# FAST VOLUME RENDERING USING A SHEAR-WARP FACTORIZATION OF THE VIEWING TRANSFORMATION

**Philippe G. Lacroute**

**Technical Report: CSL-TR-95-678**

**September 1995**

This research has been supported by ARPA/ONR under contracts N00039-91-C-0138 and 175-6212-1, NSF under contract CCR-9157767, and grants from Software Publishing Corp., SoftImage, and the sponsoring companies of the Stanford Center for Integrated Systems.

# FAST VOLUME RENDERING USING A SHEAR-WARP FACTORIZATION OF THE VIEWING TRANSFORMATION

**Philippe G. Lacroute**

**Technical Report: CSL-TR-95-678**

September 1995

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055
e-mail: pubs@shasta.stanford.edu

## Abstract

Volume rendering is a technique for visualizing 3D arrays of sampled data. It has applications in areas such as medical imaging and scientific visualization, but its use has been limited by its high computational expense. Early implementations of volume rendering used brute-force techniques that require on the order of 100 seconds to render typical data sets on a workstation. Algorithms with optimizations that exploit coherence in the data have reduced rendering times to the range of ten seconds but are still not fast enough for interactive visualization applications. In this thesis we present a family of volume rendering algorithms that reduces rendering times to one second.

First we present a scanline-order volume rendering algorithm that exploits coherence in both the volume data and the image. We show that scanline-order algorithms are fundamentally more efficient than commonly-used ray casting algorithms because the latter must perform analytic geometry calculations (e.g. intersecting rays with axis-aligned boxes). The new scanline-order algorithm simply streams through the volume and the image in storage order. We describe variants of the algorithm for both parallel and perspective projections and a multiprocessor implementation that achieves frame rates of over 10 Hz.

Second we present a solution to a limitation of existing volume rendering algorithms that use coherence accelerations: they require an expensive preprocessing step every time the volume is classified (i.e. when opacities are assigned to the samples), thereby limiting the usefulness of the algorithms for interactive applications. We introduce a data structure for encoding spatial coherence in unclassified volumes. When combined with our rendering algorithm this data structure allows us to build a fully-interactive volume visualization system.

**Key Words and Phrases:** Volume Rendering, Scientific Visualization, Medical Imaging, Software Algorithms, Parallel Algorithms.

Copyright © 1995

Philippe Lacroute

# Acknowledgements

Many people have contributed to this work and provided support during my years at Stanford. First of all, I would like to express my sincerest gratitude to my advisor, Marc Levoy, who sparked my interest in computer graphics and provided encouragement at key times. Marc always stays involved with his students, whether during marathon brain storming sessions or on canoeing expeditions in the swamps of Florida. I would also like to thank Pat Hanrahan and Sandy Napel for serving on my reading committee and for their interest in my work. Their comments greatly improved this manuscript. I am also indebted to Mark Horowitz who provided guidance during my first few years at Stanford and who helped shape my interest in computer systems.

Next, I would like to thank my colleagues in the Stanford Graphics Group for providing a stimulating and fun research environment. In particular, I thank Brian Curless, Reid "Ankles" Gershbein, Craig Kolb, Matt Pharr, and especially Eric Veach for their comments on this manuscript. Maneesh Agrawala contributed to the implementation of the parallel volume renderer described in Chapter 6.

I would also like to thank the members of the DASH, FLASH and TORCH groups. In particular, J.P. Singh helped me to make sense of my parallel performance results, and Margaret Martonosi, Steve Woo, Mark Heinrich, and Dave Nakahira answered my countless questions about their cache simulators and the DASH multiprocessor.

Many thanks to Charlie Orgish, the only system administrator I know who lets the students mis-administer the machines and then is always on call to fix the problems. Thanks also to Sarah Beizer for cheerfully dealing with the University bureaucracy.

Silicon Graphics, Inc. provided time on the Challenge multiprocessor for the performance results in Chapter 6. The volume data sets used in this work were provided by Pat

Hanrahan, Sandy Napel, Siemens Medical Systems, Inc., and the North Carolina Memorial Hospital. My graduate work was supported by ARPA/ONR under contracts N00039-91-C-0138 and 175-6212-1, NSF under contract CCR-9157767, and grants from Software Publishing Corp., SoftImage, and the sponsoring companies of the Stanford Center for Integrated Systems.

Finally, I would like to thank my parents, Bernard and Ronni Lacroute, who have provided love, encouragement, and home-cooked meals whenever my fridge was empty. I dedicate this dissertation to them.

# Contents

**Acknowledgements** **iii**

**1 Introduction** **1**

1.1 Volume Rendering . . . 3

1.1.1 The Volume Rendering Equation . . . 3

1.1.2 The Volumetric Compositing Approximation . . . 8

1.1.3 Data Representation and Sampling . . . 11

1.1.4 The Visualization Process . . . 13

1.2 A New Family of Fast Volume Rendering Algorithms . . . 15

1.3 Organization . . . 16

**2 Prior Work** **18**

2.1 Volume Rendering Algorithms . . . 18

2.1.1 Ray Casting . . . 19

2.1.2 Splatting . . . 20

2.1.3 Cell Projection . . . 21

2.1.4 Multipass Resampling . . . 22

2.2 Acceleration Techniques . . . 24

2.2.1 Spatial Data Structures . . . 24

2.2.2 Early Ray Termination . . . 26

2.3 Chapter Summary . . . 28

**3 The Shear-Warp Factorization** **29**

3.1 An Overview of the Factorization . . . 30

3.2 The Affine Factorization . . . 33
3.3 The Perspective Factorization . . . 35
3.4 Properties of the Factorization . . . 39
3.5 Existing Shear-Warp Algorithms . . . 40
3.6 Chapter Summary . . . 43

**4 Three Fast Volume Rendering Algorithms 44**
4.1 Parallel Projection Rendering Algorithm . . . 44
4.1.1 Overview of the Parallel Projection Algorithm . . . 44
4.1.2 The Run-Length Encoded Volume . . . 47
4.1.3 The Run-Length Encoded Image . . . 49
4.1.4 Resampling the Volume . . . 53
4.1.5 Warping the Intermediate Image . . . 54
4.1.6 Opacity Correction . . . 54
4.1.7 Implementation of the Parallel Projection Algorithm . . . 56
4.2 Perspective Projection Rendering Algorithm . . . 60
4.2.1 Overview of the Perspective Projection Algorithm . . . 61
4.2.2 Resampling the Volume . . . 62
4.2.3 Opacity Correction . . . 62
4.2.4 Implementation of the Perspective Projection Algorithm . . . 63
4.2.5 Limitations of the Perspective Projection Algorithm . . . 66
4.3 Fast Classification Algorithm . . . 68
4.3.1 Overview of the Fast Classification Algorithm . . . 69
4.3.2 The Min-Max Octree . . . 71
4.3.3 The Summed-Area Table . . . 74
4.3.4 Implementation of the Fast Classification Algorithm . . . 76
4.3.5 Limitations of the Fast Classification Algorithm . . . 79
4.4 Switching Between Modes . . . 81
4.5 Chapter Summary . . . 83

**5 Performance Analysis 84**
5.1 Performance of the Shear-Warp Algorithms . . . 86

5.1.1 Speed and Memory Performance . . . 86
5.1.2 Image Quality . . . 95
5.2 Comparison of Coherence-Accelerated Volume Rendering Algorithms . . 98
5.2.1 Asymptotic Complexity . . . 99
5.2.2 Experimental Methodology . . . 103
5.2.3 Comparison of Speedups from Algorithmic Optimizations . . . 105
5.2.4 Costs of Coherence Accelerations . . . 110
5.2.5 Memory Overhead . . . 114
5.2.6 Analysis of the Shear-Warp Coherence Data Structures . . . 116
5.3 Low-Coherence Volumes . . . 119
5.3.1 Categories of Volume Data . . . 119
5.3.2 Voxel Throughput . . . 121
5.3.3 The Impact of Coherence on Rendering Time . . . 123
5.3.4 The Role of Coherence in Visualization . . . 126
5.4 Chapter Summary . . . 127

**6 A Multiprocessor Volume Rendering Algorithm 128**
6.1 Multiprocessor Rendering Algorithm . . . 130
6.1.1 Image and Object Partitions . . . 131
6.1.2 Task Shape . . . 132
6.1.3 Load Balancing . . . 133
6.1.4 Data Distribution . . . 134
6.1.5 Overall Algorithm . . . 134
6.2 Implementation . . . 136
6.2.1 Hardware Architectures . . . 136
6.2.2 Software Implementation . . . 137
6.3 Results . . . 138
6.3.1 Rendering Rates . . . 138
6.3.2 Performance Limits on the Challenge Multiprocessor . . . 141
6.3.3 Performance Limits on the DASH Multiprocessor . . . 144
6.3.4 Memory Performance . . . 146

6.3.5 Load Balancing . . . 148
6.3.6 Related Work . . . 149
6.4 Chapter Summary . . . 150

**7 Extensions** **152**
7.1 Flexible Shading With Lookup Tables . . . 153
7.1.1 Shade Trees and Lookup Tables . . . 153
7.1.2 Implementation of Shading Functions . . . 155
7.1.3 General Shade Trees . . . 160
7.2 Fast Depth Cueing . . . 160
7.2.1 Depth Cueing . . . 160
7.2.2 Factoring the Depth Cueing Function . . . 162
7.2.3 Implementation of Fast Depth Cueing . . . 164
7.3 Rendering Shadows with a 2D Shadow Buffer . . . 165
7.3.1 Algorithms for Rendering Shadows . . . 165
7.3.2 Implementation of the Shadow Rendering Algorithm . . . 167
7.3.3 Performance of the Shadow Rendering Algorithm . . . 169
7.4 Rendering Mixtures of Volume Data and Polygons . . . 173
7.5 Clipping Planes . . . 175
7.6 Chapter Summary . . . 176

**8 VolPack: A Volume Rendering Library** **177**
8.1 System Architecture . . . 178
8.2 Volume Representation . . . 180
8.3 Classification and Shading Functions . . . 181
8.4 Viewing Model . . . 182
8.5 Functionality Provided by VolPack . . . 182
8.6 Chapter Summary . . . 183

