INTERACTIVE DIRECT VOLUME RENDERING OF CURVILINEAR AND UNSTRUCTURED DATA

BY

PETER LAWRENCE WILLIAMS

B.S., University of California, Berkeley, 1960
M.S., University of Lowell, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGMENTS

A very special thanks to my advisors, Dennis Gannon and Herbert Edelsbrunner, and my unofficial advisor at NCSA, Michael Norman. Thanks also to the members of my committee, L.V. Kale, William Kubitz, and David Kuck.

The financial support of the Center for Supercomputing Research and Development and the National Center for Supercomputing Applications is gratefully acknowledged.

A special thanks to Donna Cox, Melanie Loots, Nelson Max, Peter Shirley, and Allan Tuchman for their support with this work, and also to Silicon Graphics Inc. for an equipment loan.

Thanks also to the many people at the University of Illinois for advice and support: Henry Neeman, Kyle Gallivan, Cathy Warmbier, Randall Bramley, Kelvin Sung, Parris Egbert, Ernst Mücke, Ray Idaszak, Mike Krogh, Mat Arrott, Bill Sherman, Gautam Mehrotra, George Francis, Mark Washburn, Tiow-Seng Tan, Tony Baylis, Randy Flight, Dennis Parsons, Charles Tucker, Bob Haber, Kurt Larson, Terri Stewart, Mike McNeill, Amy Swanson, Don Hearn, Harald Rosenberger, Dan Reed, J. Dantzig, Joyce Woodworth, Uday Reddy, Mark Bajuk, Fen-Lien Juang, Carol Song, Chris Song, Dan Brady, Amhed Sameh, Dave Semarero, S. Balachandar, George Cybenko, Linda Ashby, Lou Wicker, Paul Saylor, Michael Heath, John Gray.

Thanks to the people in the graphics, computational geometry, and mesh generation community who supplied me with data sets or gave me information on their work, and/or feedback on my own: Tim Baker, Sia Meshkat, Vincent Argiro, Dimitri Mavriplis, Marshall Merriam, Bruno Nitrosso, Tracy Williams, Sharon Rose Fisher, Chris Wagner, Mike Chapman, Tom O'Connor, Rob Wolff.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Volume rendering[1] is used to show the characteristics of the interior of a solid region in a 2D image [31]. This thesis focuses on volume rendering as a technique for visualizing three dimensional space-filling[2] scientific data sets.

A scientific data set may consist of the results from a supercomputer simulation (a computational data set) or it may be a set of observed data (scattered or empirical data).[3] These data sets are defined over a mesh.[4]

Meshes can be classified by the structure of their cells as rectilinear, curvilinear or unstructured (irregular). Curvilinear and unstructured meshes are also referred to as nonrectilinear meshes. These terms as well as others are defined in Chapter 2. Two dimensional examples of each type of mesh are shown in Figure 1.1.

[1]Also known as volumetric rendering.

[2]Three dimensional space-filling data means the data is defined over a region of non-zero measure, also referred to as a volume of E^3, as opposed to over 3D surface or shell. More formally, the region is a 3-manifold embedded in E^3. Such data is sometimes referred to as a volume of data or volumetric data.

[3]A typical simulation is the use of the finite element method to solve a problem in computational science such as computational fluid dynamics. Scattered data sets come from various scanning methods used in areas such as the earth sciences or biomedicine.

[4]Scattered data has no specified connectivity between the data points. For the purpose of visualization, the data points can be triangulated, resulting in an unstructured (irregular) mesh.

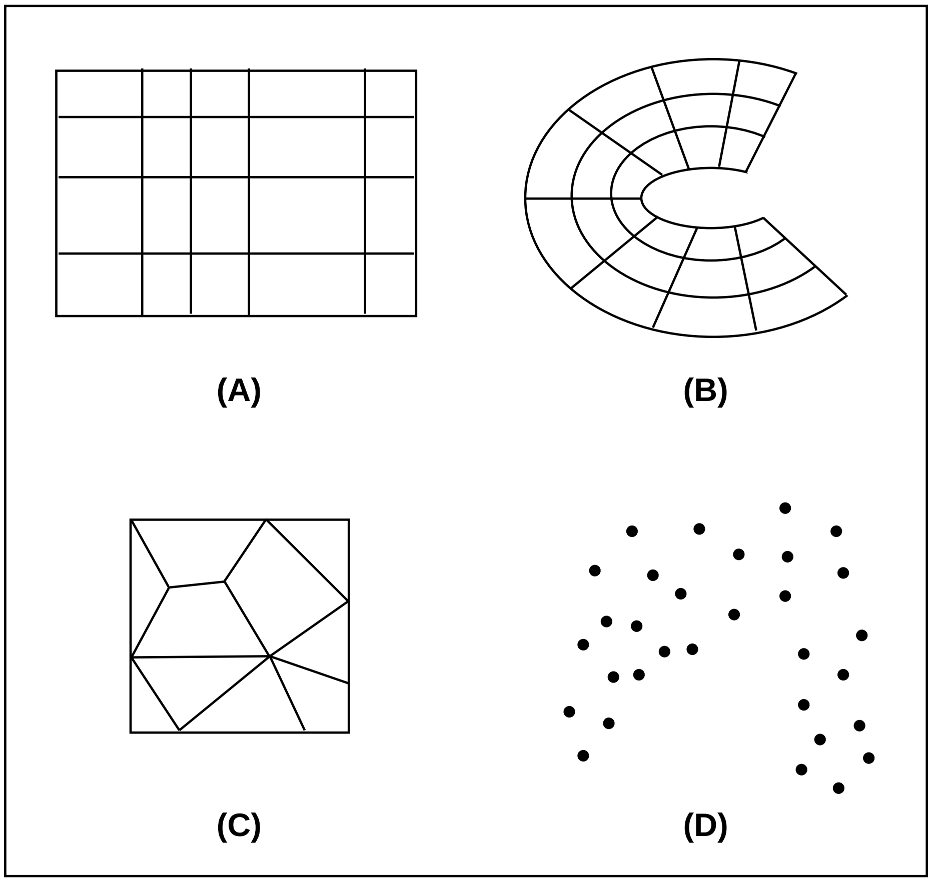


Figure 1.1: Two dimensional examples of (A) a rectilinear mesh, (B) a curvilinear mesh, (C) an unstructured (irregular) mesh, and (D) the points of an empirical or scattered data set. Scattered data can be treated as an unstructured mesh if its points are triangulated.

A good deal of attention has been given to volume rendering rectilinear data sets [17, 20, 36, 47, 55, 57, 59, 60, 61, 66, 76, 88, 89, 93, 97, 101, 103]. However, much less has been published dealing with nonrectilinear data [9, 38, 40, 61, 93, 94, 96, 102, 104, 105]. Nonrectilinear data is often rendered volumetrically by first interpolating it to a rectilinear mesh. The focus of this thesis is on volume rendering nonrectilinear data without interpolating it to a rectilinear mesh.

There are three basic techniques for displaying volume data [60], the use of 2D cross sections or slices of the data (*cross section rendering*); 3D level surfaces (*isosurfaces*) or *threshold rendering*, which can either be opaque or semitransparent; or *direct volume rendering* where a 2D projection of a colored semitransparent 3D volume or cloud is displayed. We concentrate on direct volume rendering.

Direct volume rendering is a method for rendering 3D scalar fields by directly displaying the data without first extracting intermediate representations, such as isosurfaces. The volume of data as a whole is rendered. One way to do this is to display a 2D projection of a colored semitransparent 3D volume or cloud, where the color and opacity are functions of the scalar field. These functions can be used to highlight desired features in the data, such as the extrema or hot spots, or other regions of interest such as shock waves. Thus a holistic view of the entire field can be given with the brightly colored extrema gleaming through the cloud. A feeling for the spatial orientation of the field and the relative locations of areas of interest is given by rotating the image, hence the importance of interactivity.

When discussing the optical properties of this colored semitransparent cloud or volume, it is sometimes referred to as a *volume density.* This subject is discussed at length in Chapter 3.

There are three classes of techniques for direct volume rendering: ray tracing or ray casting [9, 36, 55, 59, 60, 89, 97, 102], projection methods [17, 57, 61, 93, 97, 100, 101, 103, 104, 105], and hybrid methods [38, 40].

In ray tracing, rays are cast out from the viewer through the screen pixels. The contributions to the pixel from the points [59] or regions [89] along the ray are calculated. This approach is simple to implement, but can be liable to the aliasing problems common to ray tracing. Ray traced images typically take from 15 min to several hours to generate and their complexity depends on the size of the image in pixels. It is possible that the aliasing problem may be overcome if adaptive supersampling is used [70].

This thesis focuses on the use of projection methods for nonrectilinear data where the data is rendered without interpolating it to a rectilinear mesh. I refer to this henceforth as *Direct Projection Volume Rendering* (DPVR). Methods for DPVR are discussed in [61, 93, 104, 105].

In DPVR, each cell of the mesh is projected onto the screen in back-to-front or front-to-back order. To do this, an algorithm is required to visibility order the cells of the mesh. The cell's color and opacity contribution to each pixel is calculated and then blended with the pixel's existing color and opacity.

If the cells are output in layers from back to front, then the image unfolds as if a cutting plane perpendicular to the line of sight was sweeping over the image towards or away from the viewer. During the rendering process, useful information can be gained by watching the image being generated. When the data sets are large, it is as if one is watching an animation.

A very accurate, but computationally intensive, DPVR algorithm is given by Max, Hanrahan and Crawfis [61]. A fast approximation to this process, sometimes called a *splatting* algorithm, is given by Shirley and Tuchman [93]. The Shirley and Tuchman splatting algorithm, which they refer to as the Projected Tetrahedra (PT) Algorithm, is used as the basis for the work herein. Wilhelms and Van Gelder [103] discuss a number of issues very relevant to DPVR.

A brief overview of splatting is given here; it is described in more detail in Chapter 6. In this technique, each cell is projected onto the screen in visibility order from back to front to build up a semitransparent image. Westover [100] first referred to this process as splatting, as in splatting a snowball against a wall. See Figure 1.2. The contribution of each cell to the image is proportional to the thickness of the splat. This means the opacity is zero at the periphery of the splat.

The splat is rendered as a set of up to four triangles which have a common vertex at the point of maximum thickness of the splat. At this common vertex, the opacity is nonzero; at all other vertices the opacity is zero. The opacity and color at the vertices are interpolated over the splat. Wilhelms and Van Gelder [103] describe three possible interpolation methods for this purpose.

To highlight areas of interest in the scalar field and de-emphasize other areas, user-specified color and density transfer functions, are used to map the scalar field value to a color and density. As explained in Chapter 3, the density is used in a ray integration process to calculate the opacity. Some typical transfer functions are shown in Figure 1.3. Typically, the color and density maps are implemented as lookup tables.

The goal of this thesis is: (1) to develop a method for visibility ordering the cells of meshes of any shape and cell structure, and (2) to investigate techniques for achieving interactive performance with a DPVR splatting algorithm, even when the data sets are very large. The techniques investigated are parallelization, graphics hardware support, a suite of splatting approximations, and mesh filtration.

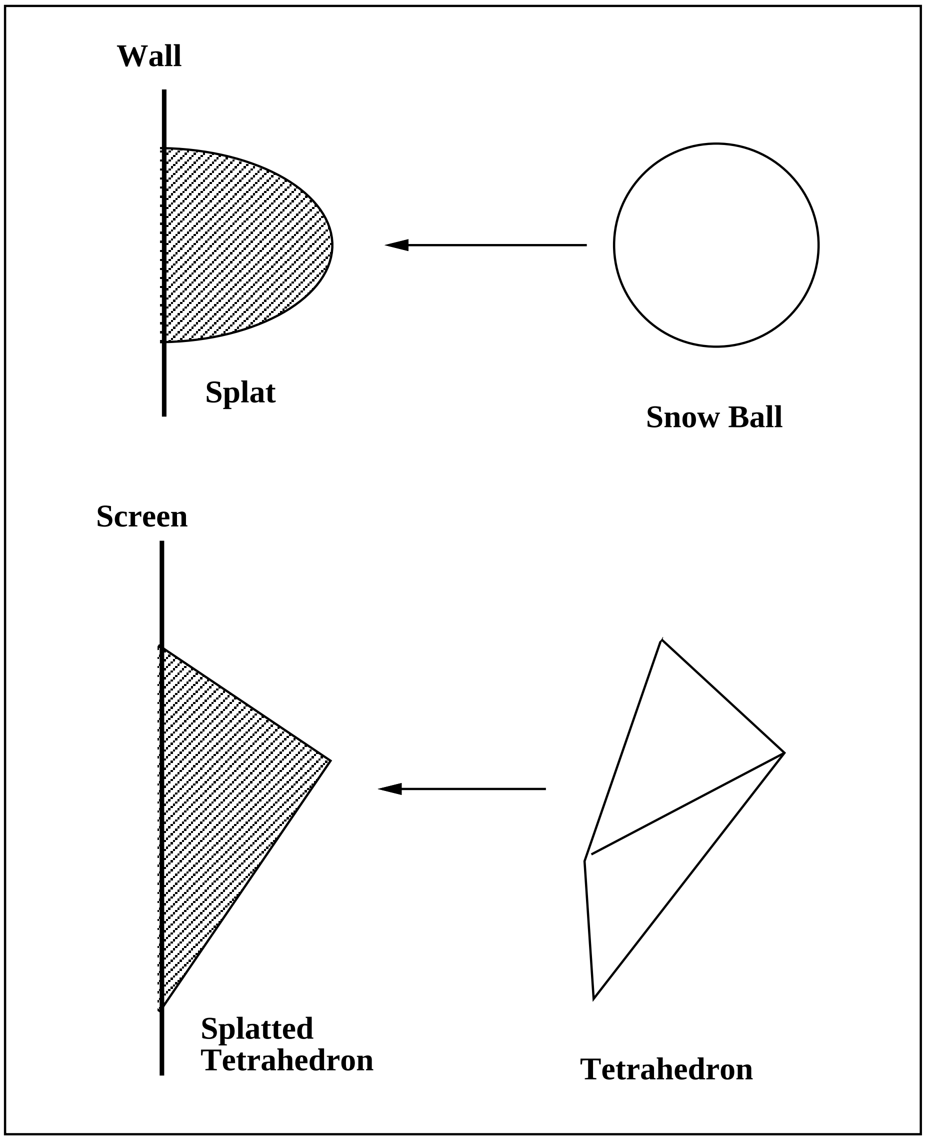


Figure 1.2: The splatting process.

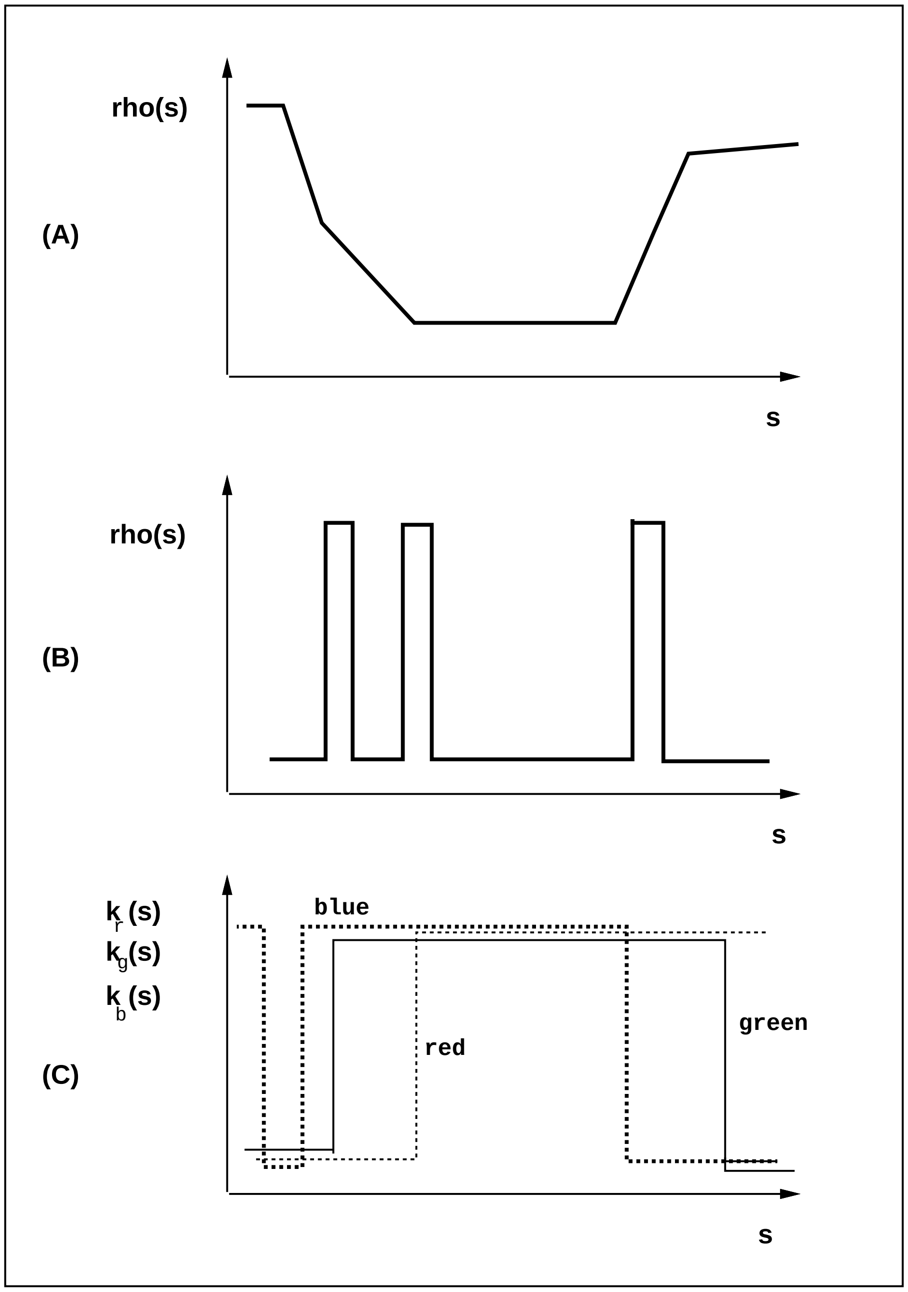


Figure 1.3: Example transfer functions which map a scalar value s, such as temperature or pressure, into density ρ and color κ. (A) Shows the density function being used to emphasize the extrema and de-emphasize the mid band. (B) Shows the density transfer function used to create three isosurfaces. (C) Shows color transfer functions. These color transfer functions were used in the creation of the image shown in Figure 3.5.

The suite of splatting approximations along with the PT algorithm and highly accurate volume rendering methods, such as ray tracing and the DPVR algorithm by Max, Hanrahan and Crawfis [61], form a hierarchy of rendering methods that tradeoff image accuracy/quality and generation time.

Parallel volume rendering algorithms that include visibility ordering for both convex and nonconvex irregular meshes are investigated and results are given for several versions of parallel algorithms. A performance analysis of one of these algorithms on a high performance MIMD 3D graphics workstation is presented.

Even with parallelization, fast graphics hardware and the use of rendering approximations, it still may not be possible to interactively generate images of very large data sets. To achieve this goal, it may be necessary to reduce the number of cells rendered by filtering techniques.

Projection methods were chosen for this research because they seemed to have the best potential for interactive performance. Using the methods described herein, the DPVR splatting algorithm has generated volume rendered images of data sets with over 1,000,000 cells interactively (in less than 15–30 seconds)[5]. Using the filtering methods described herein, this performance is possible for even larger data sets. These results justify the choice of a projection method, since they are significantly faster than any results published to date for any of the methods of direct volume rendering.

Giertsen's papers [38, 40] on the hybrid method of volume rendering do not state the complexity of his algorithm; and it is difficult to calculate bounds based on the information published. It appears that the complexity depends not only on the size of the mesh but also on the size of the image in pixels, as in ray tracing. The timings reported are for relatively small

[5]I distinguish between *interactive* and *real-time* performance. By real-time, I mean an object on the screen can be rotated smoothly under the control of a mouse.

meshes. Based on this limited information, it appears that this hybrid method is an order of magnitude slower than the algorithms reported on herein. Some advantages of this hybrid algorithm are that it allows adjacent cells in the mesh not to be aligned, it doesn't require a depth buffer, and it can get arbitrary precision in alpha blending.

Since considerable research is being devoted to parallel ray tracing, it will be interesting to compare the results reported herein with those from a ray tracing volume rendering algorithm for irregular volumes, such as the one reported by Garrity [36], that uses the latest parallel ray tracing algorithms and the most efficient hardware available.

The thesis is organized as follows. In Chapter 2, we define a number of terms and discuss concepts needed for future chapters. Chapter 3 discusses various optical models that are used as a theoretical basis for volume rendering; and a new model for interactive volume rendering is introduced. Algorithms and techniques for visibility ordering the cells of nonrectilinear meshes are presented in Chapter 4. Chapter 5 discusses hardware support for high performance polygon rendering and its role in interactive DPVR. Cell projection methods are discussed in Chapter 6 and a suite of fast approximations to the DPVR splatting process is introduced; results from the various approximation methods are presented. In Chapter 7, parallelization of the visibility ordering algorithms, the PT algorithm, and the suite of splatting approximations are discussed. Filtering techniques are discussed in Chapter 8. In Chapters 9 and 10 respectively, it is shown how the visibility ordering algorithms, presented in Chapter 4, can be used to solve the spatial point location problem and how they can assist in domain decomposition of finite element meshes for parallel processing. Finally, in Chapter 11, we discuss how well the goal of this thesis was achieved and what work still needs to be done.

CHAPTER 2

PRELIMINARY DEFINITIONS, TERMS AND CONCEPTS

In this Chapter, we define some terminology and concepts relevant to this thesis.

2.1 Meshes, Fields and Manifolds

In Chapter 1, for the purpose of visualization, meshes were classified by the structure of their cells as rectilinear, curvilinear or irregular (unstructured). For volumetric data, a *rectilinear mesh* is one whose cells are right-angled parallelepipeds. Examples of nonrectilinear meshes are curvilinear meshes and irregular meshes. A *curvilinear mesh* is a nonorthogonal mesh in $\mathbf{E}^3$ which is a transformation of a rectilinear mesh in some curvilinear coordinate system. Haber, Lucas and Collins [45] classify meshes by their topology and geometry.

A mesh can have a regular or irregular topology and a regular or irregular geometry. A curvilinear mesh has a regular topology but may have either a regular or irregular geometry. An *unstructured* or *irregular mesh* is one which has an irregular topology and an irregular geometry. Examples of different classes of meshes are shown in Figure 1.1.

A mathematically rigorous definition of a regular or irregular geometry and/or topology is not given. However, intuitively, a regular geometry is one in which the vertices of the mesh form a lattice. A regular topology means the connectivity of the mesh is regular.

For the purpose of visibility ordering, it is assumed that if any cells have curved bounding surfaces, then these surfaces will be approximated by flat surfaces, and if any cells are nonconvex, then these cells will be decomposed into convex cells.

A scalar function $f(x, y, z)$ defined over some domain D implies that at every point $p = (p_1, p_2, p_3)$ of D the function defines a scalar given by $f(p) = f(p_1, p_2, p_3)$. The totality of points p and scalars $f(p)$ is called a *scalar field.* If the field is defined at every point of D then it is a *continuous scalar field.* A *discrete scalar field* is a field which is defined at a discrete set of points in D.

Formally, the domain D, referred to above, is a manifold. A *manifold* M is a set of points in $\mathbf{E}^n$ so that each point of M has a neighborhood homeomorphic to $\mathbf{E}^k$. Intuitively, each point in the manifold has a local environment that is like a piece of $\mathbf{E}^k$. M is called a k-dimensional manifold embedded in n-space. The mapping from $\mathbf{E}^k$, the *coordinate domain*, to $\mathbf{E}^n$, the *embedding space*, is called a *parametrization*, the inverse mapping is called a *chart.*

The calculus of manifolds serves as a useful mathematical model for scientific visualization because it allows for a uniform treatment of data (scalar, vector or tensor) defined on meshes of different dimensionality, geometry and topology. Haber, Lucas and Collins [45] have developed this model to some extent.

For example, a curvilinear mesh, as used in the finite difference method, exists as a rectilinear mesh in the coordinate domain. An irregular mesh with distorted elements, from the finite element method, exists as an undistorted irregular mesh in the coordinate domain where the parametrization is the isoparametric mapping function. Often visualization calculations can be done more efficiently in the coordinate domain, where the mesh is simpler, than in the embedding space.

A scalar field defined at the nodes or vertices of a mesh is an example of a discrete scalar field. An interpolation function can be used to interpolate the nodal field values over the boundary and interior of each cell. Thus, a discrete scalar field along with an appropriate interpolation function becomes a continuous scalar field. Interpolation functions are discussed in Chapter 3.

2.2 The Voxel Model

A *voxel* or volume element is a small right-angled parallelepiped whose interior is considered to have one color, density, or scalar value, as the case may be. This is analogous to a *pixel* or picture element in 2D which is a small rectangle whose interior is all the same color.

Rectilinear data sets are often referred to as *voxel data*, and the visualization model for rectilinear data as the *voxel model*. Much has been published on the voxel model of volume rendering [17, 36, 55, 57, 59, 60, 61, 89, 93, 97, 101, 103]. Biomedical data has usually been the center of attention.

The voxel model is not generally applicable to curvilinear or irregular data unless the data is interpolated onto a rectilinear mesh. Therefore, many direct projection volume rendering techniques developed for the voxel model are not directly applicable to curvilinear or irregular data.

Interpolating nonrectilinear data to a rectilinear mesh has the drawback that if the non-rectilinear mesh is graded then the mesh scale of the rectilinear mesh needs to be as small as the finest gradation of the nonrectilinear mesh. This can be very expensive for meshes with highly refined regions. Some workers are investigating hierarchical rectilinear meshes, using a quadtree or octree concept, which may compensate for this problem [102]. If a satisfactory solution can be found, then the many volume rendering techniques developed for the voxel model will become applicable to nonrectilinear meshes.

The goal of this thesis is to visualize irregular volume data without interpolating the data to a rectilinear mesh.

CHAPTER 3

VOLUME DENSITY OPTICAL MODEL

3.1 Introduction

An exact simulation of light interacting with a volume density or cloud is quite complex and requires the use of Radiative Transport Theory [10, 53]. However, for the purpose of scientific visualization, especially interactive previewing, less complex simulations are satisfactory.

The optical model is the most crucial part of a volume renderer but it also can be the most confusing part. Therefore it is important that the underlying model be clearly understood. Current models such as in [89, 103] lack some generality and/or are not easy to comprehend. This chapter presents a new continuous model which is rigorous and quite general, yet is intuitive and easy to understand.

The next section discusses an early cloud model upon which many subsequent cloud models are based. Then Section 3.3.1 reviews the particle model and also makes some aspects of its derivation more rigorous. In Section 3.3.2, the continuous optical model for a volume density is presented. This model is suitable either for ray tracing or for projection methods and allows maximum flexibility in setting color and opacity. An expression for the light intensity along a ray through a volume, in terms of six user-specified transfer functions, three for optical density and three for color, is derived. Closed form solutions under several different assumptions are presented, including a new exact result for the case that the transfer functions vary piecewise linearly along a ray segment within a cell. A method is described which allows isosurface shading within a volume rendering.

3.2 Early Cloud Models

One of the first computer graphics models for clouds was reported by James Blinn [4] of the Jet Propulsion Lab. He described a method to synthesize an image of the rings of the planet Saturn using data from Voyager 1. The rings of Saturn consist of clouds of reflective ice particles in orbit about the planet.

Blinn used his model to calculate the amount of light reflected/transmitted by the cloud, that is, to show the effect of light incident on the cloud from the same side as the viewer and also from the opposite side. The model assumes a cloud of spherical reflecting particles positioned randomly in a layer as shown in Figure 3.1 (A).

Blinn's model deals with the scattering, shadowing and transmission of light propagating through the cloud. It assumes that a ray of light is reflected (scattered) by only a single particle, i.e. multiple reflections are considered negligible. This simplifying assumption will be true if the reflectivity (albedo) of each particle is small (less than 0.3). This model also deals with the shadowing or blocking effect of other particles after a light ray has been scattered by a single particle. See Figure 3.1 (B). And, it deals with the transparency (transmittance) of the cloud layer, that is, the amount of light coming from behind the cloud not blocked off by particles. See Figure 3.1 (C).

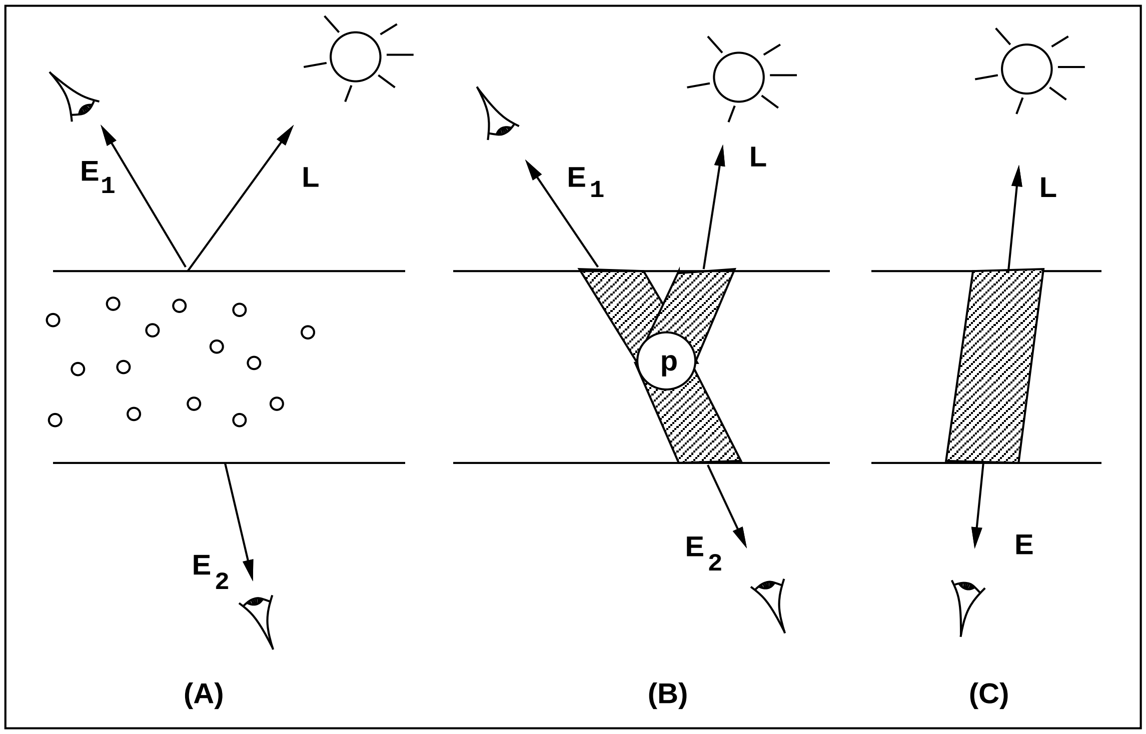


Figure 3.1: (A) Geometry of Cloud Layer. (B) Scattering Conditions. Shaded areas must contain no particles in order for the light ray to enter from L and escape to E_1 or E_2 after being scattered by particle p. (C) The transparency of the layer is the probability that the shaded area has no particles in it.

In order for light reflected from particle p in Figure 3.1 (B) to be visible, there must be no other particles in the shaded area. Statistically, the attenuation of light traversing the shaded volume V is $P(0;V)$, the probability of zero particles in the volume V. If n is the number of particles per unit volume, then the expected number of particles in V is nV. If n is small, the distribution of particles can be modeled by a Poisson distribution $P(x;V) = \frac{e^{-\lambda}\lambda^x}{x!}$. In this case, λ is the expected number of particles in V, so $\lambda = nV$, and so $P(0;V) = e^{-nV}$. The exponent is sometimes referred to as the *optical density* or *optical depth* ρ. Therefore, in this model, light passing through an absorbing medium is attenuated by the factor $e^{-\rho}$.

Kajiya and Von Herzen [50] give an alternative model which deals with multiple scattering against particles with high albedo; and they further develop Blinn's low albedo model and give a ray tracing algorithm for it. Light propagating through clouds is also discussed by Max [62, 63], Rushmeier and Torrance [87] and Ebert and Parent [20]; however, these techniques are not directly applicable to volume rendering. Ruder et al [86] discuss the use of line of sight integration for visualization of 3D scalar fields.

3.3 Current Volume Density Models

Up to this point in the development, the light sources have been outside the cloud and the model has described how the particles in the cloud scatter, absorb and transmit this light.

For the purpose of volume rendering for scientific visualization, a slightly different volume density model is used in which the cloud itself emits light. Two basic models can be used. Both yield very similar results but differ in the degree of flexibility in setting color and opacity. In the first model, the *particle model*, the scalar field being visualized is modeled as a cloud of light emitting particles. In the other model, the *continuous model*, the scalar field is represented as a cloud expressed as continuous glowing medium. Each point of the medium both emits and absorbs light.

The two models are really two different explanations of the same physical phenomenon. For a single transfer function, which is all the particle model allows, the two models give the same mathematical formula for the derived intensity. By neglecting shadowing on the way in, Blinn's single scattering model turns out to be the same as the particle model if the phase factor is neglected.

