear that $\Omega(k)$ is a lower bound, since k is the size of the output from the ray casting.

Let us start with the case of c convex components in the mesh S, each made up of a set of convex cells. Assume that one of the rays to be traced lies exactly along the z-axis. In fact, we can assume that there is only one pixel, at the origin, in the image plane. Then the only ray to be cast is the one along the z-axis, and k simply measures how many cells it intersects. To show a lower bound of $\Omega(c \log c)$, we simply note that any ray tracing algorithm that outputs the intersected cells, in order, along a ray can be used to sort c numbers, z_i. (Just construct, in $O(c)$ time, tiny disjoint tetrahedral cells, one centered on each z_i.)

Now consider the case of a *connected* mesh S, all of whose cells are convex. We assume that all local connectivity of the cells of S is part of the input mesh data structure. (The claim of the theorem is that, even with all of this information, we still must effectively perform a sort.) Again, we claim that casting a single ray along the z-axis will require that we effectively sort n numbers, $z_1, \ldots, z_n$. We take the unsorted numbers z_i and construct a mesh S as follows. Take a unit cube centered on the origin and subtract from it a cylinder, centered on the z-axis, with cross sectional shape a regular $2n$-gon, having radius less than $1/2$. Now remove the half of this polyhedral solid that lies above the x-z plane. We now have a polyhedron P of genus 0 that we have constructed in time $O(n)$. We refer to the n (skinny) rectangular facets that bound the concavity as the "walls". Now, for each point $(0, 0, z_i)$, create a thin "wedge" that contains $(0, 0, z_i)$ (and no other point $(0, 0, z_j)$, $j \neq i$), such that the wedge is attached to wall i (and touches no other wall). Refer to Figure 17. We now have a polyhedron P, still of genus 0, of size $O(n)$, and this polyhedron is easily decomposed in $O(n)$ time into $O(n)$ convex polytopes.

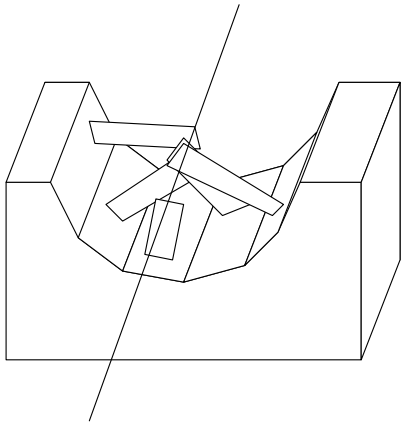


Figure 17: *Lower bound construction.*

Further, the z-axis intersects (pierces) all n of the wedges, and does so in the order given by the sorted order of the z_i's. Thus, the output of a ray tracing algorithm that has one ray along the z-axis must give us the sorted order of the n wedges, and hence of the n numbers z_i. The $\Omega(n \log n)$ bound follows. □

Upper Bounds

The previous theorem establishes lower bounds that show that, in the worst case, any ray casting method will have complexity that is superlinear in the problem size — essentially, it is forced to do some sorting. However, the pathological situations in the lower bound constructions are unlikely to arise in practice.

We now examine upper bounds for the running time of the sweep algorithm we have proposed, and we discuss how its complexity can be written in terms of other parameters that capture problem instance complexity.

First, we give a worst-case upper bound. In sweeping 3-space, we have $O(n)$ vertex events, plus N "events" when we stop the sweep and process the 2-dimensional slice corresponding to a scanline. Each operation (insertion/deletion) on the priority queue requires time $O(\log M)$, where M is the maximum size of the event queue. In the worst case, M can be of the order of n, so we get a worst-case total of $O(N + n \log n)$ time to perform the sweep of 3-space.

For each scanline slice, we must perform a sweep as well, on the subdivision $\mathcal{S}$, which has worst-case size $O(n)$. The events in this sweep algorithm include the $O(n)$ vertices of the subdivision (which are intersections of the slice plane with the edges of the mesh, S), as well as the N "events" when we stop the sweep-line at discrete pixel values of x, in order to output the ordering (of size $k_{i,j}$ for the ith pixel in the jth scanline) along the sweep-line, and pass it to the rendering module. Thus, in the worst case, this sweep of 2-space, for each scanline slice, requires overall time $O(\sum_{i,j} k_{i,j} + Nn \log n) = O(k + Nn \log n)$. Overall, then, we get $O(k + Nn \log n)$.[1]

Now, the product term, Nn, in the bound of $O(k = Nn \log n)$ is due to the fact that each of the N slices might have complexity roughly n. However, this is a pessimistic bound for practical situations. Instead, we can let n_s denote the total sum of the complexities of all N slices; in practice, we expect n_s to be much smaller than Nn, and potentially n_s is considerably smaller than n. (For example, if the mesh is uniform, we may expect each slice to have complexity of $n^{2/3}$, as in the case of a $n^{1/3}$-by-$n^{1/3}$-by-$n^{1/3}$ grid, which gives

[1] The upper bound of $O(k+Nn \log n)$ should be contrasted with the bound $O(N^2 n \log n)$ obtained from the most naive method of ray casting, which computes the intersections of all N^2 rays with all $O(n)$ facets, and then sorts the intersections along each ray.

rise to $n_s = O(Nn^{2/3})$.) If we now write the complexity in terms of n_s, we get worst-case running time of $O(k + n \log n + n_s \log n)$.

Theorem 2 (Upper Bound) *Ray casting for an irregular grid having n edges can be performed in time $O(k + n \log n + n_s \log n)$, where $k = O(N^2 n)$ is the size of the output (the total number of facets crossed by all cast rays), and $n_s = O(Nn)$ is the total complexity of all slices.*

Note that, in the worst case, $k = \Omega(N^2 n)$; e.g., it may be that every one of the N^2 rays crosses $\Omega(n)$ of the facets in the mesh. Thus, the output size k could end up being the dominant term in the complexity of our algorithm. Note too that, even in the best case, $k = \Omega(N^2)$, since there are N^2 rays.

The $O(n \log n)$ term in the upper bound comes from the sweep of 3-space, where, in the worst case, we may be forced to (effectively) sort the $O(n)$ vertices (via $O(n)$ insertions/deletions in the event queue).

Consider the sweep of 3-space with the sweep-plane. We say that vertex v is *critical* if, in a small neighborhood of v, the number of connected components in the slice changes as the sweep-plane passes through v. (Thus, vertices that are locally min or max are critical, but also some "saddle" points may be critical.) Let n_c denote the number of critical vertices. Now, if we conduct our sweep of 3-space carefully, then we can get away with only having to sort the critical vertices, resulting in total time $O(n + n_s + n_c \log n_c)$ for constructing all N of the slices. The main idea is to exploit the topological coherence between slices, noting that the number of connected components changes only at critical vertices (and their y-coordinates are sorted, along with the N scanlines). In particular, we can use depth-first search to construct each connected

component of $\mathcal{S}$ within each slice, given a starting "seed" point in each component. These seed points are obtained from the seed points of the previous slice, simply by walking vertically ($+y$ direction) from one seed to the next slice (in total time $O(n)$, for all walks); changes only occur at critical vertices, and these are local to these points, so they can be processed in time linear in the degree of the critical vertices (again, overall $O(n)$). This sweep of 3-space gives us the slices, each of which can then be processed as already described. (Note that the extremal vertices within each slice can be discovered during the construction of the slice, and these are the only vertices that need to be sorted and put into the initial event queue for the sweep of a slice.)

Another potential savings, particularly if the image resolution is low compared with the mesh resolution, is to "jump" from one slice to the next, *without* using the sweep to discover how one slice evolves into the next. We can instead construct the next slice from scratch, using a depth-first search through the mesh, and using "seed" points that are found by intersecting the new slice plane with a critical subgraph of mesh edges that connects the critical vertices of the mesh. Of course, we do not know a priori if it is better to sweep from slice i to slice $i+1$, or to construct slice $i+1$ from scratch. Thus, we can perform a "lock step" algorithm (doing steps in alternation, between the two methods), to achieve asymptotically a complexity that is the minimum of the two. This scheme applies not just to the sweep in 3-space, but also to the sweeps in each slice.

As an illustration of how these methods can be quite useful, consider the situation in Figure 18, which, while drawn only in 2 dimensions, can depict the cases in 3-space as well. When we sweep from line 2 to line 3, a huge complexity

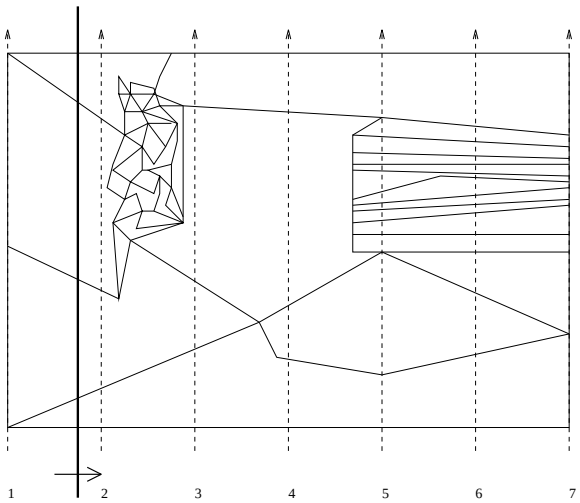


Figure 18: *Illustration of a sweep in one slice.*

must be swept over, and this may be costly compared to rebuilding from the scratch the slice along line 3. On the other hand, sweeping from line 5 to line 6 is quite cheap (essentially no change in the geometry and topology), while constructing the slice along line 6 from scratch would be quite costly. By performing the two methods in lock step (possibly in parallel, if a second processor is available), we can take advantage of the best of both methods. The resulting algorithm exploits coherence in the data and has a running time that is sensitive, in some sense, to the complexity of the visualization task.

4.5 Experimental Results

We have implemented the main algorithm described in the previous sections. Our implementation handles general disconnected grids, and has most of the advantages of the complete algorithm described already, but we have not yet

implemented the "lock step" idea (used to avoid worst-case complexity in disparate data sets), and our code does not currently handle perspective projections. (The implementation of *perspective projection* will be done soon and is conceptually very simple, requiring only that the priority values in the queue be based on an appropriate dot product.) Further, in our initial implementation, we have assumed that cells of the mesh are tetrahedra (simplices). Our method does not require convex cells, even though they do make some of the implementation issues simpler.

The rendering algorithm consists of about 5,000 lines of C code. It is fairly naive in terms of optimization, so we expect that it can be further improved. An interesting aspect of the code is the way it cleanly handles geometric degeneracies. The major modules of the program include: *3D sweep*, which sweeps the vertices of the input mesh along a given direction, while maintaining two dynamic sweep status data structures — the active tetrahedra list (ATL) and the active edge list (AEL); *2D sweep*, which orders the 3D edge intersections, and is complemented with the code that incrementally depth sorts the segments along the current ray. We also have a graphics module that sets up the transformations and manages the other modules, and the transfer function and the optical integration (or simple shading) modules. We do not attempt to describe the implementation in detail, but we shall explain some of the most relevant issues.

Due to the large sizes of irregular grids, efficient data structures can substantially influence the performance of the implementation. For priority queues (we use two of them, one for the incremental 3D sweep sort, another for the 2D sweep), we use a simple *heap* implementation (the same code is shared for

3D/2D). Instead of performing the view-dependent $O(n)$ search for extremal vertices, we simply preprocess the external vertices of the grid and place them in the heap before starting the sweep (all the internal vertices are still inserted incrementally – see Figure 19 – in order to avoid the need for substantial extra storage). In order to keep the ATL and AEL, we need a dictionary data structure that allows efficient insertion/deletion. We have experimented using a hash table and binary trees. The hash tables performed much better than the binary trees in our examples, because of lower overhead, both in time and space.

During the 2D sweep, a binary tree stores the sweep status. Edges are inserted in depth order, and for rendering at the pixel locations, the binary tree is sent to the shader. The handling of the binary tree is tricky, since a consistent ordering of all the segments along each ray must be maintained as edges are inserted and deleted during the sweep. Due to degeneracies, geometric tests alone are not sufficient to keep a consistent ordering; edges may have the same *geometrical* properties, but *topologically* they are different, which causes inconsistencies in the tree – for instance, an edge might be inserted along a certain binary tree path when its first endpoint is reached, but might not be found in the tree when its second endpoint is reached due to the insertion of another edge along that path, resulting in an inconsistent sweep-status state. This problem is solved by assigning a computational ordering, that is, explicitly using an ordering function that depends on the memory position of the edges (which are fixed for each scanplane), to break geometric "ties", therefore forcing a globally consistent ordering among edges.

Another place where degeneracies have to be avoided is during the final

rendering. The problem arises because several vertices may lie on the same plane. This leads to intersections that may have non-closed and/or null primitives (e.g., a triangle with two coincident sides). The solution is to keep track of the current status of the priority queue and only perform the rendering once all the events with values lower than or equal to the current y-value (or x-value when in ray casting) have been processed. This solution is conceptually simple, correct and easy to implement.

Geomview, from the Geometry Center of the University of Minnesota, was instrumental in the development of our renderer, helping to create animations and visually debug our code. Without visual debugging it would have been virtually impossible to write this code.

Datasets

The code currently handles datasets composed of tetrahedral grids. The input format is analogous to the Geomview "off" file. It simply has the number of vertices and tetrahedra, followed by a list of the vertices and a list of the tetrahedra, each of which is specified using the vertex locations in the file as an index. This format is compact, can handle general grids (including disconnected), and it is fairly simple (and fast) to recover topological information. Maintaining explicit topological information in the input file would waste too much space.

For our test runs we have used tetrahedralized versions of the well-known Blunt Fin and Liquid Oxygen Post datasets, originally in NASA Plot3D format. The Blunt Fin contains 40-by-32-by-32 data points (40,960 vertices),

from which we create 187,395 tetrahedra by breaking each cell into 5 tetrahedra. Figure 21 depicts the decomposition used, and Figure 22 shows a running configuration of the algorithm. The Post dataset contains 38-by-76-by-38 data points (109,744 vertices) and 513,375 tetrahedra after conversion. We have generated several other artificial datasets for debugging purposes; in particular, we generated simple datasets that have disconnected components.

Memory Requirements

Our algorithm is very memory efficient. The dataset is stored as a collection of vertices and tetrahedra. Each tetrahedron only stores indices to its vertices, and a single flag that identifies the external faces (no topological information is saved at the tetrahedra). Each vertex contains, besides its position and scalar value, a flag, used during the algorithm for various purposes, and a list of the tetrahedra it belongs to. Because each tetrahedron contains four vertices, the overall increase in memory cost for the topological information is minimal.

Besides the input dataset, the only other memory consumption is in the priority queues, which are very small in practice. (For the Blunt Fin, the extra storage is below half a megabyte.) This low storage requirement is due to our incremental computations, which only touch a cross section of the dataset at a time. The overall memory consumption for rendering the Blunt Fin is about 8MB of memory total, of which over 95% is the dataset itself (about 36% is topology information). For the Post dataset, the storage requirement is a bit over 21MB, of which 97% is the dataset itself (about 35% is topology information).

Performance Analysis

Our primary system for measurements was a Sun UltraSparc-1. We present numbers for the tetrahedralized version of the Blunt Fin and Post datasets, described above. It is important to notice, that our rendering times will clearly be higher than algorithms that treat the either dataset as a curvilinear grid composed of hexahedral cells.

Reading the Blunt Fin dataset off a local disk takes 9.8 secs (seconds) on the UlraSparc. The Post dataset takes 27.32 secs. Our ASCII input files require parsing; thus, processing time dominates, not the actual disk access time. (Our tetrahedralized Blunt Fin version has almost 6MB, and the Post has over 16MB.) The use of binary files would likely improve efficiency, but using ASCII files simplifies the manual creation of test samples. In a preprocessing phase, we recover the adjacency information of the grid, and separate the external vertices into a list (for the Blunt Fin, we classify 6,760 vertices as externals, for the Post, 13,840 vertices). The complete preprocessing takes 2.95 secs for the Blunt Fin, and 8.48 secs for the Post.

Rendering can be decomposed into several stages: 3D sweep, 2D sweep and 1D ray casting (including shading). All of them are embedded inside each other. The complexity of the 3D sweep is independent of the image size; it just depends on how many points need to be processed. For instance, without performing any rendering, just sweeping (the 3D sweep) the Blunt Fin grid takes about 3.5 secs. During this time, the ATL and AEL are being updated at every event (the binary tree implementation takes over three times as long). The AEL is used during the 2D sweep for calculating and ordering the intersections for the final ray casting (see Figures 19 and 20).

Since the Blunt Fin projection is not square, it is not meaningful to give performance numbers on a square screen. Instead we give numbers for a 527-by-200 screen (105,400 pixels), which matches the aspect ratio of the Blunt Fin for the direction in which we are looking. Figure 19 contains the number of active edges for each scanline. It is easy to see that, because the structure of the grid is irregular, the number of edges varies quite a bit. For the Post, we used a 300-by-300 screen (90,000 pixels).