**9 Conclusions** **185**
9.1 Final Summary . . . 185
9.2 Future Directions for Performance Improvements . . . 187

9.3 Hardware Support for Volume Rendering . . . . . 188
9.4 Interactive Volume Rendering . . . . . 190

**A Mathematics of the Shear-Warp Factorization** **192**
A.1 Coordinate Systems and Definitions . . . . . 192
A.2 The Affine Factorization . . . . . 194
A.2.1 Finding the Principle Viewing Axis . . . . . 194
A.2.2 Transformation to Standard Object Coordinates . . . . . 196
A.2.3 The Shear and Warp Factors . . . . . 197
A.2.4 Projection to the Intermediate Image . . . . . 199
A.2.5 The Complete Affine Factorization . . . . . 201
A.3 The Perspective Factorization . . . . . 202
A.3.1 Finding the Principle Viewing Axis . . . . . 202
A.3.2 Transformation to Standard Object Coordinates . . . . . 204
A.3.3 The Shear and Warp Factors . . . . . 205
A.3.4 Projection to the Intermediate Image . . . . . 207
A.3.5 The Complete Perspective Factorization . . . . . 209

**Bibliography** **211**

# List of Tables


5.1 Characteristics of the data sets used for performance testing. . . . . 86
5.2 Rendering parameters used for performance testing. . . . . 88
5.3 Rendering time and memory usage results. . . . . 88
5.4 Timing results for the 2D warp. . . . . 89
5.5 Speedup of the shear-warp algorithm relative to a ray caster. . . . . 110
5.6 Timing breakdowns for coherence-accelerated rendering algorithms. . . . 111

6.1 Characteristics of the multiprocessors used for performance testing. . . . . 136
6.2 Rendering rates and tasks sizes on the SGI Challenge. . . . . 140
6.3 Rendering rates and task sizes on the Stanford DASH Multiprocessor. . . 140
6.4 Comparison of load balancing algorithms. . . . . 149

7.1 Performance results for the shading algorithm. . . . . 159
7.2 Performance results for fast depth cueing. . . . . 164

# List of Figures


1.1 Example of a volume rendering. . . . . 2
1.2 A physical model for volume rendering. . . . . 4
1.3 A simplified model for volume rendering. . . . . 7
1.4 Examples of sampling grids in 2D. . . . . 11
1.5 The visualization process. . . . . 14

3.1 Sheared object space for a parallel projection. . . . . 30
3.2 Sheared object space for a perspective projection. . . . . 31
3.3 The brute-force shear-warp algorithm. . . . . 32
3.4 Determining the affine shear coefficients. . . . . 34
3.5 The perspective shear transformation. . . . . 36
3.6 The perspective scale transformation. . . . . 38
3.7 Resampling a translated voxel slice. . . . . 40
3.8 Ray casting with the template optimization. . . . . 42

4.1 Run-length encoding the intermediate image. . . . . 45
4.2 Resampling and compositing run-length encoded scanlines. . . . . 46
4.3 Run-length encoding the volume. . . . . 48
4.4 A tree data structure to represent an opaque pixel run. . . . . 50
4.5 The path compression optimization. . . . . 51
4.6 Opacity correction. . . . . 55
4.7 Plot of the opacity correction function. . . . . 56
4.8 Pseudo-code for the parallel projection algorithm. . . . . 57
4.9 Sampling the volume in the perspective projection algorithm. . . . . 61

4.10 Pseudo-code for the perspective projection algorithm. . . . . 64
4.11 Splitting the volume to support a wide field of view. . . . . 66
4.12 Visiblity errors in the perspective rendering algorithm. . . . . 68
4.13 Goal of the fast classification algorithm. . . . . 69
4.14 Data structures for the fast classification algorithm. . . . . 70
4.15 Diagram of a min-max octree. . . . . 72
4.16 Diagram of a 2D summed-area table. . . . . 74
4.17 Pseudo-code for the fast classification algorithm. . . . . 77
4.18 A family of volume rendering algorithms. . . . . 82

5.1 Volume rendering of an MR scan of a brain. . . . . 87
5.2 Volume rendering of a CT scan of a head. . . . . 87
5.3 Timing results for a rotation sequence. . . . . 90
5.4 Execution time for each stage of rendering. . . . . 92
5.5 Color volume rendering of a CT scan of a head. . . . . 93
5.6 Color volume rendering of a CT scan of an engine block. . . . . 93
5.7 Color volume rendering of a CT scan of a human abdomen. . . . . 94
5.8 Perspective volume rendering of a CT scan of an engine block. . . . . 94
5.9 Image quality comparison of the shear-warp algorithm with a ray caster. . 95
5.10 Image quality comparison of a bilinear filter with a trilinear filter. . . . . 96
5.11 Timing breakdowns without coherence optimizations. . . . . 106
5.12 Timing breakdowns with coherence optimizations. . . . . 108
5.13 Comparison of timing breakdowns for several data sets. . . . . 109
5.14 Pseudo-code for tracing a ray through an octree. . . . . 112
5.15 Memory overhead without coherence optimizations. . . . . 115
5.16 Memory overhead with coherence optimizations. . . . . 115
5.17 Impact of individual coherence optimizations. . . . . 117
5.18 Categories of volume data. . . . . 119
5.19 Results of the voxel throughput experiment. . . . . 122
5.20 Test volumes for the voxel coherence experiment. . . . . 123
5.21 Results of the voxel coherence experiment. . . . . 124

6.1 Pseudo-code for the multiprocessor algorithm. . . . . 135
6.2 Block diagram of a 2 x 2 DASH system. . . . . 137
6.3 Test data sets for the multiprocessor algorithm. . . . . 139
6.4 Interactive user interface for the multiprocessor algorithm. . . . . 139
6.5 Timing results for a rotation sequence on the DASH multiprocessor. . . . 141
6.6 Speedup curve on the Challenge multiprocessor. . . . . 142
6.7 Overhead vs. task size on the Challenge multiprocessor. . . . . 143
6.8 Overhead vs. number of processors on the Challenge multiprocessor. . . . 143
6.9 Speedup curve on the DASH multiprocessor. . . . . 145
6.10 Overhead vs. number of processors on the DASH multiprocessor. . . . . 146

7.1 An example shade tree. . . . . 155
7.2 Quantizing normal vectors. . . . . 158
7.3 Volume rendering with depth cueing. . . . . 161
7.4 Factoring the depth cueing function. . . . . 162
7.5 Diagram of Grant's shadow mask sweep algorithm. . . . . 166
7.6 Pseudo-code for the shadow rendering algorithm. . . . . 167
7.7 Volume rendering with the shadow rendering algorithm. . . . . 170
7.8 Limitations on the sampling rate in the shadow rendering algorithm. . . . 171
7.9 Aliasing in the shadow buffer. . . . . 172
7.10 Rendering mixtures of volume data and polygons. . . . . 174

8.1 Schematic diagram of a VolPack rendering context. . . . . 179

A.1 Coordinate systems used in the derivation of the shear-warp factorization. 193
A.2 Determining the affine shear coefficients. . . . . 198
A.3 Definition of the intermediate image coordinate system. . . . . 200
A.4 The shear transformation in the perspective shear-warp factorization. . . . 205
A.5 The scale transformation in the perspective shear-warp factorization. . . . 206

# Chapter 1

# Introduction

Three-dimensional arrays of data are a convenient and widely-used representation for information. Medical imaging technologies such as magnetic resonance (MR) and computed tomography (CT) can produce 3D arrays of data containing detailed representations of internal organs, allowing doctors to make a diagnosis without invasive surgery. Confocal microscopy can produce data revealing the 3D internal structure of individual cells in a biological sample with details as small as 0.1 microns. In seismic exploration, acoustic waves triggered by an explosive can be used to produce 3D maps of geologic structures below the surface of the Earth. As a final example, simulations based on computational fluid dynamics produce data on a 3D grid that can be used to predict the aerodynamics of a proposed automobile design or the regional weather.

All of these measurement and simulation techniques can produce very large arrays of numbers that are difficult to understand due to the sheer quantity of data. Data visualization techniques specialized for large 3D arrays are necessary. One emerging technique is volume rendering, a method for producing an image from a 3D array of sampled scalar data. Figure 1.1 shows a volume rendering produced from a CT scan of a human head. Volume rendering is a robust and versatile method for visualizing 3D arrays, but it has one key disadvantage that has prevented its widespread use: existing volume rendering algorithms are computationally expensive.

The high computational cost of volume rendering stems from the large size of typical

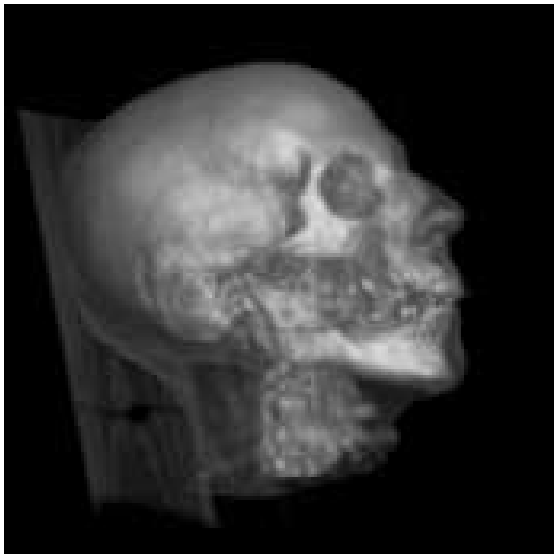


Figure 1.1: Example of a volume rendering produced from a CT scan of a human head. The input data contains 15 million samples.

data sets. Arrays containing many megabytes of data are common, and rendering such quantities of data quickly enough for interactive applications requires large amounts of processing power and high memory bandwidth. Past approaches to speeding up volume rendering have included trading off image quality for speed, or using large, expensive multiprocessors or special-purpose hardware.

This thesis presents a family of new volume rendering algorithms with lower computational costs and improved memory performance. The performance improvements stem from high-level algorithmic optimizations that exploit coherence in the data more efficiently than previous approaches. The new algorithms achieve rendering rates that are fast enough for interactive applications on desktop workstations, without using specialized hardware and without significantly compromising image quality. Parallel versions of the new algorithms running on small- to medium-scale multiprocessors achieve real-time rendering rates.