For the remainder of this chapter, it is assumed that the volume over which the scalar field is defined is subdivided into cells and that the scalar field is continuous over the volume.

3.3.1 The Particle Model

Paolo Sabella [89] first described a particle model for volume rendering which he called the density emitter model. It is based on Blinn's model but assumes the particles emit their own light, rather than scattering light from a source. This model is not related to Reeves [81] particle system in which particles are modeled individually. Sabella models the density of particles, not the particles themselves. The size of the particle is considered to be small compared to other dimensions so the density of the particles can be regarded as a continuous function.

In Sabella's model, the density of particles at any point $a = (x, y, z)$ is defined by considering a volume element of the cloud dV centered at a. If dV_P is the volume occupied by the particles in dV, then the density function is defined to be $\rho(x, y, z) = dV_P/dV$. If v_p is the volume of a single particle, then the expected number of particles in a region R of the cloud is:

$$N_R = \int_R \frac{dV_P}{v_p} = \int_R \frac{\rho(x, y, z)\, dV}{v_p} \tag{3.1}$$

A volume rendered image is created by setting a pixel's color to the intensity of light perceived by the eye along a ray from that pixel to the eye through the volume. Consider a cylinder with cross section σ whose axis is a ray to the eye, parameterized by length t, which enters the cloud at t_1 and exits at t_2. Assume the density function is parameterized along the ray as $\rho(x(t), y(t), z(t))$, or just as $\rho(t)$. An infinitesimal segment S of this cylinder, centered at t and of length dt, has volume $dV = \sigma\, dt$, and contains $N_{dV} = \frac{\rho(t)\, dV}{v_p}$ particles. If the particles are all spheres with radius r, then $v_p = \frac{4}{3}\pi r^3$, and each has projected area πr^2. So if they all glow diffusely on their surfaces with intensity κ, the total light power crossing the front surface of S is:

$$\kappa\pi r^2 N_{dV} = \kappa\pi r^2 \rho(t)\sigma\, dt/(\frac{4}{3}\pi r^3) = \frac{3\kappa\sigma}{4r}\rho(t)\, dt$$

The power is distributed over an area σ, so the intensity (power per unit area) contributed by this segment is $\frac{3\kappa}{4r}\rho(t)\, dt$. We have assumed dt is infinitesimal, so that the particles do not occlude each other. But on the path from the interior position t to the front edge t_2 of the cloud, occlusion can take place. To calculate the probability that this ray from t to t_2 is unoccluded, take another cylinder C about it, of radius r, the particle radius. The ray will be unoccluded if there are no particle centers within C. From Equation 3.1, the expected number N_C of particles inside C is:

$$N_C = \int_C \frac{\rho(x,y,z)\, dV}{v_p} = \int_t^{t_2} \rho(u)\pi r^2\, du/(\frac{4}{3}\pi r^3) = \frac{3}{4r}\int_t^{t_2} \rho(u)\, du \qquad (3.2)$$

If the density is small enough that the chances of mutual overlap are small, the particles can be assumed to be independently distributed. Then the probability $P(0; C)$ that there are no particle centers in the cylinder is given by the Poisson distribution formula $P(0; C) = e^{-N_C} = e^{-\frac{3}{4r}\int_t^{t_2} \rho(u)\, du}$. Therefore, the intensity of light reaching the eye due to dV is:

$$\frac{3\kappa}{4r}\rho(t)\, dt\, e^{-\frac{3}{4r}\int_t^{t_2} \rho(u)\, du}$$

The total intensity I reaching the eye due to all contributions between t_1 and t_2 is:

$$I = \frac{3\kappa}{4r} \int_{t_1}^{t_2} \rho(t) e^{-\frac{3}{4r} \int_t^{t_2} \rho(u)\, du}\, dt$$

If we assume the intensity at t_1 is zero, that is, there is a black background, and we let $\tau = \frac{3}{4r}$ and $c = \kappa\tau$, we get:

$$I = c \int_{t_1}^{t_2} \rho(t) e^{-\tau \int_t^{t_2} \rho(u)\, du}\, dt \tag{3.3}$$

For Equation 3.3 to hold, the density must be small as required by the Poisson distribution. The density can either be set equal to the scalar field S which is being visualized, or a single user defined transfer function f can be used, i.e. $\rho(x, y, z) = f(S(x, y, z))$. Sabella uses numerical methods to estimate the integral in Equation 3.3 by sampling along the ray.

Since the modeled light intensity is monochromatic, Sabella's images using the above model are gray scale pictures. He introduces color in a novel way by the use of the HSV color model. V, the value, is set to the intensity I in Equation 3.3. S, the saturation, is mapped to the distance D into the volume where the peak density is encountered along the ray. And H, the hue, is the entry in a user-defined color map corresponding to value of the peak density M. See Figure 3.2. Sabella also allows for lighting from several point light sources exterior to the volume by incorporating a diffuse lighting term into Equation 3.3.

When the indefinite integral $\int_0^t \rho(u)\, du$ can be tabulated or calculated analytically, and when c and τ are constants, Max, Hanrahan and Crawfis [61] show how Equation 3.3 can be simplified to a closed form expression which can be evaluated for each cell through which the ray passes. In addition, they take c and τ as vectors with three components, red, green and blue. This has the effect of scaling the single transfer function differently for each of the three components of color. The images created by this method are very accurate renderings of the volume density;

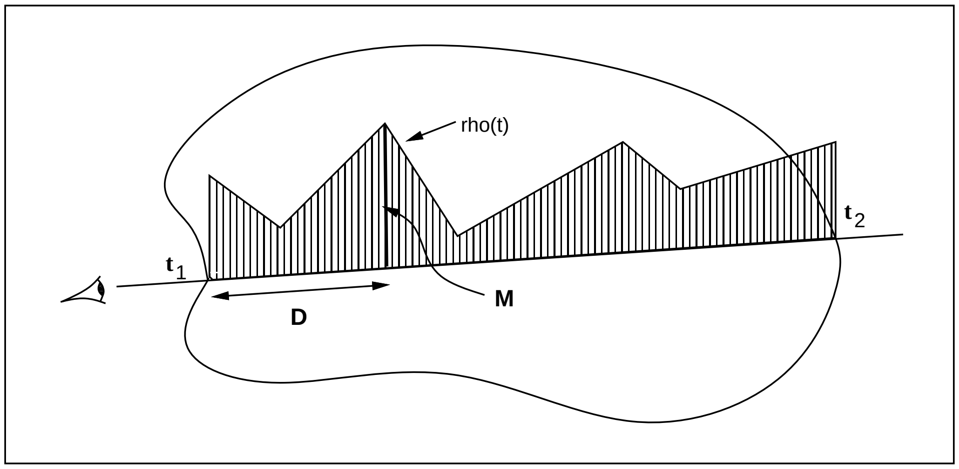


Figure 3.2: A 2D view of a density profile along a ray.

however, the process is computationally intensive and really is intended to be implemented in microcode.

It may be possible to modify Sabella's model to include three separate transfer functions for red, green and blue emitted light by assuming that a fraction of the particles emit light of a certain color. Then the density of the red-emitting particles, for example, will vary with a density function $\rho_r(x, y, z)$, and similarly for the blue and green particles. However, these three color particle densities need to be related to the attenuation particle density ρ which appears in the exponent in Equation 3.3. It seems easier to make this generalization in the continuous model which is described below.

3.3.2 The Continuous Model

We now consider the second model, the continuous model for a volume density. This formulation and development is new and has not been presented before. For visualization, it offers more flexibility than the particle model described in the last section. The volume renderer described in this thesis is based on this theoretical model. We benefit greatly from the earlier work of Shirley and Tuchman [93] and Wilhelms and Van Gelder [103].

The goal is to provide a simple, but accurate, formal model on which to base direct volume rendering of scalar fields defined on irregular meshes and to maximize the flexibility of use of transfer functions. It is intended for use with scalar field data from the finite element method, or scattered data, as opposed to scanned data sets where material classification is involved. The model is suitable either for ray tracing or for projection methods. The model is simplified to the bare minimum needed to clearly display the internal structure of the scalar field. No attempt is made to produce a highly realistic simulation of an actual cloud.

3.3.3 Model Development

In this model, the volume density can be thought of as a luminous or glowing gas cloud, such as neon or a glowing plasma, that selectively absorbs light of certain wavelengths and emits self-generated light. The gas cloud has two physical properties, optical density and chromaticity, both of which are functions of the scalar field being visualized.

The optical density of the gas at any point is wavelength λ dependent and is given by the function $\rho(x, y, z, \lambda) \geq 0$. The chromaticity is specified by a chromaticity function $\kappa(x, y, z, \lambda) \geq 0$. These two functions are defined in terms of six user-specified transfer functions ρ_r, ρ_g, ρ_b, κ_r, κ_g, κ_b, so for example, $\rho(x, y, z, red) = \rho_r(S(x, y, z))$, where S is the scalar field being visualized.

Let $P(t)$ be a ray to the eye parameterized by length t which enters the cloud at $P(t_0)$ and exits at $P(t_n)$, and let t be a point on the ray centered in the interval $(t + \frac{\Delta t}{2}, t - \frac{\Delta t}{2})$; see Figure 3.3. Let the notation $\rho(t, \lambda)$ stand for $\rho(P(t), \lambda)$, and similarly for $\kappa(t, \lambda)$.

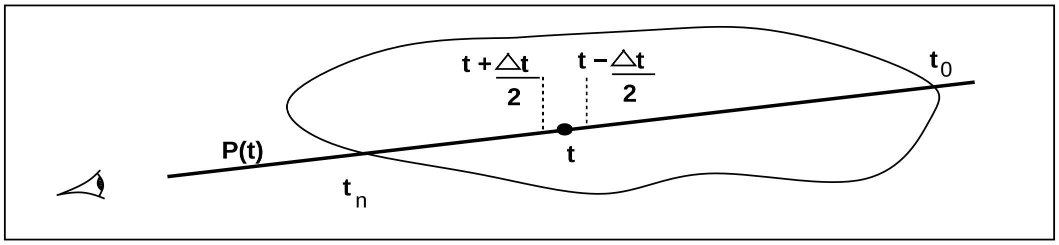


Figure 3.3: A 2D view of a cloud with a ray $P(t)$ to the eye parameterized by length t.

The meaning of the optical density $\rho(t, \lambda)$ is that, in the limit as Δt goes to zero, $\rho(t, \lambda)\Delta t$ is the fraction of light of wavelength λ entering Δt that is occluded over the distance Δt. The chromaticity $\kappa(t, \lambda)$ has the meaning that, in the limit as Δt approaches zero, $\kappa(t, \lambda)\rho(t, \lambda)\Delta t$ is the intensity of light of wavelength (color) λ emitted at the point $P(t)$. Henceforth, $I(t, \lambda)$ will represent the cumulative intensity of light of wavelength λ at t due to all contributions up to the point t.

The intensity of light reaching $t + \frac{\Delta t}{2}$ is:

$$I(t + \frac{\Delta t}{2}, \lambda) = I(t - \frac{\Delta t}{2}, \lambda)(1 - \rho(t, \lambda)\Delta t) + \kappa(t, \lambda)\rho(t, \lambda)\Delta t \tag{3.4}$$

Simplifying, we get:

$$\frac{I(t + \frac{\Delta t}{2}, \lambda) - I(t - \frac{\Delta t}{2}, \lambda)}{\Delta t} = -\rho(t, \lambda)I(t - \frac{\Delta t}{2}, \lambda) + \kappa(t, \lambda)\rho(t, \lambda)$$

In the limit as Δt goes to zero, we get:

$$\frac{dI(t, \lambda)}{dt} = -\rho(t, \lambda)I(t, \lambda) + \kappa(t, \lambda)\rho(t, \lambda) \tag{3.5}$$

Equation 3.5 is instantiated once for each of the three component wavelengths of light: red, green and blue. In each of these equations, ρ, κ and I are functions only of t; therefore, Equation 3.5 is really a set of three linear first order differential equations. For example, for red light:

$$\frac{dI_r(t)}{dt} + \rho_r(t)I_r(t) = \kappa_r(t)\rho_r(t)$$

These equations can be solved by numerical methods, for example by the fourth-order Runge-Kutta method, or by linking to a ODE solver subroutine.

Alternatively, by use of the integrating factor $e^{\int_{t_0}^{t} \rho(u,\lambda)\, du}$, we can rewrite Equation 3.5 as:

$$e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} \frac{dI(t,\lambda)}{dt} + \rho(t,\lambda) e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} I(t,\lambda) = e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} \kappa(t,\lambda)\rho(t,\lambda)$$

then,

$$\frac{d}{dt}\left[e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} I(t,\lambda) \right] = e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} \kappa(t,\lambda)\rho(t,\lambda)$$

Integrating both sides from t_0 to t_n, using the boundary condition that the intensity at t_0 is $I(t_0,\lambda)$, yields:

$$\left[e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} I(t,\lambda) \right]_{t_0}^{t_n} = \int_{t_0}^{t_n} e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} \kappa(t,\lambda)\rho(t,\lambda)\, dt$$

and so,

$$e^{\int_{t_0}^{t_n} \rho(u,\lambda)\, du} I(t_n,\lambda) - e^{\int_{t_0}^{t_0} \rho(u,\lambda)\, du} I(t_0,\lambda) = \int_{t_0}^{t_n} e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} \kappa(t,\lambda)\rho(t,\lambda)\, dt$$

which simplifies to:

$$I(t_n,\lambda) = e^{-\int_{t_0}^{t_n} \rho(t,\lambda)\, dt} \int_{t_0}^{t_n} e^{\int_{t_0}^{t} \rho(u,\lambda)\, du} \kappa(t,\lambda)\rho(t,\lambda)\, dt + I(t_0,\lambda) e^{-\int_{t_0}^{t_n} \rho(t,\lambda)\, dt}$$

(The limits of integration of the integrating factor were chosen so as to satisfy the boundary condition.) By combining the two exponentials in the first term, we get:

$$I(t_n,\lambda) = \int_{t_0}^{t_n} e^{-\int_{t}^{t_n} \rho(u,\lambda)\, du} \kappa(t,\lambda)\rho(t,\lambda)\, dt + I(t_0,\lambda) e^{-\int_{t_0}^{t_n} \rho(t,\lambda)\, dt} \tag{3.6}$$

Equation 3.6 can not be solved in closed form for the general case. However, if the transfer functions vary piecewise linearly along a ray segment within a cell, then this equation can be integrated exactly on a cell by cell basis. This solution is given in Section 3.3.5 after parameter functions are introduced in Section 3.3.4.

If the chromaticity is assumed to be constant along a ray then, by the same method that Max [61] used to simplify Equation 3.3 or by a simple transformation of Equation 3.6, a closed form solution for Equation 3.6 can be obtained:

$$I(t_n, \lambda) = \kappa(\lambda)(1 - e^{-\int_{t_0}^{t_n} \rho(u,\lambda),\, du}) + I(t_0, \lambda) e^{-\int_{t_0}^{t_n} \rho(t,\lambda)\, dt} \tag{3.7}$$

If we further assume that both the optical density and chromaticity are constant along a ray, then from Equation 3.7 or by integrating Equation 3.5 by separation of variables, we get:

$$I(t_n, \lambda) = \kappa(\lambda) \underbrace{(1 - e^{-\rho(\lambda)(t_n - t_0)})}_{A} + I(t_0, \lambda) \underbrace{e^{-\rho(\lambda)(t_n - t_0)})}_{B} \tag{3.8}$$

The term labeled B is the transmittance of the region, the fraction of light of wavelength λ entering the region at t_0 that reaches t_n. The opacity α is one minus the transmittance or $1 - e^{-\rho(\lambda)(t_n - t_0)}$ and represents the fraction of light of wavelength λ that enters the region at t_0 that is occluded while passing through the region. Term A represents the attenuation of the light emitted within the region itself. Equation 3.8 is the alpha compositing formula described by Porter and Duff [77] which they call the *atop* operator.

The restriction that the optical density and/or chromaticity be constant along a ray is not too serious since the cloud is discretized into cells. The ray integration can be evaluated by discretizing the ray in the same way that is used in finite element analysis and then integrating on a cell by cell basis, thus the density and/or chromaticity can vary from cell to cell.

Equation 3.8 can be evaluated for each cell by letting $\kappa(\lambda)$ and $\rho(\lambda)$ be the average chromaticity and density respectively along the ray between t_1 and t_2, the points where the ray enters and exits the cell:

$$\kappa_{avg}(\lambda) = \frac{\kappa(\lambda, t_1) + \kappa(\lambda, t_2)}{2} \tag{3.9}$$

and similarly for the optical density. Since it is common to linearly interpolate the scalar field within a cell, this approximation is acceptable. For further efficiency, the opacity

$$\alpha = 1 - e^{-\rho(\lambda)(t_2 - t_1)} \tag{3.10}$$

can be approximated by

$$\alpha = \rho(\lambda)(t_2 - t_1) \tag{3.11}$$

provided $\rho(\lambda)(t_2 - t_1) \ll 1.0$. The calculation can be further simplified if the optical density is assumed to be independent of wavelength, then only four user-defined transfer functions are required. If Equation 3.10 is used, then the exponential need be evaluated only once per cell. Figure 3.4 compares $\alpha = (1 - e^{-x})$ with $\alpha = x$.

Images generated using the approximation for cell opacity given in Equation 3.11 were compared with images generated using Equation 3.10. The images were very similar; often no difference could be detected. See Figures 3.5 and 3.6 where the images shown are volume renderings of the density field from a simulation of a binary star formation which is defined on a mesh of 593,920 tetrahedra. The MPVO algorithm was used for the visibility ordering; and the PT algorithm was used for splatting.

If it is desired to specify the opacity and color in the range $(0, 1)$, the optical density and chromaticity whose range is $(0, \infty)$ can be normalized to the range $(0, 1)$, as $\hat{\rho} = 1 - e^{-\rho}$ and $\hat{\kappa} = 1 - e^{-\kappa}$, where $\hat{\rho}$ is the normalized density and $\hat{\kappa}$ is the normalized chromaticity.

3.3.4 Shading, Gradients & Parameter Functions

For shading contour (level) surfaces, the intensity at a point on the surface can be made to vary as a function of the angle between the surface normal vector and a vector to a point light source. One way to incorporate surface shading into the model is discussed at the end of Section 3.3.5.

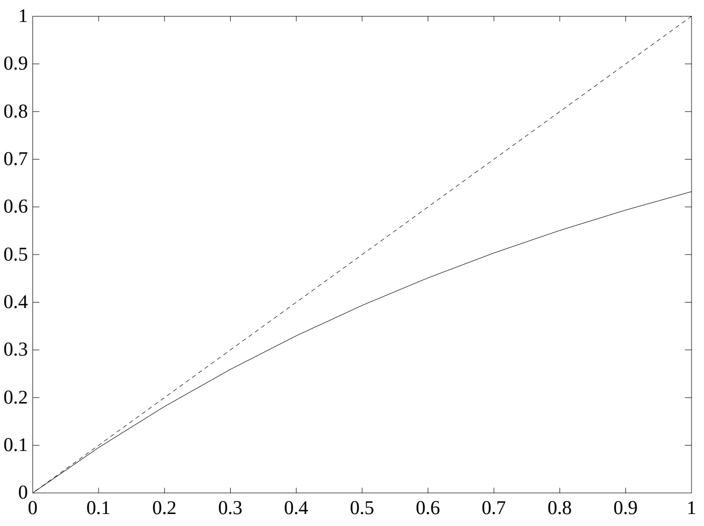


Figure 3.4: Comparison of $\alpha = (1 - e^{-x})$, the solid line, with $\alpha = x$, the dotted line.

The surface normal at any point p on a level surface of a scalar field S is the direction of the gradient of S at p. For tetrahedra, the gradient will be constant throughout the cell. For other types of cells used in the finite element method, the gradient can be computed from the parameter function for the type of cell involved. The parameter function S_c, sometimes referred to in visualization as the interpolation function, gives the value of the scalar field within any given cell. The parameter function may be linear, quadratic, cubic, etc. For example, for a 4 node tetrahedron, the parameter function is linear; and so the scalar field within a cell is given by the parameter function:

$$S_c(x, y, z) = c_1 + c_2x + c_3y + c_4z \tag{3.12}$$

Figure 3.5: Top image uses Equation 3.10. Bottom image uses Equation 3.11 (no exponential term).

Figure 3.6: Top image uses Equation 3.10. Bottom image uses Equation 3.11 (no exponential term).

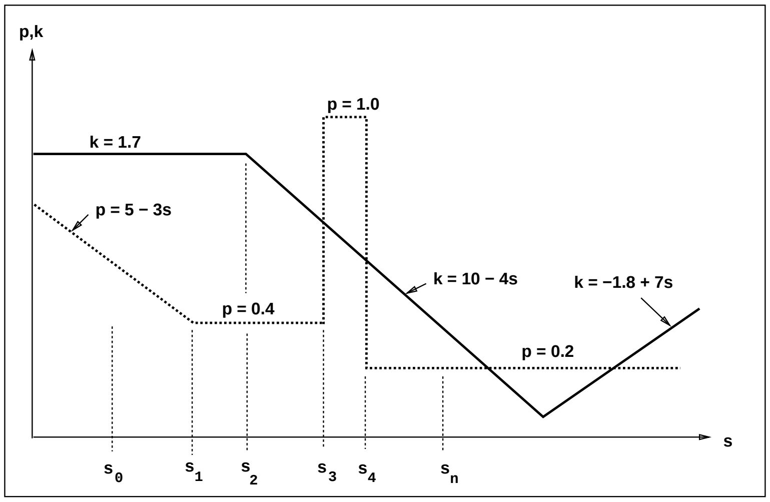


Figure 3.7: Example piecewise linear transfer functions ρ_r and κ_r, for red light. s_0 and s_n are the values of the scalar field at the points where the ray enters and exits the cell.

Since the scalar field value is known at the four vertices of the cell and the coordinates of the vertices are known, Equation 3.12 becomes a set of 4 simultaneous equations which can be solved for c_1, c_2, c_3 and c_4.[1]

[1]For the other three most common 3D elements (cells), the parameter functions are as follows. For an undistorted 10 node (quadratic) tetrahedron:

$$S_c(x,y,z) = c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6xy + c_7y^2 + c_8yz + c_9z^2 + c_{10}xz$$

for an 8 node (linear) brick:

$$S_c(x,y,z) = c_1 + c_2x + c_3y + c_4z + c_5xy + c_6yz + c_7xz + c_8xyz$$

for a 20 node undistorted (quadratic) brick:

$$\begin{aligned} S_c(x,y,z) &= c_1 + c_2x + c_3y + c_4z + c_5xy + c_6xz + c_7yz + c_8x^2 + c_9y^2 + c_{10}z^2 + \\ &\quad c_{11}xyz + c_{12}x^2y + c_{13}xy^2 + c_{14}x^2z + c_{15}xz^2 + c_{16}y^2z + c_{17}yz^2 + c_{18}x^2yz + c_{19}xy^2z + c_{20}xyz^2 \end{aligned}$$

When the elements are distorted by the use of an isoparametric mapping function, then the mapping function needs to be inverted and the interpolation performed in the undistorted element using the parameter function. (In order to assure convergence in the finite element method, the mapping function must be invertible.) The topic of visualization of distorted elements is not dealt with here but it is an important issue that needs to be faced.

The parameter function can also be expressed in terms of the scalar values s_i at the i vertices of the cell:

$$S_c(x,y,z) = \sum_i N_i(x,y,z)s_i$$

For a linear tetrahedron, this becomes:

$$S_c(x,y,z) = N_1(x,y,z)s_1 + N_2(x,y,z)s_2 + N_3(x,y,z)s_3 + N_4(x,y,z)s_4$$

The coefficients $N_i(x,y,z)$ are referred to as the *shape* or *basis functions* for the cell. They are polynomials of the form discussed above; for example on the 4 node tetrahedron they are linear.

The parameter functions will generally not be C^1-continuous at the boundary between cells; and so the gradient will not be C^0-continuous between cells. If the change in the gradient between cells is large then the shading will show anomalies from cell to cell. This can provide useful feedback to the scientist regarding the quality of his/her mesh. However, if smooth shading is desired, an average gradient at each vertex may be calculated by averaging the gradients at the centroids of all cells that share the vertex.[2] The surface normal at any point in a cell can then be calculated by interpolating the gradients at the cell's vertices. A more accurate average vertex gradient can be calculated by weighting the gradients at the surrounding centroids by the inverse of the distance to the centroid. Other methods are described in [35].

3.3.5 Exact Solution for Linear Parameter and Transfer Functions

Equation 3.6 can be integrated exactly on a cell by cell basis if the transfer functions vary piecewise linearly along a ray segment within a cell. This can be done as follows.

If the scalar field data has been generated by the finite element method, then, within a cell, the scalar field is given by the parameter function $S_c(x, y, z)$. This is shown for a linear tetrahedron in Equation 3.12. We would like to express S_c as a function of t, the ray parameter. In parametric form, a ray through the cell is expressed as: $x = \alpha_1 + \alpha_2 t$, $y = \beta_1 + \beta_2 t$, $z = \gamma_1 + \gamma_2 t$, where $(\alpha_1, \beta_1, \gamma_1)$ is the point where the ray enters the cell and $(\alpha_2, \beta_2, \gamma_2)$ is a unit vector along the direction of the ray.

Substituting these three equations into Equation 3.12 gives, for the case of linear tetrahedral elements:

$$S_c(t) = v + wt \tag{3.13}$$

[2]When a mesh is rectilinear, a finite difference scheme [97, 59, 100] can be used to approximate the gradient.

where $v = c_1 + c_2\alpha_1 + c_3\beta_1 + c_4\gamma_1$ and $w = c_2\alpha_2 + c_3\beta_2 + c_4\gamma_2$.

Now the transfer functions must be considered. Let $s_0 = S_c(t_0)$ be the scalar field value at the point where the ray enters the cell, and $s_n = S_c(t_n)$ be the value at the exit point. If the transfer functions are piecewise linear, as they are in Figure 3.7, then they can be considered to be composed of $m+1$ piecewise linear intervals (s_0, s_1), (s_1, s_2) , . . . , (s_i, s_{i+1}), . . . , (s_m, s_n). See Figure 3.7 where five intervals are shown. The intensity of light at t_n can be calculated by integrating Equation 3.6 over each of the $(m+1)$ intervals. The value of t at $s_1, s_2, \ldots, s_m$ can be found from Equation 3.13. For example: $t_1 = (s_1 - v)/w$.

For red light, the terms involving ρ_r and κ_r in Equation 3.6 are $\rho_r(s) = a + bs$ and $\kappa_r(s)\rho_r(s) = f + gs + hs^2$ or in terms of t, using Equation 3.13:

$$\rho_r(t) = a + bv + bwt \qquad \kappa_r(t)\rho_r(t) = f + gv + hv^2 + gwt + 2hvwt + hw^2t^2$$

over any interval on which κ and ρ are both linear. Hence a table lookup procedure may be used to return the coefficients a, b, f, g and h for any given interval; v and w are calculated from the ray entry and exit points and the parameter function for the cell. For example, for the first interval (s_0, s_1) shown in Figure 3.7, $a = 5$, $b = -3$, $f = 8.5$, $g = -5.1$ and $h = 0$.

For red light, Equation 3.6 can then be written as:

$$I_r(t_{i+1}) = \int_{t_i}^{t_{i+1}} e^{-\int_t^{t_{i+1}}(a_i+b_iv+b_iwu)\,du}(f_i + g_iv + h_iv^2$$
$$+g_iwt + 2h_ivwt + h_iw^2t^2)\,dt + I_r(t_i)e^{-\int_{t_i}^{t_{i+1}}(a_i+b_iv+b_iwt)\,dt}$$

simplifying, we get:

$$I_r(t_{i+1}) = \int_{t_i}^{t_{i+1}} e^{-\int_t^{t_{i+1}}(q_1+q_2u)\,du}(q_3 + q_4t + q_5t^2)\,dt + I_r(t_i)e^{-\int_{t_i}^{t_{i+1}}(q_1+q_2t)\,dt}$$

where $q_1 = a_i + b_iv$, $q_2 = b_iw$, $q_3 = f_i + g_iv + h_iv^2$, $q_4 = g_iw + 2h_ivw$ and $q_5 = h_iw^2$.

The second term can be integrated easily to give:

$$I_r(t_i)e^{-(q_1 t_{i+1}+\frac{q_2 t_{i+1}^2}{2}-q_1 t_i-\frac{q_2 t_i^2}{2})}$$

The first term can be evaluated by completing the square of the exponent and then using integration by parts. With the help of Nelson Max [personal communication] and *Mathematica*, version 2.0, the first term evaluates to:

$$\frac{q_2q_4 - q_1q_5 + q_2q_5t_{i+1}}{q_2^2} - \frac{q_2q_4 - q_1q_5 + q_2q_5t_i}{q_2^2}e^{(t_i-t_{i+1})(2q_1+q_2t_i+q_2t_{i+1})/2} +$$
$$\frac{\sqrt{\frac{\pi}{2}}(q_2^2q_3 - q_1q_2q_4 + q_1^2q_5 - q_2q_5)}{q_2^{2.5}e^{(q_1+q_2t_{i+1})^2/(2q_2)}} \times (\mathrm{Erfi}(\frac{q_1+q_2t_{i+1}}{\sqrt{2q_2}}) - \mathrm{Erfi}(\frac{q_1+q_2t_i}{\sqrt{2q_2}}))$$

In this expression, the argument to Erfi is either real, when $q_2 > 0$, undefined if $q_2 = 0$, or pure imaginary, when $q_2 < 0$. Erfi(x) is shorthand for two functions, depending on whether x is real or pure imaginary. If x is real, $\mathrm{Erfi}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{u^2}\,du$, while if x is pure imaginary, of the form $x = ib$, with real b, $\mathrm{Erfi}(ib) = i\frac{2}{\sqrt{\pi}}\int_0^b e^{-u^2}\,du$. When $q_2 < 0$, the i in the latter expression cancels the i in the denominator $q_2^{2.5}$. Thus, we need only prepare one dimensional tables for these two integrals, or use subroutines to approximate them. (The second integral is closely related to the integral of the Gaussian error function, for which subroutines and tables exist.)

If $q_2 = 0$, then the first term takes a different form, and integration by parts or *Mathematica* gives:

$$\frac{(q_1^2q_3 - q_1q_4 + 2q_5 + q_1^2q_4t_{i+1} - 2q_1q_5t_{i+1} + q_1^2q_5t_{i+1}^2)}{q_1^3} -$$
$$\frac{(q_1^2q_3 - q_1q_4 + 2q_5 + q_1^2q_4t_i - 2q_1q_5t_i + q_1^2q_5t_i^2)e^{q_1(t_i-t_{i+1})}}{q_1^3}$$

This is a new result. It remains to be seen if these expressions can be evaluated so as to permit interactive rendering. Certainly, they can be used to generate benchmark images against which images generated by approximate solutions, such as those given in Equations 3.7 and 3.8, can be compared. And, they can be used for the generation of presentation quality images. Accurate approximations need to be investigated.

Surface shading is helpful for understanding the orientation of (iso)surfaces that may appear in a volume rendered image. However, for the remainder of the image shading is not appropriate and may even make the image harder to understand. An interval in the density map which produces an isosurface is distinctive, for example a tall narrow rectangular pulse or a delta function. If such intervals are flagged, surface shading can be turned on only for those intervals.

Wilhelms and Van Gelder [103] outline a continuous model and develop differential equations for cumulative intensity and transmittance based on it. Our model benefits from their development, but also differs from their model in a number of respects. For example, in our model the opacity does not appear as a factor in the denominator of the cumulative intensity; and color intensity is linked to opacity, i.e. color and optical density are multiplicative factors. The value of the former property is that the cumulative intensity is well behaved for very small or zero opacity. The latter property means that if the transfer functions are defined such that (a) the optical density at a point in the volume is very small, i.e. the medium is almost totally transparent at that point, and (b) the color intensity is maximum at that point, then the overall contribution will be minimal. These differences need further investigation.

Our new exact solution needs to be implemented and the rendering time and image quality compared with those resulting from implementations of Equations 3.7 and 3.8 and with results from other volume density models. Also, the utility of the three density transfer functions introduced by the continuous model needs to be investigated.

3.3.6 Other Models and Considerations

Texturing and/or a general shading model with exterior lights may be used to highlight and accentuate surfaces which may occur in the volume density, either level surfaces of the scalar field or surfaces based on material classification, such as bones or flesh, or oil, water or sand, etc. As Westover [100] states, [the volume density has now become] "a reflective light emitting semi-transparent blob".

The standard equations for surface reflection due to ambient light $I_a(\lambda)$ and i point light sources $I_{p_i}(\lambda)$ can be written as:

$$I(\lambda) = \kappa_a(\lambda)I_a(\lambda) + \kappa_d(\lambda)\sum_i I_{p_i}(\lambda)(\vec{N}\cdot\vec{L_i}) + \kappa_s\sum_i I_{p_i}(\lambda)(\vec{N}\cdot\vec{H_i})^n \tag{3.14}$$

where κ_a, κ_d, κ_s are the coefficients of ambient, diffuse and specular reflectivity respectively. $\vec{N}, \vec{L_i}$ and $\vec{H_i}$ are the normal, light and bisector vectors, respectively. See Figure 3.8. The value of n determines the shininess of the surface.