Rendering a scanline involves computing the intersection points, sorting them along the direction of the scanline, and then performing a 1D sweep (or sort) along each ray incrementally (which basically involves processing events and shading). Figure 20 shows the rendering times for the 2D sweep, for each scanline. The performance numbers indicate: the time to process a given scanline is directly correlated to the number of active edges on that plane; the cost per scanline varies depending on the complexity of the plane being rendered; (and most important for future optimization) the event handling time dominates the total time spent per scanline.

The event handling time is clearly the bottleneck of the rendering speed. This was puzzling at first, specially because it is just performing a sweep of a few thousand vertices (less than 5,500). In the 3D sweep, we handle over 40,000 vertices in about 3.5 secs. Profiling the code showed that "CompareEdge" (a function that tells which of two edges is closer to the screen) is called over 68 million times, consuming over 40% of the *overall* rendering time. Further study shows that the reason for such a high number of calls to "CompareEdge" is related to the depth of the binary tree used to save the ordering. Because the Blunt Fin comes from a curvilinear grid, it has lots of vertices that lie

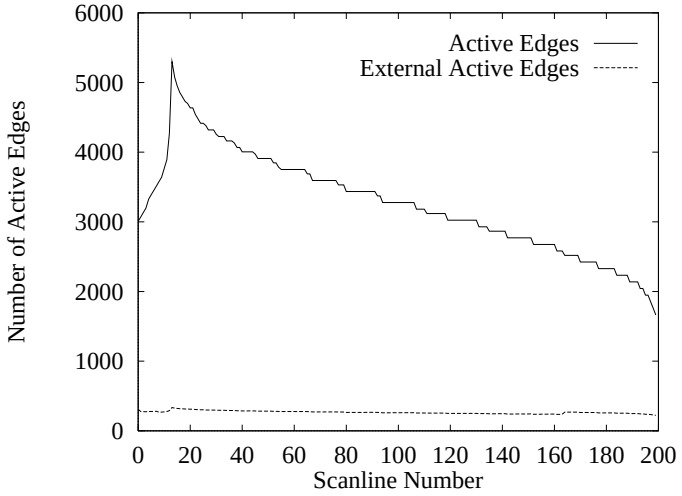


Figure 19: *Number of active edges as a function of the scanline: the active edges are the edges intersected at a given scanline by the current scanplane during the 3D sweep. The number of external active edges is also shown.*

(degenerately) on common planes, which causes extremely bad behavior in our binary tree sorting. This indicates that we can potentially obtain a dramatic improvement in performance, just by changing the data structure used (e.g., by employing a standard 2-3 tree or a Red-Black tree [18]). Another reason the 2D sweep is taking so long is the fact that there is a scanline component on its rendering time. As discussed later, the most time consuming parts of it can be eliminated by making incremental changes to the depth sorting on the segments.

Performance Comparisons

The total rendering time of our algorithm is 70 secs for a 190,000 tetrahedra cell complex (the Blunt Fin), for a 527-by-200 image with almost complete pixel coverage (see Figure 23 – the picture was actually padded with a black frame

after rendering for printing purposes). For the Blunt Fin, the performance of our current code is 373μs (microseconds) per tetrahedron, and 664μs per pixel. For the Post, a 500,000 tetrahedra cell complex, it takes 145 seconds (see Figure 24) to render a 300-by-300 image.

The most recent report on an irregular grid ray caster is Ma [66], from October 1995. Ma is using an Intel Paragon (with superscalar 50MHz Intel i860XPs). He reports rendering times for two datasets, an artificially generated *Cube* dataset with 130,000 tetrahedra and a *Flow* dataset with 45,500 tetrahedra. He does not report times for single CPU runs, always starting with two nodes. With two nodes, for the *Cube*, he reports taking 2,415 secs (2234 secs for the ray casting – the rest is parallel overhead) for a 480-by-480 image, for a total cost of 10.5 (9.69) ms (milliseconds) per pixel. The cost per tetrahedron is 18.5 (17.18) ms. For the *Flow* dataset he reports 1593 (1,585) ms (same image size), for a cost of 6.9 (6.8) ms per pixel, and 35.01 (34.8) ms per tetrahedron. All his performance numbers reflect the use of 2 processors. Giertsen [38] reports running times of 38 secs for 3,681 cells (10.32 ms per cell). His dataset is too small (and too uniform) to allow meaningful comparisons, nevertheless our implementation handles a cell complex that has over 100 times the number of cells he used, at a fraction of the cost per cell. Yagel et al. [132] reports rendering the Blunt Fin, using an SGI with a Reality Engine^2 in just over 9 secs, using a total of 21MB of RAM, using 50 "slicing" planes; with 100 planes, he reports the cost increases to 13–17 secs. (Their rendering time is dependent on the number of "slicing" planes, which, of course, affects the accuracy of the picture generated.) For a 50 slice-rendering of the Post, it takes just over 20 secs, using about 57MB RAM.

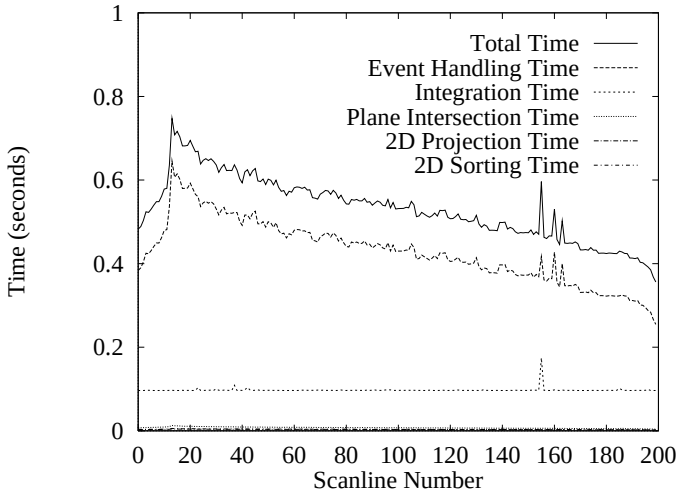


Figure 20: *Total rendering time as a function of the scanline: the intersection time is the time it takes to calculate the 2D point of intersection of the active edges with the scanplane; the event handling time is the time to process every active edge after sorting, including the 1D ray sorting necessary for integration. Note that event handling time dominates the total cost of a scanline.*

Optimizations

There are at least a couple of directions for optimization of the current code that may make it even more competitive. First, improvement in the data structures for keeping the sorted rays should lower the cost of using "CompareEdge". Second, at this time, we are starting the 2D sorting process over for every scanplane, not using the previously sorted information.

4.6 Conclusions and Future Work

In this chapter, we propose a new algorithm for rendering irregular grids. Our algorithm is carefully tailored to exploit spatial coherence even if the

image resolution differs substantially from the object space resolution. We have also discussed some of the theoretical upper and lower bounds on ray casting approaches.

We have reported timing results showing that our method compares favorably with other ray casting schemes, and is, in fact, a couple orders of magnitude faster than other published ray casting results. Another advantage of our method is the fact that it is very memory efficient, making it suitable for use with very large datasets.

It is difficult to compare our method with hardware-based techniques (e.g., [132]), which can yield impressive speed-ups over purely software-based algorithms. On the other hand, software-based solutions broaden the range of machines on which the code can run (e.g., much of our code was developed on a small laptop, with only 16MB of RAM). Also, we are optimistic that implementation of the optimizations suggested in the last section will further improve the performance of our software. More experimentation should help us quantify exactly how our algorithm compares with other methods.

An interesting possible extension of our work would be to investigate issues involving out-of-core operation. The spatial locality of our memory accesses indicates that we should be able to employ *pre-fetching* techniques to achieve fast rendering when the irregular grids are much larger than memory. Also, our method is a natural candidate for parallelization.

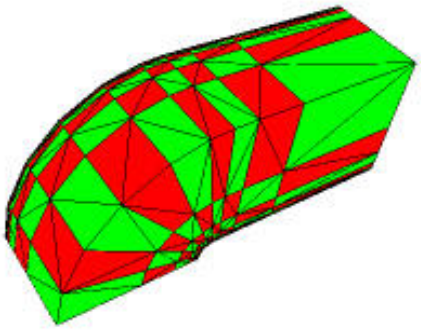


Figure 21: *Outside faces of a lower resolution version of the Blunt Fin are shown to demonstrate the tetrahedralization process. Red and green cells have to be tetrahedralized in opposite direction to allow for correct matching between cells.*

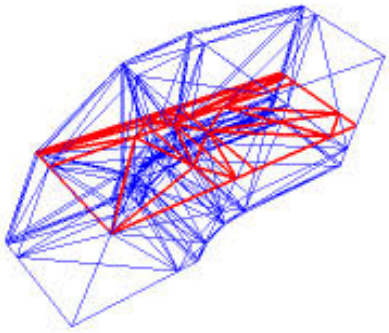


Figure 22: *A typical configuration during the sweep is shown in red. (A lower resolution version of the Blunt Fin is used to avoid excessive cluttering.)*

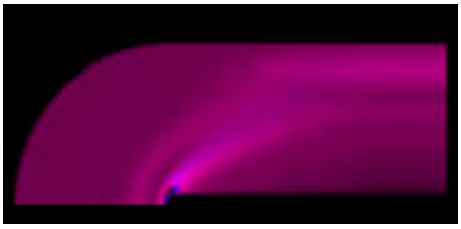


Figure 23: *A volume rendering of the Blunt Fin dataset generated with our method.*

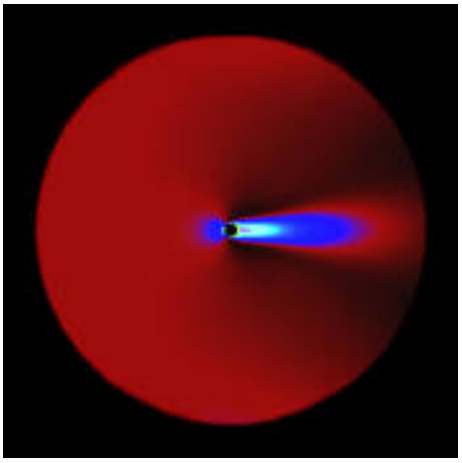


Figure 24: *A volume rendering of the Liquid Oxygen Post dataset generated with our method.*

Chapter 5

Parallel Rendering of Irregular Grids

In this chapter we present a distributed-memory MIMD machine parallelization of the method proposed in Chapter 4. Our parallelization is a *true* distributed-memory parallelization, in the sense that each rendering node gets only a portion of the dataset.

5.1 Introduction

Because irregular grids tend to be very large and they lack the simple geometric coherency that regular grids have, there is a natural push for parallel methods. The largest irregular grids currently being renderered are just breaking the 1,000,000 cell barrier. If only data sample points are taken into account, this is equivalent to a 100-by-100-by-100 regular grid. Memorywise, such a grid requires more than 50MB of memory, while its regular counterpart only needs

1MB. This is due in part to the fact that in irregular grids there is a need to store several floating point values per cell, instead of a single 8-bit index value. Regular grids of this size can be rendered in interactive time using off-the-shelf components (i.e., using Lacroute's Shear-Warp technique [55]), while the irregular grids of this size take a considerable amount of time to render. (Using accurate ray casting techniques it would take over an hour to render such a dataset.)

The sizes of irregular grids of interest of computational scientists are much larger than one million cells. Using our efficient sweep-based ray casting, it takes about 150 seconds to render a 500,000-cell complex. Assuming a linear scaling behavior of our rendering algorithm (what is not quite correct) it would takes over 10 minutes to generate images of a 2,000,000 cell complex. Even though these numbers are quite reasonable, given the previous techniques (Ma [66] needed over 40 minutes to render a dataset with only 130,000 cells), it is still far from interactive.

Our goal is to develop a method that is both faster and scalable to larger dataset. We believe the next generation of supercomputers, such as the ACSI TeraFlop machine to be installed at Sandia National Labs later in the year, will make it possible to generate much larger datasets. For instance, the ACSI machine will have at least an order of magnitude more memory than the current Intel Paragon installed at Sandia and even more usable memory (i.e., not taking OS overhead into account). This will enable the generation of very large grids, possibly in ranges of 10,000,000-100,000,000 cells or larger.

Clearly, with improvements in rendering algorithms, we should be able to steadily improve the rendering times. But in order to offset such a large jump

in dataset size, a similar jump is necessary for the rendering machines. The other main reason for the use of parallel machines comes from the size of the datasets. The largest workstations available to us have 1GB–3GB of memory, what is very small when compared to the 300GB of memory expected in the ASCI machine. Yet another reason for parallel rendering is to avoid moving large quantities of data in and out of the parallel machines. Network and disk bandwidth are the biggest bottlenecks for moving such large quantities of data.

With all the points raised above in mind, we present our algorithm for rendering irregular grid data in place, directly on the distributed-memory MIMD machine.

5.2 Previous Work

Work on rendering irregular grid data on distributed memory architectures has been quite sparse. This area has received relatively little attention. This might be due to the fact that rendering irregular grids is much harder than regular grids. Also, efficient irregular grid rendering packages are not publically available.

Uselton has parallelized his original ray tracing work [122] in a shared memory multiprocessor SGI, and reported that the implementation scales linearly up to 8 processors. Challinger [13] reports on a parallel algorithm for irregular grids, implemented on a shared-memory BBN-2000 Butterfly. Giertsen [37] has also parallelized his sweep algorithm on a collection of IBM RS/6000, using a master/slave scheme and total data replication in the nodes. His reported scalability results are for just a handful of nodes (less than ten).

Ma [66] proposed a parallelization technique similar to ours. It is unfortunate that he used a sequential ray casting technique that is shown to be at least two orders of magnitude slower than the one we use. Because of this, he was not able to find any interesting bottlenecks of the parallelization technique.

His technique works by breaking up the original grid into multiple, disjoint cell complexes using Chaco [45], a graph-based decomposition tool developed at Sandia National Labs. Chaco-based decompositions have several interesting and important properties for parallelization of computational methods. In general, Chaco decomposes the graphs into equal sized partitions using some simple metrics. For instance, one can potentially use Chaco to break a graph into equal-volume pieces. However, Ma did not make use of advanced Chaco features. Also, it is unclear the extra overhead of using Chaco has actually any influence on the rendering speed of the parallelization.

For the overall rendering organization, Ma uses a similar (but simpler and less general — he has no notion of clusters, etc.) rendering pipeline to the one presented in Chapter 3. He divides the nodes into two classes: rendering and compositing nodes. The rendering nodes, essentially use Garrity's method [36] to compute each ray of an image, but due to the fact that rays might be partitioned among nodes, the rendering nodes output a set of stencils instead of a final color. After each ray is computed, they are sent to the compositing nodes for further sorting and the final accumulation. Each compositing node is assigned a set of rays to be composited. Ma reports that because the rendering is so slow as compared to compositing, there was no need for any further optimizations of the parallel algorithm.

5.3 Algorithm Overview

Our algorithm is similar to Ma [66]. We keep the same framework introduced in Chapter 3 and used in our PVR system. This allows a considerable amount of code re-use, specially in the cluster and pipeline implementation. Also, this provides a framework for integrating regular and irregular grid rendering into the same pipeline.

Dataset Decomposition

In order to subdivide the dataset among the nodes, we use a hierarchical decomposition method, with a similar flavor to our load balancing scheme for regular grids. Starting with the bounding box of the complete cell complex, we make cuts in this box, taking two things into account: the aspect ratio of the cuts, and the number of vertices. At every step, we cut along the largest axis in such a way as to break the number of vertices in half in each stage of the cutting. Because cells might belong to more than one of these convex space decomposition (or box, for short), we assign the cell to the box that has the biggest percentage of it (e.g., in the number of vertices, with ties broken in some arbitrary, but consistent way).

The basic rendering operation is to sample and shade the stencils. With the dataset decomposed into these boxes, the natural thing to do is assign each box to a different processor. Because we expect each box will have the same number of primitives, this should minimize the total rendering time for the complete irregular grid, and optimize the overall load balancing.

This decomposition method is not optimized for incremental rendering

computations (where only parts of the dataset need to be sampled and only rough pictures are generated). We plan to explore this kind of rendering in the future.

In our decomposition each processor should have roughly the same number of primitives, each of which, approximately confined to a rectangular grid of almost bounded aspect ratio (because of the largest-axis cutting).

Rendering

The rendering performed at each node is a variation of our sweep-based ray casting technique presented in Chapter 4. There is a single significant difference: Instead of generating an image, every node generates a *stencil* data-structure during rendering. All nodes work concurrently on generating complete stencil scan-lines before sending them for final compositing.