The rest of this chapter describes volume rendering in more detail, motivates the need for fast algorithms, and summarizes the contributions of this thesis.

## 1.1 Volume Rendering

The basic steps in any volume rendering algorithm consist of assigning a color and an opacity to each sample in a 3D input array, projecting the samples onto an image plane, and then blending the projected samples. The foundation of this visualization technique is a physically-based model for the propagation of light in a colored, semi-transparent material. In Section 1.1.1 we develop the model by writing an energy balance equation and reducing it to the volume rendering equation. In Section 1.1.2 we show how to simplify the volume rendering equation to a form known as the volumetric compositing approximation. This approximation is the starting point for the algorithms in this thesis.

### 1.1.1 The Volume Rendering Equation

The input to a volume rendering algorithm is a 3D array of scalars (as opposed to vectors which have a directional component). The array is called a *volume* and each element of the array is called a *voxel*. Although some volume rendering algorithms assume that each voxel represents a small cube in space with a constant scalar value, we will assume a voxel represents a point sample of a continuous scalar function.

Volume rendering is an approximate simulation of the propagation of light through a *participating medium* represented by the volume (Figure 1.2). The medium can be thought of as a block of colored, semi-transparent gel in which the color and opacity are functions of the scalar values in the input array. As light flows through the volume it interacts with the gel via several processes: light can be absorbed, scattered or emitted by the volume. Many other types of interactions are also possible, such as phosphorescence (absorption and re-emission of energy after a short time delay) and fluorescence (absorption and re-emission of energy at a different frequency). However since the goal of volume rendering is data visualization, not accurate simulation of the physics, we will omit many parts of the full optical model that are not necessary for the visualization. Given a simplified optical model, a volume rendering algorithm produces an image by computing how much light reaches each point on an image plane.

Light transport is governed by a special case of the Boltzmann equation from *transport theory*, which is the study of how a statistical distribution of particles (such as photons) flows

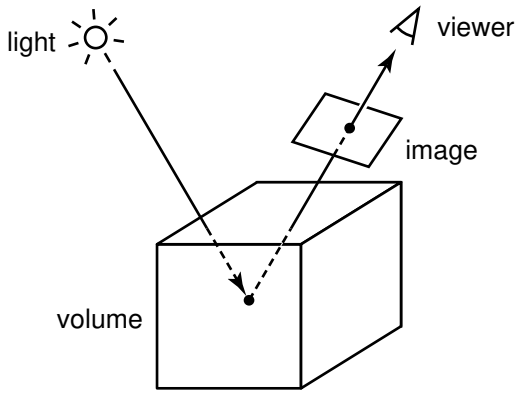



Figure 1.2: A physical model for volume rendering: A light ray propagates through a cube of semi-transparent gel and scatters onto the image plane.

through an environment. Arvo [1993], Glassner [1995, Ch. 12] and Cohen & Wallace [1993, Ch. 2] contain introductions to transport theory and its application to computer graphics, and Siegel & Howell [1992] contains a more detailed treatment of the underlying physics. Here we summarize the major results.

The flow of photons in a fixed environment rapidly reaches equilibrium, so the number of photons flowing through a given region of space in a particular range of directions must be constant over time. Thus if we consider the photons in a differential volume about a point $\mathbf{r}$ traveling in a differential range of angles about a direction $\vec{\omega}$ and we sum the change in the number of photons due to each possible type of interaction with the volume then the net change must be zero.

Instead of counting photons we will write an energy balance equation in terms of *radiance*, designated by $L(\mathbf{r}, \vec{\omega})$. Radiance describes the density of power transmitted by the photons at a particular point in a given direction. More precisely it is the power per unit area (projected onto the direction of flow) per unit solid angle, and it has units[1] of W/m$^2$-sr.

[1] Units will be expressed using the MKS system. The meter (m) is the unit of distance and the watt (W) is the unit of power (energy per unit time). The steradian (sr) is the unit of solid angle.

The energy balance equation states that the directional derivative of the radiance along a particular ray equals the sum of three terms representing losses and gains due to interactions with the volume:

$$\vec{\omega} \cdot \nabla L(\mathbf{r}, \vec{\omega}) = -\phi_t(\mathbf{r}) L(\mathbf{r}, \vec{\omega}) + \epsilon(\mathbf{r}, \vec{\omega}) + \int_{\mathcal{S}^2} k(\mathbf{r}, \vec{\omega}\,' \rightarrow \vec{\omega}) L(\mathbf{r}, \vec{\omega}\,')\, d\omega' \tag{1.1}$$

$\phi_t(\mathbf{r})$ is the *extinction coefficient* (with units of $\mathrm{m}^{-1}$) which equals the probability per unit distance that a photon traveling along the ray will be either absorbed or scattered into a different direction by the volume.

$\epsilon(\mathbf{r}, \vec{\omega})$ is the *emission function* (with units of W/$\mathrm{m}^3$-sr) that accounts for photons emitted within the volume.

$k(\mathbf{r}, \vec{\omega}\,' \rightarrow \vec{\omega})$ is the *scattering kernel* (with units of $\mathrm{sr}^{-1}\,\mathrm{m}^{-1}$) which is the probability per unit solid angle per unit distance that a photon moving in direction $\vec{\omega}\,'$ will scatter into direction $\vec{\omega}$. We integrate this function against the incoming radiance from all directions (denoted by $\mathcal{S}^2$) to account for the photons scattered into the ray.

Equation 1.1 is a first order differential equation known as the *differential form of the equation of transfer*. It can be written in an equivalent *integral form* as follows:

$$L(\mathbf{r}, \vec{\omega}) = e^{-\tau(\mathbf{r}, \mathbf{r_B})} L_B(\mathbf{r_B}, \vec{\omega}) + \int_{\Gamma(\mathbf{r}, \mathbf{r_B})} e^{-\tau(\mathbf{r}, \mathbf{r}')} Q(\mathbf{r}', \vec{\omega})\, dr' \tag{1.2}$$

$\tau(\mathbf{r}, \mathbf{s})$ is the integral of the extinction coefficient along the straight-line path between points $\mathbf{r}$ and $\mathbf{s}$:

$$\tau(\mathbf{r}, \mathbf{s}) \equiv \int_{\Gamma(\mathbf{r}, \mathbf{s})} \phi_t(\mathbf{r}')\, dr'$$

The path between the two points is denoted by $\Gamma(\mathbf{r}, \mathbf{s})$. We note that $e^{-\tau(\mathbf{r}, \mathbf{s})}$ is the integrating factor used to convert Equation 1.1 into Equation 1.2.

$L_B(\mathbf{r}, \vec{\omega})$ is a function specifying boundary conditions over a closed surface surrounding the volume. The point $\mathbf{r_B}$ is the intersection between the closed surface and the ray from $\mathbf{r}$ in direction $\vec{\omega}$.

$Q(\mathbf{r}, \vec{\omega})$ is short-hand for the sum of the emission and scattering terms:

$$Q(\mathbf{r}, \vec{\omega}) \equiv \epsilon(\mathbf{r}, \vec{\omega}) + \int_{S^2} k(\mathbf{r}, \vec{\omega}\,' \rightarrow \vec{\omega}) L(\mathbf{r}, \vec{\omega}\,')\, d\omega'$$

This function can be thought of as a generalized source function that accounts for all of the gains in the energy balance equation.

The integral form of the equation of transfer states that the radiance at any point along a ray equals a contribution from the radiance entering at the boundary of the volume plus a sum of contributions from the source terms along the ray. All of these contributions are multiplied by an exponential attenuation function that depends on the distance between the source and the measurement point (located at $\mathbf{r}$).

Equation 1.2 is a generalization of many equations used in rendering. Kajiya's rendering equation [Kajiya 1986] is equivalent to the equation of transfer specialized for environments consisting of surfaces (represented by boundary conditions) separated by empty space, in which case volume absorption, emission and scattering drop out of the equation. Zonal radiosity is a method for solving the equation of transfer assuming that the scattering and emission functions are isotropic, i.e. independent of $\vec{\omega}$ [Rushmeier & Torrance 1987]. Finally, with a different set of assumptions we arrive at the volume rendering equation.

First, we only model single scattering, i.e. all of the photons reaching the image are assumed to have scattered only once after leaving a light source. Second, we ignore absorption between the light source and the scattering event (but not between the scattering event and the image plane). Third, we assume isotropic absorption. Fourth and finally, we choose a simple boundary condition: we assume that the only energy entering the volume comes from a finite set of point light sources.

The first two assumptions allow us to drop the scattering integral from the equation of transfer since we can use the emission term to model the single scattering event. In this model the light sources deposit energy at each voxel and the voxels then re-emit the energy. The emission function describes the amount of re-emitted energy as a function of the viewing angle. Typical volume rendering algorithms compute the value of the emission function at a voxel using a local illumination model that is a function of the voxel's scalar value and the positions of the light sources. Since we have ignored absorption between the light source

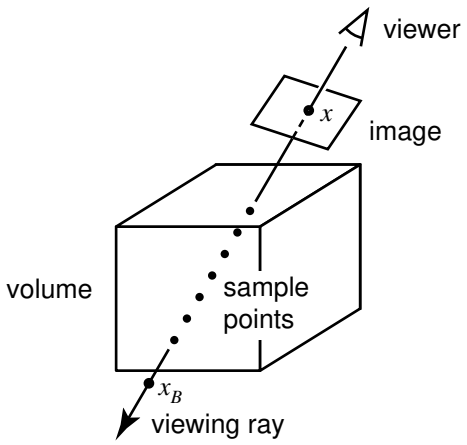



Figure 1.3: A simplified model for volume rendering: Each voxel in the volume emits light (computed using a local illumination model) and absorbs a fraction of the light passing through it, but scattering is ignored. The value of a pixel is computed by sampling the voxel values along a viewing ray from a point $x$ on the image to a point $x_B$ on the opposite boundary of the volume and then numerically evaluating the volume rendering integral.

and the voxel this model cannot be used to produce shadows. We discuss volume rendering algorithms that correctly render shadows in Chapter 7.