The cumulative intensity $I(t, \lambda)$ derived in Section 3.3.2 can be considered to be the first term in Equation 3.14, that is the ambient term. Upson and Keeler [97] add a contribution to the diffuse reflection due to surface texture $M(x, y, z, \lambda)$, and use depth cueing to weight both the ambient and diffuse terms.

Westover [100] provides the user additional flexibility by offering a number of transfer functions in addition to opacity and color, such as reflected color tables and opacity modulation tables. For example, setting specular reflectivity to zero turns off specular shading. The reader is referred to Westover's paper for interesting examples of applications of these additional transfer functions.

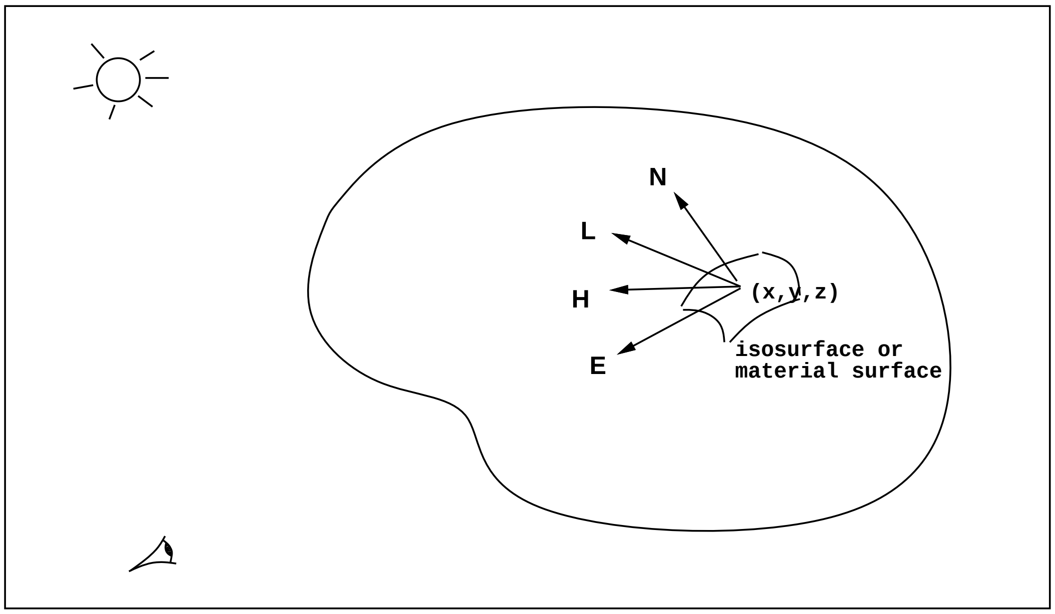


Figure 3.8: Surface shading diagram. N is the normal vector to the surface at the point (x, y, z). H is a vector halfway between the light vector L and the eye vector E.

CHAPTER 4

VISIBILITY ORDERING MESHED POLYHEDRA

4.1 Introduction

A *visibility ordering* of a set of objects from some viewpoint is an ordering such that if object a obstructs object b, then b precedes a in the ordering. Certain visualization techniques, particularly direct volume rendering based on projection methods [17, 57, 61, 93, 97, 101, 102, 104, 105] require a visibility ordering of the polyhedral cells of a mesh so the cells can be rendered using color and opacity blending. Visibility ordering the cells of rectilinear meshes (or certain classes of regular meshes based on a decomposition of a rectilinear mesh) is straightforward [33]. However, for other types of meshes, such as curvilinear or unstructured meshes, it is not immediately obvious how to compute this ordering.

This chapter describes a simple and efficient algorithm for visibility ordering the cells of any acyclic convex set of meshed convex polyhedra. This algorithm, called the Meshed Polyhedra Visibility Ordering (MPVO) algorithm, orders the cells of a mesh in linear time using linear storage. Preprocessing techniques and/or modifications to the MPVO algorithm are described which permit nonconvex cells, nonconvex meshes (meshes with cavities and/or voids), meshes with cycles, and sets of disconnected meshes to be ordered. The MPVO algorithm can also be used for domain decomposition of finite element meshes for parallel processing. The data structures for the MPVO algorithm can be used to solve the spatial point location problem.

The MPVO algorithm was used to generate the picture of a scientific data set of approximately 70,000 tetrahedra which is shown on the cover of [93]. Other images generated using the MPVO algorithm and the MPVO algorithm for nonconvex meshes are shown in Chapter 6. The basic ideas of the MPVO algorithm were suggested by Herbert Edelsbrunner in a conversation regarding his paper on the acyclicity of cell complexes [26]. A similar algorithm to the MPVO algorithm, also based on Edelsbrunner's suggestions, was developed independently by Max, Hanrahan and Crawfis [61].

The Binary Space Partition (BSP) tree algorithm [34] is not suitable for visibility ordering large meshes because the algorithm uses splitting planes, (even when not required to break cycles). Since the cells are meshed, a large number of cells could be split, resulting in a potential explosion in the total number of cells. Analysis of the BSP tree algorithm by Paterson and Yao [72] suggests that its performance could be $O(f^2)$, where f is the number of faces in the original mesh. An A-buffer [7] is also not suitable for visibility ordering large meshes because there are too many transparent cells at each pixel, making memory requirements prohibitive with current hardware.[1] Goad [41] describes a special purpose program written in LISP for visibility ordering polygons. This approach might be adapted for polyhedra; further investigation may be warranted. The *brute force* algorithm, for visibility ordering an acyclic mesh, calculates an obstructs relation[2] for every pair of the n cells in the mesh, and requires at least $\Omega(n^2)$ time.

[1]An off the cuff estimate indicates that an A-buffer could require a minimum of 30 MBytes for a 64,000 cell mesh rendered as a 250x250 pixel image; a 1,000,000 cell mesh could require 300 MBytes of A-buffer memory for a 500x500 pixel image.

[2]An obstructs relation can be calculated for two convex polyhedra by projecting them onto a plane and checking their projections for intersection.

This Chapter is organized as follows. An overview of the MPVO algorithm is given in the next section, and some preliminary definitions in Section 4.3. Section 4.4 contains a formal description of the MPVO algorithms. Implementation details of the algorithms are discussed in Section 4.5. A time and storage analysis is given in Section 4.6. Section 4.7 deals with the effects of numerical error and degeneracy. Sections 4.8 and 4.9 give methods for dealing with cyclically obstructing polyhedra and show how any cyclic and/or nonconvex mesh can be converted into an acyclic convex mesh. Other applications of the MPVO algorithms are given in Section 4.10.

4.2 Overview of the MPVO Algorithm

An intuitive overview of the MPVO algorithm is as follows. First, the adjacency graph for the cells of a given convex mesh is constructed. Then, for any specified viewpoint, a visibility ordering can be computed simply by assigning a direction to each edge in the adjacency graph and then performing a topological sort of the graph. The adjacency graph can be reused for each new viewpoint and for each new data set defined on the same static mesh.

The direction assigned to each edge is determined by calculating a behind relation for the two cells connected by the edge. Informally, the behind relation is calculated as follows. Each edge corresponds to a face shared by two cells. That face defines a plane which in turn defines two half-spaces, each containing one of the cells. If we represent the behind relation by an arrow through the shared face, then the direction of the arrow is towards the cell whose half-space contains the viewpoint; see Figure 4.1[3]. To implement this, the plane equation for the shared face can be evaluated at the viewpoint. The adjacency graph and the plane equation coefficients

[3]All figures in this paper, except Figures 4.8 and 4.9, are two dimensional, representing polyhedra as polygons.

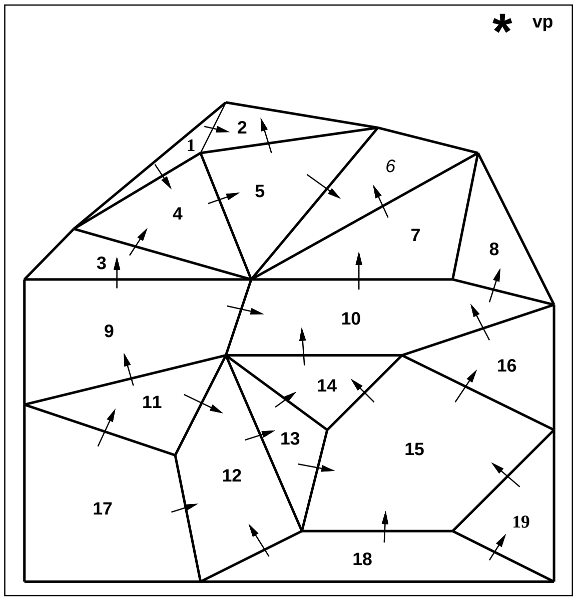


Figure 4.1: Visibility ordering of the cells of a mesh relative to viewpoint *vp*.

can be computed and stored in a preprocessing step. The MPVO algorithm can be extended to order many nonconvex meshes; this is described in detail in Section 4.4.2.

4.3 Preliminary Definitions

A *convex polyhedron* in $\mathbf{E}^3$ is the intersection of a finite set of closed half-spaces so that its interior is nonempty (it has positive volume). A polyhedron can therefore be unbounded; however, for purposes of this paper we will assume that all polyhedra are bounded.[4] Henceforth the terms *cell* and *polyhedron* will be used synonymously and, unless specified to the contrary, will mean a bounded convex polyhedron. The *interior* of a polyhedron P is the largest open subset in P.

A *set of meshed polyhedra* or a *mesh* is a finite set S of polyhedra, where (1) the intersection of any two polyhedra is either empty or a face, edge or vertex of each, and (2) for any partition

[4]Technically, a bounded polyhedron is called a *polytope*, a term not used in this paper.

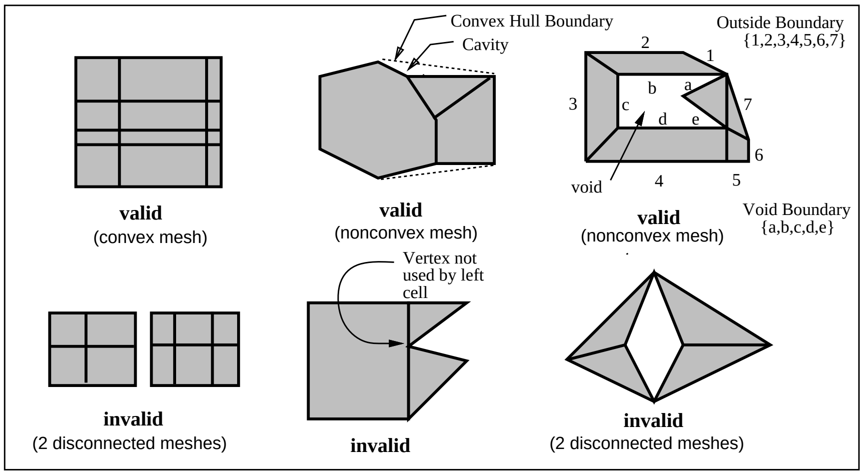


Figure 4.2: Examples of valid and invalid meshes. In 3D a void is completely enclosed by faces of the mesh; a cavity may be a tunnel or network of tunnels.

of S into two subsets, there is always at least one polygon that is a face of a polyhedron from each subset.[5] Two meshes are *disconnected* if they do not share any faces. See Figure 4.2.

If a face f of some cell in a mesh S is not shared by any other cell in S, then f is an *exterior face*. The union of all exterior faces of S constitutes the *boundary* of S. A face that is not an exterior face is an *interior face*. An *exterior cell* has at least one exterior face. The *convex hull of a mesh* S is the smallest convex set containing the cells of S. 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.

Let S be a nonconvex mesh. A polyhedron p, not necessarily convex, which is not a member of S, such that each face of p is shared by some cell in S, is called a *void.* If p is convex, then we call it a *convex void.* A nonconvex mesh may have zero or more voids. The union of all faces of a void is referred to as the *void boundary.* The union of the faces in the boundary of S that are not faces of a void is the *outside boundary* of S. A nonempty region between the boundary

[5] A mesh corresponds to a cell complex, a standard notion in computational geometry, provided the cell complex has property (2) and the cells are convex polyhedra.

of the convex hull of a mesh S and the outside boundary of S is referred to as a *cavity*. See Figure 4.2. In some cases a cavity may be a tunnel or network of tunnels.

4.4 The Visibility Ordering Algorithms

DEFINITIONS: A *viewpoint* is some point in $\mathbf{E}^3$. The *obstructs* relation is defined as follows. Let vp be a viewpoint, p_1 and p_2 be two distinct cells of a mesh S, and $\mathrm{int}(p_1)$ and $\mathrm{int}(p_2)$ be the interiors of p_1 and p_2. Relative to viewpoint vp, p_2 *obstructs* p_1 if there is a half-line hl starting at vp so that $hl \cap \mathrm{int}(p_1) \neq \emptyset$, $hl \cap \mathrm{int}(p_2) \neq \emptyset$, and every point of $hl \cap \mathrm{int}(p_2)$ lies between vp and any point of $hl \cap \mathrm{int}(p_1)$ [26]. Let us define the *behind* relation $<_{vp}$ such that $p_1 <_{vp} p_2$ if and only if p_1 and p_2 are adjacent, that is, share a face, and p_2 obstructs p_1. Diagrammatically, we represent $p_1 <_{vp} p_2$ as an arrow through the face shared by p_1 and p_2 pointing from p_1 to p_2. See Figure 4.1. We denote the transitive closure of $<_{vp}$ as $<^*_{vp}$.

We can assume, without any loss of generality, that the viewpoint does not lie in any plane spanned by a face of the mesh. Otherwise the viewpoint can be perturbed infinitesimally to eliminate such degenerate cases. Since a mesh has a finite number of faces, the viewpoint can be perturbed infinitesimally without changing the obstructs relation for any face whose plane does not contain the viewpoint. Thus, every pair of adjacent cells can be related by $<_{vp}$. A symbolic perturbation technique that illustrates this is given in Section 4.7. Practical considerations about degeneracy, floating point arithmetic and numerical error are discussed in Sections 4.5.1.2 and 4.7.

DEFINITIONS: Let S be a mesh. A *cycle* is a sequence $a <_{vp} b <_{vp} \cdots <_{vp} c <_{vp} a$ of cells of S. The obstructs relation on S is *acyclic* if, for every viewpoint, no cycles exist; then, we say S is an *acyclic mesh*. Let S be an acyclic mesh, and vp be a viewpoint. A *visibility ordering* of

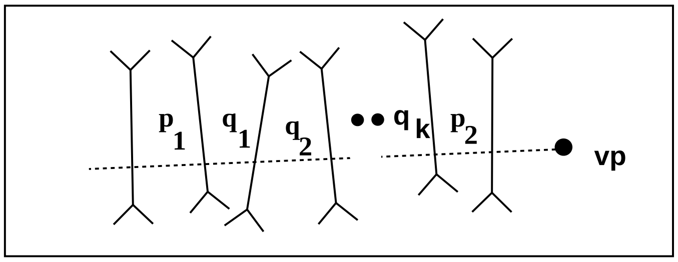


Figure 4.3: Diagram for proof of Lemma

S relative to vp is defined to be an ordering of the cells of S such that if cell p_2 obstructs cell p_1, then p_1 precedes p_2 in the ordering.

LEMMA: Given a viewpoint vp, any total ordering which is consistent with the ordering defined by the $<_{vp}$ relation on an acyclic convex mesh S is a visibility ordering of the cells of S relative to vp. In other words, for any cells p_1 and p_2 in S, if p_2 obstructs p_1 then $p_1 <^*_{vp} p_2$.

PROOF: Let $p_1, p_2, q_1, q_2, \ldots, q_k$ be cells of an acyclic convex mesh S and vp be any viewpoint. If p_1 and p_2 are adjacent and p_2 obstructs p_1 relative to vp, then $p_1 <_{vp} p_2$ by definition. If p_1 and p_2 are not adjacent and p_2 obstructs p_1, then it is possible to draw a ray starting from vp such that the ray intersects both int(p_1) and int(p_2). We can assume that by proper selection of a point in p_1 through which the ray passes, that the ray will intersect no edges and vertices of the mesh, nor will the ray lie in any plane spanned by a face of the mesh. By the definition of the obstructs relation, the ray will first intersect p_2, and then exit p_2. The face through which the ray exits p_2 has the relation $q_k <_{vp} p_2$, where q_k is the other cell that shares this face. Because a mesh has a finite number of cells and is convex, this process will continue until we reach q_1 which shares a face with p_1. This face will have the relation $p_1 <_{vp} q_1$. See Figure 4.3. Therefore there exists a sequence of cells $q_1, q_2, \ldots, q_k$ such that $p_1 <_{vp} q_1 <_{vp} q_2 <_{vp} \cdots <_{vp} q_k <_{vp} p_2$, i.e. $p_1 <^*_{vp} p_2$. □

4.4.1 The MPVO Algorithm

THE MPVO ALGORITHM: Given an acyclic convex mesh S, construct the adjacency graph G for the cells of S. For a given viewpoint vp, for every pair of adjacent cells p_1 and p_2 in S, compute the $<_{vp}$ relation. If $p_1 <_{vp} p_2$, direct the corresponding edge in G from p_1 to p_2, otherwise from p_2 to p_1. G is now a directed acyclic graph (DAG). Perform a topological sort of G as described in Section 4.5.1.3; the resulting ordering is a visibility ordering of the cells of S.

The directed graph constructed by the algorithm is a representation of the ordering defined by the $<_{vp}$ relation. A topological sort generates a total ordering of a DAG. Therefore, by the lemma, the MPVO algorithm will output a visibility ordering of the cells of S relative to vp. The domain of the MPVO algorithm is the set of all acyclic convex meshes. Sections 4.8.2 and 4.9 describe methods to preprocess meshes which are cyclic and/or nonconvex such that the resulting mesh can be ordered by the MPVO algorithm. If cells of a computational mesh have curved bounding surfaces, the cells can be approximated by sets of tetrahedra [54]; nonconvex cells can be tetrahedralized [13, 93]. Thus, the domain of the MPVO algorithm may be extended to any mesh.

4.4.2 The MPVO Algorithm for Nonconvex Meshes

The MPVO algorithm can be adapted to visibility order many acyclic nonconvex meshes. This modified algorithm, called the MPVO algorithm for nonconvex meshes, is a heuristic; its limitations are discussed below.

DEFINITIONS: If a DAG G has an edge from node a to node b, then we say a is a *predecessor* of b and b is a *successor* of a. If certain nodes of G are marked, the set *upreds*$^*(b)$ is defined

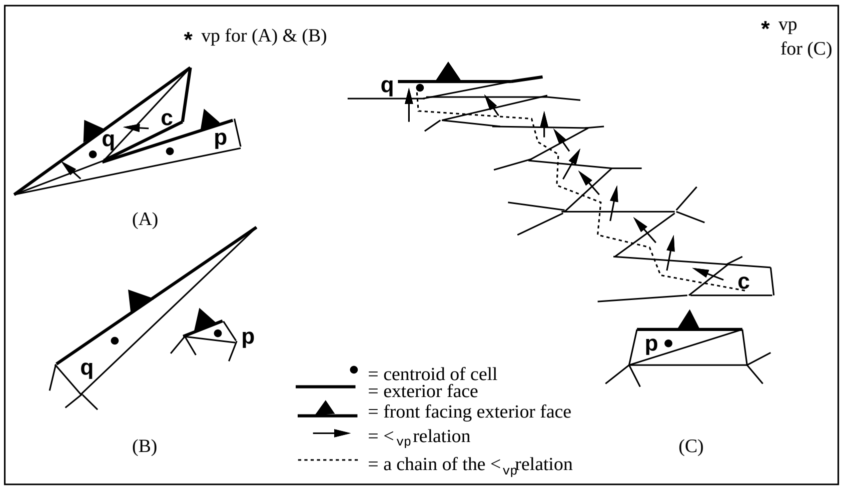


Figure 4.4: The examples of boundary anomalies shown in (A) and (B) cause only a few cells to be incorrectly ordered. However, the example in (C) could cause large numbers of cells to be incorrectly ordered.

as follows. If b is marked, then upreds$^*(b) = \emptyset$, otherwise upreds$^*(b)$ is the union of $\{b\}$ and upreds$^*(a)$ over all predecessors a of b. Let S be a mesh and vp a viewpoint, if p is an exterior cell in S which has an exterior face f such that the plane defined by f separates vp and p, then we say p is a *front facing* exterior cell.

DEFINITION: Let *dist*() be the Euclidean metric and *cen*() be the centroid. Let S be an acyclic mesh. If for some viewpoint vp, (1) there exist front facing exterior cells p and q in S such that $\text{dist}(vp, \text{cen}(q)) \geq \text{dist}(vp, \text{cen}(p))$, and either (a) there exists a cell $c <^*_{vp} q$ such that c obstructs p, and $p \not<^*_{vp} c$, as in Figure 4.4 (A) and (C), or (b) q obstructs p, and $p \not<^*_{vp} q$, as in Figure 4.4 (B), and (2) for all front facing exterior cells r in S with $\text{dist}(vp, \text{cen}(r)) > \text{dist}(vp, \text{cen}(q))$, $p \not<^*_{vp} r$, then we call this an instance of a *boundary anomaly.* See figure 4.4.

THE MPVO ALGORITHM FOR NONCONVEX MESHES: Given an acyclic mesh S, construct the adjacency graph G for the cells of S. For a given viewpoint vp, initialize a list L called the *start node list* to empty. Place all front facing exterior cells in S on L. Sort the cells on L by decreasing distance from vp to the centroid of the cell. For every pair of adjacent cells

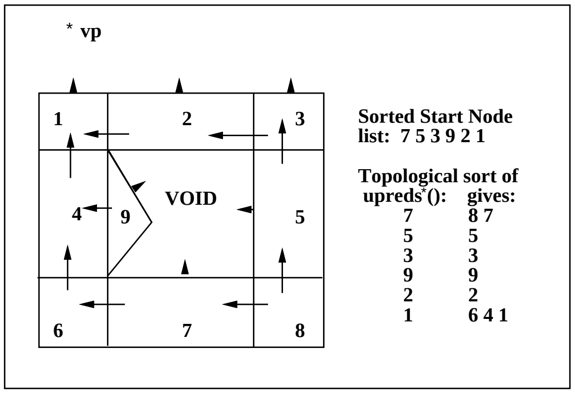

* vp
1
2
3
4
9
VOID
5
6
7
8
Sorted Start Node list: 7 5 3 9 2 1
Topological sort of upreds*(): gives:
7 8 7
5 5
3 3
9 9
2 2
1 6 4 1

Figure 4.5: Example of the MPVO algorithm for nonconvex meshes. The cells with exterior faces facing *vp* are sorted by decreasing distance from *vp*.

p_1 and p_2 in S, compute the $<_{vp}$ relation. If $p_1 <_{vp} p_2$, direct the corresponding edge in G from p_1 to p_2, and otherwise from p_2 to p_1. For each cell c on L, starting with the first and continuing in order, mark and output in topological order the cells of upreds$^*(c)$. See figure 4.5. (If a cell d is marked, then by definition of a topological sort, all cells q such that $q <^*_{vp} d$ will also be marked.) Provided S has no boundary anomalies, the output is a visibility ordering of the cells of S.

Increasing the distance from the viewpoint to the mesh tends to reduce the incidence of boundary anomalies. An efficient procedure for determining if a mesh has a boundary anomaly for any viewpoint is not known. The MPVO algorithm for nonconvex meshes has been tested on several actual computational meshes and found to give a correct ordering for all viewpoints tried. However, it is easy to generate counterexamples by hand, as is shown in figure 4.4. This heuristic is regarded as an interim algorithm which is quick and easy to implement and that can be used until the methods for converting a nonconvex mesh to a convex mesh described in Section 4.9 are implemented.

4.5 Some Implementation Details

4.5.1 The MPVO Algorithm

The domain of the MPVO algorithm is the set of all acyclic convex meshes. If a mesh has a cycle, it will be detected by the implementation of the MPVO algorithm described below. No test is made for convexity. The MPVO algorithm consists of three phases. The first phase, a preprocessing step done once for a given mesh, constructs an adjacency graph for the mesh and calculates the plane equation coefficients for each face. At runtime, for each given viewpoint, phase II calculates the $<_{vp}$ relation for all adjacent pairs of cells in the mesh, thus converting the adjacency graph into a DAG; and, phase III generates a visibility ordering of the cells of the mesh by performing a topological sort of the DAG. Two different algorithms for topologically sorting a DAG are described, one uses a depth first search (DFS) and the other a breadth first search (BFS).

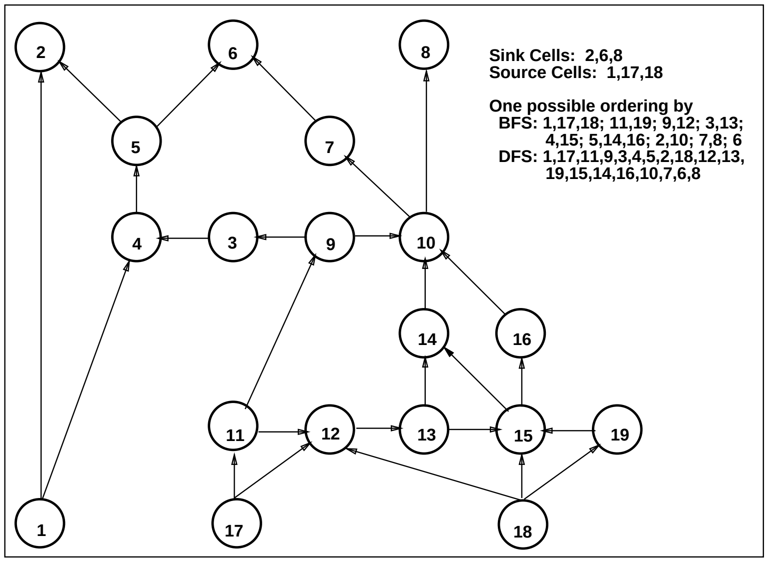
One possible set of data structures for this algorithm consists of two arrays. One array has a record for each face. The record has four fields, one for each coefficient of the *plane equation* $pe(x, y, z) = Ax+By+Cz+D$. The other array, indexed by cell number, has a record or pointer to a record for each cell. Each cell record has fields for recording the cell's vertices, faces, the cell numbers of the cells adjacent to it, and for marking each face with the $<_{vp}$ relation, the arrow referred to in Section 4.4. The arrow field can have the markings 'inbound', 'outbound' or 'none', which are explained in Section 4.5.1.2. One field per cell is required for marking the cell 'imaginary' as described in Sections 4.8.2 and 4.9. If the DFS algorithm will be used in phase III, then each cell should have two additional fields to mark whether a cell: (1) has ever been 'visited', and (2) has been 'visited on this descent'. As indicated in Section 4.5.1.3, the first marking is used to guide the search and the second is used to check for cycles. If the

BFS algorithm will be used in phase III, one additional field per cell for holding the number of inbound arrows to the cell is helpful. If the cells have a varying number of faces, one field per cell should be used to hold the number of faces.

4.5.1.1 PHASE I of the MPVO Algorithm (Preprocessing Step)

Construct and initialize the data structures described above. If the cells of a mesh are non-convex, or if it is desired that all the cells be converted to tetrahedra, then the cells can be triangulated during this phase.

When computing the plane equation coefficients, a suitable convention should be used so the sign of the plane equation is consistent for all cells and also with the usage described in the next section. For tetrahedral meshes the following technique can be used. For each cell, index each face f by the local vertex number (0 .. 3) of the vertex v_f not used to define f. So, for example, face 1 has vertices {0,2,3}; face 3 has vertices {0,1,2}; etc. Using the convention that for vertices (i, j, k), $A = y_i(z_j - z_k) + y_j(z_k - z_i) + y_k(z_i - z_j)$, $B = z_i(x_j - x_k) + z_j(x_k - x_i) + z_k(x_i - x_j)$, $C = x_i(y_j - y_k) + x_j(y_k - y_i) + x_k(y_i - y_j)$, and $D = -x_i(y_j z_k - y_k z_j) - x_j(y_k z_i - y_i z_k) - x_k(y_i z_j - y_j z_i)$, calculate the plane equation coefficients for face 3 of any nondegenerate cell, using the vertex ordering (0,1,2). If $pe(v_3) > 0$, swap the first two vertices of face 3 in that cell's data structure. Propagate this ordering of the vertices to all the other cells in the mesh. For a right-handed coordinate system, the vertices are now recorded so they can be enumerated in counterclockwise order when viewed from outside the cell provided the following ordering is used: face 0: (3,2,1); face 1: (0,2,3); face 2: (3,1,0); face 3: (0,1,2). Now calculate the plane equation coefficients for each face of each cell using this vertex ordering. Mark all degenerate cells 'imaginary'.



2
6
8
5
7
4
3
9
10
14
16
11
12
13
15
19
1
17
18
Sink Cells: 2,6,8
Source Cells: 1,17,18
One possible ordering by
BFS: 1,17,18; 11,19; 9,12; 3,13; 4,15; 5,14,16; 2,10; 7,8; 6
DFS: 1,17,11,9,3,4,5,2,18,12,13, 19,15,14,16,10,7,6,8

Figure 4.6: The DAG for the mesh in Figure 1. Edge 7-8 is not included since vp is coplanar with the shared face between cells 7 and 8.

4.5.1.2 PHASE II of the MPVO Algorithm

In this phase the adjacency graph built in phase I is converted into a DAG by calculating the $<_{vp}$ relation for each shared face; see Figure 4.6. If a perspective projection is used, the $<_{vp}$ relation is calculated as shown below using the plane equation coefficients computed in phase I. As discussed in Section 4.7, cycles can be caused by a combination of degeneracy and the effects of numerical error due to floating point arithmetic. Therefore, when $-\varepsilon \leq pe(x_{vp}, y_{vp}, z_{vp}) \leq \varepsilon$, where ε is some small number, we assume that the corresponding cells are not related by the $<_{vp}$ relation and so mark the arrow field of the corresponding edge 'none'.

```
PHASE II ALGORITHM:
    read in a viewpoint $vp = (x_{vp}, y_{vp}, z_{vp})$
    for each cell $c$
        for each interior face $f$ of $c$
            let $c'$ be the cell that shares $f$ with $c$
```

```
if the cell number of $c <$ the cell number of $c'$
    if $pe_f(x_{vp}, y_{vp}, z_{vp}) > \varepsilon$ then
        mark arrow field for $c.f$ 'outbound'
        mark arrow field for $c'.f$ 'inbound'
    else if $pe_f(x_{vp}, y_{vp}, z_{vp}) < -\varepsilon$ then
        mark arrow field for $c.f$ 'inbound'
        mark arrow field for $c'.f$ 'outbound'
    else mark $c.f$ and $c'.f$ 'none'
```

If an orthographic projection is used, then let (r_x, r_y, r_z) be a vector along the direction of projection, and evaluate $-Ar_x - Br_y - Cr_z$ instead of $pe_f(x_{vp}, y_{vp}, z_{vp})$ in the above algorithm; where A, B, C are the first three coefficients of pe_f. If the DFS algorithm will be used in phase III, mark all cells 'not visited' and 'not visited this descent'; find all sink nodes, nodes which have no arrows leaving them, and place them on a list called the sink cell list. If the BFS algorithm will be used, count the number of inbound arrows for each cell and record this in the appropriate field; and, find all the source nodes, nodes which have no arrows entering them, and put them in a queue, called the source cell queue, in any order. All these operations can be done within the above for loop.

4.5.1.3 PHASE III of the MPVO Algorithm

In this phase a total ordering of the cells in visibility order is obtained by topologically sorting the DAG. The graph search methods DFS and BFS can be used. A DFS of a DAG yields a reversed topological sort of the nodes. If the DFS is reversed, as in the phase III-DFS algorithm below, an unreversed topological sort is obtained. The use of the phase III-DFS algorithm has the advantage that it will output all cells in the mesh regardless of the existence of cycles; however, it is not significantly parallelizable. On the other hand, the phase III-BFS algorithm can be parallelized and its output consists of sequences of cells that do not obstruct each other and therefore can be rendered concurrently; but, it will halt on encountering a cycle. The cycle issue is discussed further in Section 4.8.1.

```
PHASE III-DFS ALGORITHM:
    for each cell on the sink cell list dfs( cell ).
    dfs( cell ):
        mark cell 'visited';
        mark cell 'visited this descent';
        for each predecessor p of cell
            if p is not marked 'visited' then dfs( p );
            else if p is marked 'visited this descent' then output cycle warning;
        if cell is not marked 'imaginary' then output( cell );
        remove the mark 'visited this descent' from cell.
```

```
PHASE III-BFS ALGORITHM:
    while the source cell queue Q is not empty
        let h be the cell at the head of Q
        if h not marked 'imaginary' then output h
        remove h from Q;
        for each successor s of h
            if number inbound arrows into $s > 1$ then
            decrement number inbound arrows into $s$;
        else put $s$ on the end of $Q$;
    if (number cells output $\neq$ total number cells) output cycle warning.
```

4.5.2 The MPVO Algorithm for Nonconvex Meshes

The MPVO algorithm for nonconvex meshes is implemented in the same way as the MPVO algorithm described above except for the following modifications. In phase I, calculate the centroid of each exterior cell and store it in an array called the centroid list. For each exterior cell, set one of the unused adjacent cell fields to the negative of the index of the cell's entry in the centroid list. Calculate the plane equation coefficients for all faces including exterior faces. In phase II set the arrow field for all faces including exterior faces. Place all exterior cells that have one or more exterior faces with the arrow field marked 'outbound' on a list L, the start node list. Sort the cells on L by decreasing distance from vp to the centroid of the cell. In phase III call dfs() for each cell on L that is marked 'not visited', starting with the first and continuing in order.