The stencil representation of a scan-line is a linked-list of color and depth of cells, which have been lazily composited as follows: If two stencils share an end point — $(\vec{a}, \vec{b})$ and $(\vec{b}, \vec{c})$ — they are composited into a single stencil $(\vec{a}, \vec{c})$ representing the whole region. Each stencil contains information about its position, color and opacity. In the end of a scan-line rendering computation, each node potentially has a collection of stencils. Because of the way we created our cell decomposing among the nodes, it is expected that the stencil fragmentation is minimal. This is necessary in order to minimize communication. The more connected the stencils are (in number of pieces), less communication is needed to transmit to the compositing nodes.

Compositing

In Ma [66], compositing nodes are responsible to complete scan-lines. This minimizes the amount of communication during compositing (there is actually no communication with exception of the final image assembly).

Our approach is different. We try to achieve better performance by creating a tree of compositing nodes (such as the one we use for the parallel rendering of regular grids in Chapter 3). Every compositing node is responsible for a certain region of space (i.e., one of the original box decompositions proposed above) that belongs to a global BSP-tree.

It is the responsibility of the rendering nodes to respect the BSP-tree boundaries and send the data to the correct compositing nodes, possibly breaking stencils that span across boundaries.

Once the data of each scan-line is received in the compositing nodes, the final depth sorting can be efficiently performed by merging the stencils into a complete image. An efficient pipeline scheme can be implemented on a scan-line by scan-line basis, with similar properties to the one implemented (with an image at a time approach) for the regular grid case.

5.4 Summary

In this chapter, we have presented a parallel method for rendering irregular grids algorithm based on our previous work (presented in Chapters 3 and 4). We apply several of the ideas used in those chapters in order to achieve a simple, practical and efficient parallel rendering method for irregular grids.

An interesting extension would be to change the current regular grid implementation to generate stencil data in order to integrate regular and irregular grids into a single pipeline. A harder problem is to allow for overlapping volumes (both regular and irregular), in this case it would be necessary to synchronize all the rendering nodes for every scan-line as to guarantee that the compositing has taken all the data into account.

Chapter 6

Simplification

In this chapter we discuss our work on the simplification of irregular grids. A major portion of the chapter deals with a new method for the automatic generation of triangular irregular networks (TINs) from dense terrain models using the same greedy principle used to compute minimum-link paths in polygons. The algorithm works by taking greedy cuts ("bites") out of a simple closed polygon that bounds the yet-to-be triangulated region. The algorithm starts with a large polygon, bounding the whole extent of the terrain to be triangulated, and works its way inward, performing at each step one of three basic operations: ear cutting, greedy biting, and edge splitting. We give experimental evidence that our method is competitive with current algorithms and has the potential to be faster and to generate many fewer triangles. Also, it is able to keep the structural terrain fidelity at almost no extra cost in running time and it requires very little memory beyond that for the input height array.

In the end of the chapter, we present algorithmic extensions of the our

method that can be used to simplify surfaces of general topological type. Extensions to three-dimensional irregular grids are left for future work.

6.1 Introduction

A *terrain* is the graph of a function of two variables. The function gives the *elevation* of each point in the domain. Terrain models are widely used in visualization and computer graphics applications, such as flight simulators, financial visualization tools, strategic military analyzers, geographic information systems, and video games. Thus, it is of the utmost importance that primitive operations can be performed in real-time. Several factors may affect the efficiency of algorithms that operate on terrain: The most important are probably the *size* of the input and its underlying data structure.

The most common source of digital terrain elevation data is the DEM (*Digital Elevation Model*), supplied by the U.S. Geological Survey. A DEM is a two-dimensional floating point height array. It contains an extremely high level of redundancy, which usually forbids real-time applications from using it. Several alternative data structures have been proposed, including contour lines, quad-trees, and TINs. TINs stand out as being one of the most convenient to use for rendering and for geometric manipulation operations. A TIN is a set of contiguous non-overlapping triangles whose vertices are placed adaptively over the DEM domain [32]. The automatic generation of TIN models from DEM models is an important area of research and is the main topic of this article. Several factors are important in judging the quality of the TIN representation of a given DEM (list partially adapted from [100, 102]):

- *Numerical accuracy* – measured as maximum, mean, or standard deviation error;
- *Visual accuracy* – usually assessed by inspection and by number of "slivery" triangles;
- *Size of the model* – measured as the number of output triangles;
- *Algorithm complexity* – measured in terms of the time to generate the TIN and the memory requirement.

Fowler and Little [32] have introduced one of the first (and still very popular) methods to address the problem of automatic generation of TINs directly from DEMs. Their method is very simple. First, they classify the points by automatically choosing some "important" features of the terrain, such as ridges and peaks. They describe this phase of the algorithm as constructing the "structural fidelity" of the model (i.e., the TIN representation should have the same geographical features as the DEM). Then, they incrementally compute a triangulation of the points; in their case, they chose to use the Delaunay triangulation. At each step a new point is added to the triangulation until no points are farther from the original surface than a certain predefined threshold. This phase is designed to preserve the "statistical fidelity" (i.e, to make it fit the specified error bound).

Franklin [33] has proposed a similar approach in 1973. It appears that his method had no notion of structural fidelity, and he did not use the Delaunay triangulation as the basis for his method. A new version of his code is publically available, and we compared it against our method. A detailed

description of his algorithm and code is given in Section 6.4. Recently, substantial research has been conducted on creating hierarchical structures on top of TINs [27, 103], and on techniques to improve the quality of TIN meshes [101]. Scarlatos' dissertation [100] gives a good survey of terrain modeling and representation. A very recent approach to building hierarchical models of terrains is given by de Berg and Dobrindt [26], who apply a hierarchical refinement of the Delaunay triangulation to represent terrain TINs at many levels of detail. [56, 57] describe the "drop heuristic" and provides comparisons with other methods. Common to all these methods is the need to have a complete starting triangulation that is either *refined* by adding new points, or *decimated* [106] by removing redundant points. These approaches require that the algorithm maintain in memory a complete triangulation representation of the input, extended with various pieces of global information (e.g., most deviant point per triangle). The need for *global* information impacts the running time and memory requirements of these algorithms.

Our work is based on an entirely different approach for the triangulation and simplification of the data. It is based on an idea in the method developed by Mitchell and Suri [79], where a greedy set cover approach has been developed for approximating convex surfaces, and used recently by Varshney [124] in heuristics for simplifying CAD models. We can consider the input DEM to be an instance of a TIN with very high resolution. In particular, each pixel of the DEM corresponds to four elevation data points, and we consider these to define two adjacent triangles of a surface. (A square pixel can be triangulated in one of two ways. We triangulate all pixels uniformly, with diagonals at 45-degrees.) Our goal is to simplify this input TIN surface to create a new TIN

that has far fewer triangles, but is still within a specified error bound of the original surface. From an algorithmic point of view, terrain simplification is hard (NP-hard) [22, 21], but some polynomial-time algorithms are known for computing a nearly-optimal (i.e., nearly minimum-facet) approximating surface, guaranteed to be within a factor $O(\log n)$ of optimal (see [5, 16, 76, 79]), or within a constant factor of optimal, if the surface is convex (see [10]). Unfortunately, the polynomial-time bounds for these theoretically good approaches is rather high (at least cubic). In contrast, from the practical point of view, most of the previous computer graphics and geography research in the area is based on heuristics for generating triangulations that "fit" the original data, but have no guarantees, either in terms of the closeness to optimal or in terms of the worst-case running time.

The principle that drives our method (and is related to that of [16, 79, 124]) is the same greedy principle that is used to compute minimum-link paths in simple polygons. This problem is well studied in computational geometry [44, 78, 120] and can be used to find an optimal piecewise-linear approximation to a function of a single variable (see [41]). Our problem is of one higher dimension. We use a *greedy-facet* approach, selecting large triangles (bites) by which to extend an approximating surface, based on their feasibility (i.e., they must lie within an ϵ-fattening of the original surface) and on their size (e.g., area of projection in the x-y plane). The use of greedy algorithms is known to give provably good approximation results in many combinatorial optimization problems, for example, the *set cover* problem is approximated within a log factor of optimal by a natural greedy algorithm, and this fact leads [79] to a provably good approximation algorithm for the convex case

of our problem. We have not yet been able to prove that our algorithm has a guaranteed effectiveness with respect to optimal, but we are hopeful that interesting properties can be proved about its performance. Currently, our code only handles inputs in the form of elevation arrays, but in principle, there is no reason why it cannot be extended to arbitrary polyhedral terrains, or, for that matter, polyhedral surfaces in general. Extensions to higher dimensions also seem possible, that is, for simplifying piecewise-linear functions of three variables defined over tetrahedralizations of 3-space.

Instead of a top-down approach that starts with a feasible Delaunay triangulation and tries to generate finer and finer Delaunay triangulations by adding points to the already created triangulations, our algorithm works bottom-up. At each step a greedy cut is taken from an untriangulated polygon. The greedy cuts are an attempt to sample the data at the lowest possible resolution, thus minimizing the number of triangles in the output. A full description of our algorithm is given in the next section.

6.2 The *Greedy-Cuts* Algorithm (Terrain Case)

This section gives a high-level description of our algorithm. The problem definition is as follows:

Given an input array, H, of heights $H(x, y)$, $0 \leq x < m$ and $0 \leq y < n$, whose data points are sampled from a regular grid on a rectangle R, and some $\epsilon > 0$ specifying an error tolerance. Find a triangulated surface that represents a terrain on R, such that it has a small number of triangles (T_i), and each data point given by the array $H(x, y)$ lies within vertical distance ϵ of the TIN.

The algorithm maintains a list of *untriangulated simple polygons*, $\mathcal{P}$, which represents the portion of R over which no triangulated surface has yet been constructed. At each step, our goal is to select a maximum area triangle T within one of the polygons $P \in \mathcal{P}$, such that (1) the vertices $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2)$, and $v_3 = (x_3, y_3)$ of T are grid points (points (x, y) for which we have the altitude $H(x, y)$); (2) at least two of these vertices are vertices of P (i.e., T shares at least one edge with P); and (3) the triangle T corresponds to a triangle T' in space (with coordinates $(x_1, y_1, H(x_1, y_1))$, $(x_2, y_2, H(x_2, y_2))$, $(x_3, y_3, H(x_3, y_3))$) such that T' is "feasible" with respect to ϵ (see below for a precise definition). Because input data is sampled using a regular grid, the area of T is a good estimation of its combinatorial coverage (how many data points it covers). The ideal version of our algorithm searches all candidate triangles T and picks the best at each stage. However, for the sake of efficiency, the implemented version of our algorithm does not search all possible triangles T; instead, we do an approximate (limited) search for the best T, based on three basic operations, which will be described below.

Since each polygon $P \in \mathcal{P}$ corresponds to an independent subproblem, we can work on each separately. (There is no particular ordering in how we store the polygons $P \in \mathcal{P}$.) Thus, at each step of the algorithm, a *bite* (triangle) T is taken out of the polygon P at the head of the list $\mathcal{P}$, until P is reduced to a single feasible triangle, or it is divided into two new simple polygons, each of which is inserted into the list. The final result of our algorithm is the list of all triangles (bites), $\mathcal{T}$. There is no need to store in memory the list $\mathcal{T}$ of triangles as it is generated. Each triangle can be written out directly to a file. No triangle connectivity information is saved at this point. Each polygon

$P \in \mathcal{P}$ is saved as a simple list of vertices, in counter-clockwise order. Thus, only very small and simple data structures are required.

We ought to define precisely what we mean by a triangle (in space) being "feasible" for input terrain H, with respect to a given ϵ. As already mentioned, we can consider the input DEM H to be an instance of a TIN (a polyhedral surface, S), even though no triangulation is explicitly given. Specifically, to fix that one of the many triangulations we consider to be the input surface, we consider point $(x, y, H(x, y))$ to have six neighbors, namely, those data points corresponding to $(x \pm 1, y \pm 1)$ (the standard four grid neighbors) and the diagonal points $(x + 1, y + 1)$ and $(x - 1, y - 1)$.

We say that a triangle T' (in space) satisfies *weak feasibility with respect to* ϵ if, for every grid point (x, y) that lies within the projection T of T' onto the (x, y)-plane, T' intersects the vertical segment joining $(x, y, H(x, y) - \epsilon)$ and $(x, y, H(x, y) + \epsilon)$. In other words, T' fits the function at the relevant internal grid points. Note that if T' has a very "skinny" or "small" projection (e.g., so that T contains no grid points at all), then it will certainly satisfy weak feasibility.

We say that triangle T' (in space) satisfies *strong feasibility with respect to* ϵ if T' lies completely above the surface $S^{-\epsilon}$ and completely below the surface $S^{+\epsilon}$, where $S^{-\epsilon}$ (resp., $S^{+\epsilon}$) is the polyhedral surface (TIN) obtained by shifting S downwards (resp., upwards) by ϵ. Note that if T' satisfies strong feasibility, then it certainly satisfies weak feasibility (but the converse is clearly false). The notion of strong feasibility applies directly to approximating arbitrary input terrains (e.g., given by a TIN rather than a DEM).

In order to test weak feasibility of T', we only have to examine the elevations at grid points internal to the projected triangle T. Such internal grid points are identified using a standard scan conversion of T. In Figure 25, we indicate these grid points with small squares. Strong feasibility, however, requires that we also check the altitudes corresponding to those points (indicated with circles in Figure 25) that lie at the intersections of an edge of T with a grid edge.

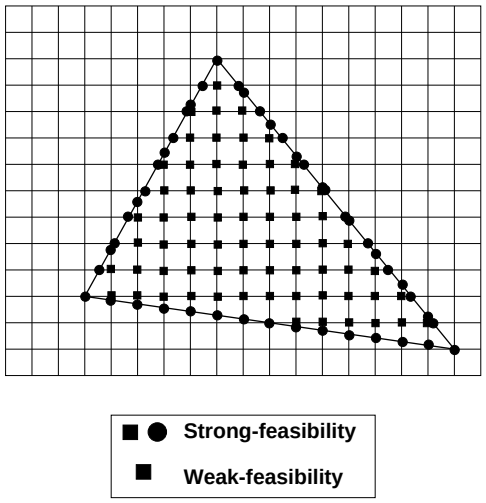


Figure 25: *Weak and strong feasibility.*

The algorithm works by performing three basic operations, one at a time: ear cutting, greedy biting, and edge splitting. Each operation is applied to a current active polygon. The next sections describe each of these operations in more detail.

Ear Cutting

This operation traverses a polygon $P \in \mathcal{P}$ looking for possible "ears" to cut. An *ear* of a simple polygon P is a triangle contained within P that shares two of its edges with P. We simply traverse the boundary of the polygon, "cutting off" any ear which we discover that corresponds to a *feasible* triangle (i.e., one that meets the feasibility criterion for ϵ). Given a vertex v_i, we check if the edge (v_i, v_{i+2}) is an internal diagonal to the polygon, that is, it is to the inside of the polygon and it does not intersect any other edge. This operation can easily be done in linear time by a simple traversal of the boundary of P. Using a dynamic triangulation of P, and performing "ray shooting queries", one can actually check in time $O(\log k)$ if (v_i, v_{i+2}) is an ear of a simple k-gon [42], but the simple linear-time method is likely to be more practical (since k is typically small) and is what we currently have implemented.

Each cut we perform lowers the complexity (number of edges) of polygon P by one, thus taking the algorithm closer to completion. Ear cutting is essential for the algorithm to terminate. In general, it will be the final step in any run of the algorithm. Also, it has a tendency to turn obtuse angles into acute ones, which eventually leads to larger edges (hence triangles) in the triangulation. Ear cutting is the mechanism the algorithm uses to adapt itself to lower sampling rates (larger triangles).

Ear cutting fails when no more feasible ears exist. This happens when the size of the edges of P are too large, and the ears cover too much area in the polygon. In this case, there must be some way to make edges smaller, which leads to higher sampling rates. In order to adapt to more complicated terrains, we introduce two additional basic operations: greedy biting and edge splitting.

Greedy Biting

In this basic operation, we find a point v inside the polygon P and an edge, (v_i, v_{i+1}) of P, such that (v_i, v, v_{i+1}) forms a triangle, T, inside P that meets the feasibility criterion. We accomplish two things with this operation: (1) subdividing an edge of P in two (replacing (v_i, v_{i+1}) with (v_i, v) and (v, v_{i+1})), thereby achieving a higher "sampling rate"; and, (2) taking a bite out of the polygon P, thus progressing further in "eating away" all of P. The actual operation is a bit more complicated, as it needs to handle choices of v that may be a vertex of P and lead to P being split into two disjoint new simple polygons.