With all four assumptions Equation 1.2 reduces to the *volume rendering equation*:

$$L(x) = \int_x^{x_B} e^{-\int_x^{x'} \phi_t(x'')dx''} \epsilon(x')dx' \tag{1.3}$$

In this equation we have reparametrized the radiance in terms of a one-dimensional position variable $x$ representing distance along a viewing ray. The upper limit of integration is $x_B$, the point at which the ray exits the volume.

A simple ray casting algorithm based on the volume rendering equation operates by tracing rays into the volume parallel to the viewing direction as shown in Figure 1.3. The rendering algorithm then numerically evaluates the integral in Equation 1.3 along each ray. The user of the volume renderer specifies a mapping from the scalar value associated with each voxel to the parameters in the physical model, such as the absorption coefficient and the parameters of the illumination model for computing the emission function. In the next section we will describe a common method for evaluating the integral, called volumetric compositing, and we will refine the ray casting algorithm.

The volume rendering equation derived in this section is equivalent to the volume rendering models proposed by Blinn [1982], Levoy [1989] and Sabella [1988]. Other volume rendering models based on the more general equation of transfer, which includes multiple scattering, have also been proposed: Kajiya & Von Herzen [1984] derive an approximation for the case of a volume containing high-albedo (highly-reflective) particles, Krüger [1990] evaluates the full equation using Monte Carlo integration techniques, and Sobierajski & Kaufman [1994] perform volumetric ray tracing including ideal specular reflections at isosurfaces in the volume.

A key advantage of volume rendering is that the volume data need not be thresholded, in contrast to surface rendering techniques. Surface rendering techniques for volume data operate by fitting polygons to an isosurface in the volume (using, for instance, the marching cubes algorithm [Lorensen & Cline 1987]), and then rendering the polygonal model with traditional polygon rendering techniques. The surface-fitting process requires making binary decisions, i.e. the volume must be thresholded to produce regions that are either inside or outside the isosurface. If the volume contains fuzzy or cloud-like objects then a polygonal surface will be a poor approximation. In contrast, volume rendering algorithms never explicitly detect surfaces so they naturally handle fuzzy data as well as sharply-defined surfaces.

### 1.1.2 The Volumetric Compositing Approximation

In the previous section we began with an energy balance equation, we simplified the equation by ignoring some of the physical interactions, and we arrived at the volume rendering equation. In this section we derive an approximate numerical solution to the volume rendering equation called the *volumetric compositing equation*. Many volume rendering algorithms use this approximation, including the new algorithms in this thesis.

The volume rendering equation derived in the previous section is:

$$L(x) = \int_x^{x_B} e^{-\int_x^{x'} \phi_t(x'')dx''} \epsilon(x')dx'$$

The integrals in this equation can be evaluated using the rectangle rule:

$$L(x) = \sum_{i=0}^{n-1} e^{-\sum_{j=0}^{i-1} \phi_j \Delta x} \cdot \epsilon_i \Delta x = \sum_{i=0}^{n-1} \epsilon_i \Delta x \cdot \prod_{j=0}^{i-1} e^{-\phi_j \Delta x}$$

where

$$\begin{aligned} \epsilon_i &\equiv \epsilon(x + i\Delta x) \\ \phi_i &\equiv \phi_t(x + i\Delta x) \end{aligned}$$

for some sample spacing $\Delta x$ along the viewing rays. The expression on the right side of the equation can be written using the "over" operator from digital compositing [Porter & Duff 1984] as we can see by making the following substitutions. First define:

| | | | |
|---|---|---|---|
| $\alpha_i$ | $\equiv$ | $1 - e^{-\phi_i \Delta x}$ | the *opacity* of sample $i$ |
| $C_i$ | $\equiv$ | $(\epsilon_i / \alpha_i) \cdot \Delta x$ | the *color* of sample $i$ |
| $c_i$ | $\equiv$ | $C_i \alpha_i$ | the premultiplied color and opacity |

The premultiplied color and opacity is a convenient quantity for digital compositing [Porter & Duff 1984]. We have defined the "color" of a sample to be proportional to its emission normalized by its opacity. Intuitively a voxel with vanishing opacity must contain empty space so its emission should also vanish and this ratio is well-defined. This definition of color has no rigorous physical interpretation but it does have the same units as radiance (mapping radiances to displayable colors is a general problem in physically-based renderers, most of which perform some form of ad-hoc scaling just before displaying an image).

Given these definitions the volume rendering equation can be rewritten as follows:

$$\begin{aligned} L(x) &= \sum_{i=0}^{n-1} c_i \cdot \prod_{j=0}^{i-1} 1 - \alpha_j \\ &= c_0 + c_1(1 - \alpha_0) + c_2(1 - \alpha_0)(1 - \alpha_1) + \cdots \\ &\qquad + c_{n-1}(1 - \alpha_0) \cdots (1 - \alpha_{n-2}) \\ &= c_0 \text{ over } c_1 \text{ over } c_2 \text{ over } \cdots \text{ over } c_{n-1} \end{aligned} \tag{1.4}$$

Equation 1.4 is the volumetric compositing equation. It describes a method for numerically evaluating the volume rendering equation.

If we incorporate the volumetric compositing equation into the ray casting algorithm introduced in the previous section, we can render a volume as follows:

1. For each image pixel, cast a ray through the volume.

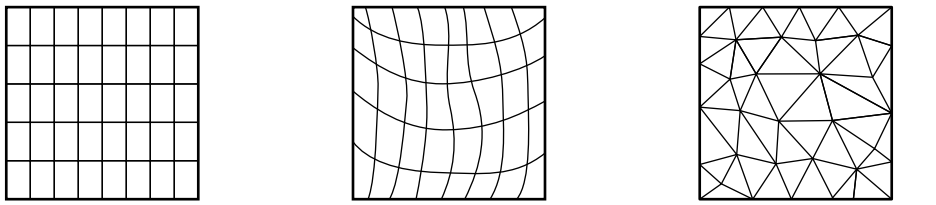
2. At regularly-spaced sample points along the ray compute a color $C_i$ and an opacity $\alpha_i$ from the scalar value in the volume.

3. Combine the colors and opacities along the ray using Equation 1.4.

This is a simple brute-force algorithm. If the volume contains $n$ voxels per side and the image is computed by casting $n^2$ rays with $n$ sample points per ray then the algorithm requires $O(n^3)$ operations. In this thesis we will propose methods to compute the same image with lower computational costs.

Several other compositing equations have been proposed for volume rendering. Upson & Keeler [1988] use the trapezoid rule and Novins & Arvo [1992] use higher-order quadrature rules to evaluate Equation 1.3 more accurately. There are also two common alternatives to volumetric compositing that model neither absorption nor scattering: X-ray projections and maximum intensity projections (MIP) [Laub & Kaiser 1988, Keller et al. 1989, Laub 1990]. In an X-ray projection the value of a pixel equals the integral of the emission function, and in a MIP rendering the value of a pixel equals the maximum emission of all samples along the ray.

X-ray projections can be computed rapidly using a theorem from Fourier analysis called the Fourier projection-slice theorem [Dunne et al. 1990, Malzbender 1993]. The advantage of the method is its low computational complexity: $O(n^2 \log n)$ operations to project an arbitrary $n^3$ voxel volume onto an arbitrary image plane. The greatest disadvantage is that the algorithm cannot be used to render images with occlusions as in conventional volume rendering since absorption is ignored, although algorithms to introduce alternative depth cues have been proposed [Levoy 1992, Totsuka & Levoy 1993]. The algorithms in this thesis can be adapted to use X-ray projections and MIP, but we will focus on methods for accelerating volume rendering algorithms based on the volumetric compositing equation.



(a) regular grid (b) curvilinear grid (c) unstructured grid

Figure 1.4: Examples of sampling grids in 2D: (a) regular grid, (b) curvilinear grid, (c) unstructured grid.

### 1.1.3 Data Representation and Sampling

In the previous discussion we have implicitly assumed that the physical parameters associated with a volume, e.g. the emission and absorption coefficients, are defined everywhere within the volume. In practice volume data is usually represented as an array of discrete samples, and the samples in the volume do not in general correspond to the sample points required to evaluate the volumetric compositing equation. Thus we must consider the representation of the volume and the method for determining values at arbitrary sample points within the volume.

There are several common sampling grids used for volume data (Figure 1.4). A *regular grid* consists of uniformly-spaced sample points located on a rectangular lattice. A *curvilinear grid* is a regular grid that has been warped with a non-linear transformation so that the sides of each cell need not be straight. Such grids are often used in computational fluid dynamics to conform the shape of the grid to the surface of an arbitrary object. An *unstructured grid* is an arbitrary collection of sample points with no implicit connectivity (although connectivity may be specified explicitly). Hybrid grids, in which several different grids are stitched together, are also possible.

In this thesis we restrict our attention to volume rendering algorithms for regular grids, which are the simplest grids and hence lead to the fastest algorithms.

Volumes sampled on other grid types can be resampled to a regular grid, although errors introduced during resampling may be a concern. Another problem is that multi-resolution hybrid grids used in computational fluid dynamics often have cells which range in size over many orders of magnitude. In this case a fixed-resolution grid with a sampling rate sufficiently high to accurately represent the highest-resolution regions of the volume would

result in an impractically large resampled data set.

A possible solution to both of these problems is to choose a regular grid with the same resolution as the display device and to resample a subset of the data set that contains the user's current region of interest. If the user zooms in or out (requiring a change of resolution) or pans to a new region of interest then the appropriate portion of the original data set must be resampled to the regular grid. The loss of accuracy due to resampling should be no worse than the inherent limits of the display device. Volume rendering algorithms for curvilinear and unstructured grids exist but are significantly slower than algorithms for regular grids [Wilhelms et al. 1990, Garrity 1990] so the computational cost of on-the-fly resampling may not be prohibitive.

Given a volume defined on a regular grid we must now find a way to determine values at arbitrary sample points within the volume. Suppose the volume was originally produced by sampling a continuous function. A basic theorem from sampling theory states that if the sampling frequency is at least twice the maximum spatial frequency in the continuous function (called the Nyquist rate) then the continuous function can be reconstructed perfectly from the samples [Bracewell 1986]. If the sampling rate is too low then *aliasing* artifacts such as jagged edges appear in the reconstructed volume.