Since a DFS is used by this algorithm, it is not significantly parallelizable; however, it is possible to parallelize the overall rendering system of which the algorithm is a part as is shown in Chapter 7 [104, 105].

4.6 Time and Storage Analysis

The *size* of a three-dimensional mesh is the total number of cells, faces, edges and vertices in the mesh. For a tetrahedral mesh, the MPVO data structures require $4f + 14n$ words of storage, where f is the number of interior faces and n the number of cells in the mesh. This includes

Computer	*71,680 tetrahedra*	*593,920 tetrahedra*	*1,208,320 tetrahedra*
IBM RS6000/530	0.8 sec.	6.8 sec.	14 sec.
Alliant FX/2800	1.0 sec.	9.5 sec.	20 sec.
Sun SPARCstn2	1.1 sec.	10 sec.	21 sec.
SGI 4D/340VGX	1.2 sec.	11 sec.	23 sec.

Table 4.1: Typical timings for the MPVO algorithm, phases II & III-BFS combined, using one CPU. Time is user time measured by getrusage().

space for the adjacency graph, cell vertices, the plane equation coefficients, the arrows, and one word per cell for flags. In addition, $3v$ words are required for the vertex coordinates, v words per scalar field for data, where v is the number of vertices in the mesh, and n words for the sink/source cell list. The space requirement is on the same order as the space required by the simulation that generates the data. For the MPVO algorithm for nonconvex meshes, f in the above equation becomes the total number of faces in the mesh; and, $3x$ words are required for storage of centroids, and another $2x$ words for the start node list, which is used instead of the sink/source cell list, where x is the number of exterior cells. A result that may be useful for data structure planning is that the maximum number of tetrahedra for a triangulation of v vertices is: $\frac{1}{2}(v^2 - 3v - 2)$ [25]. In general, for meshes whose cells are not necessarily tetrahedra, the amount of storage required is linear in the size of the mesh.

For the purpose of analysis, we assume the input mesh is in a form such that it can be converted into an adjacency graph in linear time. Let f be the number of interior faces and n the number of cells in a mesh. Each phase of the MPVO algorithm requires $O(n + f)$ time. If f becomes the total number of faces, then this bound also applies to the MPVO algorithm for nonconvex meshes provided the ratio of the number of exterior cells to the total number of cells is sufficiently small as described next. Phase II of the MPVO algorithm for nonconvex meshes requires a sort of the exterior cells; however, unless the number of exterior cells is asymptotically

54,405 cells	187,395 cells	593,920 cells
1.13 sec.	3.7 sec.	11.9 sec.

Table 4.2: Typical serial timings for the MPVO algorithm for nonconvex meshes on a SGI 4D/340VGX. The number of exterior cells range from 5% to 10% of the total number of cells.

larger than $n/\log_2 n$, sorting their centroids in time $O(m \log_2 m)$ is only $O(n)$.[6] Typically, m is way less than this threshold.

Tables 4.1 and 4.2 show typical timings for the MPVO algorithms. When the program and data fit entirely in physical memory and no other users are active, the timings shown are identical to wall clock time. For unstructured meshes, 256 MB of physical memory will hold the MPVO data structures for up to 2,500,000 cells. Phase II accounts for approximately 65% of the times shown. Phase III-DFS takes 25% longer than phase III-BFS, resulting in an 8% increase in overall time. Parallelization of the MPVO algorithm in conjunction with a volume rendering system is described in Chapter 7.

4.7 Degeneracy and Effects of Numerical Error

In phase II of the MPVO algorithm the plane equation $pe(x,y,z) = Ax + By + Cz + D$ is evaluated. Due to roundoff error, it will not be possible to reliably evaluate $pe(x,y,z)$ for a face when $-\varepsilon \leq pe(x,y,z) \leq \varepsilon$, where ε is some small number. Therefore the $<_{vp}$ relation will be indeterminate for that face, which we call an ε-face. The perturbation technique described below can not be used to eliminate ε-faces.

When an ε-face is detected, it is not known which way to direct the edge for that face or even whether there should be an edge there at all. It has been concluded that the best policy is to consider that the two cells that share an ε-face are not related. This means the possibility

[6] $O(m \log_2 m) = O(n)$ when $m = \frac{n}{\log_2 n}$, since $\frac{n}{\log_2 n} \log_2(\frac{n}{\log_2 n}) = n - \frac{n \log_2 \log_2 n}{\log_2 n} < n$.

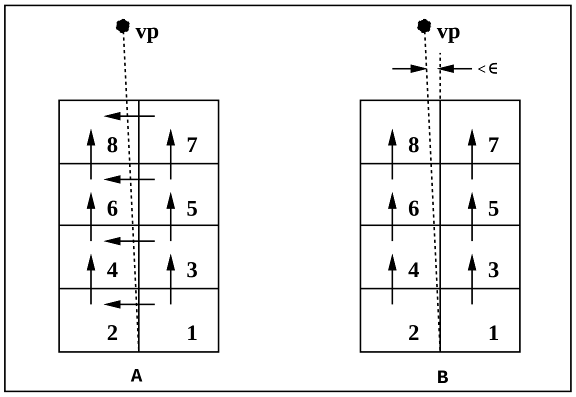


Figure 4.7: In A, the correct ordering of the cells can be obtained by topologically sorting following the arrows. In B, due to numerical error, the ordering by DFS is either 1,3,5,7,2,4,6,8 or 2,4,6,8,1,3,5,7; by BFS it is either 1,2,3,4,5,6,7,8 or 2,1,4,3,6,5,8,7. The second ordering shown for both BFS and DFS results in slivers of cells being rendered out of order.

of introducing cycles is avoided, but at the cost of possibly introducing an artifact in the form of a sliver that either may not be rendered or may be rendered out of order; see Figure 4.7.

Single precision floating point arithmetic has been used to achieve maximum speed with no noticeable degradation in image quality. Methods to experiment with to eliminate artifacts are: changing the viewpoint slightly, reducing the value of ε defined for the algorithm, moving the viewpoint farther away, and/or using double precision arithmetic. For presentation graphics, the problem can be eliminated by using integer arithmetic and tetrahedra.

If exact arithmetic is used, the following procedure can be used to relate every pair of adjacent cells by the $<_{vp}$ relation even if the viewpoint is coplanar with a face. When the viewpoint $vp = (x, y, z)$ lies in the plane of a face f, that is $pe_f(x, y, z) = 0$, we want this to be interpreted such that vp lies on one side of the plane, and we want this interpretation to be consistent over all cells of the mesh so that cycles are not introduced. This can be accomplished by the following symbolic perturbation technique based on the ideas described by Edelsbrunner and Mücke [24]. If $pe_f(x, y, z) = 0$ for a face f, then we symbolically perturb the point (x, y, z)

by some small positive amount in each of its three coordinates, let us say to $(x+\varepsilon, y+\varepsilon^2, z+\varepsilon^3)$, where ε is indeterminate, $\varepsilon > 0$ and ε can be assumed to be arbitrarily small. Thus,

$$pe_f(x+\varepsilon, y+\varepsilon^2, z+\varepsilon^3) = Ax + A\varepsilon + By + B\varepsilon^2 + Cz + C\varepsilon^3 + D$$

but $Ax + By + Cz + D = 0$, therefore,

$$pe_f(x+\varepsilon, y+\varepsilon^2, z+\varepsilon^3) = A\varepsilon + B\varepsilon^2 + C\varepsilon^3 = g(\varepsilon).$$

Since ε is arbitrarily small, $g(\varepsilon)$ can be evaluated as follows:

```
if $A > 0$ then $g(\varepsilon) > 0$
else if $A < 0$ then $g(\varepsilon) < 0$
else if $A = 0$ then
    if $B > 0$ then $g(\varepsilon) > 0$
    else if $B < 0$ then $g(\varepsilon) < 0$
    else if $B = 0$ then
        if $C > 0$ then $g(\varepsilon) > 0$
        else if $C < 0$ then $g(\varepsilon) < 0$
```

Therefore, whenever $pe_f(x, y, z) = 0$, evaluate $g(\varepsilon)$ instead. Now every pair of adjacent cells will be related. This technique is valid since $(A, B, C) \neq (0, 0, 0)$ and there exist a finite number of faces in a mesh.

4.8 Cycles and the Delaunay Triangulation

4.8.1 Cycles and Their Effect

If a mesh has cyclically obstructing polyhedra, then the only way *any* algorithm can correctly visibility order the mesh is to cut some of the cells into smaller cells so that the obstructs relation is acyclic. One way to do this is to retriangulate the vertices of the mesh such that the resulting mesh is acyclic. The MPVO algorithm is not amenable to cutting the polyhedra therefore it will output a correct ordering only if there are no cycles. A BSP tree [34] can be used to visibility order a set of polyhedra, regardless of whether they are acyclic. This can be done by using each shared face as a splitting plane to create the BSP tree. These facial splitting planes correspond to the internal nodes of the tree; and, the leafs correspond to the cells. For the reason mentioned in Section 4.1, this method is not suitable for large meshes. Furthermore, due to splitting, new cells will result with a possibly different number of faces; and, field values will need to be interpolated for the new vertices.

It is possible for cycles to occur in unstructured and curvilinear meshes. See Figures 4.8 and 4.9. However, the frequency of occurrence of cycles in actual meshes is not known. It would be useful to have a preprocessing method for determining whether or not a mesh is acyclic for all viewpoints; however, an efficient algorithm for doing this is not known. An $O(f^4)$ algorithm has been described by Edelsbrunner [personal communication], where f is the number of faces. Briefly, it works as follows. Each face in the mesh defines a plane. This in turn defines an arrangement of planes which partition space into $O(f^3)$ regions. From each region, choose any point and use that point as a viewpoint for a cycle test; a cycle test for a convex mesh takes $O(f)$ time.

The MPVO algorithm using the phase III-DFS algorithm will still output an ordering of the cells of a mesh with cycles but the ordering will not be a correct visibility ordering for some of the cells involved in the cycle. It is quite possible, due to the high degree of abstraction inherent in the process of direct volume rendering, that, in some cases, cycles may have no noticeable effect on the image. If the number of cells involved in the cycle is small in comparison to the

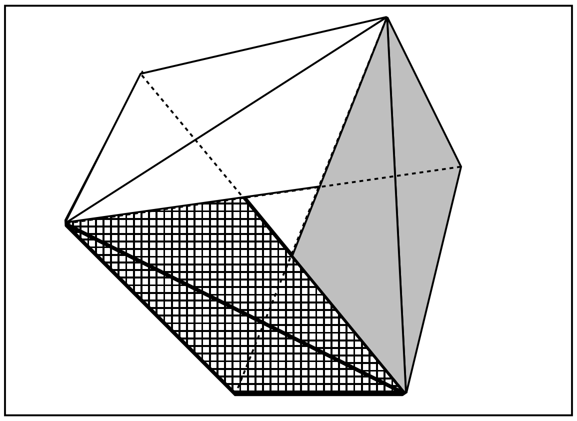


Figure 4.8: These three tetrahedra, which bound Schönhardt's polyhedron [92], form a cycle. By adding a point in the 'middle' of the polyhedron and then triangulating, a convex mesh of tetrahedra is created which has a cycle. Such a situation could occur in an unstructured mesh, possibly a finite element mesh.

total number of cells, then an image with cyclic artifacts might be considered acceptable. This tradeoff between robustness and accuracy needs investigation.

When the phase III-BFS algorithm is used, which can be preferable for the reasons given in Section 4.5.1.3, it will halt on encountering a cycle. It would be nice to find a way to break cycles encountered and thus output all the cells even though the resulting ordering would not be entirely correct. It is known that heuristics must be used to do this since the general problem of minimizing the number of cells incorrectly ordered, which is related to the feedback arc set problem, is NP-complete. One heuristic to use when a cycle is detected is to delete the inbound edges of a cell with a minimum number of such edges. For either phase III algorithm, if the mesh has cycles, the user is given a warning. The user then has the following options: use the output anyway, try a different viewpoint, or abort and either retriangulate all or a portion of the mesh or use a different visibility ordering algorithm.

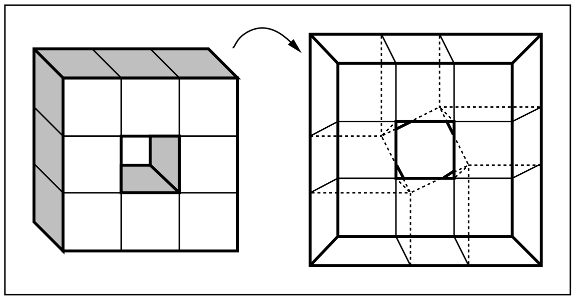


Figure 4.9: A simple curvilinear mesh with a cycle. The cycle is over the entire mesh for the viewpoint from which the figure is shown. If the curved bounding surfaces of the cells are approximated by triangles, the resulting cells can be tetrahedralized so that the cycle remains.

4.8.2 The Delaunay Triangulation

An acyclicity theorem for the obstructs relation for meshed polyhedra resulting from a Delaunay triangulation (DT) [19] has been proven by Edelsbrunner [26]. A DT of a set S of points in $\mathbf{E}^3$ is a triangulation such that a sphere circumscribed about the four vertices of any tetrahedron in the triangulation contains no other points in S. Therefore, in order to eliminate cycles from a meshed data set or to be sure it does not contain cycles, one can construct a DT of the vertices of the mesh thereby redefining the cells of the mesh such that they are guaranteed to be acyclic. For unusual point distributions, a DT of n points can be $O(n^2)$ both in time and in size; however, for uniform point distributions, over certain domains, a DT can be expected to have size $O(n)$ and [1, 49] report expected running times of $O(n^{4/3})$. It is possible that most scientific data sets will fall into the category of uniform point distributions; but, this requires investigation, especially for graded meshes.

The boundary of a DT of a set of points P is the boundary of the convex hull of P. If a nonconvex mesh is retriangulated with a DT, then the boundary of the new triangulation will not be the same as the original boundary; and, it will be necessary to ensure that the faces in the original boundary are also faces in the new triangulation, and to mark all tetrahedra outside the original boundary as 'imaginary'. A *conformed DT* is a DT which is required to have certain faces, in this case the original boundary faces. Since a DT of a given point set is unique, usually a conformed DT is possible only if new points are added. The process of adding points and retriangulating is repeated until no tetrahedron intersects the original boundary. Points are usually added only on the original boundary using heuristics. The convergence properties of this process in $\mathbf{E}^3$ is an unsolved problem.[7] An implementation of such a process for floating point DTs has been developed by Meshkat et al at IBM [64]. Although they are not able to prove convergence for their algorithm, they have tested it on hundreds of objects and found that it converges rapidly. The convergence issue in $\mathbf{E}^3$ is discussed somewhat in [69].

The DT is important in computational geometry and also is considered to be one of the three most attractive methods for 3D automatic mesh generation [73]. In computational geometry, the DT is usually implemented using integer arithmetic in order to properly handle degenerate cases. Three-dimensional DT algorithms are described in [49, 68, 85]. Mesh generation algorithms used by computational scientists usually are implemented in floating point; see for example Baker's 3D DT algorithm [1]. The proof of acyclicity of the DT is based on exact arithmetic and the slightest inaccuracy can destroy the acyclicity property. The Simulation of Simplicity (SoS) technique developed by Edelsbrunner and Mücke [24], and now implemented as a C library by Mücke, uses integer arithmetic, and cleanly and transparently handles all degenerate cases that may arise in the implementation of the DT algorithm. It is possible to implement a floating point front and back end to the SoS library which will make the use of integer arithmetic transparent to the user.

[7]Edelsbrunner and Tan [22] have demonstrated an $O(m^2n)$ bound on the number of points in a Delaunay triangulation in $\mathbf{E}^2$ that conforms to m line segments and n vertices.

A number of algorithms have been published which are called Delaunay triangulations of nonconvex domains or sometimes *constrained* DTs, e.g. [52]; however, these triangulations are not rigorous DTs due to relaxing the DT criteria at the boundary, therefore they may have cycles.

Delaunay triangulations are a special case of regular triangulations [58]. A *regular triangulation* in $\mathbf{E}^d$ is a triangulation that can be obtained by projecting the boundary complex of a convex polyhedron in $\mathbf{E}^{d+1}$. Edelsbrunner has proven the acyclicity of the obstructs relation for regular triangulations [26]. One of the best algorithms for constructing a regular triangulation in any dimension is given by Edelsbrunner and Shah [23]. When more is known about the properties and the implementation of regular triangulations, it may be the case that a conformed regular triangulation will be more suitable than a conformed DT for the purpose of the work taken up in this thesis.

The impact of retriangulation on the interpolation of data defined on the mesh should be investigated. Due to the high degree of abstraction in direct volume rendering, interpolation errors introduced by retriangulation may be acceptable. If the computational scientist uses a mesh generated by a DT, then this problem will not arise.

4.9 Nonconvex Meshes

Curvilinear and unstructured meshes often are nonconvex. The MPVO algorithm for nonconvex meshes can be used; however, it has some shortcomings. It takes time $O(n \log n + f)$ rather than $O(n + f)$ if many cells are exposed, it is not parallelizable, and it will not give a correct ordering if a boundary anomaly exists. Therefore, an important area of research is to find ways to convert nonconvex meshes into convex meshes so the regular MPVO algorithm can be used.

By use of a conformed DT as described in Section 4.8.2, any mesh can be converted into an acyclic and convex one. Alternatively, the following techniques can be used. A nonconvex mesh all of whose cavities or voids are convex can be converted into a convex mesh simply by treating the voids and cavities as 'imaginary' cells in the mesh. If a mesh has nonconvex voids and/or cavities they can be triangulated or decomposed into convex cells and then marked 'imaginary'. Cells marked 'imaginary' are not output by the MPVO algorithms. Two or more disconnected meshes can be combined into one nonconvex mesh using these techniques.

Some of these preprocessing techniques are predicated upon being able to determine computationally whether a mesh is convex or nonconvex. If the mesh is nonconvex, then it is necessary to computationally locate each void or cavity and determine its surface, and then determine if these regions are convex or nonconvex. By tracing adjacent exterior (unshared) faces via their shared edges and calculating the dihedral angle between these faces we can determine if the mesh is convex. The mesh is convex if all exterior faces are connected to each other and all dihedral angles are greater than or equal to 180 degrees. If there is more than one connected set of exterior faces, this implies the existence of voids or that the mesh is disconnected. To perform this computation it is helpful to have a data structure containing edge adjacency information, and to order the vertices so as to have facial orientation information. One such data structure is given in [56]. If more than one connected set is found, the set which is the outside boundary can be determined by comparison with the convex hull or by the following method. Select any face in any of the connected sets, draw a half-line from that face and intersect the half-line with all the faces in all of the *other* connected sets. If the half-line intersects an odd number of faces, then the face from which the half-line originates is a member of a void surface, otherwise it is the outside boundary. The use of the SoS library greatly simplifies this method by eliminating degenerate cases.

To define the set of faces for each cavity: trace the outside boundary; when it diverges from the convex hull a cavity face has been located; start from this face and gather all adjacent exterior faces which are not a part of the convex hull, stopping, and backing up if necessary any time a face or edge in the convex hull boundary is reached. If necessary, the convex hull faces which cap the cavity can also be gathered, bearing in mind that a cavity may have more than one cap if the cavity is a tunnel or network of tunnels. The dihedral angle examination can be used to determine if each void or cavity is convex/nonconvex.

An $O(n \log n)$ time convex hull algorithm for n points in $\mathbf{E}^3$ has been described by Preparata and Hong [78] and implemented by Day [18]. Day writes that he found the task "... definitely not a trivial exercise ..." due to degeneracies and special cases. However, if an $O(n^2)$ algorithm [27, section 8.4] is satisfactory, a much simpler implementation is possible. Even an $O(n \log n)$ implementation can be quite clean if the Guibas and Stolfi quad edge data structure [43] and the SoS library is used. Algorithms by Chazelle [11] or Chazelle and Palios [13] can be used to partition any nonconvex cavities or voids into convex regions; however, there is no guarantee of acyclicity.

A collection of observed data with no specified connectivity between the data points is called *scattered data*. In order to visualize this data it is very helpful to triangulate it. The Delaunay triangulation is an excellent method for doing this since the resulting triangulation is acyclic and so can be visibility ordered by the MPVO algorithm. When the shape of the data set is nonconvex, for example a set of samples gathered over the land mass surrounding the Gulf of Mexico, the Delaunay triangulation can create tetrahedra which are not amenable to accurate interpolation. Consider for example a tetrahedron created such that three vertices are in Mexico and one in Florida. The α-shape concept discussed in [21, 27, 68] can be a useful way to make a DT conform to the implicit boundary of the data, which in the example given would be the shore line of the Gulf.

The implementation of the preprocessing methods, described in this section, for converting a nonconvex mesh into a convex mesh could take a very significant amount of time; they are by no means trivial. The implementation of a 3D conformed Delaunay triangulation is still a research question at this time. Therefore, the MPVO algorithm for nonconvex meshes, which has been found to be easy to implement, may fill an immediate need despite its shortcomings.

4.10 Other Applications of the MPVO Algorithms

The MPVO algorithm may be used for domain decomposition of finite element meshes for parallel processing. This topic is covered in Chapter 10. The MPVO algorithm data structures and the $<_{vp}$ relation can be used to solve the point location problem. This is described in Chapter 9.

The data structure for the MPVO algorithm can be used to easily create a mesh previewer so the mesh geometry can be observed. Either the exterior surface of the mesh can be viewed or the surfaces of any voids. If the computer supports hidden surface removal, then output the exterior faces (or the void faces) and render them as bordered polygons; otherwise, use the appropriate MPVO algorithm to visibility order the mesh and render only the front facing exterior faces (or back facing void faces).

4.11 Conclusion and Remarks

The MPVO algorithm can visibility order any acyclic convex mesh in time linear in its size. Therefore it can be useful for direct volume rendering of large 3D curvilinear or unstructured data sets, making rotation, zooming and animation a possibility. It will be especially useful if polyhedron rendering hardware becomes available. An acyclic nonconvex mesh can be visibility ordered by the MPVO algorithm for nonconvex meshes if it has no boundary anomalies. However, a better approach seems to be to convert the mesh into a convex mesh as discussed in Section 4.9. Visibility ordering and point location can then be performed in a straightforward manner. Comparative timings for these two approaches are given in Chapter 7. If a mesh is cyclic, a Delaunay triangulation, which is acyclic, can be used to retriangulate the vertices of the mesh; if the cyclic mesh is nonconvex, then a conformed Delaunay triangulation is required.

Rectilinear meshes with embedded rectilinear meshes can be visibility ordered by the recursive use of a method such as given in [33]. Nonrectilinear meshes with embedded meshes can be ordered by treating the embedded mesh as a cell in the embedding mesh. Apply the MPVO algorithm to the embedding mesh; when a cell is encountered that is an embedded mesh, apply the MPVO algorithm recursively to it.

We would like computational scientists to consider the following generation technique for unstructured meshes. Whenever possible, generate a mesh over a convex domain even though the domain of interest is nonconvex; such a technique is used by Baker [2]. Retain the vertices and cells which lie outside the domain of interest for use at visualization time; send the remaining vertices and cells to the finite element solver. By doing this, the data set can be immediately and efficiently visualized using the MPVO algorithm. Since this algorithm has been found to be easy to implement, the scientist can then view his or her own data with minimal effort. This technique of saving the vertices and cells lying outside the domain of interest also applies to meshes created with a conformed Delaunay triangulation. The Gehäuse and ONERA meshes discussed in Chapters 6 and 7 are examples of the application of this technique.

The basis for the MPVO algorithm came from theoretical work in the field of computational geometry by Herbert Edelsbrunner, and bears testimony to the practical applicability of theoretical research in computer science. This paper has shown that computational geometry is a rich source of solutions for many of the problems encountered in dealing with volume visualization of nonrectilinear meshed data sets; and also that investing in a one time preprocessing of a data set yields high dividends in terms of run-time efficiency.

It would be especially helpful to have a suite of efficient, correct and relevant 3D computational geometry programs in the public domain, such as implementations of the SoS library, a parallel version of the brute force visibility ordering algorithm, an $O(\log^2 f)$ spatial point location algorithm, as well as code to triangulate a nonconvex polyhedron, to compute a convex hull, an α-shape, a (conformed) Delaunay triangulation, etc. These programs would provide a valuable platform on which to build research experiments. As it is now, researchers face many months of labor to implement one or more of these algorithms before they can begin to investigate the issues in computer graphics or computational science that really interest them. A team consisting of researchers from the fields of computational geometry, computational science and computer graphics/visualization, with a grant from a national funding agency, could make a most valuable contribution to science by undertaking such a project.

CHAPTER 5

HARDWARE SUPPORT FOR GRAPHICS

5.1 Introduction

High-end graphics workstations offer hardware support for high performance polygon rendering. At the present time, polyhedron rendering engines are not available; although, experimental architectures for rendering voxels have been developed. As noted in Chapter 2, the voxel model does not apply when volume rendering nonrectilinear data.

The splatting technique for DPVR described in Chapter 6 decomposes each tetrahedron into triangles. Therefore, high performance polygon rendering engines can be harnessed to help achieve interactive rates of rendering.

Shirley and Tuchman [93] implemented their Projected Tetrahedra (PT) splatting algorithm for volume rendering on a Sun 4/490 workstation. They wrote a scan conversion algorithm to create and display the image on a Pixar monitor. They reported that volume rendering a data set with 71,680 tetrahedra took 19 seconds for the execution of the PT algorithm, 3 seconds for the MPVO Algorithm to do the visibility ordering, plus the time to render the image.

The rendering time was considerable since the scan conversion, Gouraud shading and opacity blending were all done in software.[1]

These last three procedures, scan conversion, Gouraud shading and opacity blending, as well as viewing transformations, lighting calculations and clipping, are precisely what are implemented in hardware and microcode on high performance 3D graphics workstations such as the Apollo DN10000VS, the Pixel machine and the Silicon Graphics Power Series. Therefore, to achieve the goal of interactive volume rendering of very large data sets, it is natural to utilize hardware support for high performance polygon rendering.

For the reasons given in Chapter 7, the Silicon Graphics Power Series (SGIPS) was chosen for this research. The SGIPS is an MIMD machine.

In the next section we describe some architectural details of the SGIPS that relate to hardware support for polygon rendering and to the parallelization issues that will be discussed in Chapter 7. In Section 5.3 we discuss software support for these hardware features and some other related software issues. Knowledge of both of these sections will be helpful in reading Chapters 6 and 7. The inaccuracies introduced into volume rendering by the use of graphics hardware are discussed in Chapter 6, Section 6.5.

5.2 SGIPS Architectural Description

The Silicon Graphics Power Series have up to eight 33 MHz MIPS R3000 RISC main processors and eight floating point coprocessors, as well as 85 dedicated proprietary graphics processors in the VGX graphics subsystem. The specific model utilized for my work was the 4D/360VGX

[1]For the purpose of this work, Gouraud shading means the color and opacity at each vertex of the triangle are linearly interpolated across the pixels in the interior of the triangle. Opacity or alpha blending is the process of blending a portion of a pixel's existing color with a portion of the new color to achieve the effect of transparency.

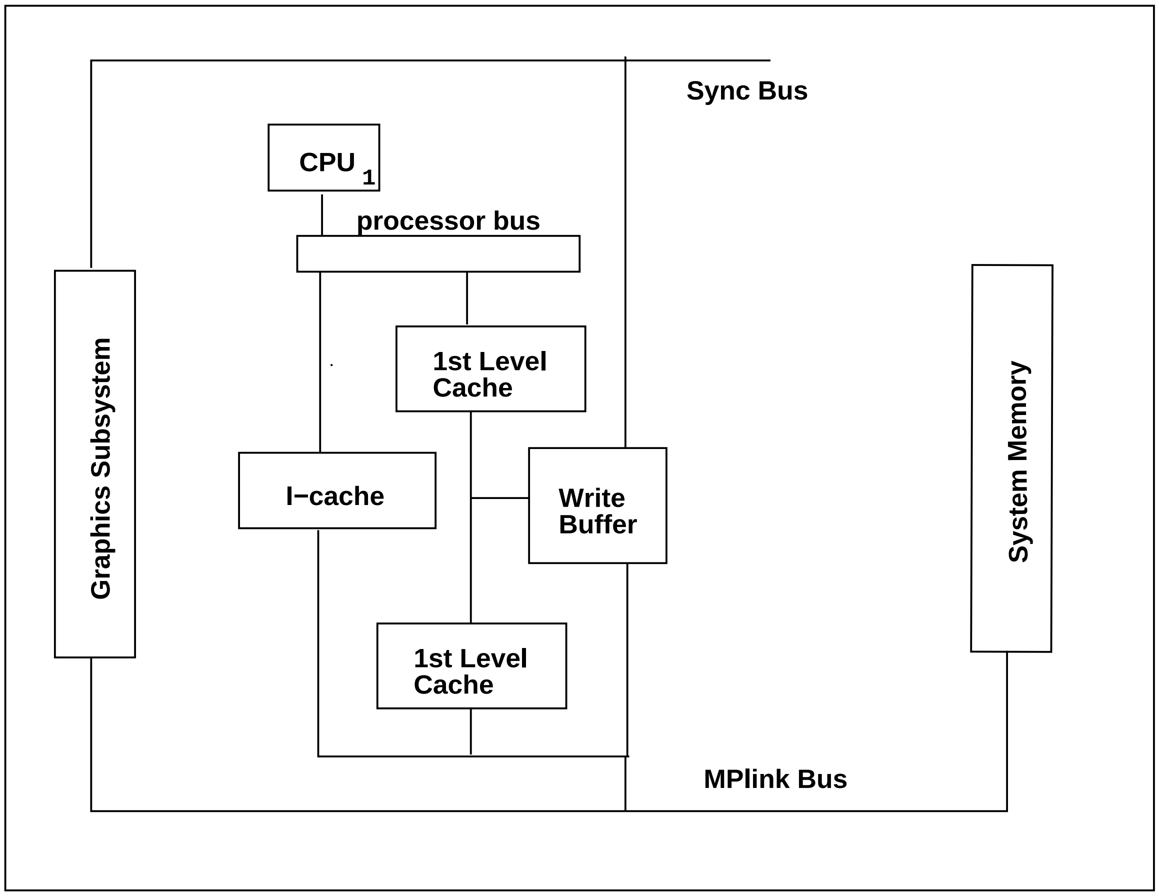


Figure 5.1: Block diagram of the Silicon Graphics Power Series architecture. Figure shows one CPU and its associated caches. There are up to eight such CPU configurations in a SGIPS machine.

with six CPUS, henceforth referred to as the SGI-VGX. It had 128 MB of physical memory. The fundamental design of the SGI-VGX is a tightly coupled symmetric shared memory architecture. See Figure 5.1.

The Sync Bus provides high speed synchronization between the main CPUS in support of fine grain parallelism. It supports 65,000 individual test-and-set variables which are in a special part of the physical address space. These provide the user with very fine grained hardware spin locks. Any one process share group can use up to 4,096 hardware locks. As described in Chapter 7, these locks were heavily used in my algorithms and found to be quite efficient.

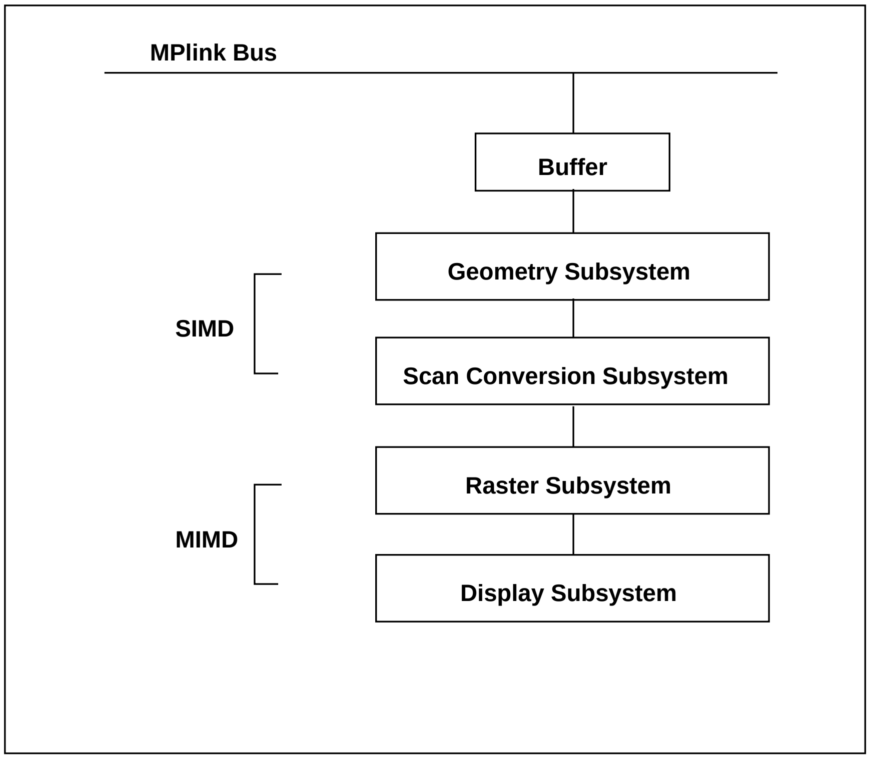


Figure 5.2: The graphics subsystem of the SGI-VGX

The Processor Bus can support sustained data transfers at 8 bytes every clock cycle. Thus an eight processor system has a total processor to cache bandwidth of 2100 megabytes per second. The first level caches, both instruction and data, are 64 kilobytes. The second level caches are 256 kilobytes, organized as 16 lines of 16 bytes each. The second level cache watches every transaction on the MPlink bus and checks for transactions in its data storage. The first level cache is always a subset of the second level cache so data consistency is guaranteed. Both caches use physical addresses rather than virtual addresses.