The greedy biting operation works as follows:

- *Bite.* For the polygon P, for each edge (v_i, v_{i+1}) search for a point $v \in P$ such that (v_i, v, v_{i+1}) corresponds to a feasible triangle. For efficiency, we search for such a point v in a neighborhood of (v_i, v_{i+1}). Currently, we limit the search to grid points along (close to) the vector perpendicular to (v_i, v_{i+1}) at the midpoint of (v_i, v_{i+1}). We use a binary search, starting at a point whose distance from (v_i, v_{i+1}) is roughly $|v_i v_{i+1}|$, then halving the distance at each step until a point is found (or we fail). (By trying other search strategies for v, we can likely improve the algorithm performance. This is being investigated.)

- *Split.* If the "Bite" step succeeds in finding a point v for which (v_i, v, v_{i+1}) corresponds to a feasible triangle, we will potentially split polygon P. We search for the closest edge (v_j, v_{j+1}) to v. If the triangle (v_j, v, v_{j+1})

also corresponds to a feasible triangle, we subdivide (split) the polygon P into two simple polygons, outputting both triangles ((v_i, v, v_{i+1}) and (v_j, v, v_{j+1})); otherwise, we simply output (v_i, v, v_{i+1}) without splitting P.

Edge Splitting

It may happen that both ear clipping and greedy biting fail to find a feasible triangle. In this case, our algorithm attempts to split some edge of the polygon P. Checking each edge of P in succession, starting with the longest, we look for an edge to split (roughly) in half (or possibly in smaller pieces, if splitting in half fails). When we split edge (v_i, v_{i+1}) at a (grid) point v, we are actually creating a skinny (feasible) triangle, (v_i, v, v_{i+1}). Since the triangles created in this way are small or "slivery", we prefer not to perform this operation very often. Indeed, in practice this phase of the algorithm is seldomly needed.

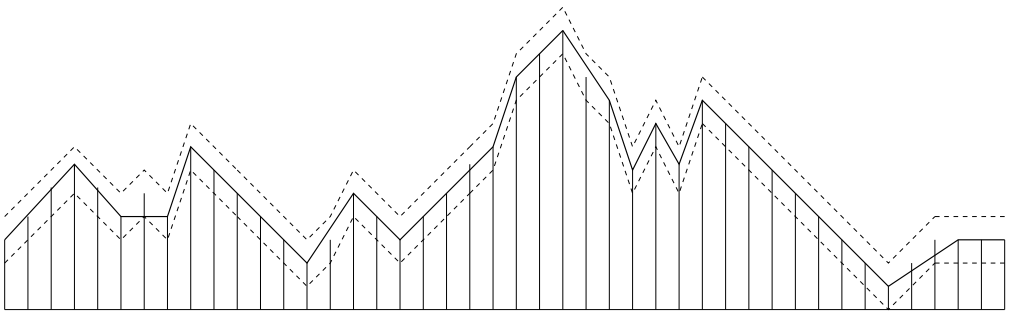


Figure 26: *The solid line is calculated by a greedy method. In linking data points, go as far as possible without exiting the strip defined by the dashed lines.*

Initialization

Each phase of our algorithm works to triangulate the interior of a simple polygon P, with feasible triangles. In order to generate the first such polygon, bounding the whole domain R, we apply a one-dimensional version of our algorithm in each of the four cross sections (defined by the vertical planes $x = 0, m,\ y = 0, n$) that correspond to the boundary of the region R. The algorithm can be considered to be a simplified version of the standard min-link path method of Suri [120], applied to the discrete data points between the offset curves obtained by shifting the terrain surface up/down by ϵ. See Figure 26.

Greedy Cuts Algorithm

The algorithm simply applies the above three operations, one at a time, giving priority (in order) to ear cutting, greedy biting, and then edge splitting. A complete description of our algorithm is outlined as follows:

(0) Initialize $\mathcal{P}$ to be a list of one element – the single polygon obtained by the initialization procedure above.

(1) While $\mathcal{P}$ is not empty, do

- **(a)** Let $P \in \mathcal{P}$.
- **(b)** If P is a single feasible triangle, output this triangle, and remove P from $\mathcal{P}$.
- **(c)** Else, while P is not fully triangulated,

(i) Perform ear cutting on P, until no feasible ears exist.

(ii) Perform greedy biting on P. If this results in a greedy bite that splits P, then remove P from $\mathcal{P}$, add the two new polygons to $\mathcal{P}$, and go to (1). Otherwise, if at least one greedy bite is found (for some edge of P), go to (1) (without splitting P).

(iii) Perform an edge split for P.

6.3 Miscellaneous Topics

6.3.1 Terrain Sampling

One of the most interesting properties of the Greedy Cuts algorithm is the way it samples the dataset. It generates large triangles in places of relatively little change and small triangles in areas of more radical change. It is interesting to try to analyze how this happens, and here is where we can see the nice coupling of properties between the ear cutting phase and the others. If the terrain is largely uniform, ear cutting generally leads to longer and longer edges of P, until we encounter a region of high complexity, at which point edges are subdivided by greedy biting or edge splitting (a method of increasing the sampling resolution). Once we triangulate the high complexity region, ear cutting again makes the edges on the boundary larger and larger (i.e., making the triangles larger). Our algorithm therefore has a natural mechanism for minimizing the number of triangles required. Of course, as we have already said, our algorithm is not guaranteed to find a true minimum (an NP-hard problem). The strategy of where/when to apply each of our three operations

affects which regions get sampled at higher resolutions. Thus, we continue to experiment with further variants of our search strategy in hopes of obtaining better and smaller triangulations.

6.3.2 Maintaining Structural Fidelity

A primary objective in any algorithm that simplifies (compresses) data is to maintain as much of the important structure of the input as possible. Our algorithm generates a TIN that is close to the input DEM, according to the given tolerance ϵ. However, beyond the constraint of being ϵ-close, one may wish to place further restrictions on the structural fidelity; for example, one may wish to preserve a selected set of point features or of edge features, requiring that the surface approximation include these points and segments in the output TIN. In top-down algorithms, such requirements can be incorporated using constraints; for example, line segments can be preserved using constrained Delaunay triangulation (e.g., [26]). In our bottom-up algorithm, we can incorporate such constraints directly, at low cost, within the test for triangle feasibility: A triangle T' is not feasible if its projection, T, contains a point feature on its interior or boundary, except at a vertex, or intersects an edge feature, except if the edge is an edge of T. Further, our algorithm can maintain the structure of an edge or a ridge, at a *lower* resolution (within, say, ϵ) than the full resolution, by executing the (lower dimensional) initialization step in a vertical wall (plane) through each constraint edge.

6.3.3 Termination

It is important to consider whether or not our algorithm terminates. Could it ever get "stuck" and fail to generate any further triangles, even though the list of untriangulated regions, $\mathcal{P}$, is not empty? The answer is "no" for the case of the weak feasibility condition, assuming that greedy biting is done by searching over all possible bites. As a proof, consider a polygon $P \in \mathcal{P}$. If P has no grid points, then any ear of P is feasible. (Any simple polygon with at least 4 vertices has at least two ears, by the "Two Ear Theorem" [87].) If P has grid points in its interior, then there must exist a triangulation of these points within P (since any polygonal domain can be triangulated). All triangles in this triangulation must obey weak feasibility. In particular, there must exist a triangle T that shares at least one of its edges with P. Such a triangle is either a (feasible) ear of P (found in ear cutting) or a potential bite (found in greedy biting, assuming that we do a full search). This proves termination.

In the strong feasibility case, we get a different situation. Because of the discrete nature of the allowed output (i.e., triangles must use original data points, since we do not allow Steiner points), and the continuous nature of the strong feasibility condition (which joins data points to form a polyhedral surface constraint), there are (rare) instances in which the algorithm, as implemented, can go into an infinite loop when using strong feasibility. In response to this, we have implemented a simple feature that will guarantee termination in all cases. If the algorithm cannot find a feasible triangle, then it relaxes the feasibility condition in ear cutting, and finds, instead, an ear that has the smallest deviation from the original DEM.

6.3.4 Complexity

We first remark that our algorithm uses very little internal memory. Other than the input data array, we keep track only of the list $\mathcal{P}$ of polygons, each of which is (typically) very small. Triangles that we generate do *not* need to be stored, but can be written out directly to disk. In contrast, methods that rely on triangulation refinement must maintain some sort of topological data structure for the full set of triangles. Typically, one would expect that if the output size (number of triangles) is k, then the boundary of the polygons $\mathcal{P}$ at any given instant will have roughly size $\sqrt{k}$.

It is difficult to prove a bound on the expected run time of the algorithm. Clearly, the *worst-case* running time is polynomial in the input size, since each primitive test or computation can easily be performed, usually in worst-case linear time (linear, generally, in the size of $P \in \mathcal{P}$). However, our experimental evidence suggests that the algorithm runs in time roughly linear in the input size.

The output complexity for our algorithm is again hard to estimate from a theoretical point of view. The problem we are trying to solve approximately is known to be NP-hard, in general. Thus, the best we can hope for is that we may be able to prove a worst-case bound on the ratio of our output size (number of triangles) to the number of triangles in an optimal TIN. There is good theoretical basis (e.g., from greedy set cover heuristics) to suggest that our algorithm (or a close variant thereof) will never produce more than a small (e.g., logarithmic) factor more triangles than is possible for a given ϵ. Proving such a fact remains an open (theoretical) problem. Perhaps the best indication we have of the effectiveness of the algorithm is the experimental

data we have, which suggests that our algorithm is obtaining substantially fewer (roughly 20-30 percent) triangles than the competing algorithm for the same error tolerance ϵ.

6.4 Experimental Results

The Greedy Cuts algorithm is relatively simple to implement. Our C implementation has about 4,000 lines of code. The code uses several computational geometry primitives, many of which come from O'Rourke [87], including segment intersection testing, diagonal classification, point classification (point location with respect to a simple polygon). With these primitives in hand, and routines to handle simple polygon operations (e.g., splitting an edge of a polygon, inserting a vertex), it is fairly easy to implement the algorithm described in Section 6.2. As with all geometric algorithms, care has to be taken with special (degenerate) cases that arise from collinearities.

In order to study its performance, we have conducted tests of our algorithm and compared it with Franklin's algorithm, which is a top-down approach. We compared the speed, average error bound (over all the triangles), and the complexity of the output (measured in the number of triangles). We ran both algorithms on the following types of input: real terrain datasets, artificially generated terrains arising from performing cuts to generate faults, and artificially generated terrains arising from lifting triangulations.

Franklin's algorithm

Franklin's algorithm is described in [33], and is a nice and efficient example of a top-down triangulation method. Initially, the algorithm approximates the DEM by 2 triangles. Then, a general step of the algorithm involves finding the most deviant point in each generated triangle and inserting this new point into the triangulation, splitting one triangle into three. Each time a point is inserted, the algorithm checks each quadrilateral that is formed by a pair of adjacent triangles, at least one of which is a new triangle (one of the three incident on the new point). A local condition on the quadrilateral determines whether or not to perform a diagonal swap. The original code works by performing a pre-determined number of splits. We have changed the code to make as many splits as necessary in order to meet a prespecified error bound ϵ. Franklin's implementation gives emphasis on efficiency and for the sake of speed, it makes wide use of internal memory.

Experimental Data

Our experiments were conducted on a Silicon Graphics ONYX, equipped with two 100Mhz R4400 processors and 64MB of RAM. Only one of the processors was used. The time to read the terrain datasets from the disk was not included in our runtimes. In Table 1, we show the results of running three algorithms on seven real terrain datasets. We ran Franklin's algorithm (f), and two versions of Greedy Cuts algorithm — one using weak feasibility (w), and one using strong feasibility (s). The table shows the choice of ϵ, the running times, and the total number of triangles in the output TIN, for each of the seven

terrains. The input terrains were all scaled to be 120-by-120 elevation arrays, for uniformity of testing.

In summary, *greedy cuts* with *weak-feasibility* beats Franklin's code in the number of output triangles in all instances. *Greedy cuts* with *strong-feasibility* loses in most cases, but it applies a stricter accuracy requirement than Franklin's algorithm (which uses weak feasibility). Franklin's optimized code is usually faster than our (relatively naive) implementation. We expect that with fine tuning and optimization, our algorithm will be able to run much faster. But perhaps more significant is the comparison of memory requirements. On average, Franklin's algorithm used more than an order of magnitude the memory Greedy Cuts require.

Figures 27 and 28 show rendering examples of real terrain rendered with both Franklin's and our algorithm. Notice that our algorithm generates considerably larger polygons.

Figures 29 to 33 contain images with the Denver digital terrain at different resolutions. It shows the scaling of the TINs generated by our algorithm as the error bound gets smaller.

6.5 Algorithm Extensions and Optimizations

The work presented so far can be extented in several different ways. In this section we describe some possible extension. In order to make the algorithm run on more general inputs, such as polyhedral terrains, and general polyhedral surfaces, there are at least a couple of different directions. The conceptual treatment of polyhedral terrains is trivial. In the case the polyhedral terrain is

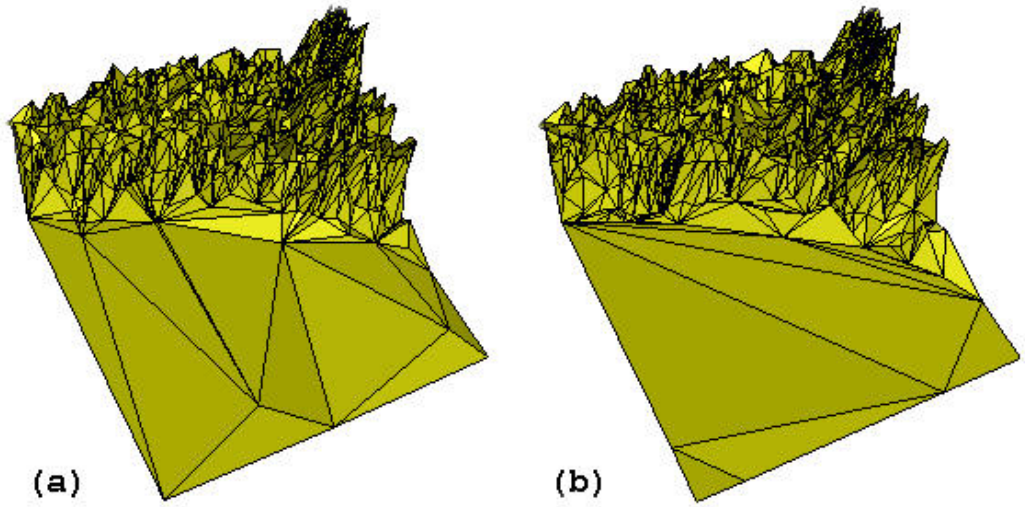


Figure 27: *Buffalo terrain triangulated with (a) Franklin's algorithm, (b) our algorithm (strong-feasibility).*

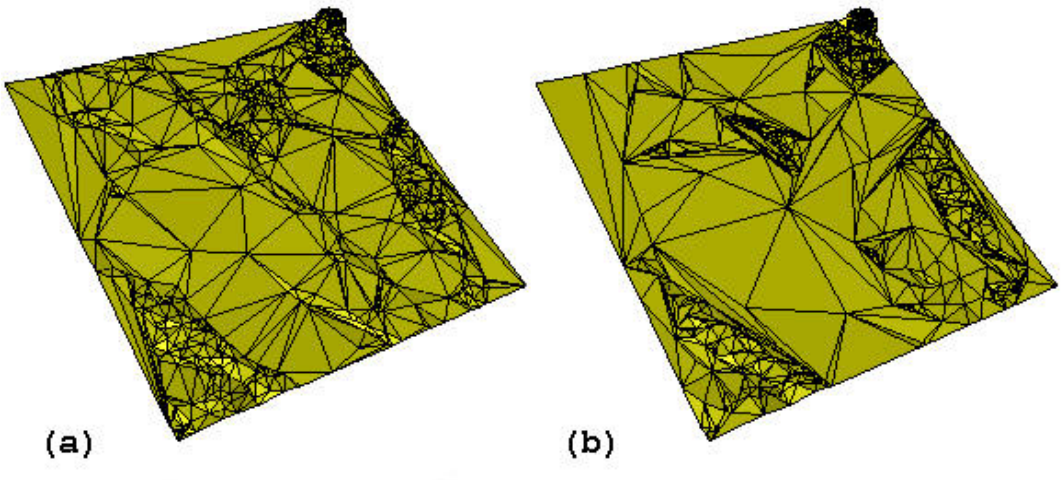


Figure 28: *Jackson terrain triangulated with (a) Franklin's algorithm, (b) our algorithm (strong-feasibility).*