In practice two problems prevent perfect reconstruction. First, the continuous function must be *band limited*, meaning that it cannot contain arbitrarily-high frequencies. To guarantee this property the original function must be low-pass filtered *before* sampling, and we can only reconstruct the low-pass filtered version of the original function. Many sources of volume data, such as medical imaging scanners, produce a sample by averaging their input signal over a small region of space (thereby applying a low-pass filter). We will therefore assume that the samples in a volume represent a band-limited continuous function.

The second problem is that the ideal reconstruction procedure (using a sinc filter [Bracewell 1986]) is expensive. The sinc filter uses a weighted contribution of every input sample to compute one output value. Cheaper, less-accurate filters use a small number of samples in the vicinity of the output location to compute each output value. The number of samples required by the filter is called the *support* of the filter, and the location of the samples in the input array is called the *footprint* of the filter. The most common reconstruction filters used for volume rendering are the nearest-neighbor and trilinear interpolation filters,

which produce acceptable results provided the frequencies in the band-limited signal are well below the Nyquist rate.

Reconstruction is one part of the procedure used to calculate samples along a viewing ray from samples in the original volume. The complete process is called *resampling*. We call the coordinate system of the volume samples *object space* and the coordinate system of the viewing ray samples *image space*. Resampling consists of four conceptual steps: (1) reconstruction of the continuous signal from the object-space samples, (2) warping of the continuous signal to image space, (3) band limiting the warped signal (only if the image-space sampling rate is less than the object-space sampling rate), and (4) sampling the band-limited signal to produce image-space samples. In real implementations the image-space samples are usually computed directly from the object-space samples without explicitly computing the intermediate signals. The four steps are combined by convolving the two filters into a single filter and then evaluating the combined filter at the image-space sample points.

Sampling issues will become important in Chapter 5 when we discuss image quality.

### 1.1.4 The Visualization Process

Since the goal of this thesis is to enable interactive volume rendering applications it is important to understand the steps involved in visualizing a data set. Figure 1.5 is a flow chart that shows the actions a user performs in a typical visualization session.

First the user acquires an array of input data. The acquisition process may include preparation steps such as resampling the volume to a regular grid, interpolating missing voxel values, and applying image processing operators to improve contrast.

Next the user *classifies* the volume. Classification is the process of assigning an opacity to each voxel ($\alpha$ in Equation 1.4). We will focus primarily on classification algorithms in which the user chooses a transfer function to compute the opacities from the scalar values. An alternative method is to partition the volume into specific structures (using a *segmentation* algorithm) and then to assign opacities to each structure.

Proper choice of the classification function is difficult. A good classification function reveals structures in the volume or highlights some subset of the volume in an informative

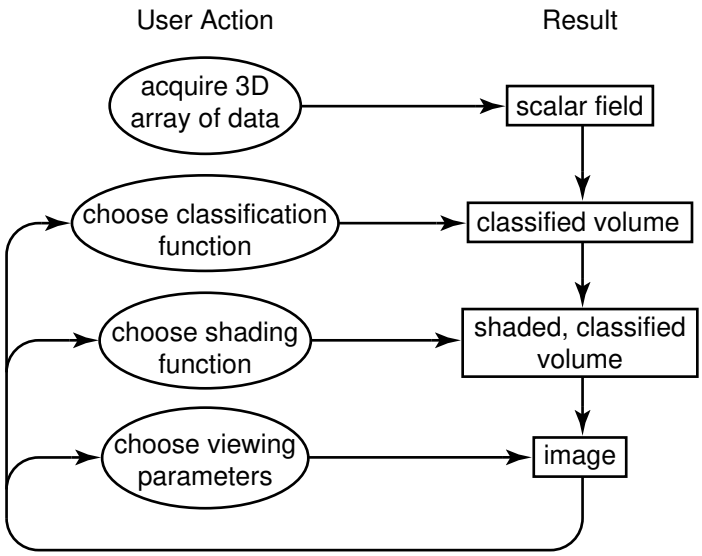



Figure 1.5: The visualization process is a feedback loop involving the user and three steps in the volume rendering algorithm: classification, shading, and rendering an image.

way. The appropriate choice depends both on the data itself and on the goals of the user. Unsuccessful classification can result in an image portraying a cloud-like fog which provides little information about the data. Experimentation with different classification functions is often necessary to produce an informative visualization, and interactive exploration can also lead to unexpected discoveries about the data.

After classifying the data the user chooses a shading function. The shading function specifies the illumination model and a rule for determining the color of each voxel. The rendering algorithm uses this information to compute $C$ in Equation 1.4. Proper choice of the shading function is also a subjective process, depending partly on the information the user wishes to convey and partly on characteristics of the human visual system. Careful use of visual cues, such as specular highlights, depth cueing and shadows can greatly improve the effectiveness of a visualization.

Finally the user chooses viewing parameters: the viewpoint, the type of projection (parallel or perspective), clipping planes, and so on. With this information and the shaded, classified volume, the rendering algorithm produces an image.

At this point the user will almost certainly want to adjust the parameters to change or improve the image, as indicated by the feedback arcs in Figure 1.5. Because of the subjective

choices involved in classification and shading it is difficult to decide on a set of parameters *a priori*, so the user must rely on trial and error to produce a good visualization. Furthermore, the user often does not know exactly what information exists in a new data set before exploring it with a visualization tool. The visualization application should support exploration of the parameter space in search of new insights about the data. User interaction with immediate visual feedback is essential for this process.

In the past, applications have used 2D slices or low-quality 3D renderings to give users visual feedback during selection of classification and shading parameters, coupled with a slow non-interactive volume renderer to produce final images. However, direct interaction with a 3D volume via a high-quality rendering provides better feedback and a simpler user interface. Our goal is to support interactive manipulation of all rendering parameters while immediately updating the rendered image. We wish to achieve this goal for high-resolution volumes using readily-available hardware. Given the current state of technology, we define our canonical problem as follows: we want to render a volume with $256^3$ eight-bit voxels on a mid-range desktop workstation[2] in less than one second. Achieving this goal requires a volume renderer that is an order of magnitude faster than the best previous methods.

## 1.2 A New Family of Fast Volume Rendering Algorithms

Many acceleration techniques for volume rendering have been proposed. The most successful have used spatial data structures to reduce the number of voxels processed, thereby reducing the computational cost of volume rendering by an order of magnitude compared to brute-force methods [Levoy 1990a, Subramanian & Fussell 1990, Zuiderveld et al. 1992]. Such algorithms achieve rendering times of several seconds to several tens of seconds for our canonical problem on current workstations. While an improvement, these algorithms are still not fast enough for a fully interactive application. Multiprocessors can deliver real-time volume rendering, but only very large and expensive machines have the necessary processing power.

[2]Current mid-range machines have integer performance ratings of about 100 SPECint92 and floating point performance ratings of about 100 SPECfp92. The machine we use for our performance results is a 150 MHz SGI Indigo2.

The primary contribution of this thesis is a new family of fast volume rendering algorithms. The algorithms achieve sub-second rendering times on a desktop workstation without graphics accelerators, and can generate images at real-time frame rates on small- to medium-size general-purpose parallel processors. The algorithms achieve these speed improvements through high-level algorithmic optimizations that reduce computational costs. As in previous work we use spatial data structures to exploit coherence in the volume. However, the new algorithms use an efficient scanline-order traversal to stream through the data structures in storage order, thereby reducing overhead incurred by accessing the data structures. These scanline algorithms rely on a factorization of the viewing transformation into a 3D shear, a projection to form a 2D intermediate image, and a 2D warp. The factorization enables simultaneous scanline-order access to both the volume and the image.

We also introduce separate algorithms optimized for different stages in the rendering process: an algorithm for rendering while experimenting with classification functions, an algorithm for rendering while experimenting with shading functions and viewpoints, and variants of the algorithms for rendering parallel and perspective projections. The family of algorithms proposed in this thesis addresses each step of the visualization process and enables an interactive volume visualization system on general-purpose hardware.

## 1.3 Organization

Chapter 2 begins with a brief survey of previous volume rendering algorithms and acceleration techniques. We present a simple taxonomy of volume rendering algorithms and highlight the advantages and disadvantages of each class in the taxonomy.

Next, in Chapter 3 we describe the technique that forms the basis of our new algorithms: volume rendering using the shear-warp factorization. The factorization allows us to combine the advantages of the different classes of existing algorithms. We describe previous uses of the factorization for rendering parallel projections and present a new extension to perspective projections.

Chapter 4 introduces three new volume rendering algorithms based on the shear-warp factorization. The algorithms use the shear-warp factorization and spatial data structures to maximize efficiency with minimal loss of image quality.

Chapter 5 contains a performance analysis of the shear-warp algorithms, including a detailed comparison to brute-force algorithms and an octree-accelerated ray casting algorithm. This chapter uses an optimized ray caster as an example to show how the shear-warp algorithms eliminate overheads inherent in other volume rendering algorithms. Chapters 3-5 expand on material previously published in Lacroute & Levoy [1994].

Chapter 6 describes a parallel volume rendering algorithm for shared memory multiprocessors based on the new serial algorithms. We describe implementations for two multiprocessors, the SGI Challenge and the Stanford DASH Multiprocessor, and analyze the performance of the algorithms. This chapter appears in slightly-modified form in Lacroute [1995].

Chapter 7 describes a collection of extensions to the basic rendering algorithms. The extensions include a fast, flexible shading system based on lookup tables, a fast depth cueing algorithm, and a method for rendering shadows with the shear-warp algorithm. The chapter also sketches algorithms for rendering volumes with clipping planes and mixed data sets containing both volume data and polygons.

Chapter 8 gives an overview of the design of VolPack, a volume rendering software library based on the algorithms in this thesis. The chapter focuses on the tradeoff between flexibility and speed when designing an application programmer's interface for volume rendering.

Finally, Chapter 9 summarizes the conclusions of this work and discusses the implications for future research.