The MPlink bus supports the above cache consistency scheme and provides communication between the main processors, memory, the I/O system and the graphics subsystems. The Illinois cache consistency protocol is used. The MPlink bus, which is pipelined, has a 64 megabyte sustained bandwidth.

The graphics subsystem is shown in Figure 5.2. It receives a description of 3D objects in the form of polygons from the main CPUS. Each polygon is expressed in terms of its vertices. Each vertex is specified by its coordinates and an optional color, opacity and normal vector.

The graphics subsystem, composed of 85 processors, is divided into four subsystems: the Geometry Subsystem, the Scan Conversion Subsystem, the Raster Subsystem and the Display Subsystem. The first two of these have a SIMD pipeline organization and can process four graphics primitives simultaneously. The primitives accepted are points, line, polygons and polygon meshes. The latter two stages have an MIMD organization and allow up to 200 million pixels to be processed per second.

The Geometry Subsystem performs viewing transformations, lighting calculations and clipping. The Scan Conversion Subsystem converts the output of the Geometry Subsystem into pixel RGBA data for each pixel on the screen (1280x1024). Vertex color and opacity is linearly interpolated across the interior of lines or polygons.

The Raster Subsystem contains up to 40 processors, each of which controls up to 1/40 of the pixels on the screen. Each pixel has at least 144 bits of data, which includes 32 bits of RGBA data for each of two buffers, a 24 bit Z-buffer, an 8-bit stencil plane used to extend the capability of the Z-buffer so various prioritizing algorithms can be performed, two 4-bit underlay/overlay planes for use by the window manager, and two sets of 4-bit window planes. This subsystem is responsible for hidden surface removal and various blending and texturing operations.

The Display Subsystem reads the frame buffer and displays the image on the screen. It allows the display of multiple images simultaneously in an overlapping window environment, in single or double buffered RGB modes.

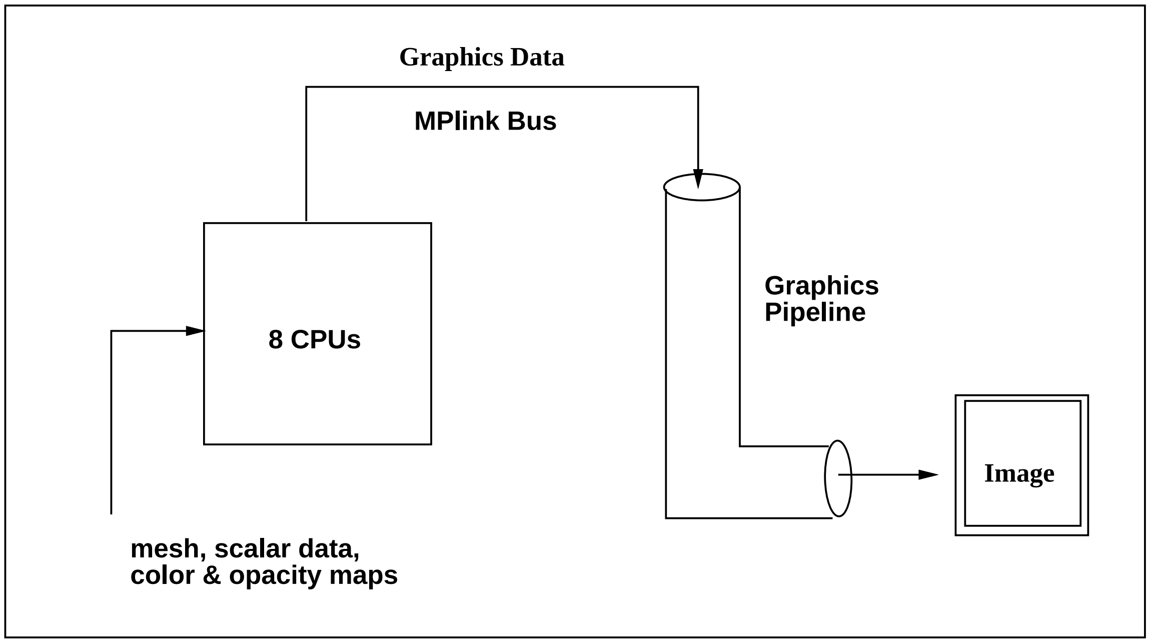


Figure 5.3: Flow abstraction of the SGI-VGX

The graphics pipeline is entirely under the control of microcode. The user feeds the graphics pipe with data using calls from the SGI GL graphics library. See Figure 5.3. At this level of abstraction, there are two processes going on in parallel, the graphics pipeline and the main CPUS. Thus while the user's algorithm runs in parallel on the main CPUS, at the same time the data output from the user's algorithm is is being processed in parallel by the graphics pipeline.

When the system is being used at its maximum efficiency, one CPU is used to fill the graphics pipe full time and the remaining CPUS are used to calculate the data being fed into the graphics pipe. Whether or not the system can be used at its maximum efficiency is application dependent.

As described in the performance analysis in Chapter 7, it was determined that the system could not be used at its maximum efficiency for this application due to a slow graphics pipe. One of the reasons for this is discussed in the next section. The bottleneck was further pinpointed to the Scan Conversion Subsystem of the graphics pipe.

Most of the frequently made graphics calls are compound calls. For example, to render a triangle, where the coordinates for vertex i are stored at location vip and the RGBA data for that vertex are stored at cip, requires the following sequence of calls. The c4f() call means the color is specified as 4 floating point numbers, r, g, b and opacity. The v3f() call means the vertex is specified by 3 floating point coordinates, x, y, z. If lighting is used, then a normal vector is specified for each vertex in addition to a color.

```
bgnpolygon();
    c4f(c1p);
    v3f(v1p);
    c4f(c2p);
    v3f(v2p);
    c4f(c3p);
    v3f(v3p);
endpolygon();
```

Since graphics calls can be made by any of the CPUS, compound graphics calls must be atomic and so mutual exclusion must be used; hardware spin locks may be used for this purpose.

The flow of data from the main CPUS to the graphics pipeline is maximized by the use of burst DMA which moves four 32-bit words of data. Typically this transaction will involve the vertex coordinates $< x, y, z, w >$ or colors $< r, g, b, a >$. Thus, for maximum performance, these data should be quadword aligned.

5.3 Software Issues

The SGI graphics library (GL) provides the functionality to utilize the SGIPS graphics hardware. To achieve maximum performance from the graphics hardware, surfaces are described as meshes of polygons, either triangles or quadrilaterals. Often a surface will be described by several hundred or several thousand such polygons. When the basic element is a triangle, the mesh is called a tmesh.

The advantage of this form of description can be seen be comparing the following code fragments. The first fragment uses GL calls to render a single triangle. The second fragment uses the tmesh to render three triangles; see Figure 5.4. The coordinates of vertex i are stored at memory location vip and the color for that same vertex at cip.

To render a single triangle requires eight GL calls. Whereas, using a tmesh, three triangles can be rendered using only 12 calls. When rendering very large tmeshes the savings can be very significant.

CODE TO RENDER 1 TRIANGLE:

```
bgnpolygon();
    c4f(c1p);
    v3f(v1p);
    c4f(c2p);
    v3f(v2p);
    c4f(c3p);
    v3f(v3p);
endpolygon();
```

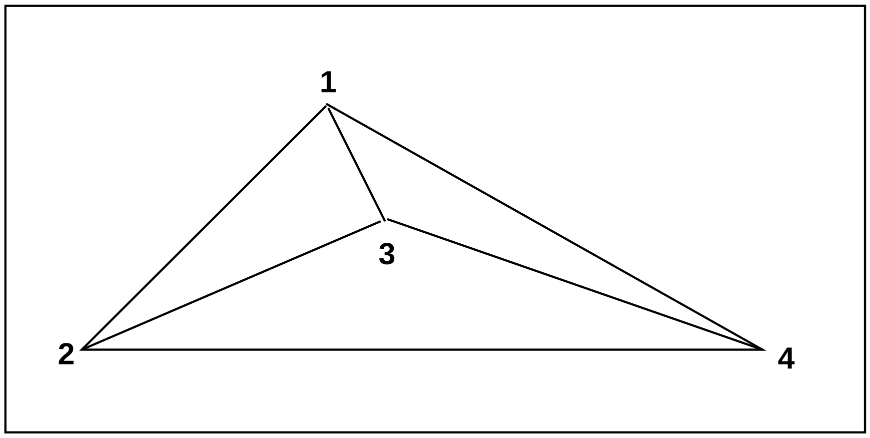


Figure 5.4: The three triangles used for the tmesh example. This corresponds to a case 1 splatting decomposition as defined in Chapter 6

CODE TO RENDER 3 TRIANGLES USING A TMESH:

```
bgntmesh();

    c4f(c1p);

    v3f(v1p);

    c4f(c2p);

    v3f(v2p);

    c4f(c3p);

    v3f(v3p);

    c4f(c4p);

    v3f(v4p);

    c4f(c1p);

    v3f(v1p);

endtmesh();
```

In the splatting process described in Chapter 6, a tetrahedral cell is projected into from one to four triangles and then each triangle is rendered. Therefore, using a tmesh rather than rendering each triangle individually results in increased efficiency. When a tetrahedron projects into three triangles, if a tmesh is used, 12 graphics calls are needed instead of 24.

Using the tmesh whenever possible resulted in a performance improvement of 35% when rendering in parallel using 6 CPUS with the Projected Tetrahedra splatting algorithm for 71,680 tetrahedra. The improvement in performance for serial rendering was even higher.

To use the rendering hardware at its maximum efficiency it is necessary to use tmeshes which consist of hundreds or thousands of triangles. Because our volume renderer had at most 4 triangles per tmesh, it was unable to use the graphics hardware at its maximum efficiency.

The software support for parallel processing on a SGIPS is described in Chapter 7.

CHAPTER 6

CELL PROJECTION METHODS

6.1 Introduction

In this Chapter, we present a suite of fast rendering approximations that support the goal of interactive volume rendering. A modified version of Shirley and Tuchman's [93] Projected Tetrahedra (PT) splatting algorithm is used as the basis for these approximations and also as a standard of comparison, both for image information content and rendering speed.

When considering the validity of these approximated images, it is important to keep in mind that interactive DPVR is intended to give the scientist a working tool to provide a general idea of the spatial distribution of the scalar field and roughly identify areas of interest, e.g. extrema or hot-spots, and not necessarily to create a highly realistic or precise image for publication purposes.

In other words, this environment is intended to be a data previewer for the scientist. He or she might use it in an analogous way to a professional photographer who uses a Polaroid camera to quickly check several different shots before using a 4x5 view camera for the final photo. These fast approximations can also be useful for selecting and tuning the color and density transfer functions.

There is a high degree of abstraction inherent in DPVR; every cell in the mesh contributes to the image. Therefore approximation errors if randomly distributed over the image may not be significant. However, if the mesh is highly regular, then these errors can be magnified and be highly noticeable.

For this thesis, only tetrahedral cells are considered. The MPVO algorithm can deal with any cells. The PT splatting algorithm, described below, deals only with tetrahedra. Wilhelms and Van Gelder [103] discuss the splatting of right-angled parallelepipeds (bricks). Alternatively, nontetrahedral cells can be tetrahedralized so the PT algorithm can be used.

We start with an overview of the splatting process, then describe Shirley and Tuchman's PT algorithm, then describe our suite of four fast splatting approximation algorithms, then discuss distortions introduced by the use of graphics hardware, and finally present comparative timings and discuss image quality.

6.2 Overview of the Splatting Process

In splatting, as in splatting a snowball against a wall [100], each cell is projected onto the screen in visibility order from back to front to build up a semitransparent image. The contribution of each cell to the image is proportional to the thickness of the splat. The splat is rendered as a set of up to four triangles which have a common vertex at the point of maximum thickness of the splat. At this common vertex, the opacity is nonzero; at all other vertices the opacity is zero. The opacity and color at the vertices are interpolated over the splat; Wilhelms and Van Gelder [103] describe three possible interpolation methods.

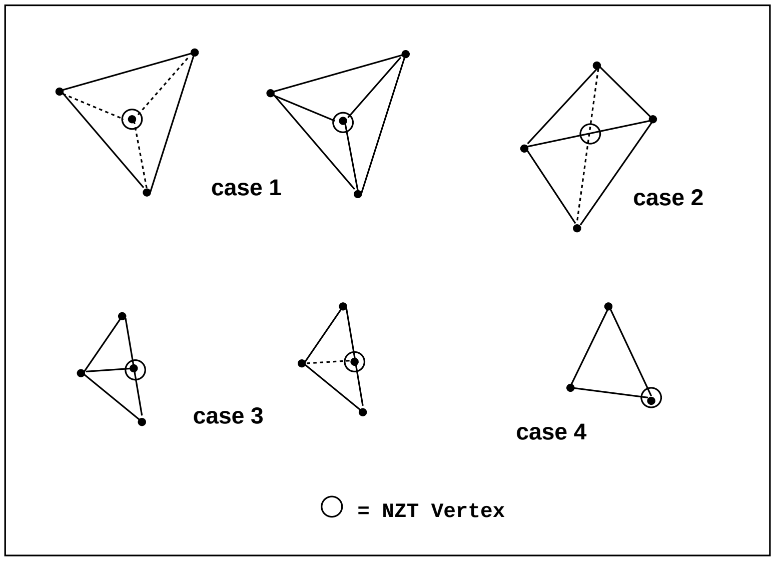


Figure 6.1: Projections of a tetrahedron onto the plane. There are four cases or footprints. The point marked as NZT is used as a non-zero thickness vertex for the triangles.

6.3 The Projected Tetrahedra Algorithm

The PT algorithm [93] works by compositing into an image the polygonal projection of each cell, called the cell's *footprint*, onto the viewing plane. Each footprint is subdivided into from one to four triangles, depending on the projection, and then rendered as a set of triangles.

For tetrahedra, there are only two combinatorially different projections; they are shown in Figure 6.1 as cases 1 and 2. Cases 3 and 4 are degenerate instances. My experiments show that for irregular meshes case 1 occurs approximately 40% of the time, on average, and case 2 approximately 60%. Cases 3 and 4 occur less than 5% of the time. Only cases 1 and 2 are discussed here; the extension to the degenerate cases is obvious.

The PT method uses the assumption, described in Chapter 3, that the optical density and chromaticity are constant along a ray through a cell. See Equation 3.8. Only one density transfer function ρ is used, rather than three.

Each 2D footprint has a “nonzero thickness” (NZT) vertex as shown in Figure 6.1. For each cell, the opacity at the NZT vertex is computed using either Equation 3.10 or 3.11. In these equations, t_1 and t_2 are the entry and exit points of a viewing ray through the NZT vertex. The opacity at the remaining vertices is set to zero. The color is calculated using Equation 3.9; a similar equation is used for the density.

The splatting calculation time was slightly faster when the opacity approximation given in Equation 3.11 was used rather than Equation 3.10, but not significantly so. For consistency, Equation 3.11 has been used for all results reported herein.

The Shirley and Tuchman PT algorithm was modified and optimized for an MIMD architecture with hardware support for high performance polygon rendering.

The opacity and color at the vertices of each triangle is linearly interpolated over the interior of the polygon by the graphics hardware (Gouraud shading). Also, the hardware is used to blend the new and old color for each pixel (alpha blending). The remainder of the splatting process, cell visibility ordering, decomposition of the polyhedral cell into polygons, and calculation of color and opacity for each polygonal vertex is done in software.

Figures 6.2 through 6.7 show volumetrically rendered images using the the modified PT algorithm on a Silicon Graphics 4D/360VGX workstation. Other images were shown in Figures 3.5 and 3.6 in Chapter 3. The images in Figures 3.5 and 3.6 were generated using the MPVO Algorithm. The images shown in Figures 6.2 through 6.7 were generated using the MPVO Algorithm for nonconvex meshes. Figure 6.2 shows volume rendered images of a simulated temperature field and of hot-spots in a different temperature field, both defined on an nonconvex irregular mesh of 13,499 tetrahedra comprising a MBB-Gehäuse solid modeling benchmark. The mesh was generated by a conformed Delaunay triangulation [65].

Figure 6.3 shows two views of an energy field from a simulation of airflow around an airfoil defined on a mesh of 287,962 tetrahedra. The data is courtesy of ICASE, NASA Langley. Figure 6.4 shows the energy and the magnitude of velocity from a CFD simulation of incompressible flow around multiple posts defined on a curvilinear mesh of 513,375 tetrahedra [84]. The images shown in Figure 6.5 show an N3S simulation of velocity magnitude of coolant flow in a component of the cooling system in Electricité de France's Super Phoenix nuclear reactor. The data field is defined on a mesh of 12,936 tetrahedra. Figure 6.6 shows a composite image of the Mach number field from two different viewpoints and also an image of the pressure field for airflow around an ONERA M6 wing with freestream Mach number 0.84, and a 3.06 angle of attack. The mesh has 362,712 tetrahedra and was constructed using a conformed Delaunay triangulation [2]. The top image in Figure 6.7 shows the density field defined on a curvilinear mesh of 187,395 tetrahedra from a simulation of a blunt-fin induced shock wave and turbulent boundary layer separation [48].

6.4 Suite of Rendering Approximations

To investigate the feasibility of even faster methods than the PT algorithm, a suite of four different rendering approximation methods have been developed. They tradeoff image accuracy/quality for faster image generation time. Along with highly accurate methods for direct volume rendering, the PT algorithm and the suite of fast approximations form a hierarchy of rendering methods as shown in Figure 6.8. Color photographs of the images generated by each of the approximations are shown in Figures 6.7 to 6.12.

For all these approximations, it is assumed that the transfer functions $\kappa_r(S(x,y,z))$, $\kappa_g(S(x,y,z))$, $\kappa_b(S(x,y,z))$, and $\rho(S(x,y,z))$ exist, where is $S(x,y,z)$ is the scalar field being visualized.

Figure 6.2: Two different temperature fields are shown. Each is defined on an nonconvex unstructured mesh of 13,499 tetrahedra comprising a MBB-Gehäuse solid modeling benchmark. The mesh was generated by a conformed Delaunay triangulation.

Figure 6.3: Volume rendered images of an energy field for a simulation of airflow around an airfoil defined on a mesh of 287,962 tetrahedra.

Figure 6.4: The top image shows the energy field and the bottom image shows the magnitude of velocity from a CFD simulation of incompressible flow around multiple posts defined on a curvilinear mesh of 513,375 tetrahedra.

Figure 6.5: Volume rendered images of simulation of velocity magnitude in a component of the cooling system from the Super Phoenix nuclear reactor defined on a mesh of 12,936 tetrahedra.

Figure 6.6: Flowfield for an ONERA M6 wing with freestream Mach number 0.84, and a 3.06 angle of attack on a mesh of 362,712 tetrahedra. The top image shows two views of the Mach number field; the bottom image shows the pressure field.

Figure 6.7: Volume rendered images of the density field defined on a curvilinear mesh of 187,395 tetrahedra from a simulation of a blunt-fin induced shock wave and turbulent boundary layer separation. The top image was generated using the modified PT splatting algorithm. The VOX approximation was used to generate the bottom image.

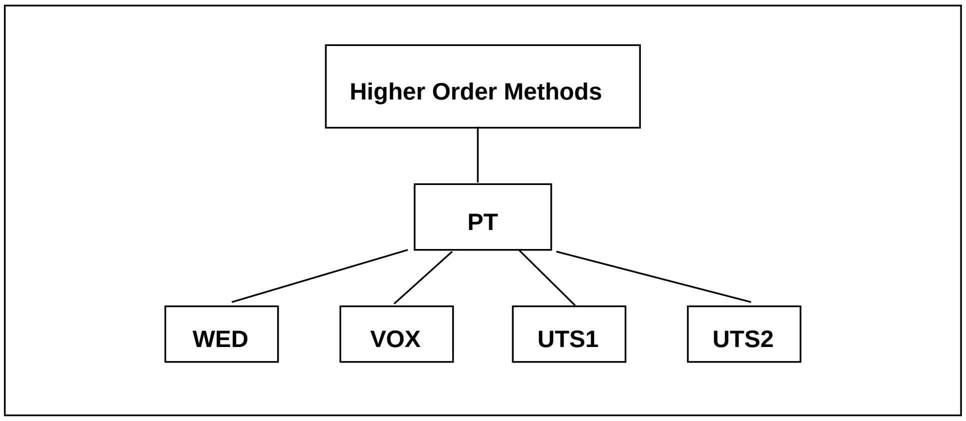


Figure 6.8: Hierarchy of approximations to the highly accurate methods for volume rendering, such as ray tracing and the projection algorithm of Max et al [61]. The approximations tradeoff image accuracy/quality for faster image generation time. PT is the Projected Tetrahedra algorithm of Shirley and Tuchman [93]. The other four methods WED, VOX, UTS1 and UTS2 are described in this Section.

6.4.1 The Voxel Approximation

The first approximation considered, the *voxel* (VOX) approximation, precomputes an average color and opacity for each cell based on the average of the scalar data at the four vertices of the cell. This value is stored as a single packed 32-bit integer in the MPVO data structure along with the cell's vertex, adjacency information, etc.

The color intensity for the red channel for a cell c is calculated as:

$$\kappa_{r_c} = \kappa_r(avgScalarData(c))$$

similarly for the other two channels. The opacity is calculated as:

$$\alpha_c = a\rho(avgScalarData(c))$$

where a is an empirically determined constant called the *attenuation factor*. Typically, $0.05 \leq a \leq 1.5$ for the VOX method. The average scalar data is calculated by averaging the data values at the four vertices of the cell.

The attenuation factor a for a given data set is calculated by adjusting a until the approximated image matches as closely as possible the corresponding image produced by the PT method. Once the attenuation factor has been determined in this way for a particular data set, it is valid for all viewpoints and viewing parameters.

For case 1, only one triangle is output, the single front-facing or back-facing triangle, for case 2, the two front-facing triangles. Each triangle is rendered with a single color and opacity, the precomputed average value for the cell described above.

Images generated using the VOX approximation are shown at the bottom of Figure 6.7 and at the top of Figure 6.9. In the image at the bottom of Figure 6.7 $a = 0.7$ and at the top of Figure 6.9 $a = 0.075$.

6.4.2 The Uniform Thickness Slab Approximation

The next approximation, the *uniform thickness slab* (UTS1) approximation, treats each cell as a slab of uniform thickness.

The opacity for each vertex v is calculated as $\alpha = a\rho(S(v))$, where a is an empirically determined attenuation factor. Interesting and useful visual effects can be achieved by varying a, between 0.1 and 2.5. The color for each vertex is simply $\kappa_r(S(v))$, $\kappa_g(S(v))$ and $\kappa_b(S(v))$. The vertex colors and opacities are calculated in a preprocessing step and stored as a single packed 32-bit integer for each vertex.

Rather than perform a case analysis to determine the splat profile, all front-facing faces are rendered. This technique is fast since the MPVO data structure contains information about which faces are front-facing.

Figure 6.9: The top image is generated using VOX approximation. The bottom image is generated with the UTS1 approximation. For comparison, an image using the modified PT algorithm is shown in Figure 3.5

Figure 6.10: Volume rendered image using the UTS1 approximation.

Images generated using the UTS1 method are shown in Figures 6.9 through 6.12. The top image in Figure 6.11 was rendered using the UTS1 approximation with $a = 0.4$. The bottom image used the UTS1 approximation with $a = 1.8$.

A variation of this method, called the UTS2 method, is to render the single front-facing or back-facing face for case 1 projections. This requires extra calculation to determine the footprint, but it results in about 25% fewer polygons being rendered. Figure 6.12 compares an image created using the UTS1 method with one created by the UTS2 approximation.

6.4.3 The Wedge Approximations

The final method considered here, the *wedge* (WED) approximation, decomposes each splat into triangles as shown in Figure 6.1. Initially, the centroid was used as the single NZT vertex

Figure 6.11: The top image was rendered using the UTS1 approximation with an attenuation factor of 0.4. The bottom image used the UTS1 approximation with an attenuation factor of 1.8.

Figure 6.12: The top image was rendered using the UTS1 approximation. The bottom image used the UTS2 approximation. An attenuation factor of 0.15 was used.

for case 2 because it was faster to calculate than the NZT vertex used by the PT method. However, the image quality was not good; therefore the NZT vertex, as calculated by the PT method, was used.

The color and opacity for each vertex is calculated in the same way as in the UTS1 method. For case 1, the color and opacity for the NZT vertex is just the color and opacity for that vertex. For case 2, the color and opacity for the NZT vertex is the average of the color and opacity at the four cell vertices. The opacity is set to zero at the remaining vertices of the footprint. The color and opacity for each vertex is calculated and stored in a preprocessing step.

The WED approximation was not significantly faster than the PT algorithm; and its image quality was noticeably inferior. Therefore this approximation method was not considered useful.

6.5 Distortions Introduced by Use of Graphics Hardware

In this section, we discuss how the use of hardware support for graphics can lead to inaccuracies in certain of the calculations in the splatting algorithm. These involve the use of Gouraud shading and opacity blending.

The PT algorithm uses either Equation 3.10 or 3.11 to calculate the opacity. Equation 3.10 is reproduced here:

$$\alpha = 1 - e^{-\rho(\lambda)(t_2 - t_1)} \tag{6.1}$$

The PT algorithm calculates the opacity at the thickest point of the splat and then linearly interpolates it over the interior of the splat, with the assumption that the opacity is zero at the periphery.

Figures 6.13, 6.14, 6.15, 6.16 and 6.17 show the distribution of opacity as calculated using linear interpolation versus the actual opacity distribution calculated using Equation 6.1. It can

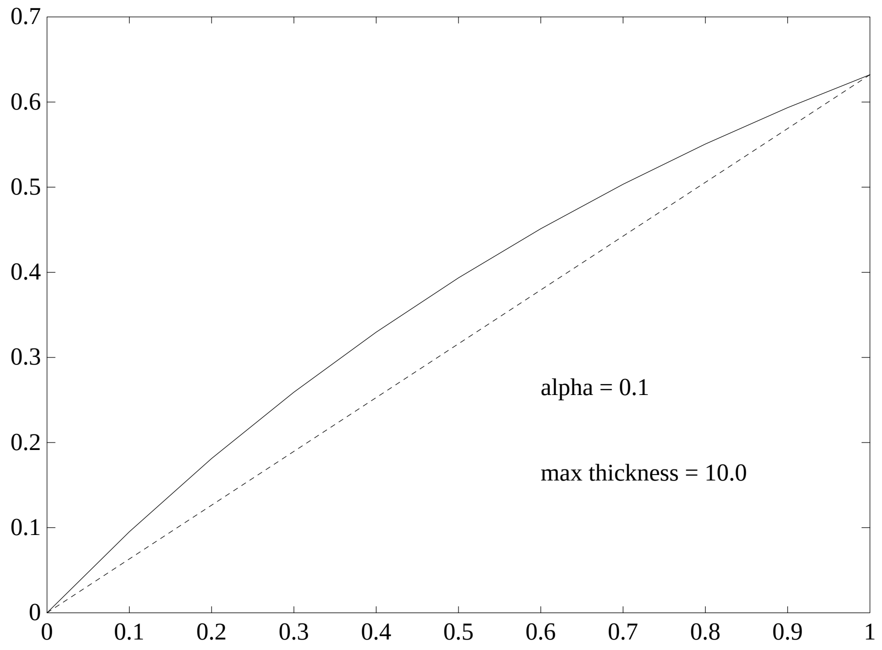


Figure 6.13: Opacity calculated using linear interpolation (dashed line) versus actual opacity calculated using Equation 6.1 (solid line).

be seen that the error in the use of linear interpolation increases with the thickness of the cell and with increasing opacity. If Equation 3.11 is used to calculate the opacity of the cell, then linear interpolation does not introduce any additional inaccuracy.

As discussed in Section 6.3, there was no perceptible difference between images produced using Equation 3.10 and Equation 3.11. However, Gouraud shading (linear interpolation) was used to create both images. To create an image using Equation 3.10 without Gouraud shading it would be necessary to either use ray tracing or else to do the scan conversion in software.

It has been pointed out [93, 103] that because only 8 bits are used for the opacity channel on many high performance workstations, this can lead to distortions in splatted images which are created using hardware opacity blending. This is due to roundoff error when a lot of cells have very small opacity. However, this was not observed to be a problem in the experiments

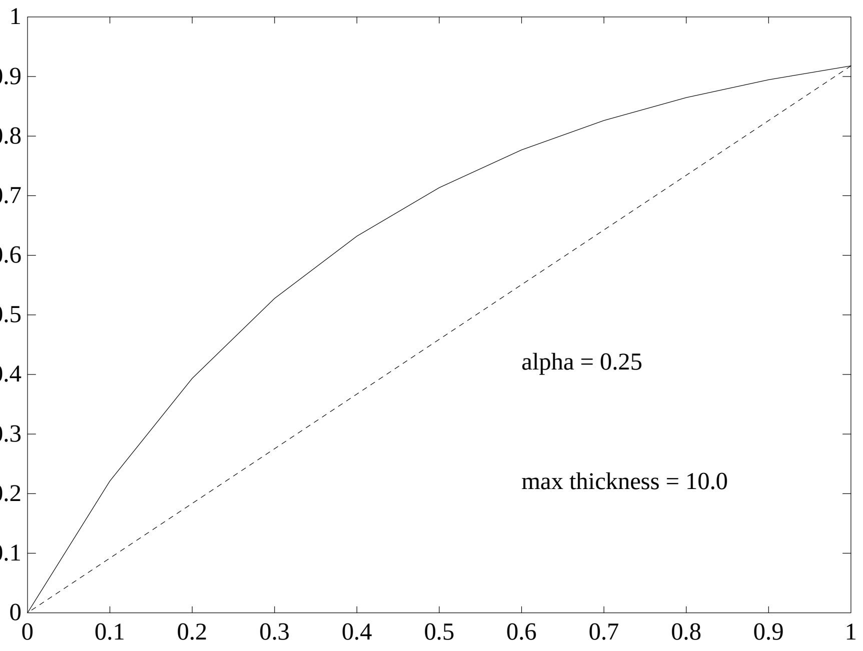


Figure 6.14: Opacity calculated using linear interpolation (dashed line) versus actual opacity calculated using Equation 6.1 (solid line).

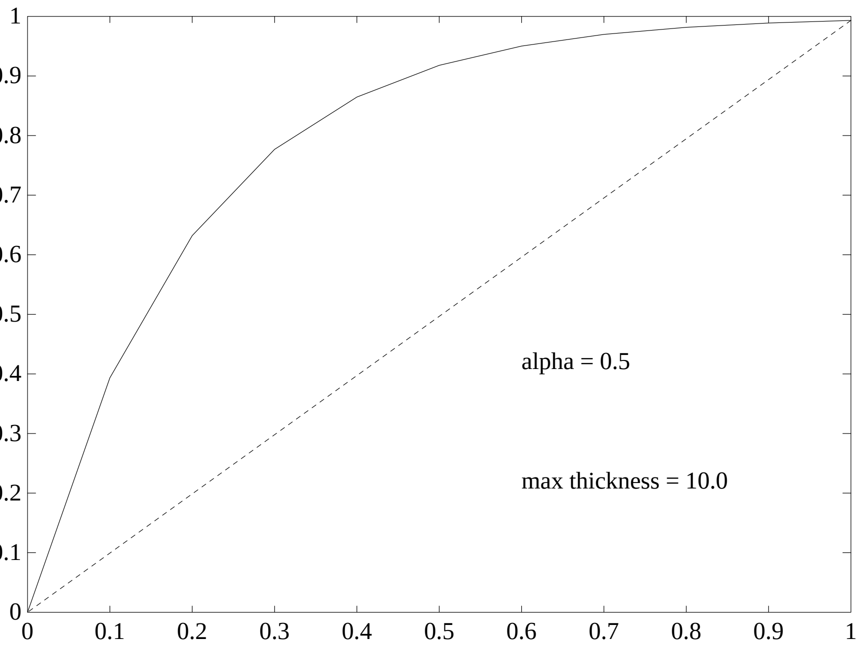


Figure 6.15: Opacity calculated using linear interpolation (dashed line) versus actual opacity calculated using Equation 6.1 (solid line).