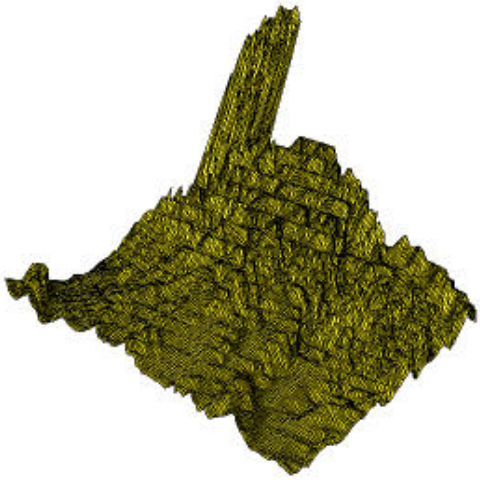


Figure 29: *Denver height-field data before simplification.*

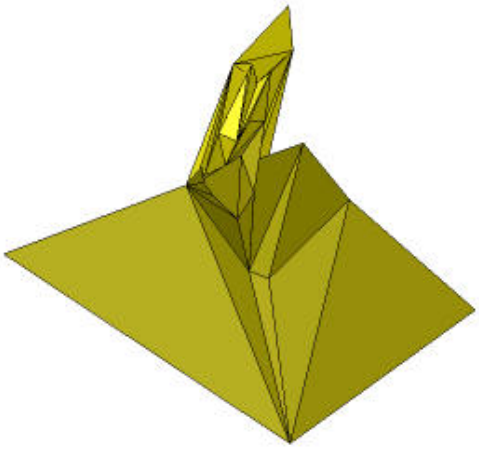


Figure 30: *Denver terrain triangulation,* $\epsilon = 20$ *units.*

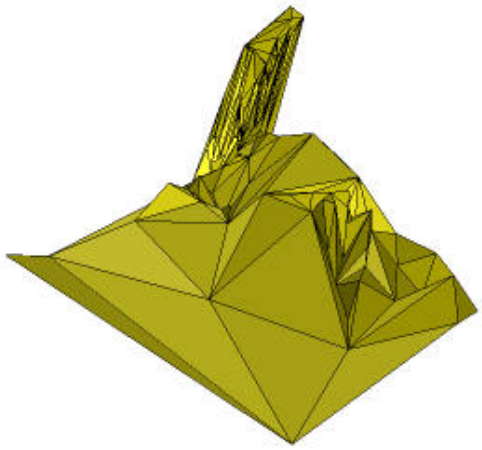


Figure 31: *Denver terrain triangulation,* $\epsilon = 10$ *units.*

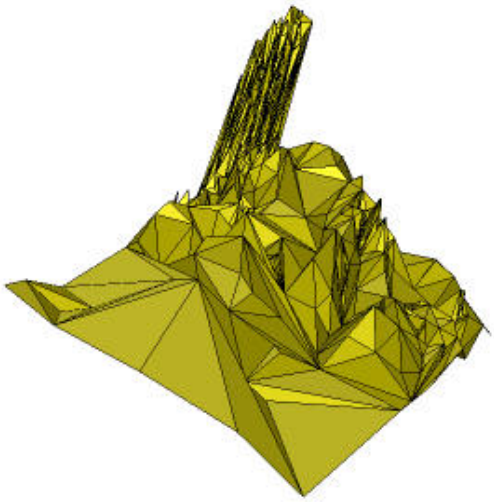


Figure 32: *Denver terrain triangulation,* $\epsilon = 5$ *units.*

Terrain	ϵ	Time	Trgs.	Memory
Buffalo	2.5 (f)	3.2	1994	6229
	2.5 (w)	8.12	1641	428
	2.5 (s)	21.86	2279	592
Denver	2.5 (f)	5.03	2688	8731
	2.5 (w)	17.38	2137	572
	2.5 (s)	27.57	2849	700
Eagle Pass	1.5 (f)	2.23	1564	4781
	1.5 (w)	4.24	1214	315
	1.5 (s)	8.1	1578	454
Grand Canyon	15 (f)	4.5	2822	8621
	15 (w)	12.87	2073	488
	15 (s)	37.96	3115	844
Jackson	0.5 (f)	2.44	1297	4084
	0.5 (w)	2.6	859	231
	0.5 (s)	3.62	1127	296
Moab	15 (f)	4.03	2561	8082
	15 (w)	10.27	1836	495
	15 (s)	21.09	2430	628
Seattle	5 (f)	5.28	2671	8365
	5 (w)	9.70	2011	486
	5 (s)	26.75	2763	672

Table 1: *Running times (in sec) of three algorithms on seven real terrain data sets.* (f) *indicates Franklin's code;* (w) *and* (s) *indicate our algorithm with weak and strong feasibility, respectively. All terrains are 120×120 elevation arrays. The error bounds (ϵ) were chosen to keep the number of triangles* (Trgs.) *in the output approximately in the 1000 to 3000 range. Memory usage is the number of 8Kbyte pages allocated.*

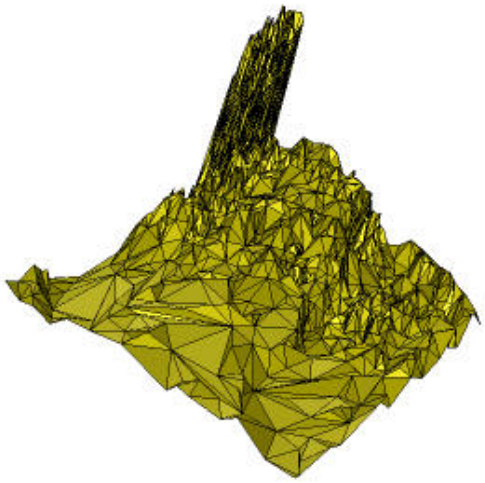


Figure 33: *Denver terrain triangulation,* $\epsilon = 2.5$ *units.*

homeomorphic to a sphere (i.e., its boundary is composed of a single piece), one can just change the feasibility checking, and the method of finding vertices for greedy biting. Extensions to general surfaces are harder, but following in the same footsteps, it is possible to break any polyhedral surface into polyhedral terrains, run the simplification algorithm in each one of these and glue the pieces back together. The gluing process is non-trivial, as it is necessary to respect boundaries across patches. Another possible solution is to extent the feasibility checking to deal with surfaces directly, without the need for the decomposition step.

Polyhedral Surface Simplification with Greedy Cuts

The Greedy Cuts algorithm works by taking greedy cuts ("bites") out of a simple closed polygon that bounds the yet-to-be triangulated region. The

algorithm starts with a large polygon, bounding the whole extent of the terrain to be triangulated, and works its way inward, performing at each step one of three basic operations: ear cutting, greedy biting, and edge splitting. Each atomic operation can only be performed if the triangles are *feasible*.

There are basically two main challenges in trying to extend the algorithm to polyhedral surfaces of general topological types. First, the algorithm assumes it starts with a polyhedral patch that is homeomorphic to S^2, that is, the boundary of the patch is connected and has no holes inside. Second, the feasibility criteria is highly dependent on the fact that the domain is a terrain, and the actual implementation there uses the fact that a height field is given.

In order to make the surface $\mathcal{S}$ suitable for the GC algorithm, we break it up into multiple disjoint surfaces $\mathcal{S}_1, \mathcal{S}_2, \ldots \mathcal{S}_n$, where each is homeomorphic to S^2. Furthermore, we force each $\mathcal{S}_i$ to be a *terrain*, in the sense that each can be re-parametrized on parameters u and v, in such a way that $\mathcal{S}_i$ is the graph of some function $g_i(u, v)$. This transformation in $\mathcal{S}$ allows each of its patches to be simplified in accordance to the GC algorithm. A problem remains on how to connect the patches as to have no "cracks" or "holes". As shown below we perform the operations as to avoid cracks and holes when we join ${\mathcal{S}'}_i$ into the final $\mathcal{S}'$.

Our modified GC for surfaces, consists of several steps. First, we need an algorithm that breaks $\mathcal{S}$ into multiple patches, each of which is a terrain and homeomorphic to S^2. Second, we need to apply a modified GC to each patch and finally we need to re-group all the patches back together avoiding cracks and holes.

Breaking Surfaces Into Terrains

Breaking (or partioning) a surface into terrains is not a trivial task. Actually the problem of breaking a general polyhedral surface into the minimal number of terrains has been conjectured to be NP-hard. Because of this, we concentrate on techniques that break a given surface into a small number of terrains.

Our general technique works by trying to grow a terrain patch by starting from a single face and introducing feasible faces to its boundary one at a time. In order to do this efficiently, we need a way of deciding if the terrain patch continues to be a terrain after the introduction of the given face to its boundary.

A simple way to accomplish this is to use a set of predetermined directions as a witness set. That is, we keep the arrangements of the projections of the chosen faces in each one of these planes, and while there is no overlap we are guaranteed that the tested face can be safely added to our patch. If there is an overlap in one plane, we can discard it and continue the procedure on the other planes. A version of this procedure was used to generate the pictures in Figures 34, 35, 36, and 37.

This approach is certainly not the best one possible. In a certain way, the normals to each triangle should be enough to decide whether a face can be safely added to a patch. There is, we should be able to use Gauss map [28]. (The Gauss map is a map from a surface to the sphere where the unit normal at every point on the surface is mapped to the sphere. Given the gauss map, one would hope to be able to determine if a patch of the surface is a terrain by checking if the convex hulls of the image of the Gauss map does not contain the

origin of the sphere, or by a similar procedure) Unfortunately, this only works for two-dimensions; we are still working on extending it to three-dimensions.

Anyway, if we regard the method of deciding if a face can be added to a terrain patch as a black box, we can simply start from any face, and from there we add faces to a patch (in our case in a sort of breadth-first search on the adjacency graph of the surface), until we try to add a face that violates our test. Then we mark that face as non-feasible and continue the search from a feasible face until the queue of possible faces is empty and we are left with a collection of connected faces that constitute a terrain patch.

One shortcoming of this technique is that the terrains might ended up with multiply connected components, making them unsuitable for our purposes. Fortunately, it is fairly simple to add a constant time check on the search to guarantee terrains that have a singly-connected boundary. We just mark already seen faces, and faces that have been added to the patch, and we can guarantee we do not close any boundaries.

Modified Greedy Cuts

Once we have multiple connected components $\mathcal{S}_1, \mathcal{S}_2, \ldots \mathcal{S}_n$, each of which is a terrain homeomorphic to a disk, we can to apply our Greedy Cuts algorithm. One possible problem remains: if each are simplified separately, it is necessary to guarantee we can glue them back together. Our method for doing this consists of breaking the boundary of $\mathcal{S}_i$ into paths α_i^j, and augmenting these paths into *maximal disjoint paths*, where each of these is a connected path of longest length that is contained in the intersection of two of the terrain patches. It is fairly simple to compute these paths, let assume γ_i^j. With γ_i^j in hand,

we apply a one dimension simplification on these (assume an ϵ-tube around them), and use these simplified paths to construct a simplified boundary β_i^j, that is suitable for running the Greedy Cuts algorithm (see Figures 34 and 35).

Gluing Patches

The last (and simpler) phase of the algorithm consists of gluing back the pieces. This is trivial as the previous phase has created simplified paths that exactly *fit* across the boundaries nicely.

Important Considerations

There are several tradeoff being made in our algorithm. The most important shortcoming being that we can not simplify a surface by more than the number of terrains it can be broken up into. A *bumpy* surface might produce terrible results. Also, the current breaking scheme can be largely improved, running our algorithm on the Stanford Bunny we notice just a few patches (around 10) have more than 99% of the triangles, but as holes were created when generating the patches, we generate another 100 or so patches, each with fewer than 10 triangles. We also need to avoid (or detects for gracefully topological breaking) self-intersections during the Greedy Cuts phase.

Instead of breaking the original surface into terrains, another solution is to avoid the decomposition process. All that is necessary is to extend the *feasibility checking* for non-terrain surfaces. Then, conceptually, one can just run the GC algorithm on the surface.

6.6 Conclusions and Future Work

We have presented a new method to generate TINs from dense terrain grids. Our algorithm differs from previous methods in its use of a bottom-up approach to terrain sampling. Its key features include:

- *Low Complexity Output TIN.* Our method generates very few triangles for a given ϵ. Indeed, a primary objective in using the greedy optimization step is the minimization of the number of triangles in the output.
- *Memory Efficiency.* It can be run on very large terrains, potentially even those whose grids cannot simultaneously fit in memory.
- *Maintenance of Structural Fidelity.* Our method is able to maintain with very little additional overhead any pre-specified set of features of the terrain, without the need for adding additional (Steiner) points.
- *Speed.* Our running times are comparable to the fastest available methods, and we can probably improve the performance dramatically with a careful refinement of our code.

Our experimental results so far have focussed on the quality of the output TIN. The running time can certainly be improved through more careful coding. Also, further experimentation with the heuristics, especially the greedy biting operation, should yield even better results with respect to the output size. On the theoretical side, we are also attempting to prove worst-case bounds on the performance of the approximation (e.g., that we obtain a number of triangles that is guaranteed to be within a small factor of optimal).

A straightforward modification of our code will permit the algorithm to work on arbitrary TIN terrain inputs, rather than just on DEM arrays. Conceptually, there are no changes needed to the algorithm. A somewhat less trivial modification will be to generalize the algorithm to approximate arbitrary (non-terrain) polyhedral surfaces and to find approximations to a minimum-facet separating surface (as done in [10, 16, 79], in the convex case).

Another straightforward extension of our method allows one to use it to build hierarchical representations of terrain. For example, we can simply start with an extremely crude terrain approximation (e.g., just two triangles), and then adjust ϵ to be smaller and smaller, making each corresponding TIN a refinement of the previous one, until we have the full resolution grid. An ideal such hierarchy would have logarithmic height, as the intermediate TINs have sizes 2, 4, 8, 16, etc.

Another extension that we are pursuing is to approximate functions (terrains) of three variables. Approximating such functions is very important in scientific visualization. One can apply our same paradigm to this problem, biting off tetrahedra that satisfy the ϵ-fitness criterion. The tricky issue in implementing this algorithm is in maintaining the regions $\mathcal{P}$ of *un*tetrahedralized domain, since this will be a polyhedral space, possibly of high genus.

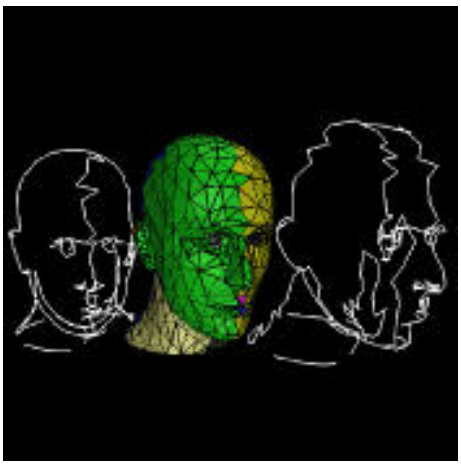


Figure 34: *Terrain decompositiong with simplified paths of the Mannequin model.*

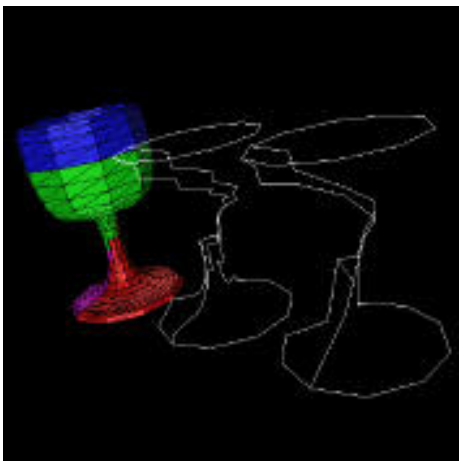


Figure 35: *Terrain decomposition with simplified paths of the Goblet model.*

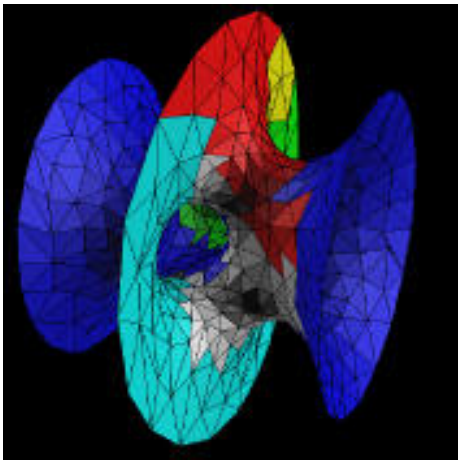


Figure 36: *Terrain decomposition of the minimal surface model.*

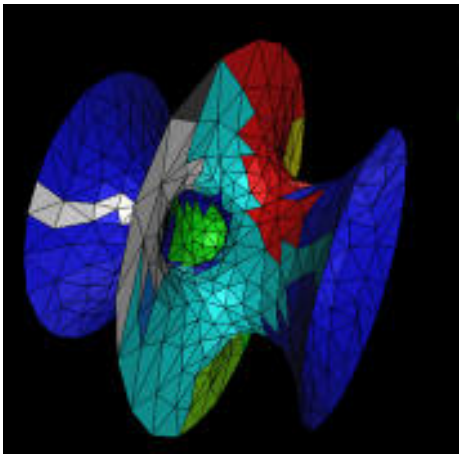


Figure 37: *Terrain decomposition of the minimal surface model into patches homeomorphic to a disk.*

Chapter 7

The PVR System

The PVR (Parallel Volume Rendering) system is an object-oriented, client/server system, developed for high performance volume rendering of very large datasets. Among its important features are its unique performance and scalability. PVR is well suited for use in a supercomputing environment, where datasets are too large to be easily archived and visualized, and where computational steering capabilities are necessary. For the scientist, PVR offers transparency from machine architecture details in achieving high performance visualization, while for the tool builder it provides an easily extensible system architecture.