# Chapter 2

# Prior Work

Existing volume rendering algorithms fall into four classes: ray casting algorithms, splatting algorithms, cell projection algorithms and multi-pass resampling algorithms. The two characteristics that distinguish each class are the order in which an algorithm traverses the volume and the method an algorithm uses to project voxels to the image. Each class of algorithms has performance advantages and disadvantages, and furthermore, some classes of algorithms are more amenable than others to particular acceleration techniques.

The volume rendering literature is too large to summarize here, so in this chapter we will focus on some representative algorithms and acceleration techniques to illustrate the differences. In Chapter 3 we will describe a method for combining the advantages of ray casting and splatting algorithms: volume rendering with the shear-warp factorization.

## 2.1 Volume Rendering Algorithms

We begin by describing the four classes of existing volume rendering algorithms: ray casting, splatting, cell projection and multi-pass resampling.

We will use a taxonomy based on loop orderings. Most volume rendering algorithms consist of six nested loops: three loops that iterate over the volume and three loops that iterate over the resampling filter kernel. The loops can be interchanged (subject to ordering constraints imposed by the volume rendering equation) to produce different algorithms. The

loop orderings are important for our purposes because they impact the performance characteristics of each algorithm.

### 2.1.1 Ray Casting

Ray casting algorithms produce an image by casting a ray through the volume for each image pixel and integrating the color and opacity along the ray, as already described in Section 1.1.2 [Levoy 1988, Sabella 1988, Upson & Keeler 1988]. Ray casters are called *image order* algorithms since their outer loops iterate over the pixels in the image:

```
for y_i = 1 to ImageHeight
  for x_i = 1 to ImageWidth
    for z_i = 1 to RayLength
      foreach x_o in ResamplingFilter(x_i, y_i, z_i)
        foreach y_o in ResamplingFilter(x_i, y_i, z_i)
          foreach z_o in ResamplingFilter(x_i, y_i, z_i)
            add contribution of Voxel[x_o, y_o, z_o] to ImagePixel[x_i, y_i]
```

The outer two loops iterate over the image pixels. The next loop iterates over sample points along a viewing ray in image space. Finally, the inner three loops iterate over the voxels required by the resampling filter to reconstruct one image-space sample. The body of the loop multiplies the value of a voxel by a resampling filter weight and adds the result to an image pixel.

Ray casters are also called *backward projection* algorithms since they calculate the mapping of voxels to image pixels by projecting the image pixels along viewing rays into the volume. Light rays flow forward from the volume to the image whereas viewing rays flow backward from the image into the volume.

The main disadvantage of ray casters is that they do not access the volume in storage order since the viewing rays may traverse the volume in an arbitrary direction. As a result, ray casters spend more time calculating the location of sample points (e.g. the voxel indices in the innermost loop body) and performing addressing arithmetic than other classes of volume rendering algorithms. Acceleration techniques based on spatial data structures further aggravate this problem. For example, when used with a ray caster some acceleration techniques require the calculation of intersections between rays and bounding boxes, as we will

see below.

A second problem is that ray casters may have higher memory overhead because they do not have good spatial locality. They do not access the volume in storage order, so the caches in modern processors are less effective at hiding memory latency. However, this problem is usually of secondary importance for software implementations because the overhead due to memory latency is insignificant compared to the amount of computation.

### 2.1.2 Splatting

In contrast to ray casting algorithms, splatting algorithms operate by iterating over the voxels [Westover 1990]. This class of algorithms computes the contribution of a voxel to the image by convolving the voxel with a filter that distributes the voxel's value to a neighborhood of pixels, a process that is called *splatting* [Westover 1989]. Algorithms of this type are called *object order* algorithms since the outer loop iterates over voxels in the object being rendered:

**for** $z_o = 1$ **to** VolumeDepth
  **for** $y_o = 1$ **to** VolumeHeight
    **for** $x_o = 1$ **to** VolumeWidth
      **foreach** $z_i$ in ResamplingFilter($x_o$, $y_o$, $z_o$)
        **foreach** $y_i$ in ResamplingFilter($x_o$, $y_o$, $z_o$)
          **foreach** $x_i$ in ResamplingFilter($x_o$, $y_o$, $z_o$)
            add contribution of Voxel[$x_o$, $y_o$, $z_o$] to ImagePixel[$x_i$, $y_i$]

Compared to the ray caster, the inner and outer loops have been interchanged. The three outer loops iterate over the voxels. The next loop iterates over the extent of the resampling filter kernel in the image-space depth dimension. Finally, the two inner loops iterate over the pixels affected by a single voxel.

Splatters are also called *forward projection* algorithms since voxels are projected directly into the image, in the same direction as the light rays.

Because splatting algorithms are object-order algorithms they can stream through the volume in storage order, an advantage compared to ray casters. However, accurately computing the filter footprint and the resampling weights is expensive because the filter is view-dependent. In a forward-projection algorithm the filter footprint must be scaled, rotated and

transformed arbitrarily depending on the viewing transformation. Moreover, in a perspective view this footprint changes from voxel to voxel. The filter kernel must also be chosen carefully to avoid gaps or excessive overlap between adjacent voxels after projection into the image. Thus it is difficult to implement a filter that is both efficient and that produces high quality results.

In theory, the splatter can produce exactly the same image as a ray caster provided both algorithms use the same filter weights. In practice, since it is difficult to compute the filter weights in the splatting algorithm, approximations must be used. Westover proposes a technique based on precomputed lookup tables for rotation-invariant Gaussian resampling filters. The method can be adapted for either fast execution or high image quality, but not both simultaneously [Westover 1990].

An alternative splatting algorithm uses 2D image warping techniques. Many volume renderers use this algorithm, including the commercial VoxelView package [Vital Images 1994] and methods based on texture mapping hardware [Cabral et al. 1994]. This algorithm partitions the volume into 2D slices and transforms each slice into image space by applying a 2D warp. The warped slices are then composited together to form the final image. This algorithm uses the same loop ordering as the standard splatting algorithm above except that the loop over the third dimension of the resampling filter kernel ($z_i$) is omitted since the resampling filter is two-dimensional. The 2D resampling filter is an approximation, but high-quality images can still be produced if the input volume has a high resolution. By using a 2D filter the algorithm reduces the complicated 3D resampling in Westover's algorithm to the well-understood problem of 2D image warping. However, the filter footprint and resampling weights are still view-dependent, so it is difficult to achieve a fast software implementation that does not compromise on image quality.

### 2.1.3 Cell Projection

A third class of algorithms consists of the cell projection techniques [Upson & Keeler 1988, Max et al. 1990, Shirley & Tuchman 1990, Wilhelms & Van Gelder 1991]. These methods are often used for volumes sampled on a non-regular grid. The first step is to decompose the volume into polyhedra whose vertices are the sample points (using a Delauney triangulation

for instance). The algorithm then sorts the polyhedra in depth order and scan converts the polygonal faces into image space. Finally, the algorithm evaluates the volume rendering integral between the front and back faces of each polyhedron to compute a color and an opacity at each pixel, and these values are composited into the image.

Cell projection algorithms are similar to splatting algorithms except that the former use polygon scan conversion to perform the projection. Software implementations of this algorithm are computationally expensive, but the polygon scan conversion can be accelerated with special-purpose graphics hardware.

In this thesis we concentrate on efficient software-only algorithms, which have the advantages of portability and flexibility, so we will not consider cell projection algorithms further.

### 2.1.4 Multipass Resampling

The fourth class of volume rendering algorithms operates by resampling the entire volume to the image coordinate system so that the resampled voxels line up behind each other on the viewing axis in image space. The voxels can then be composited together along the viewing axis as in a ray caster, except that in the resampled volume the viewing rays are always axis-aligned.

This class of algorithms uses multipass methods to resample the volume: the viewing transformation is factored into a sequence of simple shears and scales which are then applied to the volume in separate passes. Each shear or scale of the volume can be implemented with a scanline-order algorithm and a 1D resampling filter. For example, an affine transformation can be implemented using three passes. The first pass resamples scanlines in the $x$ dimension of the volume. The new volume becomes the input to the second pass which resamples scanlines in the $y$ dimension. The result then feeds into the third pass which resamples scanlines in the $z$ dimension. In terms of the loop-order taxonomy, multipass methods perform a series of object-order traversals of the volume each of the form:

```
OutputWidth = VolumeWidth × x_scale
for z = 1 to VolumeDepth
  for y = 1 to VolumeHeight
    for x = 1 to OutputWidth
      for t = t_min to t_max
        add contribution of Voxel[x/x_scale + t, y, z] to Output[x, y, z]
```

The three outer loops iterate over the volume and the inner loop iterates over a 1D filter kernel.

If the size of the filter kernel is $m$ voxels then an $n$-pass algorithm requires $nm$ operations per voxel whereas a single-pass resampling algorithm with a 3D filter requires $m^3$ operations per voxel. Thus with a filter kernel size greater than one, a three-pass resampling algorithm requires fewer operations than a single-pass algorithm.

Catmull & Smith [1980] introduced multipass resampling for warping 2D images, and the technique was first applied to volume rendering at Pixar [Drebin et al. 1988]. Hanrahan [1990] derived an efficient three-pass algorithm for volume rendering with affine viewing transformations and Vézina et al. [1992] extended it to a four-pass algorithm for perspective viewing transformations.

Multipass resampling methods are appropriate for certain types of specialized parallel hardware that take advantage of the regular communication patterns of a scanline-oriented algorithm, such as the Pixar Image Computer [Drebin et al. 1988] and SIMD array processors [Vézina et al. 1992]. However, this class of algorithms is not optimal for use on a workstation. The filter footprint of the trilinear interpolation filter used in most software volume renderers is two voxels wide, so multipass resampling has little performance advantage ($2 \cdot 3$ operations versus $2^3$ operations). Furthermore, multipass algorithms must use more expensive, higher-quality resampling filters to avoid image degradation (such as blurring) caused by the three or more resampling steps. Finally, this class of algorithms requires visiting every voxel multiple times and the volume must be transposed as the viewpoint changes (to avoid "bottleneck" problems [Catmull & Smith 1980]), leading to high costs for looping and memory overhead.