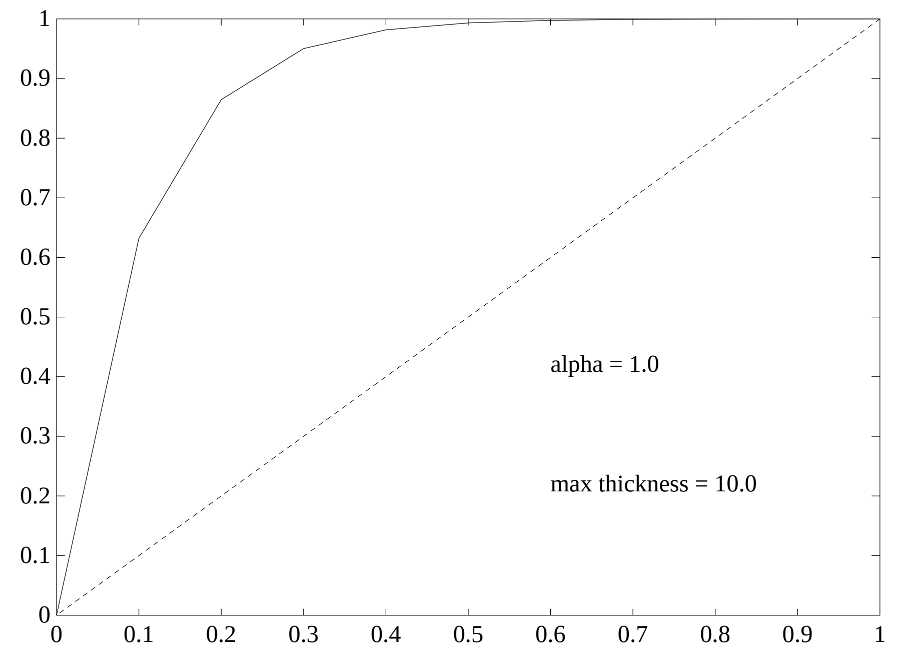


Figure 6.16: Opacity calculated using linear interpolation (dashed line) versus actual opacity calculated using Equation 6.1 (solid line).

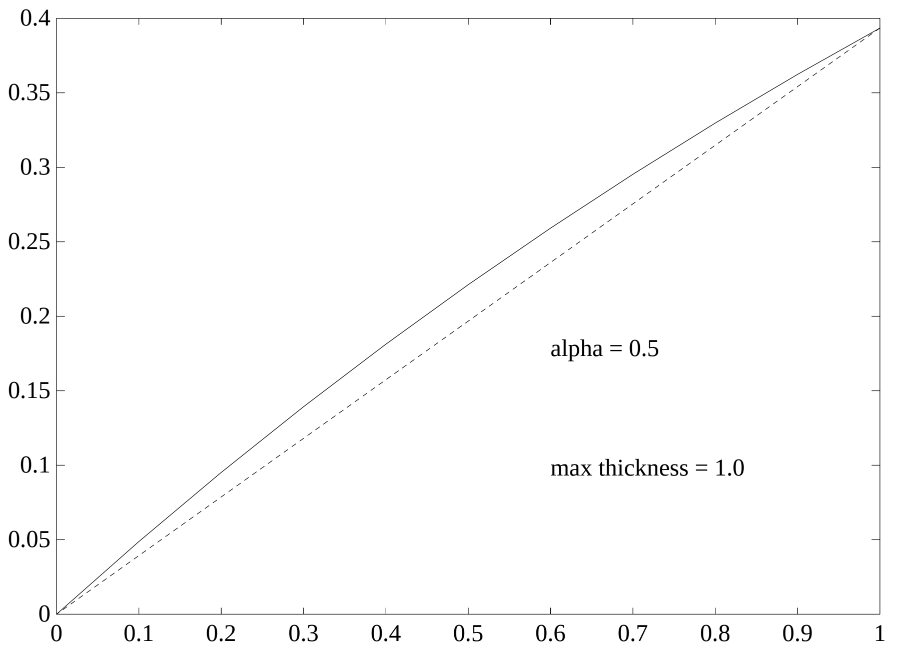


Figure 6.17: Opacity calculated using linear interpolation (dashed line) versus actual opacity calculated using Equation 6.1 (solid line).

	71,680 cells					593,920 cells				
	PT	WED	VOX	UTS1	UTS2	PT	WED	VOX	UTS1	UTS2
Serial	5.51	4.19	2.76	2.39	2.79	46.40	34.75	23.42	19.13	23.42
6CPUS	1.80	1.74	1.14	1.69	1.41	16.46	16.33	10.05	16.27	12.45

Table 6.1: Comparative volume rendering timings in seconds for the different approximation methods. All methods use the MPVO algorithm for visibility ordering and were executed on a SGI 4D/360VGX graphics workstation. Time is wall clock time.

that were performed. If an image was too transparent then the density map or attenuation factor was adjusted. It would be helpful to implement a volume renderer which did not rely on hardware blending in order to compare the images.

6.6 Comparative Timings and Image Quality Summary

Table 6.1 shows comparative timings for the PT, WED, VOX, UTS1 and UTS2 methods for 71,680 and for 593,920 tetrahedra, both for serial and parallel execution using 6 CPUS. The parallel times are discussed in Chapter 7.

The execution time for the PT splatting method, for data sets up to 600,000 cells, was brought into the interactive range by the use of graphics hardware and parallelization. Through the use of the suite of approximations described above, in particular the UTS1 method, interactivity was achieved for serial rendering of data sets of over 1,000,000 cells. Using the VOX approximation, a data set of 1,003,520 cells was rendered in parallel in 16.3 seconds. In serial mode that same data set was rendered using the UTS1 method in 34.0 seconds. In comparison, the time to render that data set using the PT method in serial was 79.6 seconds, and 30.2 seconds in parallel.

It was found that the size of the image generated had little effect on the time of generation. An image that filled the entire screen took less than 2% longer than one that was one inch square.

Since the WED approximation was not significantly faster than the PT algorithm relative to the speed of the other three approximations, it is not considered useful.

The UTS1 method was the fastest of the approximation methods for serial execution; and surprisingly, its images were the best approximation to the PT method's images, especially if the attenuation factor was carefully set in the one time process described above. Little difference was noted between the images of the UTS1 and UTS2 methods.

The VOX approximation also gave good images but usually the voxel nature of the approximation was clearly evident. The VOX approximation is more valid when the cells are relatively small in terms of pixel coverage. For parallel usage, this method is the fastest of all the approximations. An explanation of why the VOX method outperforms the other methods for parallel execution is given in Chapter 7.

CHAPTER 7

PARALLELIZATION

7.1 Introduction

Machines in the SIMD and MIMD classes of parallel architecture offer varying degrees of support for graphics. My decision to use the MIMD class for volumetric rendering of nonrectilinear data was based on an analysis of the following factors.

Massively parallel SIMD machines, such as the Connection Machine, have many thousands of processors and often have a nearest neighbor topology. The Connection Machine has a north-south-east-west (NEWS) interconnection scheme. Such machines perform best when the algorithms can be designed to use lock-step parallelism and when the data structures lend themselves to a NEWS topology. Also, machines in this class generally do not offer hardware support for 3D graphics.

The MIMD class, on the other hand, have a much smaller number of processors but offer more flexibility in terms of algorithm and data structure design. And, certain machines in this class have high performance graphics hardware.

Interactive volume rendering has been done on a Connection Machine. But it has been done for rectilinear meshes, which can take advantage of a NEWS topology because there is a one to one mapping from the mesh geometry to the processor topology. Ray integration can then proceed along a NEWS axis.

It is not clear that the splatting and graph algorithms used herein could take advantage of a SIMD architecture. The graph algorithms needed for visibility ordering an irregular mesh and the associated data structures do not seem suited to the NEWS topology.

Graph algorithms for unbounded parallelism are often based on computing the transitive closure of a boolean adjacency matrix. This is discussed further in Section 7.4.2.1.

Finding ways to effectively apply massively parallel architectures with a NEWS topology to an irregular mesh is still a open problem. If an answer is found it will benefit not only volume rendering, but more importantly, the solution of finite element problems on such meshes [29, 46, 90]. It will be interesting to investigate volume rendering using the next generation of SIMD machines which are expected to offer constant time communication between any two processors.

The two main factors that led to the choice of an MIMD architecture were the problem matching the data to the interconnection topology on SIMD machines and the availability of high performance graphics hardware on certain MIMD machines. The selection of a particular machine in the MIMD class was made based on maximizing hardware support for 3D graphics and the equipment available. This led to the use of the Silicon Graphics Power Series (SGIPS) for this work. The SGIPS is one of a class of machines that were enumerated in Section 5.

Other MIMD machines with graphics capability were considered, such as the Alliant FX/2800 and the Hypercube, but their hardware support for graphics was minimal. As will be seen below, even though the SGIPS has only eight CPUs, compared to 28 for the Alliant FX/2800 and several hundred for the Hypercube, high-end SGIPS machines have nearly 100 graphics processors that operate in parallel or pipeline at the same time the eight main CPUs are processing.

7.2 Preliminary Matters

Since it would be highly advantageous to utilize a massively parallel machine such as a Connection Machine for interactive volume rendering, we briefly review current results pertinent to this and define several terms that we will use.

The complexity of parallel algorithms can be analyzed from two different points of view. Bounded parallelism, sometimes referred to as k-parallelism, assumes the availability of k processors. Unbounded parallelism assumes the availability of an arbitrarily large number of processors. The first approach is more suited to the shared memory MIMD model of computation and the second to the massively parallel SIMD model typified by the Connection Machine.

The abstract model of parallel computation used here for discussing unbounded parallelism is the parallel random-access (PRAM) model. It is a model in which it is assumed that, in addition to the private memory for each processor, there is a shared memory, and that any processor can access any cell of that memory in unit time. This model is not physically realizable at the present time due to the complexity of linking the processors to the memory when the number of processors and the size of the memory gets very large. Nevertheless this model is commonly used to analyze parallel algorithms. Read and write conflicts can be resolved in a number of different ways; the method chosen for this discussion is the exclusive-read exclusive-write (EREW) PRAM which forbids concurrent reading or writing of a cell of shared memory.

Let the term *polylog* be defined as follows $polylog(n) = \bigcup_{k>0} O(log^k(n))$. Let S be a problem whose fastest sequential algorithm runs in time proportional to $T(n)$. A parallel algorithm for S running in time $t(n)$ with $p(n)$ processors is *optimal* if $t(n) = polylog(n)$ and the *work* $w(n) = p(n) \cdot t(n)$ is $O(T(n))$. An *efficient* parallel algorithm for S is one where the work $w(n)$ is $T(n) \cdot polylog(n)$ with time $t(n) = polylog(n)$.

The class NC^k, $k > 0$, is the class of problems that are solvable in time $O(\log^k n)$. The class $NC = \bigcup_{k>0} NC^k$ is generally accepted as a characteristic of the class of problems that can be solved with a high degree of parallelism using a feasible number of processors. In other words, the class of problems that can be solved very rapidly, in time polynomial in $\log n$, with a feasible (polynomial) number of processors coincides with NC. The class NC was first identified and characterized by Pippenger in 1979 [74] and is now called NC for Nick's class. The problems in NC^1 are the problems that are solvable by the fastest parallel algorithms.

When discussing bounded parallelism, the notation $T_k(n)$ is used to mean the time complexity of the parallel algorithm when k processors are used.

Speedup as used herein is the ratio of the time to execute an efficient serial program for a calculation to the time to execute a parallel program for the same calculation on N processors. *Efficiency*, as used henceforth, is the ratio of speedup to the number of CPUs. Efficiency for bounded parallelism has a different meaning than was given above for unbounded parallelism.

7.3 Software Support for Parallelism

On the SGIPS, the user is given the choice of parallelizing compilers or a number of explicit parallelism constructs. All algorithms herein were expressed using explicit parallelism.

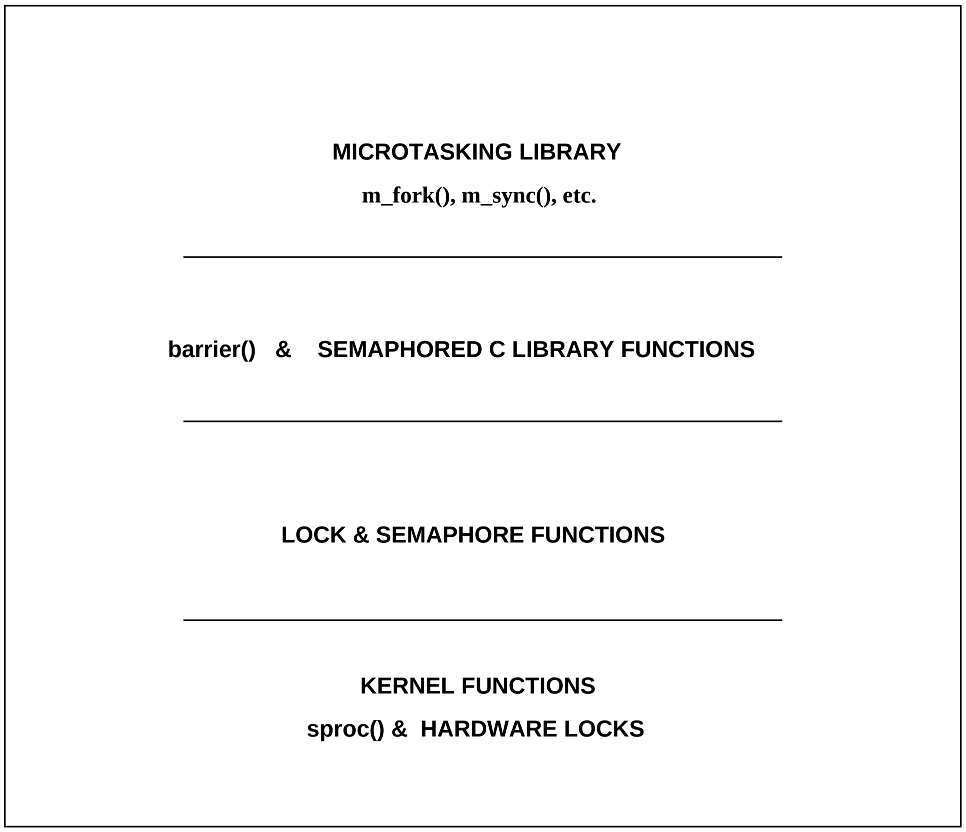


Figure 7.1: Hierarchy of support for parallelism on the SGI.

Support for explicit parallelism on the SGIPS is hierarchically arranged as shown in Figure 7.1. The functions in the higher levels of the hierarchy are based on the functionality of the lower levels.

The kernel function *sproc()* is a variant of the standard *fork()* call. *sproc()* creates threads of execution that share resources, including virtual address space, files, etc, with the parent thread and with any sibling processes in the same process share group.

At level 2, four lock functions are offered *setlock(lock)*, *csetlock(lock,spins)*, *wsetlock(lock,spins)* and *testlock(lock)*. The conditional lock *csetlock(lock,spins)* returns 1 if a lock is acquired within *spin* spins, otherwise it returns 0. The *wsetlock()* is the same as *setlock()* except the number of spins can be specified. Neither *setlock()* nor *wsetlock()* return until the lock is acquired.

The level 2 semaphores were tested and found to be very inefficient. Typically, they were an order of magnitude slower than locks.

The barrier function at level 3 creates a rendezvous point for threads of execution. It makes a process wait for the other threads to reach the same point before continuing.

The microtasking library functions are patterned after the Sequent Computer Systems parallel programming primitives. They include *m_fork()*, *m_park_procs()*, *m_sync()*, *m_rele_procs()*, *m_lock()*, etc.

All lock types were tested with a varying number of spins and waits between spins. It was found that the optimum performance was obtained with the default lock *setlock()*, which spins up to 600 times before relinquishing the CPU. The default lock was found to be 10% faster than the m_lock().

The microtasking m_fork() was tested versus the more primitive library call sproc(). Surprisingly, m_fork() performed better (on the order of 20%) for the work herein. On the other hand the microtasking barrier m_sync() was 10% slower than the primitive barrier() call.

7.4 Algorithm Parallelization

As mentioned in Chapter 5, the specific SGIPS model utilized for this work was the 4D/360VGX with six 33MHz CPUs and 128 MB of physical memory. All timings given in this Chapter are wall clock times and are based on the use of this configuration. As mentioned earlier, in addition to the 6 CPUs, there are 85 dedicated graphics processors operating in parallel.

A diagram of the overall runtime algorithm is shown in Figure 7.2. After phase II of the MPVO algorithm, as described in Chapter 4, is completed, phase III of the MPVO algorithm

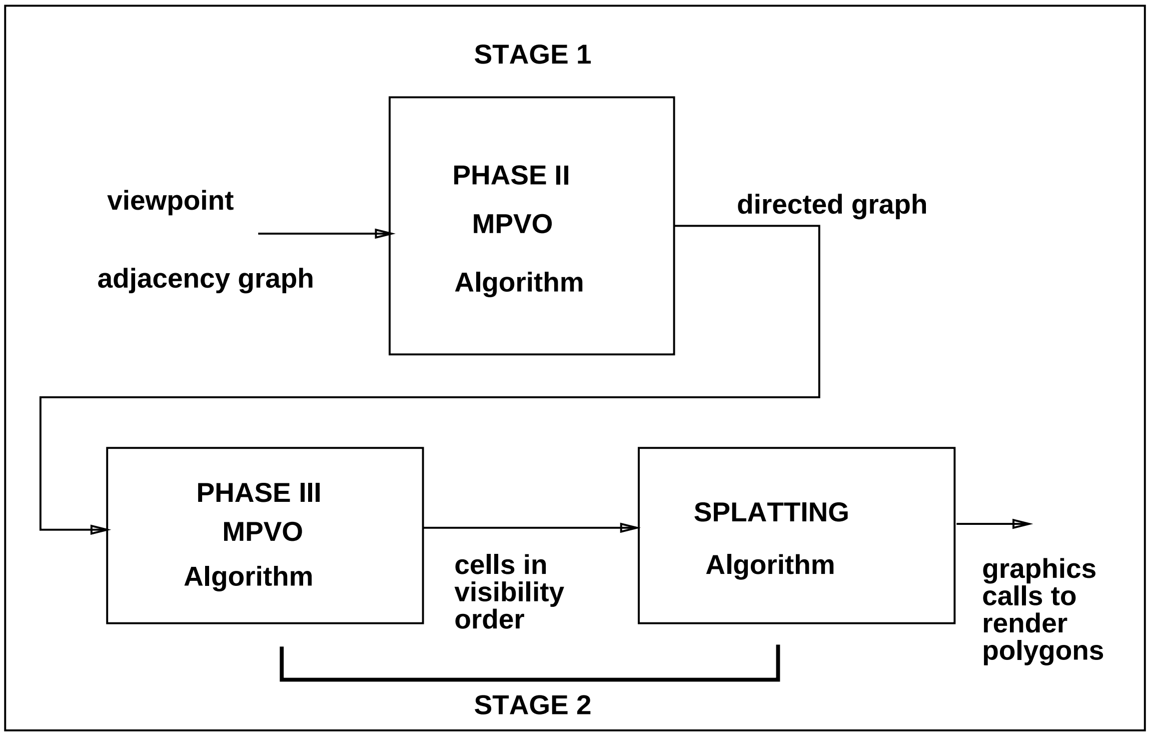


Figure 7.2: Overall runtime algorithm

and the splatting algorithm are executed as a unit. As described in Chapter 6, several different splatting methods were tested, including the PT, VOX, WED, UTS1 and UTS2 methods.

Phase II of the MPVO algorithm will now referred to as Stage 1 and the combined phase III MPVO algorithm and the splatting algorithm will be called Stage 2. Several different parallelization schemes for Stage 1 and Stage 2 will now be described. Timings reported for Stage 1 and Stage 2 combined constitute the time for the overall volume rendering process.

In what follows, it is assumed that each cell is identified by a distinct integer.

7.4.1 Parallelization of Stage 1

In Stage 1, the adjacency graph, which was created in a preprocessing step, is converted into a directed graph. This is done by evaluating the plane equation for each face with the coordinates of the viewpoint. This results in the arrow field being set for each face of each cell.

Two different versions of the parallelization of Stage 1 are presented.

Version 1 requires only one sweep of the cells and uses no locks. Therefore, a process must set all the arrows for each of the cells for which it is responsible. This means the plane equation is evaluated twice for each shared face. In spite of this, the version 1 algorithm was faster than all alternatives tried. Each process uses a shadow queue to collect the source cells it finds. At the end of the sweep all shadow queues are gathered into Output Queue. Either one process can do the gathering or it can be done in parallel.

An alternative to the use of shadow queues is to use just one queue and require a lock to access it. Usually only a minority of the cells are source cells and so the queue will not be accessed heavily. It was found that the use of one queue and locks took 20% longer, on average, than the use of shadow queues.

The details of the version 1 algorithm are shown below. It is assumed the process ids (pids) are zero based.

```
STAGE 1 PARALLEL ALGORITHM - VERSION 1
    for $cell$ = pid +1; $cell < totalNumCells$; $cell$ += $numbProcs$
        for each face of $cell$
            set arrow
        set numInbound[$cell$]
        if numInbound[$cell$] == 0
            enqueue $cell$ on shadow queue
    barrier
    move cells in shadow queues to Output Queue
```

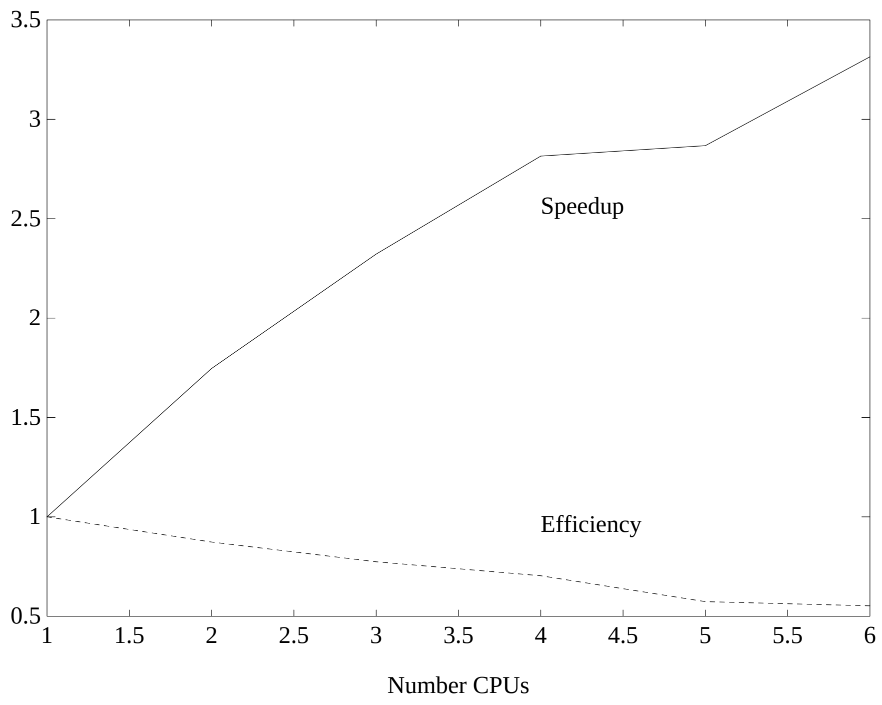


Figure 7.3: Performance data for Version 1 of Stage 1 for 1,003,520 tetrahedra.

Performance data for Version 1 of Stage 1 is shown in Figure 7.3 for 1,003,520 tetrahedra. Table 7.1 gives typical timings for 71,680, 593,920 and 1,208,320 tetrahedra. The complexity of the version 1 algorithm is $T_k(f) = O(f/k)$, where f is the number of faces in the mesh.

The cause of the flattening of the speedup curve between four and five processors is unknown; it occurred in all runs. The speedup for 71,680 and for 593,920 tetrahedra were found to be roughly equivalent to those shown for 1,003,520 tetrahedra.

The serial times given in Table 7.1 are for the most efficient serial algorithm which is the algorithm described in Section 4.5.1.2. A serial adaptation of the parallel Stage 1 version 1 algorithm described above was found to be 7% slower.

The second version of Stage 1 avoids evaluating the plane equation twice per shared face. To do this, the lower numbered cell assumes responsibility for setting the arrow for the higher

Number Processors	*time for 71,680 tetrahedra*	*time for 593,920 tetrahedra*	*time for 1,003,520 tetrahedra*
Serial	0.83 sec.	7.21 sec.	12.33 sec.
2 CPUs	0.48 sec.	4.16 sec.	7.06 sec.
3 CPUs	0.36 sec.	3.13 sec.	5.31 sec.
4 CPUs	0.30 sec.	2.59 sec.	4.38 sec.
5 CPUs	0.30 sec.	2.55 sec.	4.30 sec.
6 CPUs	0.26 sec.	2.19 sec.	3.72 sec.

Table 7.1: Typical timings for Stage 1 Version 1. Time is wall clock time.

numbered cell it shares a face with. On the one hand, this saves on evaluation of the plane equation, on the other hand, some overhead is involved since (1) when setting an arrow, the adjacent cell which shares the face must be searched to find the desired face, this is described further below, and (2) a separate sweep is required to set the numInbound field for each cell and find the source cells.

STAGE 1 PARALLEL ALGORITHM - VERSION 2

```
for $cell = pid + 1$; $cell < totalNumCells$; $cell$ += $numbProcs$
    for each $face$ of $cell$
        if ($cell <$ adjacent cell which shares $face$)
            set both arrows
barrier
for $cell = pid + 1$; $cell < totalNumCells$; $cell$ += $numbProcs$
    set numInbound[$cell$]
    if numInbound[$cell$] == 0
        lock(Qlock)
            enqueue $cell$ on Output Queue
        unlock(Qlock)
```

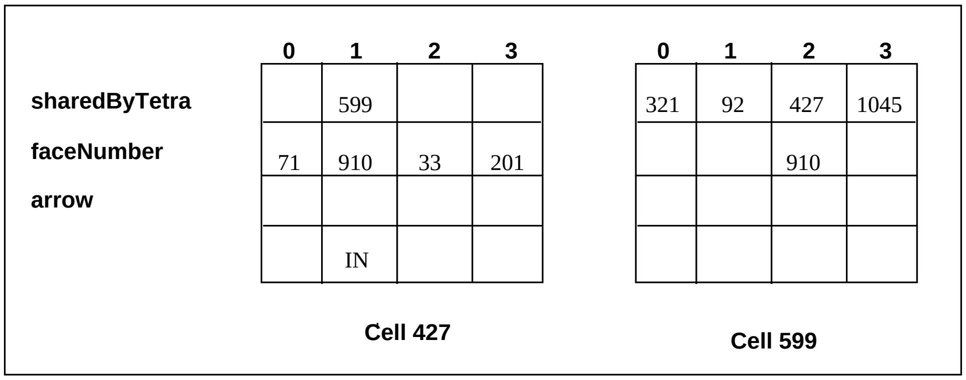


Figure 7.4: Data structures for two cells. The arrow for face 910 in Cell 427 has just been set to IN. Now it is necessary to find that same face in Cell 599 so the arrow can be set to OUT. This requires searching each sharedByTetra field in Cell 599 to find Cell 427.

Version 2 was more than twice as slow as Version 1, even when locks were not used. The factor which most adversely effected performance was the search to find the adjacent shared face; see Figure 7.4.

```
SEARCH REQUIRED IN STAGE 1 PARALLEL ALGORITHM - VERSION 2
    for each face
        if (sharedByTetra == myTetra)
            set arrow
            break
```

A field could be added to each cell's data structure to contain a pointer directly to the adjacent face. This would increase memory usage. Another possibility is to replace each existing adjacent tetrahedra (sharedByTetra) pointer with a pointer directly to the adjacent face. The problem with this approach is that it would be difficult to find other faces in the adjacent cell.

If the data structures could be set up so $face[i]$ of cell A which is shared with cell B could also be found in $face[i]$ of cell B, then a search could be avoided. Thus a more efficient algorithm would result.

For tetrahedral cells this becomes a graph coloring problem, which can be called the *Triangle 4-Coloring Problem*; it is an unsolved problem.

Let T be a triangulation in R^3. A triangle coloring of T is a 4-coloring of its triangles so that each tetrahedron of T has four differently colored triangles. The problem is: Does each triangulation in R^3 have a triangle coloring?

It is known that five colors are sufficient to guarantee that each tetrahedron can get four differently colored triangles. It is also known that there exist 3D triangulations that are not 4-colorable if the outside cell is included.

7.4.2 Parallelization of Stage 2

7.4.2.1 Introduction

In Stage 2, the directed graph created in Stage 1 is enumerated in topological order. This ordering is a visibility ordering of the cells of the mesh. As each cell is output in visibility order, it is splatted as described in Chapter 6.

The parallelization of the splatting process is straightforward. One process takes a cell through the entire splatting procedure and uses spin locks to ensure mutual exclusion when making graphics calls.

Either a depth first search (DFS) or a breadth first search (BFS) can be used for the topological sort. It was initially conjectured that a DFS was inherently sequential and so not solvable in poly-log parallel time [80, 82, 83, 95]. In 1989, however, Bongiovanni and Petreschi published a DFS algorithm using $O(n^4)$ processors which runs in time $O(\log^2 n)$. However, the algorithm is not efficient. DFS has a serial time of $O(n + e)$ [99] where n and e are the number of nodes and edges, respectively. Therefore, the processor-time product for the parallel algorithm is far in excess of the time required to solve the problem serially.

The best known bounds for a BFS algorithm are $O(\log^2 n)$ using $O(n^{2.376})$ processors [51, 37]. This means BFS is placed in NC^2. Techniques related to transitive closure are used to exhibit the algorithm. However, since BFS can be solved sequentially in time $O(n + e)$ [99], this algorithm is not efficient for the same reason as given above.

Reghbati and Corneil [82, 80] describe an algorithm for BFS using bounded parallelism. It has complexity $T_k = \sum_{i=1}^{n}(\lceil d_i/k \rceil + 1) + L\lceil \log_2 k \rceil$, where d_i is the degree of vertex i, L is the distance of the farthest node from the start node, and n is the number of nodes. They conclude that k-parallel BFS is superior to serial search when the average vertex degree is at least $\lceil \log_2 k \rceil + 5$. For a tetrahedral mesh the average vertex degree is approximately 2. This implies that k-parallel BFS may not be superior to serial search for a graph of average vertex degree 2. The parallel algorithms for BFS described below did show some speedup (1.21X – 1.46X) for four or more CPUs. However, the efficiency was relatively low.

Two versions of parallel algorithms for Stage 2 are given. Both are based on the fact that the output of the Phase III BFS algorithm given in Chapter 4 consists of sequences of cells that do not obstruct each other. We refer to each sequence as a layer. All the cells in a layer can be rendered concurrently.

When Stage 2 begins, the Output Queue (OQ) contains the source cells gathered in Stage 1. This constitutes the first layer of the BFS. All eligible successors of the current layer constitute the next layer.

7.4.2.2 Version 1

In the first version of Stage 2, all processes do the same task. Each process takes a cell from the Output Queue, splats it and then updates the graph as shown below. Each process has a shadow queue on which it enqueues the cells for the next layer. When a layer is complete, all the shadow queues are merged into the global Output Queue, and then this process is repeated.

STAGE 2 PARALLEL ALGORITHM - VERSION 1

```
while OQ not empty
        for $i$ = pid; $i$ < Length(OQ); $i$ += numbProcs
            $cell$ = OQ[$i$];
            if $cell.actual$
                splat($cell$)
                lock(gPipeLock)
                    render($cell$)
                unlock(gPipeLock)
                numRendered[pid]++
            else numVirtual[pid]++
            for each face of $cell$
                if outbound arrow
                    lock(adjCellLock)
                        if adjCell.numInbound > 1
                            decrement adjCell.numInbound
                        else enqueue adjCell on shadowQ
                    unlock(adjCellLock)
        barrier
        move cells on shadowQ's to OQ
        barrier
    end while
    if $pid == 0$
        for $i = 0$ to $numbProcs - 1$
            $total\_numRendered$ += $numRendered[i]$
            $total\_numVirtual$ += $numVirtual[i]$
        if (total_numRendered + total_numVirtual) $\neq$ total numCells
            output cycle warning
```

Times for version 1 of Stage 2, for the BFS enumeration only, are given in Table 7.2; times for the entire Stage 2 are given in Table 7.3. Performance data is given in Figures 7.5, 7.6 and 7.7.

Spinlocks are used to guarantee mutually exclusive access to a shared variable in each cell and also to the graphics pipe in the splatting routine. Only one lock is required for the graphics pipe. However, each cell has its own shared variable. If there are enough hardware locks (hardlocks), then one is used per cell. If there are not enough, there are two alternatives. Each hardlock can guard several cells, or multiplexed locks [71] can be used.

Multiplexed locks use a single hardlock to guard multiple softlocks. Each cell has its own softlock which is a boolean software variable. In order to access a cell's shared variable, a process spins on that cell's softlock. Then it spins on the associated hardlock before locking

Number Processors	*time for 71,680 tetrahedra*	*time for 593,920 tetrahedra*	*time for 1,003,520 tetrahedra*
Serial	0.45 sec.	4.34 sec.	7.61 sec.
2 CPUs	0.61 sec.	5.81 sec.	10.45 sec.
3 CPUs	0.45 sec.	4.28 sec.	7.69 sec.
4 CPUs	0.38 sec.	3.52 sec.	6.31 sec.
5 CPUs	0.34 sec.	3.13 sec.	5.57 sec.
6 CPUs	0.33 sec.	2.97 sec.	5.22 sec.

Table 7.2: Typical timings for Stage 2 version 1, for the BFS enumeration only. Time is wall clock time.

Number Processors	*time for 71,680 tetrahedra*	*time for 593,920 tetrahedra*	*time for 1,003,520 tetrahedra*
Serial	4.64 sec.	39.25 sec.	67.22 sec.
2 CPUs	3.12 sec.	30.51 sec.	54.15 sec.
3 CPUs	2.28 sec.	21.63 sec.	38.40 sec.
4 CPUs	1.80 sec.	17.53 sec.	31.05 sec.
5 CPUs	1.57 sec.	15.33 sec.	27.21 sec.
6 CPUs	1.47 sec.	14.19 sec.	25.48 sec.