7.1 Introduction

In order to allow researchers and engineers to make effective use of volume rendering in the study of complex physical and abstract structures, a coherent, powerful, easy-to-use visualization tool is needed. Furthermore, such a tool

should allow for *interactive* visualization, ideally with support for user-defined "computational steering."

There are several issues and challenges in developing such a visualization tool. First, even with the latest volume-rendering acceleration techniques running on top-of-the-line workstations, it still takes a few seconds to a few minutes to volume render an image. This is clearly far from interactive. With the advent of larger parallel machines and better scanners and instrumentation, larger and larger datasets are being generated (typically on the order of 32MB to 512MB, ranging to 16GB), some of which would not fit in memory of a workstation class machine. Second, even if rendering time is not a major concern, large datasets may be too expensive to hold in storage, and extremely slow to transfer to typical workstations over network links.

These issues lead to the question of whether the visualization should be performed directly on the parallel machine which is used to generate the simulation data or sent over to a high performance graphics workstation for post-processing. First, if the visualization software was integrated directly with the simulation software, there would be no need for extra storage, and visualization could be an active part of the simulation. Second, large parallel machines can render these large datasets faster than workstations can, possibly in real-time, or at least achieving interactive frame-rates (see Chapter 3). Finally, the integration of simulation and visualization in one tool, whenever possible, is highly desirable because it allows users to interactively "steer" the simulation. With steering, users are able to terminate or modify parameters in their simulations as the simulations progress, rather than have to wait for

painfully long simulations on extremely expensive machines, with high storage and transmission costs, only to discover during post-processing that the simulations are wrong or uninteresting.

Here we introduce the PVR (Parallel Volume Rendering) system, developed under collaboration between the State University of New York at Stony Brook and Sandia National Laboratories. PVR is a component approach to building a distributed volume visualization system. At its topmost level, it provides a flexible and high performance client/server volume rendering architecture with a unique load balancing scheme which provides a continuum of cost/performance parameters that can be used to optimize rendering speed. The original goals of PVR were to achieve a level of portability and performance for rendering beyond that of other available systems and to provide a platform that can be used for further development.

In a certain way, PVR is more than a rendering system; its components have been specially designed to be user-extensible in order to allow for user-defined computational steering. That is, the user can easily add custom computational code to PVR and just link in the rendering library. Using PVR, it is much easier to build portable, high performance, complex, distributed visualization systems. Figure 38 displays the relationship between PVR and a distributed visualization environment.

The rest of this chapter introduces the PVR client/server architecture and its components, with an emphasis on its support for volume rendering.

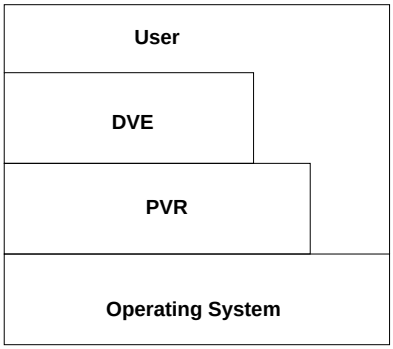


Figure 38: *The Relationship of a Distributed Visualization Environment (DVE) System and PVR.*

7.2 The PVR System

It is well known that system complexity limits the reliability of large software systems. Distributed systems exacerbate this problem with the introduction of asynchronous and non-local communication. With all of this in mind, we have used a component approach in developing our system. PVR attempts to provide just enough functionality in the basic system to allow for the development of large and complex visualization and computational steering applications. It is based on a client/server architecture, where there are, on one side, rendering/computing servers which are coupled, and, on the other side, the user acting as a client from his workstation.

The PVR client/server architecture is implemented in two main components: the *pvrsh*, which runs in the user's workstation, and the *PVR renderer*, which runs in the parallel machines. The renderer is implemented as a library and it allows for easy integration of user-defined code that can share the same processors as the rendering code. Communication across applications written with PVR are performed using the PVR protocol, and in our implementation

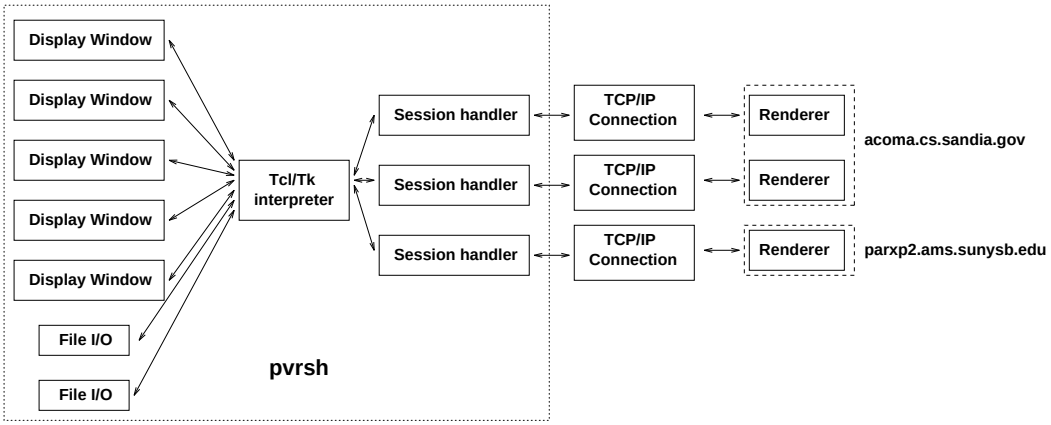


Figure 39: *PVR Architecture. The overall structure of the system is shown with an emphasis on the* pvrsh. *The Tcl/Tk core acts as glue for all the client components. Everything, with the exception of the* renderers, *runs on the user's workstation. The* renderers *run remotely on the parallel machines.*

communication is handled by separate UNIX processes (see Figure 39).

7.2.1 The *pvrsh*

The *pvrsh* provides a single new object to the user, the *PVR session*. The *pvrsh* is an augmented Tcl/Tk shell. We chose to use Tcl/Tk [88] as the system glue. Tcl, *Tool command language*, is a script language designed to be used as a generic language in application programs. It is easily extendable with new user commands, in C or Tcl, and coupled with the graphical environment Tk, it is a powerful graphical user-interface system. The use of the Tcl/Tk, which is a well-designed, debugged application language and graphical environment has contributed to reducing the overall system complexity.

The *PVR session* is an object (such as the Tk objects). It contains attributes, and corresponding methods are used to change the attributes. One

of the most important attributes is the one that *binds* a session to a particular parallel machine. Figure 39 contains an example of three sessions, two on acoma.cs.sandia.gov (a large Intel Paragon XP/S with over 1840 nodes running SUNMOS [67], installed at Sandia) and one on parxp2.ams.sunysb.edu (a small Intel Paragon with 110 nodes running Intel's version of OSF/1, installed at Stony Brook). The system is designed to handle multiple sessions using the same protocol with machines running different operating systems.

As part of its attributes, a session specifies the number of nodes it needs and the parameters that are passed to those nodes. Several pieces of informations are *interactively* exchanged between the *pvrsh* and the *PVR renderer*, such as rendering configuration information, rendering commands, sequences of images, performance and debugging information.

There is a high amount of flexibility in the specification of the rendering. Not only can simple rendering elements, such as changing transformation matrices, transfer functions, image sizes and datasets be specified, but there are commands (see Table 2) to specify in a high level format the complete parallel rendering pipeline (see Chapter 3 for details). With these parameters in hand, the *pvrsh* can be used to specify almost arbitrary scalable rendering configurations (see Section 7.2.4).

The *pvrsh* is implemented as a single process (making ports easier) in about 5,000 lines of C code. We have augmented the Tcl/Tk interpreter with TCP/IP connection capabilities (some versions of Tcl/Tk have this built in). In order to support several concurrent sessions, all the communication is performed asynchronously. We use the `Tk_CreateFileHandler()` routine to arbitrate between input from the different sessions. A UNIX `select` call and polling

could be used instead but would make the code harder to understand and, overall, more complex. Sessions work as interrupt-driven commands, responding to requests one at a time. Every session can receive events from two sources at the same time: the user keyboard and the remote machine. Locking and disabling interrupts are needed to ensure consistency inside critical sessions.

The overall structure of the code allows for user augmentation of a session functionality either by external or internal means. *External* augmentation can be performed without re-compilation, such as that used by the user interface to show images as they are received asynchronously from the remote parallel server. *Internal* augmentation requires changes to the source code. The source code is structured to allow for simple addition of new functionality. Only a single file needs to be changed to add a new session method. If it bbbchanges the *Resource Database*, two files need to be changed. New commands are added using Tcl conventions. (For details, see Part 3 of Ousterhout's Tcl/Tk book [88].)

Every PVR message is sent either as a single fixed-length message, or as two messages (the first is used to specify the size of the second). This is used to make redirection easier and to achieve optimal performance under different configurations. Look-up tables are set up with actions to be taken up on the arrival of each message type. This setup makes additions to the PVR protocol very simple.

7.2.2 The *PVR Renderer*

The *PVR renderer* is the piece of PVR that runs remotely on a parallel machine (see Figure 39). It is composed of several components, the most complex being

the rendering code itself. In order to start up multiple parallel processes at the remote machine, we use *pvrd*, the PVR daemon. This daemon runs on the parallel machine. It waits on a well-known port for connection requests. Once a request for opening a new session is made, it *forks* a handling process that is responsible for allocating processors and communicating with the session on the client. On the remote machine, the handling process allocates the computing nodes and runs the renderer code on them. The connection process is illustrated in Figure 40. One *pvrd* can allocate several processes; once it is killed, it kills all its children before exiting.

The renderer is the code that actually runs on the parallel nodes. The overall structure of the code resembles a SIMD machine [47], where there are high-level commands and low-level commands. There is one *master* node, similar to the microcontroller on the CM-2 machines, and several *slave* nodes. The functions of the slaves are completely dependent on the master. The master receives commands from the *pvrsh*, translates them, and takes the necessary actions, including changing the state of the slaves and sending them a detailed set of instructions.

For flexibility and performance, the method of sending instructions to the nodes is through *action tables* (similar to SIMD microcode). In order to ask the nodes to perform some action, the master broadcasts the address of the function to be executed. Upon receiving that instruction, the slaves execute that particular function. With this method, it is very simple to add new functionality because any new added functionality can be performed locally, without the need to change global files. Also, every function can be optimized independently, with its own communication protocol. One shortcoming of this

communication method (as in SIMD machines) is that one has to be careful with non-uniform execution, in particular because the Intel NX communication library (both OSF and SUNMOS have support for NX) has limited functionality for handling nodes as groups. For example, in setting up barriers with NX, it is impossible to select a group from the totality of the allocated nodes. Newer communication libraries, such as MPI [115], solve this shortcoming by introducing the idea of groups of nodes.

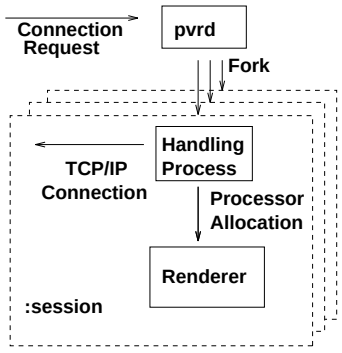


Figure 40: *In order to allocate nodes, the* pvrsh *sends a command to the* pvrd*, which in turn creates a special communication handling process and allocates a partition on the parallel machine.*

The master intrinsically divides the nodes into *clusters*. Each cluster has a specialized computational task, and multiple clusters can cooperate in groups to achieve a larger task. All that is necessary for cluster configuration is that the basic functions be specified in user-defined libraries that are linked in a single binary. During runtime, the user can use the master to reconfigure clusters according to immediate goals. The *pvrsh* can be used to *interactively* send such commands. As an example of the use of such a clustering scheme, see Figure 41, where the rendering configuration for PVR's high performance

volume renderer is depicted.

In order to achieve user-defined computational steering, one can use this clustering paradigm. It is usually necessary to add one's functionality to the action tables (e.g., linking the computational code with PVR dispatching code), and also add extra options to the *pvrsh* (usually through the `set` command) for modifying the relevant parameters interactively.

PVR volume rendering code was the inspiration for this overall code organization and is a very good application to demonstrate its features. Because in this chapter our focus is on describing the PVR system, and not on the actual volume rendering code, we only sketch the implementation to give an insight as to how to add your own code to PVR and to give you enough information for effective use of the PVR rendering facilities.

7.2.3 Volume-Rendering Pipeline

The PVR rendering pipeline is composed of three types of nodes (besides the master). These are the *rendering nodes*, *compositing nodes*, and *collector nodes* (usually just one), (see Figure 41). This specialization is necessary for optimal rendering performance and flexibility. All the clusters work in a simple dataflow mode, where data moves from top to bottom in a pipeline fashion. Every cluster has its own fan-in and fan-out number and type of messages (see Figures 10 and 11). The master configures (and re-configures) the overall dataflow using a set of user-defined and automatic load-balancing parameters.

At the top level are the rendering clusters. The nodes in a rendering cluster are responsible for resampling and shading of a given volume dataset. In general, the input is a view matrix, and the output is a set of sub-images,

each of which is a related to a node in the compositing binary tree. The master can use multiple rendering clusters working on the same image, but on disjoint scanlines in order to speed up rendering. Once the sub-images are computed, they are passed down the pipeline to the compositing clusters.

The compositing clusters are organized in a binary tree structure, matching that of the compositing tree which corresponds to the decomposition of the volume data set on the rendering nodes. The number of processors used to do compositing can actually be different than the number of nodes in the compositing tree, as we can use *virtualization* to fake more processors than allocated. Images are pipelined down the tree, with every iteration combining the results of compositing until finally all the pixels are a complete depth-ordered sequence. Those pixels are converted to RGB format and sent to the collector node(s) (at this time, we just use a single collector node).

The collector node receives RGB images from the compositing nodes and compresses them using a simple run-length encoding scheme (very fast compression is necessary). Finally, the images are either sent over to the *pvrsh* for user viewing (or saving), or locally cached on the disk. An additional option allows images to be trashed for performance analysis purposes.

The previous discussion is somewhat simplistic. There are several performance issues related to CPU speed, synchronization, and memory usage that have not been discussed. For more complete details, see Chapter 3.

7.2.4 Rendering with PVR

Figure 42 shows a simple PVR program. Several important features of PVR are demonstrated: in particular, the seamless integration with Tcl/Tk, the

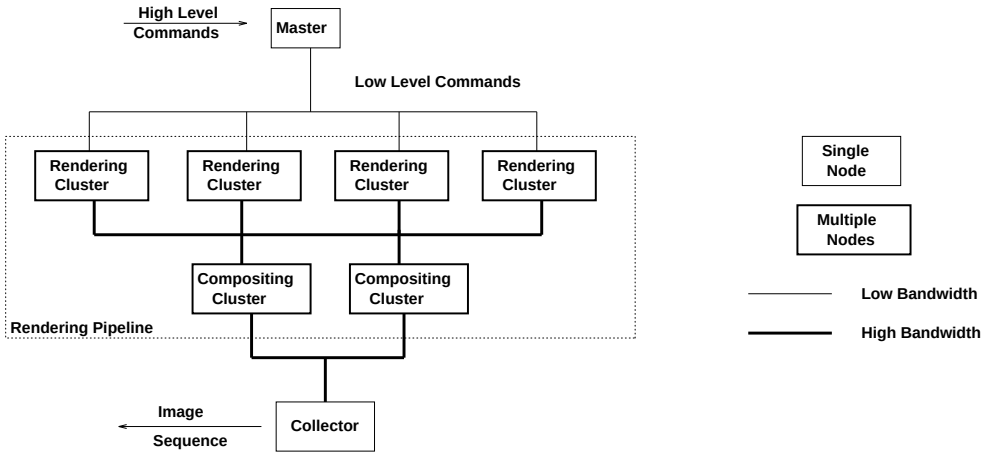


Figure 41: *The master receives high level commands that are translated into virtual* microcode *by the action tables. For rendering, the high level commands are for the generation of animations by rotations and translations, which are interpreted into simple transformation matrices commands. The rendering clusters perform rendering in parallel. The collector receives and groups images together and sends an ordered image sequence to the client application.*

flexible load-balancing scheme, and the interactive specification of parameters. The `set` command can have several options (in Figure 42, options are usually specified in multiple lines, but could be specified in a single line). For instance, `-imagesz` specifies the size of the images that are output by the system.