## 2.2 Acceleration Techniques

Many algorithmic optimizations have been proposed to speed up volume rendering. We will focus on acceleration techniques for ray casting and splatting since we saw in the previous section that they are the most appropriate methods for rendering regularly-sampled volumes on general-purpose hardware. In this thesis we also concentrate on acceleration techniques that do not trade off image quality for speed. For example, we do not consider subsampling the volume [Laur & Hanrahan 1991] or adaptive sampling [Levoy 1990a, Danskin & Hanrahan 1992]. Both of these techniques can miss or blur small features, although Laur & Hanrahan [1991] and Danskin & Hanrahan [1992] provide error bounds so that error can be controlled. In the rest of this section we consider two important optimizations that do not affect image quality: coherence acceleration using spatial data structures, and early ray termination.

### 2.2.1 Spatial Data Structures

An established acceleration technique for volume rendering is to exploit coherence in the volume by using a *spatial data structure*. For a given visualization of a data set, typically there are clusters of voxels that contribute useful information to the image and other clusters that are irrelevant. The purpose of a spatial data structure is to encode this type of coherence in such a way that the irrelevant voxels can be culled efficiently.

Spatial data structures are often used to encode the location of high-opacity voxels in a classified volume. Data structures that have been used for this purpose include octrees and pyramids [Meagher 1982, Levoy 1990a], k-d trees [Subramanian & Fussell 1990], run-length encoding [Reynolds et al. 1987, Montani & Scopigno 1990], shells [Udupa & Odhner 1993] and distance transforms [Zuiderveld et al. 1992]. Rendering algorithms use these data structures to skip transparent voxels rapidly.

Spatial data structures have also been used to encode other types of coherence. They have been used to accelerate isosurface renderings by building an octree containing min-max bounds [Wilhelms & Van Gelder 1992] or Lipschitz bounds [Stander & Hart 1994] for the scalar values within a volume, and to encode statistical information about the voxel

opacities so the sampling frequency on a viewing ray can be reduced in homogeneous regions of the volume [Danskin & Hanrahan 1992].

Algorithms that rely on spatial data structures require a preprocessing step in order to compute the data structure. Often the cost of preprocessing is acceptable if it can be amortized over many renderings. However, none of the existing coherence optimizations can be used for interactive classification of a volume since they all require preprocessing every time the voxel opacities change. Wilhelms & Van Gelder [1992] and Stander & Hart [1994] propose methods that allow the user to change a threshold value interactively during isosurface rendering but neither technique applies to general classification functions. While the preprocessing step is a disadvantage of previous algorithms, in Chapter 4 we present a new algorithm that eliminates the problem by supporting general classification functions with a classification-independent octree.

The speedup achievable with coherence optimizations is data-dependent, but typical classified volumes have a high degree of coherence. Furthermore the percentage of voxels that are transparent is typically 70-95% [Levoy 1990a, Subramanian & Fussell 1990], so the coherence can be used to eliminate a substantial fraction of work. Levoy reports that for medical data sets a ray caster with an octree to exploit coherence achieves a 3-5x speedup over a brute-force ray caster [Levoy 1990a].

These speedups are substantial, but we will see in Chapter 5 that most spatial data structures are more effective when used with an object order rendering algorithm rather than an image order algorithm. In the case of a ray caster with an octree-encoded volume, every ray must independently descend the octree, resulting in redundant computation. The costs of intersecting rays with octree nodes and switching between levels of the octree are significant overheads and limit the speedup, as we will see in Section 5.2. In contrast, a splatting algorithm with an octree only traverses the volume once and requires less-complicated addressing arithmetic since it sweeps through each octree node in storage order. Efficient use of spatial data structures is an important advantage of object-order algorithms.

### 2.2.2 Early Ray Termination

A second common acceleration technique for volume rendering is a technique called *early ray termination*. This optimization is most easily implemented in a ray casting algorithm: the algorithm traces each ray in front-to-back order and terminates the ray as soon as the accumulated ray opacity reaches a threshold close to full opacity. Any additional voxels that would have been reached by the ray must be occluded, so they need not be rendered. Two generalizations of early ray termination are Russian roulette [Arvo & Kirk 1990, Danskin & Hanrahan 1992], which terminates rays according to a probability that increases with accumulated ray opacity, and $\beta$-acceleration [Danskin & Hanrahan 1992], which decreases the sampling rate along the ray as optical distance from the viewer increases.

The goal of these optimizations is to reduce or eliminate samples in occluded regions of the volume. Levoy reports that a ray caster rendering medical data sets with early ray termination using a threshold of 95% opaque achieves speedups of 1.6-2.2x, and combined with an octree the overall speedup is 5-11x [Levoy 1990a].

Efficient implementation of early ray termination is trivial for a ray caster but is difficult for an object-order algorithm. The naive implementation of early ray termination in a splatting algorithm is to check each pixel's opacity before compositing a voxel into it. However, the algorithm would still have to visit every voxel in the data set. This optimization reduces the number of shading, resampling and compositing operations, but does not reduce the time spent traversing occluded portions of the volume.

Meagher [1982] proposes a more sophisticated algorithm that exploits coherence in the image. The algorithm uses a quadtree data structure to encode regions of the image with high accumulated opacity, in addition to an octree to encode coherence in the volume. To render a volume the algorithm traverses the octree nodes in front-to-back order and splats the voxels in each node into the image. Before splatting, the algorithm computes the visibility of a given octree node by scan-converting the silhouette of the node into image space and checking the accumulated opacity in the quadtree nodes that overlap the silhouette. If the quadtree nodes are opaque then the voxels in the octree node can be culled.

The octree and quadtree data structures significantly reduce the complexity of the algorithm. Instead of individually examining each voxel as in the naive algorithm, Meagher's

algorithm can cull an occluded octree node of arbitrary size by checking a small number of quadtree nodes. Greene et al. [1993] use Meagher's image-space quadtree technique in a rendering algorithm for very large polygonal models. However, the method is less appropriate for interactive volume rendering because the overhead required to scan convert octree nodes and traverse the quadtree data structure is likely to be large compared to the rendering time. This overhead eliminates much of the gain achieved by culling the occluded octree nodes.

Reynolds et al. [1987] propose a more efficient technique, called a "dynamic screen," for implementing early ray termination in an object-order volume rendering algorithm. Instead of using an octree and a quadtree, the technique uses run length encoding to encode coherence in both the volume and the image. Furthermore, the authors make the following observation: if the viewing transformation is a parallel projection then it transforms parallel lines in object space into parallel lines in image space. As a result, for some rotation of the image every voxel scanline projects onto a line that is parallel to the rotated image scanlines.

This observation leads to an efficient rendering algorithm: for each voxel scanline (in front-to-back order) the algorithm simultaneously traverses the voxel scanline and the image scanlines it projects onto. During the traversal, the algorithm uses the run length encoding of the voxel scanline to skip transparent voxels and the run length encoding of the image scanline to skip occluded voxels. The algorithm only composites voxels in runs containing non-transparent voxels that project onto non-opaque pixels. After the algorithm finishes traversing all of the voxel scanlines it rotates the 2D image to the correct orientation.

This algorithm never visits voxels that are occluded, which is the goal of early ray termination. It also has less overhead than Meagher's algorithm because it is less expensive to compute intersections between 1D runs than to scan-convert octree nodes and descend a quadtree. The implementation reported by Reynolds is fast compared to contemporary results but uses binary thresholding of the voxel data and point sampling, resulting in low-quality images. The algorithm is also limited to parallel projections. In Chapter 4 we will expand on this idea to support both high-quality images and perspective projections.

## 2.3 Chapter Summary

In this chapter we have seen that previous algorithms appropriate for volume rendering on general-purpose workstations fall into two classes, image-order ray casters and object-order splatting algorithms. We also observed that both types of algorithms have their own advantages: Object-order algorithms can efficiently traverse a spatial data structure to find the non-transparent voxels, but resampling is complicated because the filter is view dependent. Image-order algorithms must perform more work to traverse the spatial data structure but resampling is much simpler and these algorithms can take full advantage of early-ray termination. In the next chapter we describe a method for combining the advantages of these two classes of algorithms.

# Chapter 3

# The Shear-Warp Factorization

As we have seen in the previous chapter, image-order algorithms such as ray casters have the advantages of efficient, high-quality resampling and a simple implementation of early ray termination. On the other hand, object-order algorithms such as splatters use simpler addressing arithmetic and they achieve greater speedups from spatial data structures because they traverse the volume in storage order.

The new algorithms in this thesis combine all of these advantages. The algorithms rely on a factorization of the viewing transformation that simplifies projection from the volume to the image, allowing us to construct an object-order algorithm with the same advantages as an image-order algorithm. We call the factorization the *shear-warp factorization*.

In this chapter we first define the shear-warp factorization and motivate its use with a simple brute-force volume rendering algorithm. Next we derive separate versions of the factorization for affine viewing transformations and for perspective viewing transformations. The affine factorization has been used before in several volume rendering algorithms but the perspective factorization is new. We then discuss the properties of the factorization which we will use in the next chapter to develop our new volume rendering algorithms. Finally, we close this chapter with a discussion of previous uses of the affine shear-warp factorization.

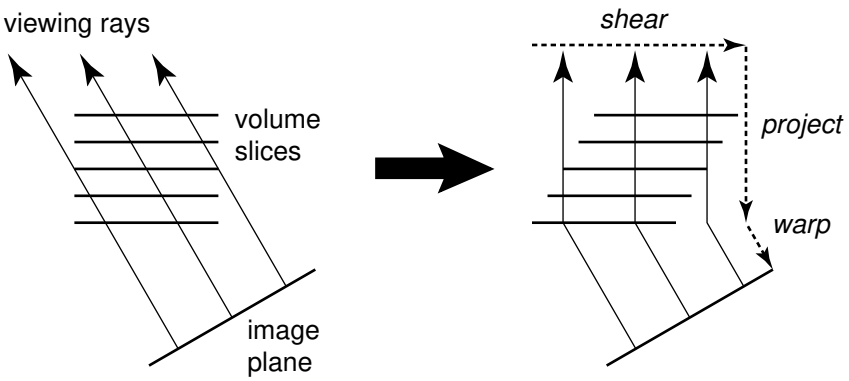



Figure 3.1: To transform a volume into sheared object space for a parallel projection we translate each slice. In sheared object space we can project the voxel slices into an image simply and efficiently.