Table 7.3: Typical timings for Stage 2 version 1 including the BFS enumeration and the PT algorithm for splatting. Time is wall clock time.

the softlock. Once the softlock is locked, the hardlock is unlocked. This procedure allows the shared variables of more cells to be accessed simultaneously.

7.4.2.3 Version 2

Version 2 of Stage 2 is an experiment to determine if one CPU can perform the BFS faster than multiple CPUs. Multiple CPUs require synchronization to access the shared variable *numInbound* in each cell. In addition, the granularity of the BFS is quite small.

Version 2, uses one CPU to traverse the graph in BFS order while the remaining CPUs do the splatting. Two queues are used as shown in Figure 7.8. When the Output Queue is emptied and the input queue is filled with the next set of cells that may be rendered concurrently, the

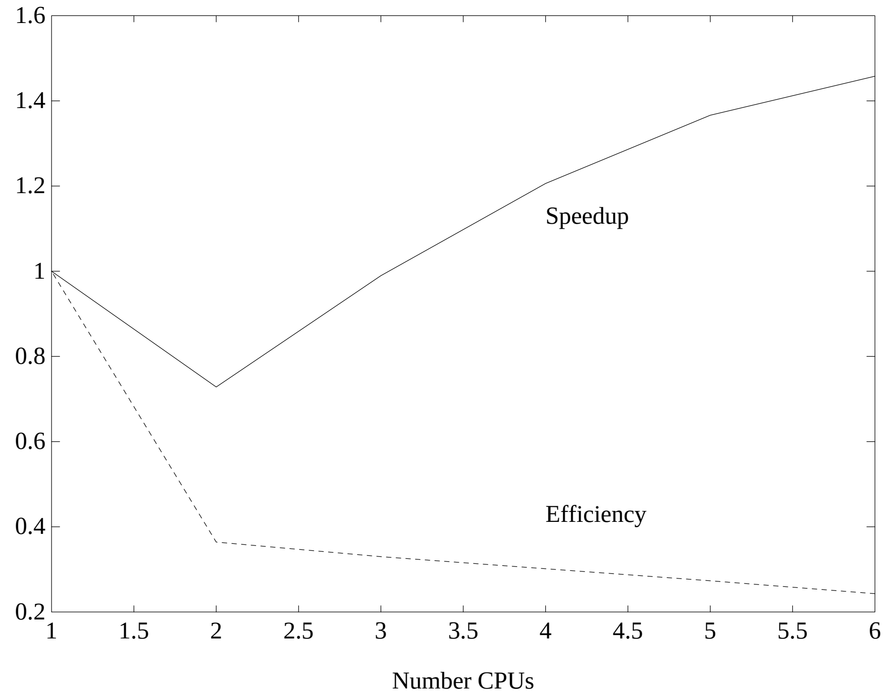


Figure 7.5: Performance data for Stage 2 version 1 for the BFS enumeration only, for 1,003,520 tetrahedra.

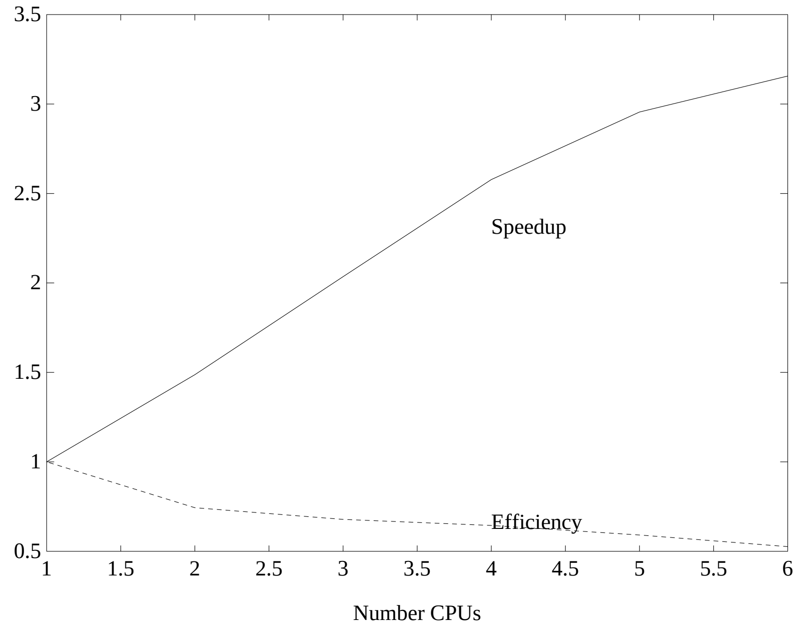


Figure 7.6: Performance data for Stage 2 version 1 for the BFS enumeration and splatting using the PT algorithm for 71,680 tetrahedra.

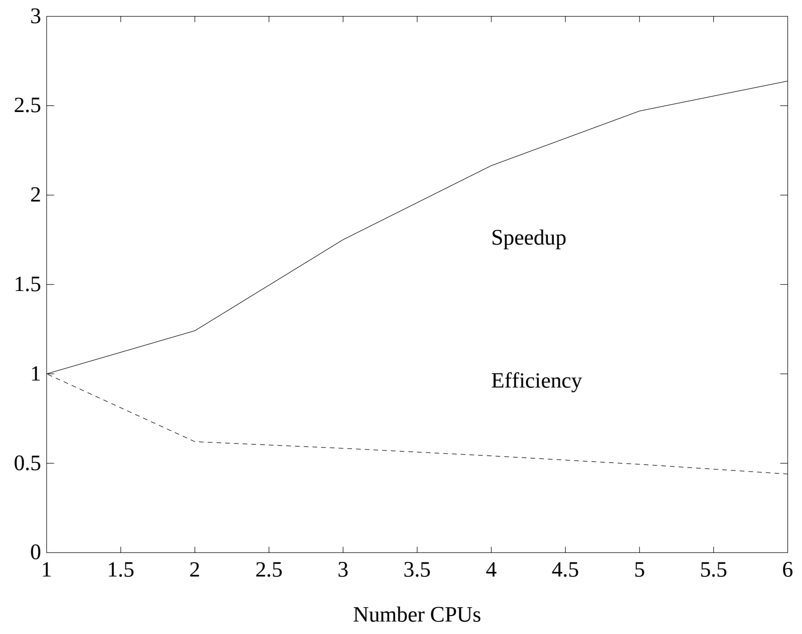


Figure 7.7: Performance data for Stage 2 version 1 including the BFS enumeration and splatting using the PT algorithm for 1,003,520 tetrahedra.

pointers to the two queues are swapped. The pseudocode for this is shown below. The cycle test and splatting procedure is not shown. It is the same as in version 1 of Stage 2 shown earlier.

STAGE 2 PARALLEL ALGORITHM - VERSION 2

```
while $OQ$ not empty
    if ($pid == 0$)
        for each $cell$ in $OQ$
            for each face of $cell$
                if arrow outbound
                    if adjCell.numInbound $> 1$
                        decrement adjCell.numInbound
                    else enqueue adjCell on input queue
```

Number Processors	*time for 71,680 tetrahedra*	*time for 593,920 tetrahedra*	*time for 1,003,520 tetrahedra*
2 CPUs	3.0 sec.	24.68	43.80 sec.
3 CPUs	2.24 sec.	17.98 sec.	31.26 sec.
4 CPUs	1.95 sec.	15.29 sec.	27.59 sec.
5 CPUs	1.82 sec.	14.00 sec.	24.41 sec.
6 CPUs	1.77 sec.	13.77 sec.	25.91 sec.

Table 7.4: Typical timings for Stage 2 version 2 using the PT algorithm, with load balancing. Time is wall clock time.

```
else /* all other processes */
    for $i$ = pid-1; $i <$ Length(OQ); $i$ += numbProcs
        splat OQ[$i$];
barrier1
switch pointers on $OQ$ and input queue
barrier2
```

Timings for Stage 2 version 2 are given in Table 7.4. The times are very similar to those shown for version 1.

The amount of time each process waited at barrier 1 was measured for Stage 2 Version 2. It was found that on average the process (*pid0*) which performed the BFS and filled the Input Queue was idle 70% of the time. To balance the load, when *pid0* finished filling the Input Queue, it was assigned to assist with the splatting. The idle time was then remeasured and no process was found to be idle more than 8% of the time. Comparative timings for Stage 2 Version 2 are given in Table 7.5. The table shows that the benefits from load balancing diminish as more processes are added. This is to be expected since only the load of one processor is being

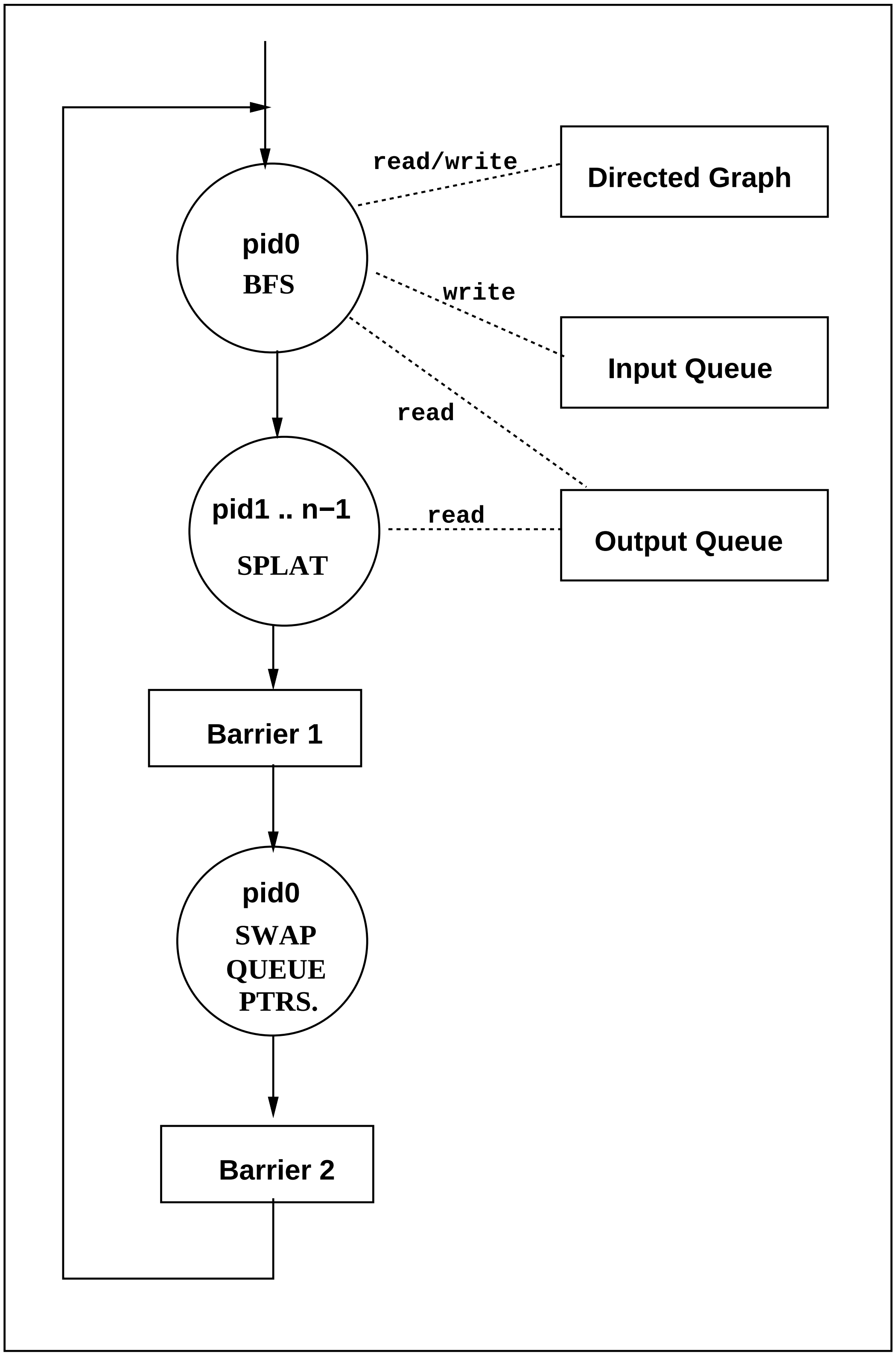


Figure 7.8: Parallelization of Stage 2 Version 2. The Output Queue is initialized with the source cells by Stage 1.

Number Processors	*Time with No Load Balancing*	*Time With Load Balancing*
2 CPUs	4.83 sec.	3.00 sec.
3 CPUs	2.75 sec.	2.24 sec.
4 CPUs	2.13 sec.	1.95 sec.
5 CPUs	1.85 sec.	1.82 sec.
6 CPUs	1.75 sec.	1.77 sec.

Table 7.5: Times for Version 2 of Stage 2, with and without load balancing.

Number Processors	*time for 71,680 tetrahedra*	*time for 593,920 tetrahedra*	*time for 1,003,520 tetrahedra*
Serial	1.28 sec.	11.55 sec.	19.94 sec.
2 CPUs	1.09 sec.	9.91 sec.	29.05 sec.
3 CPUs	0.81 sec.	7.61 sec.	13.00 sec.
4 CPUs	0.68 sec.	6.11 sec.	10.69 sec.
5 CPUs	0.64 sec.	5.68 sec.	9.87 sec.
6 CPUs	0.59 sec.	5.16 sec.	8.94 sec.

Table 7.6: Typical times for the parallelization of the MPVO algorithm

balanced, and the portion of the overall load that one processor is responsible for decreases as the number of processors increase.

7.4.2.4 Performance Results for Overall Volume Rendering System

All subsequent results are based on the use of the version 1 algorithms for both stages. The Stage 1 version 1 algorithm was clearly more efficient. Since both versions of the Stage 2 algorithm were about equal in performance, it was a somewhat arbitrary decision to use the version 1 algorithm.

Typical timings for the parallelization of Stage 1 and the BFS of Stage 2, which together comprise the MPVO algorithm, are given in Table 7.6.

	71,680 cells					593,920 cells				
	PT	WED	VOX	UTS1	UTS2	PT	WED	VOX	UTS1	UTS2
Serial	5.51	4.19	2.76	2.39	2.79	46.40	34.75	23.42	19.13	23.42
2 CPUs	3.61	3.10	1.90	1.94	2.12	34.51	28.62	17.85	18.91	18.83
3 CPUs	2.66	2.24	1.41	1.76	1.59	24.69	21.24	13.01	16.96	14.14
4 CPUs	2.15	1.90	1.28	1.72	1.45	20.02	17.85	10.99	16.53	12.78
5 CPUs	1.91	1.77	1.17	1.72	1.44	17.91	16.94	10.47	16.60	12.77
6 CPUs	1.78	1.74	1.14	1.69	1.41	16.46	16.33	10.05	16.27	12.45

Table 7.7: Comparison of typical timings, in seconds, for Stages 1 and 2 combined, utilizing the PT, WED, VOX, UTS1 and UTS2 splatting methods.

Comparative serial and parallel timings for the PT projection method and the four approximations to it, the VOX, WED, UTS1 and UTS2 methods described in Chapter 6, are given in Table 7.7. Comparative performance data is shown in Figure 7.9.

Table 7.7 shows that the UTS1 projection method is the fastest serial method and the VOX method is the fastest parallel method. This Table also shows that the VOX and UTS2 methods are faster than the UTS1 method when two or more CPUs are used. The reason for this is given below.

The VOX and the UTS2 methods calculate the footprint type of each cell in order to render the minimum number of triangles and also to determine the order of the vertices so a tmesh can be used. The UTS1 method does not do this; it just checks each face's arrow and if it is outbound, it renders that face. Therefore, the UTS1 method requires less calculation than the VOX or UTS2 methods. This is why the UTS1 method is faster for the serial case.

With additional CPUs, the rate of production of triangles increases to the point that the graphics pipe starts to saturate (this is explained below). At that point, the VOX and UTS2 methods become faster than the UTS1 method since they both produce about 20% fewer triangles than the UTS1 method.

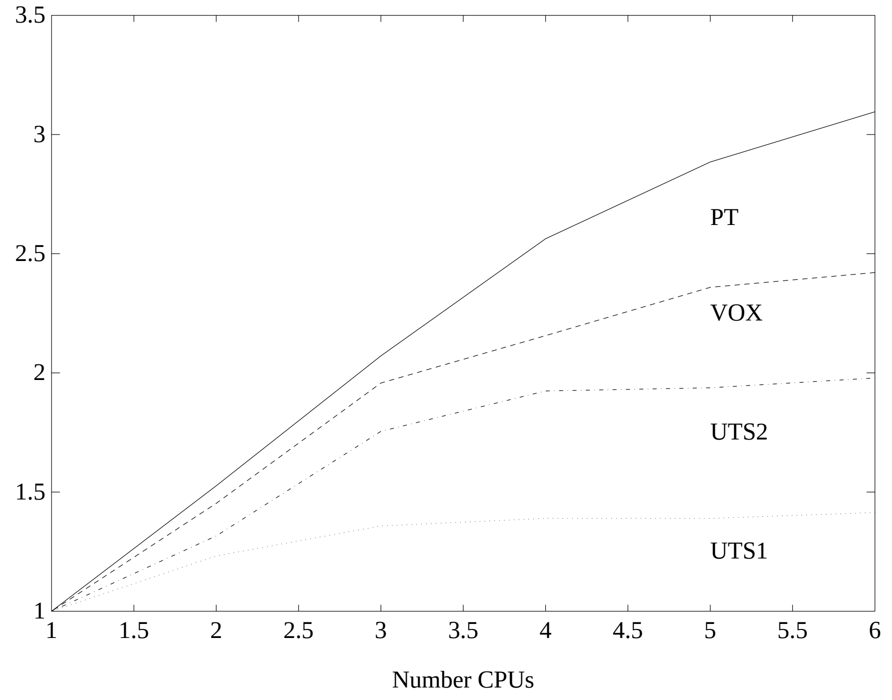


Figure 7.9: Performance data for stages 1 and 2 combined, utilizing the PT, VOX, UTS1 and UTS2 splatting methods for 71,680 tetrahedra.

	71,680 cells		593,920 cells		1,003,520 cells	
	PT	UTS1/VOX	PT	UTS1/VOX	PT	UTS1/VOX
Serial	5.5 sec.	2.4 sec.	46.4 sec.	19.1sec.	79.6 sec.	34.0 sec.
6 CPUs	1.8 sec.	1.1 sec.	16.5 sec.	10.1 sec.	30.2 sec.	16.3 sec.

Table 7.8: Typical times for the PT method, and the fastest serial and parallel methods, the UTS1 approximation and the VOX approximation, respectively, for 71,680, 593,920 and 1,003,520 cells.

The code executed by the UTS2 method and the VOX method is quite similar. However, when parallelized, the VOX approximation is faster than the UTS2 method since the VOX method uses only one color for the entire footprint. The UTS2 method specifies a color for each vertex of each triangle. Hence, for footprint number 2, which is rendered as two triangles by both methods, the VOX method makes three fewer graphics calls per splat than the UTS2 method.

Table 7.8 shows typical times for the PT method, and the fastest serial and parallel methods for 71,680, 593,920 and 1,003,520 cells.

7.5 Performance Analysis

All of the above performance curves show speedup falling off significantly with the use of five or more CPUs. The curve labeled NR (normal rendering) in Figure 7.10 shows the speedup for Stage 2 of the PT algorithm for 71,680 tetrahedra. The basis for the all speedup calculations for this figure is the parallel algorithm operating on one CPU. The curve labeled IDEAL is the maximum possible theoretical speedup. To locate the bottleneck causing the fall off in performance, a number of experiments were performed.

The possible sources of the bottleneck were: the locks, the rate at which the CPUs could compute and transfer data to the MPlink bus (for subsequent delivery to the graphics pipe), the bandwidth of the MPlink bus itself, or the flow rate through the graphics pipe. To determine which source was responsible, it was necessary to devise methods to eliminate one or more of these sources and then remeasure the performance to see if it had improved.

First, all graphics rendering calls and lock usage were commented out of the algorithms. This prevented the MPlink bus, the graphics pipe and the locks from effecting the speedup in performance. The resulting speedup is shown by the curve labeled NGC in Figure 7.10. As can be seen, this eliminated the bottleneck since the speedup curve improved significantly. Therefore the bottleneck must be either the locks, the MPlink bus or the graphics pipe.

Next, all lock usage was uncommented. The results were unchanged. This indicated that the locks were not the cause.

Then, the graphics calls were replaced by equivalent non graphics function calls with an identical number of arguments and a one line body making a reference to a global variable. (The particular graphics calls used require the CPU to communicate to the MPlink bus a single memory reference to a block of four words in main memory. The MPlink bus then uses a DMA burst to move the four words to the graphics pipe.)

The results of this experiment are shown by the curve labeled ENG in Figure 7.10. This curve shows a slight fall off in speedup but not nearly so marked as when actual graphics calls were used (curve NR). This indicated that at this level of performance the CPUs were starting to become the bottleneck.[1] However, when actually rendering, the level of performance was far below this so the real bottleneck still remained to be found. It had to be either the MPlink bus or the graphics pipe.

[1]This experiment (ENG) gave an indication of the potential speedup of my algorithms when executed on these CPUs.

If the same amount of data was transferred over the MPlink bus, but the graphics pipe was required to do less work, then an improvement in performance over normal rendering (NR) would indicate that the graphics pipe was the bottleneck. One way to do this is to render the same number of triangles but make the size of the triangles much smaller. The MPlink bus has the same loading, but the graphics pipe has less to do since it has to set fewer pixels. The result when all triangles are scaled down by 0.01 is shown by the curve labeled SCA in Figure 7.10. Since an improvement in performance was shown over normal rendering, this indicated that the graphics pipe was the bottleneck.

Another experiment performed was to double the number of triangles in each tmesh. The tmesh was described in Chapter 5. In this case the traffic over the bus increases only marginally per triangle compared to the graphics pipe. That is, the graphics pipe must render another complete triangle, whereas the bus need only make two additional DMA bursts. If this experiment yielded a speedup, then it would indicate that the graphics pipe was not the bottleneck. The results for this experiment, shown by the curve labeled BIG in Figure 7.10, indicate a significant decrease in performance. This result supported the conclusion that the graphics pipe is the bottleneck.

To get peak performance from the DMA transfers as described in Section 5.2, all data passed to the graphics pipe is quadword aligned. However, when the data was forced to be nonquadword aligned, there was no measurable change in performance. This was even further corroboration that the CPUs and the MPlink bus were not the bottleneck.

To pinpoint which graphics subsystem was the bottleneck, one additional experiment was performed.

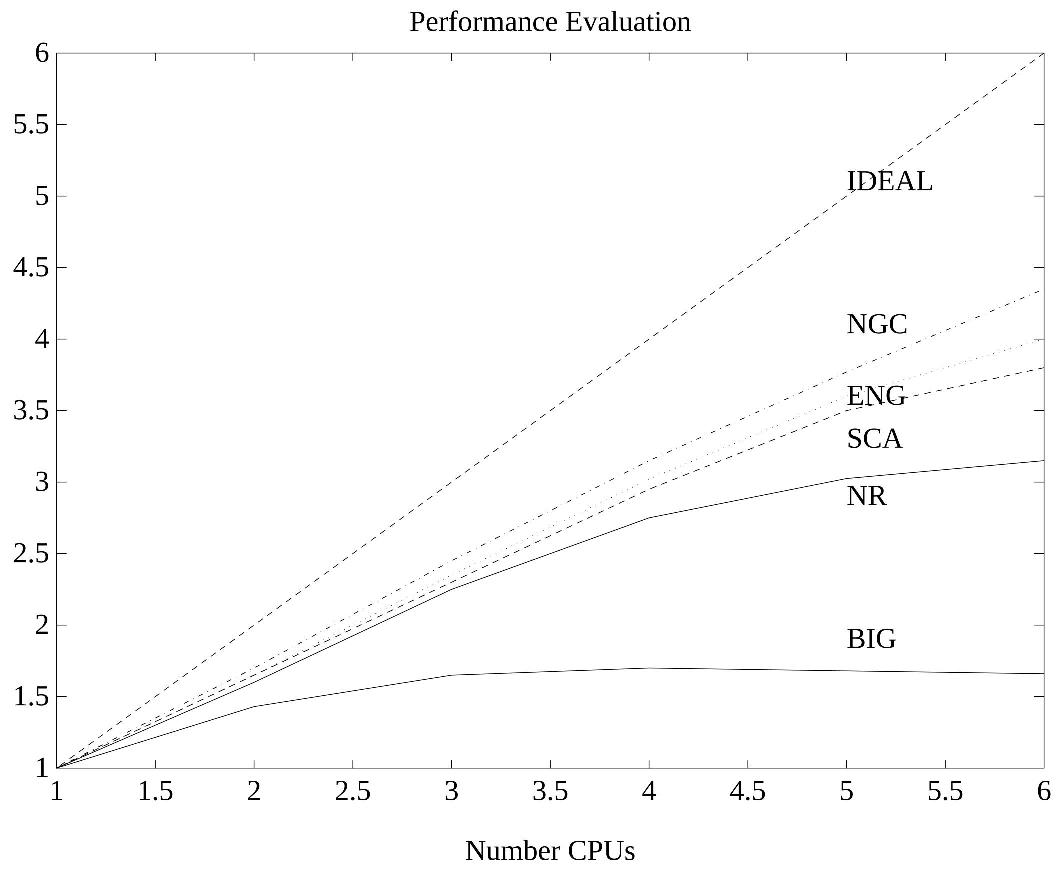


Figure 7.10: Performance speedup for Stage 2 of the PT algorithm using the modifications described in the text, for 71,680 tetrahedra.

Since decreasing the size of the triangles significantly improved performance, this suggested that the work performed by the saturated graphics subsystem was pixel intensive. This indicates that the bottleneck was either in the Scan Conversion Subsystem or the Raster Subsystem.

The algorithm used by the Raster Subsystem to fill the pixels is 8 times slower when doing opacity blending then when not doing it. If the Raster Subsystem was saturated, then turning off hardware opacity blending should give an increase in performance. When hardware blending was turned off the performance was unchanged from normal rendering (NR). This indicated that the Raster Subsystem was not the bottleneck. By elimination, the Scan Conversion Subsystem was identified as the bottleneck.

Normally, the bottleneck to high performance rendering on a SGIPS is the CPU speed. That is, the CPU's ability to put data onto the MPlink bus fast enough to feed the graphics pipe. The reason this did not apply here is that for high performance rendering large tmeshes are expected. The splatting method has only approximately four triangles per tmesh.

7.6 Parallelization of the MPVO Algorithm for Nonconvex Meshes

The next section describes the parallelization of the MPVO algorithm for nonconvex meshes. Then in Section 7.6.2 the results from the use of this algorithm are reported.

The MPVO algorithm for nonconvex meshes is described in Chapter 4, Section 4.4.2 and implementation details are given in Section 4.5.2. We will now refer to phase III of the MPVO algorithm for nonconvex meshes as the phase III algorithm. The splatting algorithm together with the phase III algorithm will be referred to as Stage 2.

7.6.1 Algorithm Parallelization

Phase II of the MPVO algorithm for nonconvex meshes accounts for a minority of the total time and it involves a sort; therefore, it was not parallelized.

The phase III algorithm requires a DFS. A BFS can not be used. Therefore, the phase III algorithm can not be significantly parallelized. However, Stage 2 as a whole can be parallelized.

A DFS does not output cells in layers that can be rendered concurrently as is the case when a BFS is used. Therefore, each cell must be rendered in the same order as it is output by the DFS. To deal with this, two queues are used, Queue1 and Queue2. One CPU, call it CPU0, is dedicated to performing the DFS; it places its output into Queue1. The remaining CPUs splat and render the cells in Queue2. After CPU0 has enqueued a predetermined number STEPSIZE of cells on Queue1, and the remaining CPUs have rendered all the cells in Queue2, then the pointers to the two queues are swapped. We refer to this as one cycle of the algorithm.

Initially, Queue2 is empty, so on the first cycle only CPU0 is busy, the remaining CPUs are idle. On the last cycle, CPU0 is idle while the remaining CPUs splat the cells in Queue2.

The DFS algorithm is normally implemented as a recursive routine. However, to allow CPU0 to synchronize with the remaining CPUs when it is time to swap the queues, the DFS algorithm is implemented without recursion using a stack. This algorithm is shown below for tetrahedra; however, it can be generalized to deal with any cells. As before, it is assumed that the process ids (pids) are zero-based; CPU0 has pid = 0, etc. The following variables are initialized to 0: *allDone, Q1idx, Q2end, doneDFS, lastCycleSplatted.* The boolean variable *notVisited* is initially set to true for each cell. The variable *allDone* should be declared as volatile.

STAGE 2 PARALLEL ALGORITHM FOR NONCONVEX TETRAHEDRAL MESHES

```
if pid == 0
    for each *cell* on sink cell list
        if cell.notVisited
            cell.notVisited = false; cell.cycleTestFlag = true
            face = 0
            push(cell,stack); push(face,stack)
            while not empty(stack)
                face = pop(stack); cell = top(stack)
                while face $\leq$ 3 /* for tetrahedra */
                    if inbound(cell.face.arrow)
                        adjCell = cell.face.sharedByCell
                        if adjCell.notVisited
                            adjCell.notVisited = false; adjCell.cycleTestFlag = true
                            push(face+1,stack); push(adjCell,stack)
                            cell = adjCell; face = 0
                        else
                            face++
                            if adjCell.cycleTestFlag then output Cycle warning
                    else face++
                end while face $\leq$ 3
                cell = pop(stack)
                if cell not marked imaginary
```

```
                    if Q1idx < STEPSIZE then Q1[Q1idx++] = cell
                    else goto BAR
L1:                 Q1[Q1idx++] = cell
                cell.cycleTestFlag = false
            end while not empty(stack)
    end for loop
    doneDFS = true
else /* remaining CPUs */
L2: for $(i = (pid - 1); i < Q2end; i += numbProcs - 1)$ splat(Q2[i])
end else
BAR: barrier()
if pid == 0
    reinitialize waitLock array
    if lastCycleSplatted then allDone = true else swap Q1 and Q2
barrier()
if not allDone
    if pid == 0
        if not doneDFS then goto L1
        else if not lastCycleSplatted
            lastCycleSplatted = true
            goto BAR /* to wait for last splatting cycle */
    else goto L2
```

To ensure the cells are rendered in the same order as they are output by the DFS, an ordering algorithm is needed to control access to the graphics pipe. This algorithm is shown below. The boolean array *waitLock[NUMBPROCS]* is initialized to all true except for *waitLock[1]* which is set to false. This allows CPU1 (pid 1) to get a lock.

It is important that the array *waitLock* be declared as volatile, otherwise an optimizing compiler might translate the first line of the following algorithm, while ($waitLock[pid]$) busy-wait, as: if ($!waitlock[pid]$) for (;;) busy-wait.

LOCK ORDERING ALGORITHM:

```
while (waitLock[pid]) busy-wait
lock(gpipe)
    make graphics calls
    waitLock[pid] = true
    if pid == (numbProcs - 1) then waitLock[1] = false
    else waitLock[pid + 1] = false
unlock(gpipe)
```

7.6.2 Results of Parallelization of Nonconvex Algorithm

The ONERA M6 wing simulation described in Chapter 6 is defined over a nonconvex mesh generated by a conformed Delaunay triangulation. Using the technique described in Chapter 4

Number Processors	*time for PT + MPVO*	*time for PT + MPVO Nonconvex*
Serial	32.66	36.01
6 CPUs	11.96	16.28

Table 7.9: Summary of timing results in seconds for volume rendering a nonconvex data set of 362,712 tetrahedra. The use of the MPVO algorithm for nonconvex meshes for the nonconvex data set is compared with the use of the regular MPVO algorithm for the comparable convex data set which has 373,654 tetrahedra. The PT splatting algorithm was used in each case. The STEPSIZE was 1000 for the parallel algorithm.

Section 4.11, the mesh was generated over a convex domain even though the domain of interest is nonconvex. The vertices and cells lying outside the nonconvex region were retained for visualization and the remaining vertices and cells were sent to the finite element solver [2].

This enables us to render the ONERA data either as a convex mesh, where the cells lying outside the nonconvex domain of interest are marked invisible, or to render it using the MPVO algorithm for nonconvex meshes. Thus we have the opportunity to compare the timing of the overall volume rendering algorithm when the MPVO algorithm for nonconvex meshes is used with the timing when the regular MPVO algorithm is used. This comparison can be made both with and without parallelization.

Timings for this experiment, both serial and parallel, using the PT algorithm for the splatting, are given in Table 7.9. The nonconvex ONERA mesh has 362,712 cells. The convex ONERA mesh has an additional 10,942 cells, the cells located inside the wing.

Typical times for volume rendering using the MPVO algorithm for nonconvex meshes and the PT Splatting algorithm, both serial and parallel, for a number of different meshes, are shown in Table 7.10. All parallel results are from the nonconvex algorithm described in Section 7.6.1.