The `-cluster` and `-group` options are unique to PVR and its flexible load-balancing scheme. With both of these options, the relative sizes of the rendering and compositing clusters can be specified together with the image calculation allocation. Several scalability strategies can be used. For instance, a rendering cluster needs to be large enough to hold the entire dataset and at least a copy of the image to be calculated. By increasing the size of the cluster (i.e., the number of nodes in the cluster), the amount of memory needed per node decreases. By *grouping* clusters (i.e., splitting the image computation

across multiple clusters), the number of scanlines a given cluster is responsible for decreases, lowering both the image memory requirements and the computational cost, thus speeding up image calculation.

The same commands can be used to configure compositing clusters. The scalability parameters for compositing clusters are very different than for rendering clusters, because of the different nature of the task. Compositing nodes need memory to hold two copies of the images, which can be quite large (our current parallel machine nodes have only between 16MB to 32MB RAM), and, also, compositing has a very high synchronization cost which increases as the number of nodes increase. Currently, the only need for multiple compositing clusters is due to the need of more memory for large images (such as 1024-by-1024).

7.3 Miscellaneous Topics

7.3.1 Related Work

The Shastra project at Purdue has developed tools for distributed and collaborative visualization [6]. The system implements parallel volume visualization with a mix of image-space and object-space load balancing. Few details of the scheme are given, and they report using up to four processors for computation, which makes it hard to evaluate the systems usability in a massively parallel environment.

Rowlan et al. [95] describe a distributed volume-rendering system implemented on the IBM SP-1. Their system seems to have several of the same

```
toplevel .rgb                          ; Tcl/Tk stuff – creates the windows
photo .rgb.p
pack .rgb.p
toplevel .c
canvas .c.c
pack .c.c
source stat.tcl                        ; stat.tcl defines actions used to
                                       ; bookkeep performance information
                                       ; and graphing them in a window
pvr_session :brain                     ; creates a session called "brain"
:brain image window .rgb.p             ; specifies the image window
:brain image callback imgCallback      ; imgCallback is called arriving image
:brain image dir ./                    ; where to place images
:brain open acoma.cs.sandia.gov        ; opens a connection with acoma
                                       ; using the default number of nodes
                                       ; the defaults are in .pvrsh
:brain set -dataset brain.slc          ; specifies the dataset
:brain set -cluster r,16 -group 0,0,1,1; 4 rendering clusters of 16 nodes
                                       ; divided into 2 groups where
                                       ; nodes in a group share
                                       ; the same image calculation
:brain set -cluster c -group 0,0       ; 2 compositing clusters of 15 nodes
                                       ; this allows for the calculation of
                                       ; large images (each cluster handles
                                       ; half of pixels)
:brain set -imagesz 512,512            ; specifies the image size
:brain render rotation 0,1,0 15,59:60  ; specifies the rendering of
                                       ; 45 images, starting from
                                       ; one quarter rotation along
                                       ; the y axis
```

Figure 42: *A simple PVR program with a set of PVR rendering commands. The commands can be put in a file and executed in batch, or can be typed interactively on the keyboard (or mixed). Tcl/Tk code (for example, "stat.tcl") can be written to take care of portions of the actions.*

characteristics as ours. In particular, it runs on a massively parallel machine, provides object-space partitioning, uses separate rendering and compositing nodes and provides a front-end GUI. Unfortunately, their paper provides few details on the actual architectural design and implementation, and even the rendering is described very briefly. As far as we can detect, their system does not provide the flexibility, portability and performance that our system does. For instance, it does not provide support for multiple rendering or compositing clusters.

Another cousin of our system is DISCOVER [62], developed at National Cheng-Kung University (Taiwan). This system has been developed for custom medical imaging applications and provides mechanisms for the use of remote processor pools. It provides a client/server architecture for a variety of clients, including support for Microsoft Windows.

7.3.2 Distributed Visualization Environments (DVEs)

DVEs can be easily developed by making use of the client/server metaphor. A DVE developed using Tcl/Tk is very portable, as Tcl/Tk has ports for almost all of the operating systems available, and TCP/IP (our communication protocol) is virtually universal. We give more details on the primitives from which DVEs can be built in Table 2.

Figure 47 shows a simple prototype GUI developed at Sandia. The complete interface is written in Tcl/Tk. The user is able to specify all the necessary rendering parameters in the right window (including image size, transfer function, etc.) and the load-balancing parameters in the left window. This simple interface uses only a single session at this time, but more functionality

is currently being added to the system.

Using the prototype GUI, users are able to add their own functionality to the system as needed. This flexibility not only makes the system more usable, because redundant bells and whistles can be discarded, but also new functionality can be added straightforwardly. The use of a portable and well-documented windows interface (e.g., Tk) is imperative. Not only do users avoid having to learn yet another programming language and graphical toolkit, but the use of Tk saved us a lot of implementation and documentation cost (Tcl/Tk is widely used and well-documented). Another important feature of Tcl/Tk for the development of prototypes is that it is freely available, enabling us to do the same for PVR.

7.3.3 Visualization Services

Our system architecture can be used to visualize time-varying data. When rendering time-varying data, we add a permanent *caching cluster* to the pipeline in Figure 41 which is responsible for distributing the volume data to the rendering nodes efficiently. The caching cluster is used to hide I/O latency from disk (or other sources). This way, the user can visualize a dataset for as long as it takes a new version of the dataset to come along. Handling data that changes too rapidly (i.e., faster than we can move it and render it) is not possible, as it would require large amounts of buffering.

Another possible use of our parallel renderer is as a visualization server for large computational parallel jobs [90]. The basic idea is to pre-allocate a set of nodes that can be shared to a limited extent by multiple users for visualizing their data. Effective use as such a server would also make use of a caching

cluster, as described above for time-varying data. The cluster, in this case, would be used to cache in alternate user data sets.

7.3.4 Results

The current version of PVR consists of about 25,000 lines of C and Tcl/Tk code. It has been used at Brookhaven National Labs, Sandia National Labs, and Stony Brook to visualize large datasets for over a year, and its reliability has been improving steadily. Below we discuss a few of the current uses and performance of PVR. The biggest challenge we have faced so far is the limited amount of memory on our Paragon nodes. It is very hard from the software engineering point of view to have consistent and reliable treatment of memory allocation issues, specially when attempting to visualize very large datasets.

We have demonstrated the capability of rendering a 500MB dataset (the 512-by-512-by-1874 CT visible human dataset – see Figure 48 – from the National Institute of Health) using approximately 128 rendering nodes and 127 compositing nodes at Supercomputing '95 in San Diego. The rendering times for a 512-by-512 image are on the order of 5 seconds/frame. It is worth pointing out that the main bottleneck for this dataset is reading the 500MB of data from the Paragon disks. Currently, it takes around 15 minutes.

Our next step is to extend the system to render the full RGB visible human (14GB) with high temporal resolution (a 72-frame rotation uses 5 degree increments. Smaller increments are desirable, but they greatly increase the size of the animation files). This will require the use of parallel I/O, a capability that currently we do not have, and dedicated use of a very large parallel machine, such as the entire 1840-node Intel Paragon at Sandia.

Figure 43 is a volume rendering of a 1024-by-1024-by-64 thymic gland tissue showing the thymic epithelial cells. Figure 44 is a volume rendering of a 100-by-110-by-92 single cell. The datasets were generated by immuno-fluorescence microscopy at the National Jewish Center for Immunology and Respiratory by C. Monks and prepared for visualization by deconvolution on Sandia's Intel Paragon by G. Davidson. The volume rendering animations were generated at multiple frames per second using PVR by B. Wylie. For further details see [80].

7.4 Conclusions

We have introduced the PVR system. The idea of developing PVR started out of frustration from trying to use a network of workstations and the Paragon as rendering engines for VolVis [7]. It was always clear that a pure distributed approach to building rendering environments would be much more powerful than special rendering tools with parallel capabilities. Here are some of the key features in our system:

- *Transparency* - PVR hides most of the hardware dependencies from the DVEs and the user.
- *Performance* - PVR provides high speed pipelined ray casting with a unique load-balancing scheme and mechanisms to fine tune performance for any given machine configuration.
- *Scalability* - All the algorithms used in the system were carefully chosen to be gracefully scalable. Scalability is not only with respect to the

machine size, but special care has been taken to allow for growth in dataset size and image size.

- *Extensibility* - The PVR architecture can be easily extended, making it easy for the DVE to add new functionality. Also, it is fairly easy for the user to add new functionality to the PVR shell and its corresponding kernel, allowing for user-defined "computational steering" coupled with visualization.

PVR introduces a new level of interactivity to high performance visualization. Larger DVEs can be built on top of PVR and yet be portable across several architectures. These DVEs that use PVR are given the opportunity to make effective use of available processing power (up to a few hundred processors), giving a range of cost/performance to end users. This is particularly important in the scientific research community, since most often the question is not *how fast but how much.* PVR provides a strong foundation for building cost effective DVEs.

As far as user interfaces are concerned, PVR introduces a much simpler way to create them. No longer does one have to spend time coding in X/MOTIF (or Windows) to create the desired user interface. The Tcl/Tk combination is much simpler, gives more flexibility, and is nearly as powerful as the other alternatives. Tcl/Tk is becoming as popular as UNIX shell programming. Different sites should be able to easily create and/or customize their own versions of the systems.

Even though we have completed a *usable* and efficient system, there is still a long wish list, on both the research and development front. We are currently

working on making the system stable enough for large scale availability. With that in mind, we are currently working on creating a more complete DVE (using VolVis as a reference) on top of PVR.

Some functionality is missing from PVR and needs to be incorporated. The most important element is probably the support for multiple data sets in a session. This would make the load-balancing scheme much more complicated, and simple heuristics might not generate well-balanced decomposition schemes. If the volumes were allowed to overlap (as in VolVis), the problem would be even harder, and the solution would require heavier processing on the compositing end. It might be necessary to have a reconfiguration phase each time a new volume is introduced into the picture. It is not yet clear how this could be done efficiently.

Command	Description
:s open *M:N*	*M* is an internet address; *N* is a port number.
:s close	Close the connection.
:s image window *W*	*W* is a Tk photo widget.
:s image callback *F*	*F* is a procedure to be called every time a new image is received.
:s image file *F*	*F* is the name of the local file name where the video stream is saved.
:s list status	Show the state of the connection and the value of internal variables.
:s set *Option Val*	Change system status.
:s set -dataset *D*	Sets the data set to be rendered.
:s set -cluster *C*	Sets the size of clusters.
:s set -group *G*	Used to group multiple clusters, for use in exploiting image-based parallelism.
:s set -imagesz *X,Y*	Sets the desired image resolution.
:s render rotate *X,Y,Z S,E:N*	Sends a rendering request. The axis of rotation and initial, end, and incremental angles are specified.
:s performance memory cluster	Returns the amount of dataset memory in each cluster.
:s performance comp cluster latency	Estimates the latency time to composite images in the current cluster configuration.

Table 2: *A list of a few external PVR commands. These commands can be typed interactively, placed in execution files, or embedded in applications.*

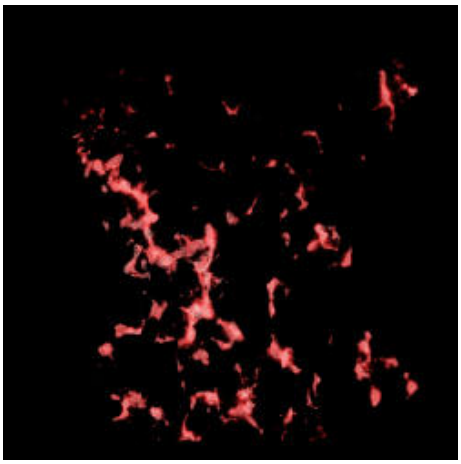


Figure 43: *Volume rendering of the Thymus tissue. An (512 × 512, 72 frame) animation was produced in just about 3 minutes using PVR on the Paragon.*

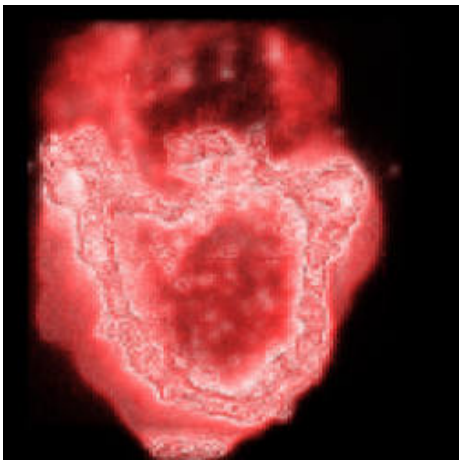


Figure 44: *A volume rendering showing T-cell receptors on an immuno-fluorescent microscopy dataset.*

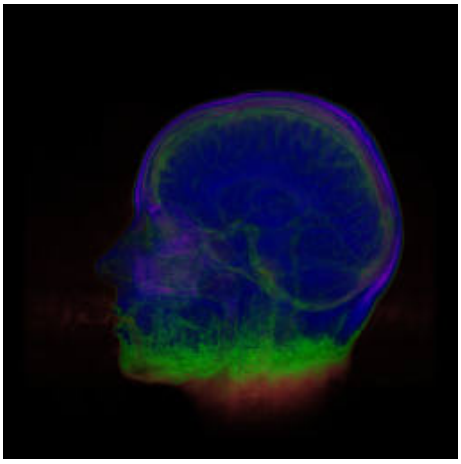


Figure 45: *Volume rendering of MR data from a human head using an unconventional transfer function in order to illustrate the flexibility of volume rendering.*

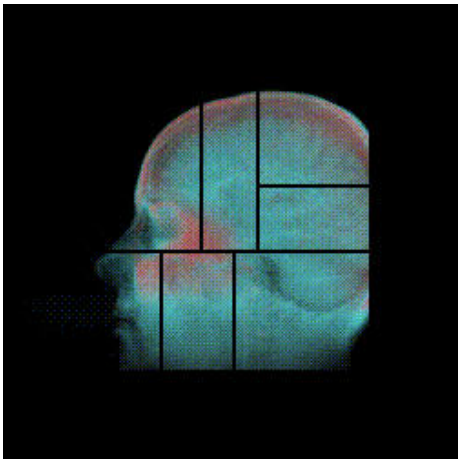


Figure 46: *A subdivision of the MR data for 8 processors is shown, illustrating our content-based load balancing.*

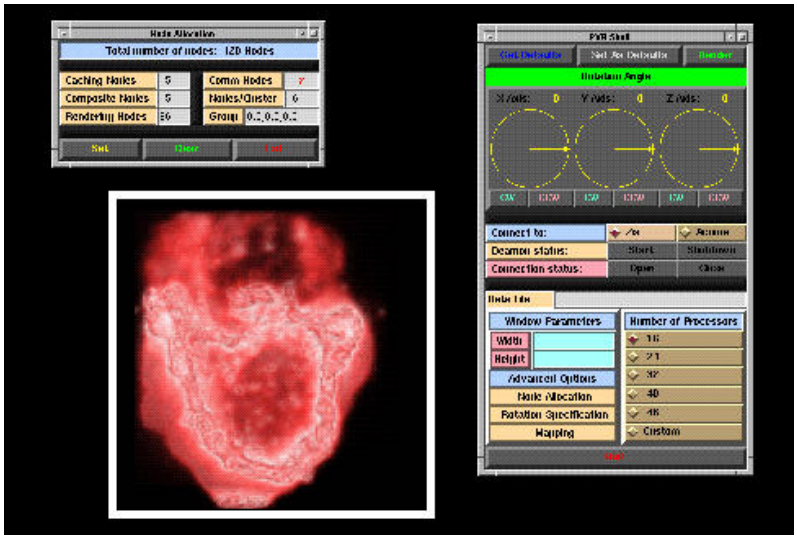


Figure 47: *A snapshot of the simple PVR GUI, with three windows. The main interface window on the right, where the user can specify general rotations. The cluster configuration window, on the left. The third window is the image of a cell calculated with PVR.*

Figure 48: *Volume rendering of the 512 × 512 × 1877 visible human.*

Chapter 8

Conclusions

In this thesis we described our work on efficient techniques for rendering large regular and irregular grids. Our contributions include the development of new algorithms and the parallelization of existing methods. Besides conceptual development, a big emphasis was given to the implementation of the techniques. In fact, over 50,000 lines of code were written specifically for the results presented here. A short review of the work presented follows.