	13,499 cells	*187,395 cells*	*287,962 cells*	*513,375 cells*
Serial	1.15	14.37	24.74	38.44
6 CPUs	0.72	7.81	13.30	20.67

Table 7.10: Typical times in seconds for volume rendering using the MPVO algorithm for nonconvex meshes and the PT Splatting algorithm. The number of exterior cells varied between 4% and 8% of the total number of cells. The STEPSIZE was 1000 for the parallel algorithm.

CHAPTER 8

FILTERING METHODS

8.1 Introduction

Even with parallelization, fast graphics hardware and the use of rendering approximations, for very large data sets, it still may not be possible for a DPVR algorithm to generate images interactively. For example the hardware rendering system may have an upper bound on the number of polygons per second that it can handle.

Therefore, to achieve interactivity, it may be necessary to reduce the number of cells rendered by techniques such as: filtering (rendering only selected cells), by coalescing adjacent cells, or by retriangulating a subset of the vertices of the mesh. I refer to this concept as *reduced resolution meshes*. How best to create such meshes is an open question.

Filtering generally requires visibility ordering the entire mesh and may result in holes or gaps in the image. The coalescing method can result in nonconvex cells not amenable to ordering by the MPVO Algorithm.

The most promising approach seems to be to retriangulate a random subset of the original vertices of the mesh using a (conformed) Delaunay triangulation [64, 105]. Therefore, this

	No Filtering	*Max Indep Set*	*20% Random*	*50% Random*
Serial	79.6 sec.	35.1 sec.	33.4 sec.	55.9 sec.
6 CPUs	30.2 sec.	13.9 sec.	13.0 sec.	21.6 sec.

Table 8.1: Comparative timings for the overall rendering process using three different filtration methods and the PT Algorithm for 1,003,520 tetrahedra.

will decrease the time for visibility ordering since fewer cells need to be ordered; the MPVO Algorithm can be used; and, there will be no holes in the image.

Since the size of data sets continues to grow with the increase in hardware computing power, the concept of reduced resolution meshes may remain valuable for some time to come.

8.2 Filtering Techniques

One way to render only selected cells is by rendering the maximal independent set[1] of the adjacency graph of the cells of the mesh. This can be done as follows. For each cell c in the mesh, render c only if none of c's neighbors have been rendered.

A disadvantage of this method is that the degree of the nodes imposes bounds on the level of filtration allowed. Typically, maximal independent set filtration results in approximately 20% of the cells being rendered.

A volume rendered image using maximal independent set filtering and the modified PT algorithm is shown at the top of Figure 8.1. The unfiltered image is shown in Figure 6.7. The regular gaps in the images shown are probably due to the order in which the cells were processed. It may be possible to avoid these gaps if the indices of the cells in the mesh are randomly permuted.

[1]An *independent set* is a set of vertices in a graph no two of which are adjacent. A *maximal independent set* is an independent set which will no longer be one when any vertex is added to the set.

Figure 8.1: Top image uses maximal independent set filtration. Bottom image uses 20% random filtration. The unfiltered image is shown in Figure 6.7.

Figure 8.2: Volume rendered image with 50% random filtration. The unfiltered image is shown in Figure 6.7.

More flexibility in the level of filtration can be achieved by selecting cells at random in the mesh. In general random selection methods have a nice distribution property. Volume rendered images using 20% and 50% random filtering are shown at the bottom of Figure 8.1 and in Figure 8.2, respectively. A comparison of all three methods is shown in Figure 8.3.

Comparative timings for these filtering methods are given in Table 8.1 for a 1,003,520 cell mesh.

Another technique is to discard every n-th cell that is output from the visibility ordering, possibly tagging adjacent cells to prevent their removal.

Figure 8.3: A comparison of the three filtering methods.

For any of these methods, it may be valuable to flag cells which have interesting or rare data values at their vertices so these cells are not filtered out. Such a determination can be made based on domain specific knowledge and/or on a histogram analysis of the data.

Low-reject, high-reject or band-reject filtering can be used to eliminate uninteresting data. Filtered cells can be flagged 'do-not-render' in a preprocessing step. Similar filtering could be achieved by the density transfer function; however, such filtration would then occur at run time.

CHAPTER 9

SPATIAL POINT LOCATION

9.1 Introduction

Given a mesh and an arbitrary point in $\mathbf{E}^3$ called the *query point*, the question can arise: In which cell of the mesh, if any, does the query point lie? This is known as the *spatial point location problem*. It can arise in volume visualization when doing interactive probing, streamline generation, or in ray tracing. When the mesh is rectilinear, the point location problem is trivial. However, when the mesh is unstructured, the problem is not straightforward.

9.2 The Meshed Polyhedra Point Location Algorithm

The Meshed Polyhedra Point Location (MPPL) Algorithm uses the MPVO Algorithm data structures and the $<_{vp}$ relation to solve the point location problem for any convex mesh. The mesh need not be acyclic. Informally, the MPPL Algorithm works like this. Set the viewpoint to the query point; and then start from any cell of the mesh and follow any 'outbound' arrow to the next cell. Keep doing this until a cell with no outbound arrows is reached; this is the *target cell*, the cell which contains the query point. If an exterior face with an outbound arrow is reached, then the point doesn't lie within the mesh. The direction of the arrows is computed on the fly and not for all cells in advance as is done in phase II of the MPVO Algorithm.

The MPPL Algorithm uses the MPVO adjacency graph described in Sections 4.5.1 and 4.5.1.1. The viewpoint is set to the query point, and then function *search* is called with any cell in the mesh. To maximize efficiency, the initial cell can be selected heuristically, to be either a centrally located cell, or, a cell in the neighborhood of the target cell, if this is known. The search path of the MPPL Algorithm is a zig-zag approximation to a ray from the start cell to the query point; no backtracking is done. It is difficult to theoretically calculate expected case bounds for this algorithm since they depend on the structure of the mesh; the time complexity can be determined empirically.

```
MPPL ALGORITHM:
    search( cell ):
        L1:for each face $f$ of cell
                calculate arrow for $f$;
                if arrow is outbound then
                    if $f$ is an interior face then
                        cell = cell which shares $f$; goto L1;
                    else
                        exit search and return -1; /* query point outside mesh */
            end for
            return cell;
```

9.3 The Meshed Polyhedra Point Location Algorithm for Nonconvex Meshes

The MPPL Algorithm can be modified to deal with nonconvex meshes; however, for efficiency it requires heuristics, and it is not efficient when the query point lies outside the mesh. Each cell is required to have a field for marking the cell 'visited' that has been initialized to the unmarked state. Between calls to the algorithm either this field needs to be reinitialized or else a unique mark needs to be used each time the algorithm is invoked. Since the cells are marked 'visited' the first time they are searched, no cell will be searched more than once. If the query point is not in any cell of the mesh, then every cell will be searched by the algorithm. Function $search1$, below, has a local variable $targetCell$; and, function $search2$ has local variables: $hasOutboundEdge$ and $targetCell$. Various heuristics and optimizations can be used; however, for sake of clarity and brevity they are not shown. Parallelization of this algorithm is worth investigating. A more efficient method would be either to convert a nonconvex mesh into a convex mesh by one of the techniques described in Section 4.9, and then use the regular MPPL algorithm, or to triangulate any nonconvex voids and then use one of the algorithms mentioned in the next paragraph.

```
MPPL ALGORITHM FOR NONCONVEX MESHES:
    $search1()$ :
        for each $cell$ in mesh
            if $cell$ not marked 'visited' then
                $targetCell = search2( cell );$
                if $targetCell$ not -1 then exit $search1$ and return $targetCell$;
    end for
    return -1; /* query point is outside mesh */


search2( cell ):
    hasOutboundEdge = FALSE;
    mark cell 'visited';
    for each face f of cell
        calculate arrow for f;
        if arrow is outbound then
            hasOutboundEdge = TRUE;
            if f is an interior face then
                if cell which shares f not marked 'visited' then
                    targetCell = search2( cell which shares f );
                    if targetCell not -1 then
                        exit search2 and return targetCell;
    end for
    if hasOutboundEdge = TRUE then exit search2 and return -1;
    else exit search2 and return cell;
```

9.4 Conclusion

Efficient spatial point location algorithms have been published by Chazelle [12] and by Preparata and Tamassia [79]; however, no known implementations of these algorithms exist. The former algorithm requires the mesh be acyclic and have no nonconvex voids; it requires $O(f)$ storage and has a query time of $O(\log^2 f)$, where f is the number of faces; no preprocessing bounds are given. The latter algorithm requires that the mesh have no nonconvex voids and requires $O(f \log^2 f)$ preprocessing time and storage, and has a query time of $O(\log^2 f)$ worst case; Goodrich and Tamassia [42] improve the storage and preprocessing bound to $O(f \log f)$.

The advantages of the MPPL Algorithms are that they are easy to implement and that they utilize the data structures of the MPVO Algorithm which may have already been built. Further, it appears that the MPVO Algorithm for Nonconvex Meshes lends itself to parallelization. Both of the MPPL algorithms described in this Chapter were implemented.

CHAPTER 10

DOMAIN DECOMPOSITION

10.1 Introduction

To maximize the speedup of a parallel finite element code, it is necessary to distribute the load as evenly as possible and to minimize interprocess communication. This means that a roughly equal number of elements should be assigned to each processor. It is also important that the subdomain of elements assigned to any one processor should be clustered coherently and the boundary minimized in order to to minimize interprocess communication. I refer to the process of partitioning the elements (cells) of a computational mesh into subdomains that can each be processed in parallel as the *domain decomposition problem.* A number of solutions to this problem have been proposed [6, 14, 28, 29, 30, 46, 67, 75, 90].

Sadayappan and Ercal [90] summarize a number of categories of approaches to the domain decomposition problem, including graph based approaches, mathematical programming based formulations, queuing theory based models and heuristic approaches such as scattered decomposition [67], simulated annealing [30] and a graph-based recursive bisection model [32].

It has been found that the MPVO Algorithm may be a candidate for domain decomposition of unstructured finite element meshes. While the resulting partitioning may not be optimal, it has the property of coherence.

The simulated annealing process for domain decomposition as described by Flower, Otto and Salama [30] can achieve an optimal mapping of finite elements to a set of parallel processors; although it does not guarantee determination of an optimal mapping. The simulated annealing method requires an initial decomposition of the mesh. The algorithm is then iterated starting from this initial decomposition. How close the resulting partition is to the optimal partition depends on the number of iterations of the annealing process and on the initial decomposition.

It is worth investigating whether the MPVO Algorithm could be valuable for creating the initial decomposition for simulated annealing.

10.2 Domain Decomposition Using the MPVO Algorithm

The MPVO Algorithm may be used for domain decomposition of an unstructured finite element mesh as follows. For k processors, and a mesh of n cells, each successive n/k cells output by the MPVO Algorithm can be assigned to a processor for execution of the finite element code.

Due to the inherent coherence of the MPVO ordering, provided n is sufficiently larger than k, the partitioning obtained might enable parallel architectures to be exploited efficiently.

For example, for the mesh shown in Figure 4.1, using phase III-DFS, for 2 CPUS, cells 1, 17, 11, 9, 3, 4, 5, 2, 18, 12 would be assigned to CPU1, and cells 13, 19, 15, 14, 16, 10, 7, 6, 8 to CPU2.

It will be interesting to see which phase III method, DFS or BFS, will prove more valuable and also to investigate the influence of the location of the viewpoint. Preliminary experiments indicate phase III-DFS with a viewpoint outside the mesh seems preferable.

Since efficiency and boundary anomalies do not appear to be serious concerns for domain decomposition, the MPVO Algorithm for Nonconvex Meshes can be used even in the presence of cycles.

By mapping the cells of each partition to a different color and using transparency, an image produced by direct volume rendering can be used to view the decomposition.

CHAPTER 11

CONCLUSION

In summary, we have examined several methods for achieving interactive direct projection volume rendering: a suite of approximations to the splatting process, the use of high performance graphics hardware, parallelization and a number of different filtering methods.

Just as importantly, we have developed a visibility ordering algorithm to order the cells of any connected mesh in linear time, using linear storage. By-products of this algorithm are a method for solving the spatial point location problem and a technique for domain decomposition of irregular finite element meshes for parallel computation.

By the use of parallelization and the suite of splatting approximations, the goal of interactive DPVR has been found to be feasible, even when the data sets are very large. Using the methods described herein, the DPVR splatting algorithm has interactively generated volume rendered images of nonrectilinear data sets with over 1,000,000 cells. Using the filtering methods described herein, this performance is possible for even larger data sets. For nonrectilinear data sets with up to 100,000 cells, volume rendered images were generated in less than 2 seconds.

Due to the tradeoff between the accuracy of the image and the time required to render it, it is not reasonable to expect high quality images of very large data sets in interactive time.

In the author's opinion, the images generated using the suite of splatting approximations should be very useful to the computational scientist. It will remain for the computational scientist to validate the utility of the volume rendering techniques described herein as to the accuracy and detail of information content. However, we can be quite certain that these techniques will be valuable for data previewing and for setting viewing parameters and the color and opacity maps.

The comparison of approximated images is subjective. The ultimate test is whether the scientist finds the image useful for understanding his or her data, and is not misled.

It would be useful to extend the current volume renderer to include a ray tracer. This would allow experimentation with more accurate ray integration methods based on the continuous volume density optical model described in Chapter 3. It would also provide a more accurate image for comparison with approximated images. In addition, the parallelization of the ray traced version would provide comparative timings for these two competing methods of direct volume rendering.

Sharp isosurfaces do not seem to be feasible for the splatting methods described herein. If the density transfer function simulates a set of very narrow band pass filters, then the density map will have a set of very narrow square pulses. Since the PT method averages the density at the front and back faces of a cell, the PT method could completely miss an isosurface passing through the cell.

Solids or polygons can be embedded in a volumetrically rendered image by generating a new mesh using a Delaunay triangulation that conforms to the vertices of the old mesh and the geometric description of the embedded objects.

When the MPVO algorithm is used, it outputs the cells in layers from front to back. Then the image unfolds as if a cutting plane perpendicular to the line of sight was sweeping over the image towards the viewer.

One of the most interesting discoveries in this work was the amount of information that could be gained by watching the image being rendered. The fine structure of the interior of the scalar field could be clearly discerned during the rendering process. It was known that watching this process could be useful. However, the degree of utility far surpassed my expectations.

The rendering process became an animation. It was quite fascinating to watch; sometimes it could be breath-taking. If the rendering time was significantly faster, or much slower, this effect would be lost. This way of experiencing data provides information that is not available by looking at a finished image, even with transparency.

BIBLIOGRAPHY

[1] BAKER, T.J. Three Dimensional Mesh Generation by Triangulation of Arbitrary Point Sets. *Amer. Inst. Aero. & Astro. Report AIAA-87-1124* 1987.

[2] BAKER, T.J. Shape Reconstruction and Volume Meshing for Complex Solids. *Intl. J. Numerical Methods in Engin. 32 4* (1991), 665–675.

[3] BALACHANDAR, S. Methods for Evaluating Fluid Velocities in Spectral Simulations of Turbulence. *Jrnl Computational Physics 83 1* (July, 1989).

[4] BLINN, J. F. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. *ACM SIGGRAPH Comput. Gr. 16 3* (July 1982), 21-29.

[5] BONGIOVANNI, G. and PETRESCHI, R. Parallel-Depth Search for Acyclic Digraphs. *Jrnl. Parallel and Distrib. Comp. 7* (1989), 383–390.

[6] BRAMLEY, R. and NITROSSO, B. Data Distribution in CFD Finite Element Codes. Center for Supercomputing Research and Development Report, University of Illinois, Urbana. (1992).

[7] CARPENTER, L. The A-buffer, an Antialiased Hidden Surface Method. *ACM SIGGRAPH Comput. Gr. 18 3* (1984), 103–108.

[8] CHALLINGER, J. Parallel Volume Rendering on a Shared-Memory Multiprocessor. Univ. Calif. Santa Cruz Dept. Comp. Sci. Report UCSC-CRL-91-23, rev March 1992.

[9] CHALLINGER, J. Parallel Volume Rendering for Curvilinear Volumes. *IEEE Comp. Soc., Proc. Scalable High Performance Computing Conference* (April 1992).

[10] CHANDRASEKHAR, S. *Radiative Transfer.* Dover, New York, 1960.

[11] CHAZELLE, B. Convex Partitions of Polyhedra: A Lower Bound and Worst Case Optimal Algorithm. *SIAM J. Comput. 13* (1984), 488–507.

[12] CHAZELLE, B. How to Search in History. *Inform. Control 64* (1985), 77–99.

[13] CHAZELLE, B. and PALIOS, L. Triangulating a Nonconvex Polytope. *Proc. 5th Ann. Sympos. Comput. Geom.* (June 1989), 393–399.

[14] CHRISOCHOIDES, N., HOUSTIS, C., HOUSTIS, E., PAPACHIOU, P., KORTESIS, S. and RICE, J. Domain Decomposer; A Software Tool for Mapping PDE Computations to Parallel Architectures. Report CSD-TR-1025 Dept. Comput. Sci. Purdue University (Sept, 1990).

[15] CLINE, H. E., LORENSEN, W. E., LUDKE, S., CRAWFORD, C. R. and TEETER, B. C. Two Algorithms for the Reconstruction of Surfaces from Tomograms. *Medical Physics, 15 3* (June 1988), 320–327.

[16] COOK, S. A. Towards a Complexity Theory of Synchronous Parallel Computation. *Enseign. Math. 27* (1981), 99–124.

[17] DREBIN, R. A., CARPENTER, L. and HANRAHAN, P. Volume Rendering. *ACM SIGGRAPH Comput. Gr. 22 4* (Aug. 1988), 65–74.

[18] DAY, A. M. The Implementation of an Algorithm to Find the Convex Hull of a Set of 3D Points. *ACM Trans. Gr. 9 1* (Jan. 1990), 105–132.

[19] DELAUNAY, B. Sur la sphère vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk 7* (1934), 793–800.

[20] EBERT, D. S. and PARENT, R. E. Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques. *ACM SIGGRAPH Comput. Gr. 24 4* (July 1990), 357–363.

[21] EDELSBRUNNER, H. and MÜCKE, E. P. Three-dimensional Alpha Shapes. Report No. UIUCDCS-R-92-1734 Dept. Comp. Sci., Univ. of Illinois, Urbana-Champaign, March 1992.

[22] EDELSBRUNNER, H. and TAN, T.-S. An Upper Bound for Conforming Delaunay Triangulations. To appear *Proc. 8th Ann. Sympos. Comput. Geom.* (1992).

[23] EDELSBRUNNER, H. and SHAH, N. R. Incremental Topological Flipping Works for Regular Triangulations. To appear *Proc. 8th Ann. Sympos. Comput. Geom.* (1992).

[24] EDELSBRUNNER, H. and MÜCKE, E. P. Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Trans. Gr. 9 1* (Jan. 1990), 66–104.

[25] EDELSBRUNNER, H., PREPARATA, F. P. and WEST, D. B. Tetrahedrizing Point Sets in Three Dimensions. *J. Symb. Comput. 10* (1990), 335–347.

[26] EDELSBRUNNER H. An Acyclicity Theorem for Cell Complexes in d Dimension. *Combinatorica 10 (3)* (1990), 251–260.

[27] EDELSBRUNNER, H. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Heidelberg, 1987.

[28] FARHAT, C. A Simple and Efficient Automatic FEM Domain Decomposer. *Comput. and Structures 28* (1988), 579–602.

[29] FARHAT, C. On Mapping of Massively Parallel Processors onto Finite Element Graphs. *Comput. and Structures 32* (1989), 347–353.

[30] FLOWER, J., OTTO,S. and SALAMA, M. Optimal Mapping of Irregular Finite Element Domains to Parallel Processors. in Parallel Computations and Their Impact on Machines, A. Noor, ed., Am. Soc. Mech. Engin., New York (1987), 239–250.

[31] FOLEY, J. D., van DAM, A., FEINER, S. K. and HUGHES, J. F. *Computer Graphics Principles and Practice.* Addison-Wesley, Reading, 1990.

[32] FOX, G. Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube. Report 327, Caltech Concurrent Cube Project, July 1985.

[33] FRIEDER, G., GORDON, D., and REYNOLDS, R. A. Back-to-Front Display of Voxel-Based Objects. *IEEE Comput. Graph. Appl. 5 1* (Jan. 1985), 52–60.

[34] FUCHS, H., KEDEM, Z. and NAYLOR, B. On Visible Surface Generation by A Priori Tree Structures. *ACM SIGGRAPH Comput. Gr. 14 3* (July 1980), 124–133.

[35] GALLAGHER, R. S. and NAGTEGAAL, J. C. An Efficient 3D Visualization Technique for Finite Element Models and Other Course Volumes. *ACM SIGGRAPH Comput. Gr. 23 3* (July 1989), 185–192.

[36] GARRITY, M. P. Raytracing Irregular Volume Data. *San Diego Workshop on Volume Visualization, Comput. Gr. 24 5* (Dec 1990), 35–40.

[37] GAZIT, H. and MILLER, G. L. An Improved Parallel Algorithm that Computes the BFS Numbering of a Directed Graph. *Inf. proc. Letters 28* (1988), 61–65.

[38] GIERTSEN, C. Volume Visualization of Sparse Irregular Meshes. *IEEE Comp. Gr. 12 2* (March 1992), 40–48.

[39] GIERTSEN, C. Creative Parameter Selection for Volume Visualization. To appear in *Jrn. Visualization and Comput. Anim.* (1992).

[40] GIERTSEN, C. and TUCHMAN, A. Fast Volume Rendering with Embedded Geometric Primitives. Manuscript, IBM Bergen Scientific Centre, Norway, 1992.

[41] GOAD, C. Special Purpose Automatic Programming for Hidden Surface Elimination. *ACM SIGGRAPH Comput. Gr. 16 3* (July 1982), 167–178.

[42] GOODRICH, M.T. and TAMASSIA, R. Dynamic Trees and Dynamic Point Location. *Proc. 23rd ACM Symp. on Theory of Computing* (1991), 523–533.

[43] GUIBAS, L. J. and STOLFI, J. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Trans. Gr. 4 2* (April 1985), 74–123.

[44] HABER, R. Visualization in Engineering Mechanics. *SIGGRAPH '88 Tutorial.*

[45] HABER, R. B., LUCAS, B. and COLLINS, N. A Data Model for Scientific Visualization with Provision for Regular and Irregular Grids. *Proc. Visualization '91* (Oct. 1991), 298–305.

[46] HAMMOND, S. and SCHREIBER, R. Mapping Unstructured Grid Problems to the Connection Machine. RIACS Report 90.22 NASA Ames Research Center, Oct. 1990.

[47] HIBBARD W. and SANTEK D. Interactivity is the Key. *Proceedings Chapel Hill Workshop of Volume Visualization* (May 1989), 39–43.

[48] HUNG, C. and BUNING, P., Simulation of Blunt-Fin Induced Shock Wave and Turbulent Boundary Layer Separation, *AIAA Paper 84-0457, AIAA Aerospace Sciences Conference, Reno NV* (Jan. 1984).

[49] JOE, B. Construction of Three-Dimensional Delaunay Triangulations Using Local Transformations. *Computer Aided Geometric Design 8* (1991), 123–142.

[50] KAJIYA, J. T. and VON HERZEN, B. P. Ray Tracing Volume Densities. *ACM SIGGRAPH Comput. Gr. 18 4* (July 1984), 165-174.

[51] KARP, R. M. and RAMACHANDRAN, V. Parallel Algorithms for Shared Memory Machines. In *Handbook of Theoretical Computer Science.* MIT Press, Cambridge, MA, 1990.

[52] KENNON, S. R. A Vectorized Delaunay Triangulation Scheme for Non-Convex Domains With Automatic Nodal Point Generation. *Amer. Inst. Aero. & Astro. Report AIAA-88-0314* (1988).

[53] KRUEGER, W. The Application of Transport Theory to Visualization of 3D Scalar Data Fields. *Proc. Visualization '90* (Oct. 1990), 273–280.

[54] KOYAMADA, K. Volume Visualization for the Unstructured Grid Data. *SPIE/SPSE Extracting Meaning from Complex Data Conf. Proc. Vol. 1259* (1990), 14–25.

[55] KOYAMADA, K. and MIYAZAWA, T. Volume Rendering of Unstructured Grid Data. Manuscript, Tokyo Scientific Center, IBM Japan Ltd (1991).

[56] LASZLO, M. J. Techniques for Visualizing 3-Dimensional Manifolds. *Proc. Visualization '90* (Oct. 1990), 342–352.

[57] LAUR, D. and HANRAHAN, P. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *ACM SIGGRAPH Comput. Gr. 25 4* (July 1991), 285–288.

[58] LEE, C. Regular Triangulations of Convex Polytopes. *Applied Geometry and Discrete Mathematics: The Victor Klee Festschrift* P. Gritzmann and B. Sturmfels, eds. Amer Math. Soc., Providence, RI, 1991.

[59] LEVOY, M. Display of Surfaces from Volume Data. *IEEE Comput. Graph. Appl. 8 3* (May 1988) 29–37.

[60] LEVOY, M. Efficient Ray Tracing of Volume Data. *ACM Trans on Graphics 9 3,* (July 1990), 245–261.

[61] MAX, N., HANRAHAN, P. and CRAWFIS, R. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *San Diego Workshop on Volume Visualization, Comput. Gr. 24 5* (Dec 1990), 27–33.

[62] MAX, N. Atmospheric Illumination and Shadows. *ACM SIGGRAPH Comput. Gr. 20 4* (Aug. 1986), 117–124.

[63] MAX, N. Light Diffusion through Clouds and Haze. *Comput Vision, Graph. and Image Proc. 33* (march 1986), 280–292.

[64] MESHKAT, S., RUPPERT, J. and LI, H. Three-Dimensional Unstructured Grid Generation Based On Delaunay Tetrahedrization. *Proc. Third Intl. Conf. Numerical Grid Generation* (June 1991), 841–851.

[65] MESHKAT, S. CDSmesh 3D Automatic Mesh Generator User's Guide and Reference. IBM Almaden Research Center, Oct. 1991.

[66] MEYERS R. J. and STEPHENSON, M. B. Ray Traced Scalar Fields with Shaded Polygonal Output. *Proc. Visualization '90* (Oct 1990), 263–272.

[67] MORISON R. and OTTO, S. The Scattered Decomposition for Finite Elements. *J. Sci. Comput. 2 1* (1987), 59–76.

[68] MÜCKE, E. P. PhD thesis in preparation. Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[69] NACKMAN, L. R. and SRINIVASAN, V. Point Placement for Delaunay Triangulation of Polygonal Domains. *Proc. Third Canadian Conf. Comput. Geom.* (Aug. 1991), 37–40.

[70] NOVINS, K., SILLION, F. and GREENBERG, D. An Efficient Method for Volume Rendering using Perspective Projection. *San Diego Workshop on Volume Visualization, Comput. Gr. 24 5* (Dec 1990), 95–100.

[71] OSTERHAUG, A. *Guide to Parallel Programming.* Sequent Computer Systems, Beaverton, OR, 1987.

[72] PATERSON, M. S. and YAO, F. F. Binary Partitions with Applications to Hidden-Surface Removal and Solid Modeling. *Proc. 5th Ann. Sympos. Comput. Geom.* (June 1989), 23–32.

[73] PERRONNET, A. Some Present Fully Automatic Mesh Generators. *Intl. Symp. on Numerical Methods in Engineering* (1989), 129–136.

[74] PIPPENGER, N. On Simultaneous Resource Bounds. *Proc. 20th Ann. IEEE Symp. on Foundations of Comput. Sci.* (1979) 307–311.

[75] POMMERELL, C., ANNARATONE, M. and FICHTNER, W. A Set of New Mapping and Coloring Heuristics for Distributed-Memory Parallel Processors. *SIAM J. Sci. Stat. Comp* (Jan 1992).

[76] PORTER, D. Perspective Volume Rendering. Univ. Minn. Supercomp. Inst. Research Report UMSI 91/149 (May 1991).

[77] PORTER, T. and DUFF, T. Compositing Digital Images. *ACM SIGGRAPH Comput. Gr. 18 3* (July 1984), 253–259.

[78] PREPARATA, F. P. and HONG, S. J. Convex Hulls of Finite Sets of Points in Two and Three Dimensions. *Comm. ACM 20* (1977), 87–93.

[79] PREPARATA, F. P. and TAMASSIA, R. Efficient Point Location in a Convex Spatial Cell Complex. To appear *SIAM J. Computing 21 2* (1992).

[80] QUINN, M. J. and DEO, N. Parallel Graph Algorithms. *ACM Comput. Surveys 16 3* (Sept. 1984), 319–348.

[81] REEVES, W. Particle Systems - a Technique for Modeling a Class of Fuzzy Objects. *ACM SIGGRAPH Comput. Gr. 17 3* (July 1983), 359–373.

[82] REGHBATI, E. and CORNEIL, D. G. Parallel Computations in Graph Theory. *SIAM J. Comput. 7 2* (1978), 230–237.

[83] REIF, J. H. Depth-first Search is Inherently Sequential. *Inform. Process. Lett. 20* (1985), 229–234.

[84] ROGERS, S., KWAK, D. and KAUL, U., A Numerical Study of Three-Dimensional Incompressible Flow Around Multiple Posts, *AIAA Paper 86-0353, Reno, Nevada* (1986).

[85] ROSENBERGER, H. Degeneracy Control in Geometric Programs. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[86] RUDER, H., ERTL, F., GEYER, F., HEROLD, H. and KRAUS, U. Line-of-Sight Integration: A Powerful Tool for Visualization of Three-Dimensional Scalar Fields. *Comput. & Graphics 13 2* (1989), 223–228.

[87] RUSHMEIER, H. E. and TORRANCE, K. E. The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium. *ACM SIGGRAPH Comput. Gr. 21 4* (July 1987), 293–302.

[88] RUSSELL, G. and MILES, R. Display and Perception of 3-D Space-filling Data. *Appl. Optics 26 6* (March 1989), 973–982.

[89] SABELLA, P. A rendering algorithm for visualizing 3D scalar fields. *ACM SIGGRAPH Comput. Gr. 22 4* (July 1988), 51–58.

[90] SADAYAPPAN, P. and ERCAL, F. Nearest Neighbor Mapping of Finite Element Graphs onto Processor Meshes. *IEEE Trans. Comput. C-36 12* (Dec. 1987), 1408–1424.

[91] Silicon Graphics Inc. 4D1-3.3 Development Release and Installation Notes. 1990.

[92] SCHÖNHARDT, E. Über die Zerlegung von Dreieckspolyedern in Tetraeder. *Math. Ann. 98* (1928), 309–312.

[93] SHIRLEY, P. and TUCHMAN, A. A Polygonal Approximation to Direct Scalar Volume Rendering. *San Diego Workshop on Volume Visualization, Comput. Gr. 24 5* (Dec 1990), 63–70.

[94] SPERAY, D. and KENNON, S. Volume Probes: Interactive Data Exploration on Arbitrary Grids. *San Diego Workshop on Volume Visualization, Comput. Gr. 24 5* (Dec 1990), 5–12.

[95] TARJAN, R. E. and VISHKIN, U. An Efficient Parallel Biconnectivity Algorithm. *SIAM J. Comput. 14* (1985), 862–874.

[96] TUCHMAN, A. M. Volume Rendering and Visualization for Scientific Data. M.S. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[97] UPSON, C. and KEELER, M. V-BUFFER: Visible Volume Rendering. *ACM SIGGRAPH Comput. Gr. 22 4* (July 1988), 59–64.

[98] UPSON, C., FAULHABER Jr., T., KAMINS, D., LAIDLAW, D., SCHLEGEL, D., VROOM, J., GURWITZ, R. and VAN DAM, A. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Comput. Graph. Appl. 9, 4* (July 1989), 30-42.

[99] van LEEUWEN, J. Graph Algorithms. In *Handbook of Theoretical Computer Science.* MIT Press, Cambridge, MA, 1990.

[100] WESTOVER, L. Interactive Volume Rendering. *Proceedings Chapel Hill Workshop of Volume Visualization* (May 1989), 9–16.

[101] WESTOVER, L. Footprint Evaluation for Volume Rendering. *ACM SIGGRAPH Comput. Gr. 24 4* (Aug. 1990), 367–376.

[102] WILHELMS, J., CHALLINGER J., ALPER N., RAMAMOORTHY S. and VAZIRI A. Direct Volume Rendering of Curvilinear Volumes. *San Diego Workshop on Volume Visualization, Comput. Gr. 24 5* (Dec 1990), 41–47.

[103] WILHELMS, J. and VAN GELDER, A. A Coherent Projection Approach for Direct Volume Rendering. *ACM SIGGRAPH Comput. Gr. 25 4* (July 1991), 275–284.

[104] WILLIAMS, P. L. Issues in Interactive Direct Projection Volume Rendering of Nonrectilinear Meshed Data Sets. Work in Progress Report, San Diego Workshop on Volume Visualization Dec. 1990, available as Report 1059, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Dec. 1990.

[105] WILLIAMS, P. L. Applications of Computational Geometry to Volume Visualization. *Proc. 3rd Canadian Conf. Comput. Geom.* (Aug. 1991), 247–251.

[106] WILLIAMS, P. L. Visibility Ordering Meshed Polyhedra. *ACM Trans. on Graphics 11 2* (April 1992).

VITA

Peter Williams was born in London, England. He graduated with a Bachelor of Science degree in Engineering-Physics from the University of California, Berkeley in June, 1960. He received a Master of Science degree in Computer Science from the University of Lowell in 1984.