First, we describe parallel methods for rendering regular grids. These algorithms have been implemented and are part of the PVR system [113], a complete high-performance parallel and distributed volume rendering system. PVR is being used at several institutions across the country, and it has successfully rendered very large datasets (e.g., on the order of half a gigabyte) at interactive rates.

We also describe a novel ray casting based method for rendering irregular grids that is two orders of magnitude faster than previous ray casting techniques. We discuss the practical and theoretical issues involved in rendering

irregular grids. We also propose a the parallelization of this rendering technique, and discuss the issues on how to implement and integrate it in the PVR system.

Finally, we present an algorithm for triangulating height-field terrain data. Besides the two-dimensional version of the algorithm, we show how to extend it to handle general polyhedral surfaces and sketch how to use it for simplifying irregular grids.

Several problems remained open in our work. One of our future goals is to integrate regular and irregular grids under the same framework within PVR. For this, it is necessary to implement better load balancing for the regular grids that generate no stalls in the pipeline, and also integrates the irregular grids rendering time (some accurate analytical model of the rendering time is needed in this case, in order to allow for uniform rendering time across all the clusters). This accurate prediction of rendering time, for different datasets and image sizes is a very challenging problem.

We have found that the most time-consuming operation in rendering irregular grids is the final depth sorting. In fact, by profiling our code, we are able to point out that re-calculating the 1D sorting of the cells inside the sweep-planes takes a big percentage of the time. We plan to explore a different formulation of the sweep algorithm, which is able to avoid the explicit re-calculation.

Greedy-Cuts is a very powerful concept that we plan to explore in detail in the future. Our terrain simplification implementation can be improved, and the polyhedral surface generalization needs to be finished. Several interesting questions remain, such as how to extend the Gauss map approach for

constant-time checking for violation of the terrain condition on the patches. Also, practical issues remain such as how to integrate texture interpolation on the simplification model. Higher dimensional variations need to be explored further, and tested thoroughly. We also plan to study its theoretical performance.

In the future, we plan to expand our research to other problems in which our techniques can be further explored. For example, we plan to use decomposition schemes (such as the ones used for parallelization) coupled with spectral wavelet methods to break up a given volume into multiple volumes, each of which is sampled at the most economical resolution (see [15] for more details). This way we should be able to decrease the rendering time, as well as storage for volumes. We hope to attack the polyhedral collision detection problem using our simplification techniques. Finally, we plan to extend our flow visualization work [109] to use a particle system in order to make it more robust to singularities in the flow field.

Bibliography

[1] P. K. Agarwal, M. J. Katz, and M. Sharir. Computing depth orders and related problems. In *Proc. 4th Scand. Workshop Algorithm Theory*, pages 1–12, 1994.

[2] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.

[3] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

[4] P. K. Agarwal and M. Sharir. Applications of a new partition scheme. *Discrete Comput. Geom.*, 9:11–38, 1993.

[5] P K. Agarwal and Subhash Suri. Surface approximation and geometric partitions. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 24–33, 1994.

[6] V. Anupam, C. Bajaj, D. Schikore, and M. Schikore. Distributed and collaborative visualization. *IEEE Computer*, 27(7):37–43, 1994.

[7] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. Volvis: A diversified volume visualization system. In *Visualization '94 Proceedings*, pages 85–92. IEEE CS Press, October 1994.

[8] R. Avila, L. Sobierajski, and A. Kaufman. Towards a comprehensive volume visualization system. In *IEEE Visualization '92*, pages 13–20. IEEE CS Press, 1992.

[9] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. volume 16, pages 21–29, July 1982.

[10] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite vc-dimension. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 293–302, 1994.

[11] E. Camahort and I. Chakravarty. Integrating volume data analysis and rendering on distributed memory architectures. In *1993 Parallel Rendering Symposium Proceedings*, pages 89–96. ACM Press, October 1993.

[12] I. Carlbom. Optimal filter design for volume reconstruction and visualization. In *IEEE Visualization '93*, pages 54–61, 1993.

[13] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *1993 Parallel Rendering Symposium Proceedings*, pages 81–88, 1993.

[14] B. Chazelle, H. Edelsbrunner, L. J. Guibas, R. Pollack, R. Seidel, M. Sharir, and J. Snoeyink. Counting and cutting cycles of lines and rods in space. *Comput. Geom. Theory Appl.*, 1:305–323, 1992.

[15] R. Chiou, M. Ferreira, A. Kaufman, and C. Silva. Using wavelets to extract information from volumetric data. In *International Conference on Information Systems Analysis and Synthesis*, pages 576–582, 1996.

[16] K. L. Clarkson. Algorithms for polytope covering and approximation. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 246–252, 1993.

[17] E. Coddington. *An Introduction to Ordinary Differential Equations.* Prentice-Hall, 1961.

[18] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[19] R. Crawfis and N. Max. Direct volume visualization of three-dimensional vector fields. *1992 Workshop on Volume Visualization*, pages 55–60, 1992.

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

[21] G. Das and D. Joseph. Minimum vertex hulls for polyhedral domains. *Theoret. Comput. Sci.*, 103:107–135, 1992.

[22] G. Das and M. T. Goodrich. On the complexity of approximating and illuminating three-dimensional convex polyhedra. In *Proc. 4th Workshop Algorithms Data Struct.*, volume 955 of *Lecture Notes in Computer Science*, pages 74–85. Springer-Verlag, 1995.

[23] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.

[24] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994.

[25] M. de Berg, M. Overmars, and O. Schwarzkopf. Computing and verifying depth orders. *SIAM J. Comput.*, 23:437–446, 1994.

[26] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C26–C27, 1995.

[27] L. De Floriani. A pyramidal data structure for triangle-based surface representation. *IEEE Comput. Graph. Appl.*, 9:67–78, March 1989.

[28] M. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[29] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 65–74, August 1988.

[30] H. Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. *Combinatorica*, 10:251–260, 1990.

[31] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice, Second Edition.* Addison-Wesley, Reading, Massachusetts, 1990.

[32] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics*, 13(2):199–207, August 1979.

[33] W. R. Franklin. Triangulated irregular network to approximate digital terrain, Section 2.3, Research Interests. Technical report, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, NY, 1994. Manuscript and code available on ftp://ftp.cs.rpi.edu/pub/franklin/, 1994.

[34] T. Fruhauf. Raycasting of nonregularly strutuctured volume data. *Computer Graphics Forum (Eurographics '94)*, 13(3), 1994.

[35] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[36] M. P. Garrity. Raytracing irregular volume data. volume 24, pages 35–40, November 1990.

[37] C. Giertsen and J. Petersen. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, 13(6):16–23, 1993.

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

[39] A. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[40] A. Glassner. *Principles of Digital Image Synthesis (2 Vols)*. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-276-3, San Francisco, CA, 1995.

[41] M. T. Goodrich. Efficient piecewise-linear function approximation using the uniform metric. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 322–331, 1994.

[42] M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths via balanced geodesic triangulations. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 318–327, 1993.

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

[44] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl.*, 3(4):383–415, December 1993.

[45] B. Hendrickson and R. Leland. The chaco user's guide (version 1.0). Tech. Rep. SAND93-2339, Sandia National Laboratories, Albuquerque, N.M., 1993.

[46] J. Hennesy and D. Paterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 1990.

[47] D. Hillis. *The Connection Machine*. MIT Press, 1985.

[48] W. Hsu. Segmented ray casting for data parallel volume rendering. In *1993 Parallel Rendering Symposium Proceedings*, pages 7–14. ACM Press, October 1993.

[49] V. Jacobson. Congestion avoidance and control. *Computer Communication Review*, 18(4):314–29, 1988.

[50] J. T. Kajiya. The rendering equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

[51] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 165–174, July 1984.

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

[53] R. Karia. Load balancing of parallel volume rendering with scattered decomposition. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

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

[55] P. Lacroute and M. Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In Andrew Glassner, editor,

Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994), Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM SIGGRAPH, ACM Press, July 1994.

[56] J. Lee. A drop heuristic conversion method for extracting irregular network for digital elevation models. In *GIS/LIS '89*, volume 1, pages 30–39. American Congress on Surveying and Mapping, 1989.

[57] J. Lee. Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models. *Intl. J. of Geographical Information Systems*, 5(3):267–285, 1991.

[58] M. Levoy. Efficient ray tracing of volume data. *ACM Transations on Graphics*, 9(3):245–261, 1990.

[59] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[60] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[61] M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, February 1990.

[62] P.-W. Liu, L.-S. Chen, S.-C. Chen, J.-P. Chen, F.-Y. Lin, and S.-S. Hwang. Distributed computing: New power for scientific visualization. *IEEE Computer Graphics and Applications*, 16(3):42–51, 1996.

[63] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer*

Graphics (SIGGRAPH '87 Proceedings), volume 21, pages 163–169, July 1987.

[64] K. Ma, J. Painter, C. Hansen, and M. Krogh. A data distributed parallel algorithm for ray-traced volume rendering. In *1993 Parallel Rendering Symposium Proceedings*, pages 15–22. ACM Press, October 1993.

[65] K. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[66] K. Ma. Parallel volume rendering for unstructured-grid data on distributed memory machines. In *IEEE/ACM Parallel Rendering Symposium '95*, pages 23–30, 1995.

[67] A. Maccabe, K. McCurley, R. Riesen, and S. Wheat. Sunmos for the Intel Paragon - A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group 1993 Annual North America Users' Conference*, October 1993.

[68] X. Mao, L. Hong, and A. Kaufman. Splatting of curvilinear grids. In *IEEE Visualization '95*, pages 61–68, 1995.

[69] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *IEEE Visualization '94*, pages 100–107, 1994.

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

[71] N. Max, R. Crawfis, and B. Becker. New techniques in 3D scalar and vector field visualization. In *First Pacific Conference on Computer Graphics and Applications*. Korean Information Science Society, Korean Computer Graphics Society, August 1993.

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

[73] N. L. Max. Efficient light propagation for multiple anisotropic volume scattering. In *Fifth Eurographics Workshop on Rendering*, pages 87–104, Darmstadt, Germany, June 1994.

[74] M. McCormick, T. DeFanti, and M. Brown. Visualization in scientific computing. Report of the NSF Advisory Panel on Graphics, Image Processing and Workstations, 1987.

[75] J. V. Miller, D. E. Breen, W. E. Lorensen, R. M. O'Bara, and M. J. Wozny. Geometrically deformed models: A method for extracting closed geometric models from volume data. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 217–226, July 1991.

[76] J. S. B. Mitchell. Approximation algorithms for geometric separation problems. Technical report, Dept, of Applied Math, University at Stony Brook, Stony Brook, NY, July, 1993.

[77] J. S. B. Mitchell, D. M. Mount, and S. Suri. Query-sensitive ray shooting. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 359–368, 1994.

[78] J. S. B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.

[79] J. S. B. Mitchell and S. Suri. Separation and approximation of polyhedral surfaces. In *Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms*, pages 296–306, 1992.

[80] C. Monks, P. Crossno, G. Davidson, C. Pavlakos, A. Kupfer, C. Silva, and B. Wylie. Three dimensional visualization of proteins in cellular interactions. In *IEEE Visualization '96*, pages 363–366, 1996.

[81] C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. In *1992 Workshop on Volume Visualization Proceedings*, pages 9–16. ACM Press, October 1992.

[82] H. Muller and M. Stark. Adaptive generation of surfaces in volume data. *The Visual Computer*, 9(4):182–199, January 1993.

[83] H. Neeman. A decomposition algorithm for visualizing irregular grids. volume 24, pages 49–56, November 1990.

[84] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *1993 Parallel Rendering Symposium Proceedings*, pages 97–104. ACM Press, October 1993.

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

[86] G. M. Nielson and B. Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. In *Visualization '91*, pages 83–91, 1991.

[87] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[88] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1993.

[89] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[90] C. Pavlakos, L. Schoof, and J. Mareda. A visualization model for supercomputing environments. *IEEE Parallel & Distributed Technology*, 1(4):16–22, 1996.

[91] M. Pellegrini. Ray shooting on triangles in 3-space. *Algorithmica*, 9:471–494, 1993.

[92] H. Pfister and A. Kaufman. Cube-4 – a scalable architecture for real-time volume rendering. In *ACM/IEEE Volume Visualization '96*, pages 47–54, 1996.

[93] T. Porter and T. Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

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

[95] J. Rowlan, E. Lent, N. Gokhale, and S. Bradshaw. A distributed, parallel, interactive volume rendering package. In *Visualization '94 Proceedings*, pages 21–30. IEEE CS Press, October 1994.

[96] S. Ramamoorthy and J. Wilhelms. An analysis of approaches to ray-tracing curvilinear grids. Tech Report UCSC-CRL-92-07, U. of California, Santa Cruz, 1992.

[97] P. Sabella. A rendering algorithm for visualizing 3D scalar fields. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 51–58, August 1988.

[98] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[99] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[100] L. Scarlatos. *Spatial data representations for rapid visualization and analysis*. Ph.D. thesis, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, 1993.

[101] L. Scarlatos and T. Pavlidis. Optimizing triangulation by curvature equalization. In *Proceedings of the 3rd 1992 IEEE Conference on Visualization, Visualization '92*, pages 333–339. IEEE, 1992.

[102] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using terrain features. In *Proceedings of the 1st 1990 IEEE Conference on Visualization, Visualization '90*, pages 168–175, IEEE Service Center, Piscataway, NJ, USA (IEEE cat n 90CH2914-0), 1990. IEEE.

[103] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographics coherence. *CVGIP: Graph. Models Image Process.*, 54(2):147–161, March 1992.

[104] P. Schroeder and J. Salem. Fast rotation of volume data on data parallel architectures. In *Visualization '91 Proceedings*, pages 50–57. IEEE CS Press, 1991.

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

[106] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH '92*, volume 26, pages 65–70, July 1992.

[107] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, New York, 1995.

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

[109] C. Silva, L. Hong, and A. Kaufman. Flow surface probes for vector field visualization. In G. Nielson, H. Mueller, and H. Hagen, editors, *Scientific Visualization: Overviews, Methodologies and Techniques*, IEEE CS Press, 1997.

[110] C. Silva and A. Kaufman. Parallel performance measures for volume ray casting. In *IEEE Visualization '94*, pages 196–203, 1994.

[111] C. Silva, J. Mitchell, and A. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *IEEE Visualization '95*, pages 201–208, 1995.

[112] C. Silva, J. Mitchell, and A. Kaufman. Fast rendering of irregular grids. In *ACM/IEEE Volume Visualization '96*, pages 15–22, 1996.

[113] C. Silva, A. Kaufman, and C. Pavlakos. PVR: High-performance Volume Rendering. *IEEE Computational Science and Engineering*, To Appear, 1996.

[114] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, 1994.

[115] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[116] L. Sobierajski and R. Avila. A hardware acceleration method for volume ray tracing. In *IEEE Visualization '95*. IEEE CS Press, 1995.

[117] L. Sobierajski and A. Kaufman. Volumetric ray tracing. In *Volume Visualization Symposium '94*, 1994.

[118] D. Speray and S. Kennon. Volume probes: Interactive data exploration on arbitrary grids. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 5–12, November 1990.

[119] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *ACM Volume Visualization Symposium '94*, pages 83–89, 1994.

[120] S. Suri. On some link distance problems in a simple polygon. *IEEE Trans. Robot. Autom.*, 6:108–113, 1990.

[121] C. Upson and M. Keeler. VBUFFER: Visible volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 59–64, August 1988.

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

[123] A. Van Gelder and J. Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. In *IEEE Visualization '93*, pages 70–77, 1993.

[124] A. Varshney. *Hierarchical Geometric Approximations.* Ph.D. thesis, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994. TR-050-1994.

[125] L. Westover. *SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm*. Ph.D. thesis, University of North Carolina at Chappel Hill, 1991.

[126] L. Westover. Footprint evaluation for volume rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 367–376, August 1990.

[127] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. Direct volume rendering of curvilinear volumes. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 41–47, November 1990.

[128] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 275–284, July 1991.

[129] P. Williams. Visibility ordering meshed polyhedra. *ACM Transations on Graphics*, 11(2), 1992.

[130] P. L. Williams and N. Max. A volume density optical model. *1992 Workshop on Volume Visualization*, pages 61–68, 1992.

[131] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, pages 19–28, 1992.

[132] R. Yagel, D. Reed, A. Law, P.-W. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Volume Visualization Symposium '96*, pages 55–62, 1996.

[133] K. Zuiderveld. *Visualization of Multimodality Medical Volume Data using Object-Oriented Methods.* Ph.D. thesis, University of Utrecht, The Netherlands, 1995.

[134] K. Zuiderveld, A. Koning, and M. Viergever. Acceleration of ray-casting using 3D distance transforms. In *Visualization in Biomedical Computing '92*, pages 324–335. SPIE, 1992.