## 3.1 An Overview of the Factorization

The arbitrary nature of the mapping from object space to image space complicates efficient, high-quality filtering and projection in object-order volume rendering algorithms and requires extra addressing arithmetic in image-order algorithms. This problem can be solved by transforming the volume to an intermediate coordinate system. We will choose a coordinate system with a simple mapping from the object coordinate system and which allows efficient projection to a 2D image.

We call the intermediate coordinate system "sheared object space" and define it as follows:

> Definition 1: By construction, in sheared object space all viewing rays are parallel to the third coordinate axis.

Figure 3.1 illustrates the transformation from object space to sheared object space for a parallel projection. The horizontal lines in the figure represent slices of the volume data viewed in cross-section. After transformation the volume has been sheared parallel to the set of slices that is most perpendicular to the viewing direction and the viewing rays are perpendicular to the slices. Since the shear is parallel to the slices, we can perform the transformation by simply translating each slice. For a perspective transformation the definition implies that each slice must be scaled as well as translated as shown schematically in Figure 3.2.

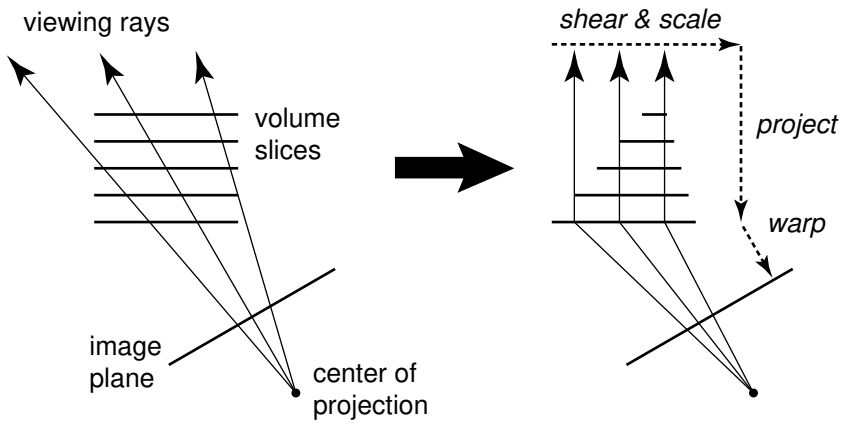



Figure 3.2: To transform a volume into sheared object space for a perspective projection we translate and scale each slice. In sheared object space we can again project the voxel slices into an image simply and efficiently.

A simple object-order volume rendering algorithm based on the transformation to sheared object space operates as follows (see Figure 3.3):

1. Transform the volume data to sheared object space by translating each slice. For perspective transformations, also scale each slice. Of the three possible slicing directions, choose the set of slices that is most perpendicular to the viewing direction. The translation and scaling coefficients are not necessarily integers so each voxel slice must be properly resampled.

2. Composite the resampled slices together in front-to-back order using the "over" operator. This step projects the volume into a distorted 2D intermediate image in sheared object space.

3. Transform the distorted intermediate image to image space by warping it. This second resampling step produces the correct final image.

Klein & Kübler [1985] and Cameron & Undrill [1992] proposed the parallel projection version of this algorithm, while the extension to perspective projections is new.

The reason for first computing a distorted intermediate image is that the properties of the factorization result in a very efficient implementation of the resampling and compositing loop. In particular, scanlines of voxels in each voxel slice are parallel to scanlines of

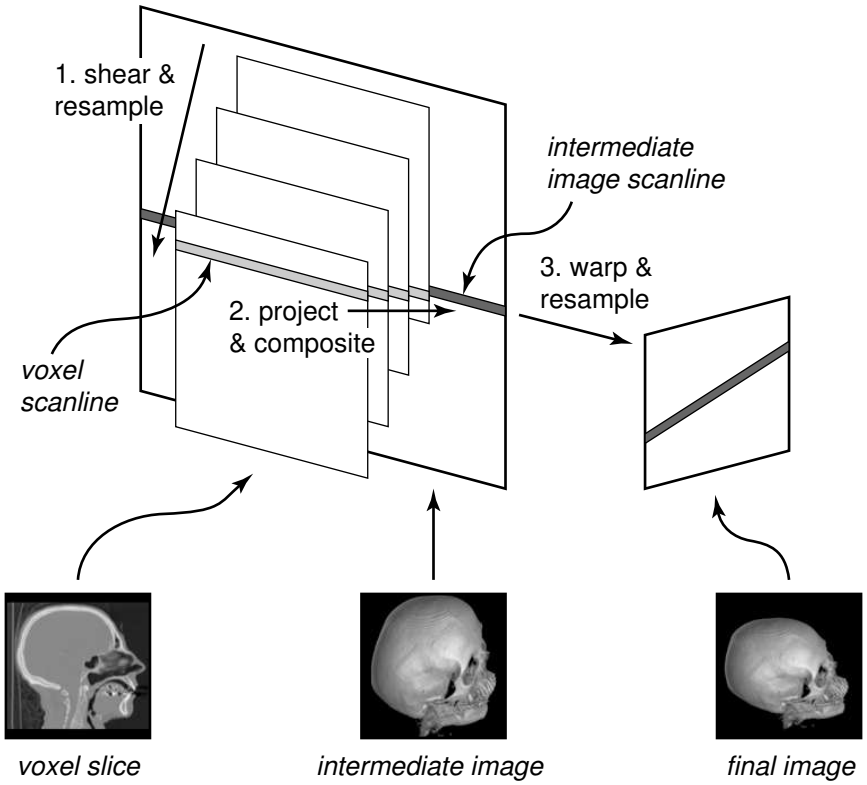



Figure 3.3: The brute-force shear-warp algorithm includes three conceptual steps: shear and resample the volume slices, project resampled voxel scanlines onto intermediate image scanlines, and warp the intermediate image into the final image. The shearing step shears the volume parallel to the volume slices in both the horizontal and vertical directions (although only a vertical shear is shown here). The horizontal shear accounts for rotation about the vertical axis and the vertical shear accounts for rotation about the horizontal axis. Rotation about the third axis is incorporated into the final warp. This algorithm works for arbitrary viewing transformations.

pixels in the intermediate image. Thus, the transformation from the object coordinate system to the intermediate image coordinate system (sheared object space) is very simple. We will elaborate on the properties of the factorization in Section 3.4. Transforming the 2D intermediate image into the final image requires a general warp but this operation is relatively inexpensive since the 2D intermediate image is much smaller than the 3D volume.

The brute-force shear-warp algorithm falls into a new category of the loop-order taxonomy introduced in the previous chapter:

```
for z_o = 1 to VolumeDepth
    for y_i = 1 to ImageHeight
        for x_i = 1 to ImageWidth
            foreach y_o in ResamplingFilter(x_i, y_i)
                foreach x_o in ResamplingFilter(x_i, y_i)
                    add contribution of Voxel[x_o, y_o, z_o] to ImagePixel[x_i, y_i]
```

The outermost loop iterates over slices of the volume. The next two loops iterate over pixels in the image. Finally, the inner two loops iterate over the voxels in the slice that contribute to a particular image pixel. There are only two loops over the resampling filter kernel because the algorithm uses a 2D resampling filter. In this algorithm the outer loop traverses the volume in object order, but the inner loops are similar to an image-order algorithm.

This loop ordering combined with the shear-warp factorization enables an algorithm that traverses the volume in object order while using nearly the same resampling filter implementation as an image-order algorithm. The only difference in the filter is that most image-order algorithms use a 3D resampling filter. The filter differences have little impact on image quality as we will demonstrate in Section 5.1.2.

## 3.2 The Affine Factorization

The shear-warp factorization for affine viewing transformations follows directly from Definition 1. Although the factorization has been used before, the derivation presented here has not been previously described.

Let $M_{\text{view}}$ be a 4x4 affine viewing transformation matrix that transforms points (represented as column vectors) from object space to image space. We want to factor $M_{\text{view}}$ into two matrices:

$$M_{\text{view}} = M_{\text{warp2D}} \cdot M_{\text{shear3D}}$$

$M_{\text{shear3D}}$ must be a shear matrix that transforms object space to sheared object space. By definition this requires shearing the coordinate system until the viewing direction becomes perpendicular to the slices of the volume.

In image space the viewing direction vector is $\vec{v}_i = (0, 0, 1)$. Let $\vec{v}_o$ be the viewing

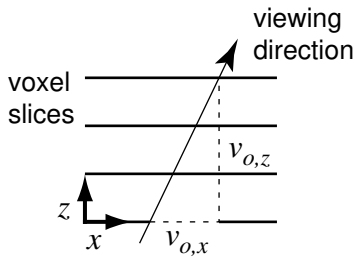



Figure 3.4: Determining the affine shear coefficients: This diagram shows a cross-section of the volume and the viewing direction projected onto the $x$-$z$ plane (for the case where the principal viewing direction is the $+z$ axis). To transform the volume to sheared object space we must shear the volume in the $x$ direction by $-v_{o,x}/v_{o,z}$. This shear causes the viewing direction to become perpendicular to the slices of the volume.

direction vector transformed to object space. It obeys the linear system of equations:

$$\vec{v}_i = M_{\text{view,3x3}} \cdot \vec{v}_o$$

where $M_{\text{view,3x3}}$ is the upper-left 3x3 submatrix of $M_{\text{view}}$. We can solve this system analytically for $\vec{v}_o$ using Cramer's Rule, yielding:

$$\vec{v}_o = \begin{bmatrix} m_{12}m_{23} - m_{22}m_{13} \\ m_{21}m_{13} - m_{11}m_{23} \\ m_{11}m_{22} - m_{21}m_{12} \end{bmatrix}$$

where the $m_{ij}$ are elements of $M_{\text{view}}$.

We define the *principal viewing axis* to be the object-space axis that is most parallel to the viewing direction. For simplicity we assume in this derivation that the principal viewing axis is the $+z$ axis of the object coordinate system as shown in Figure 3.4 (a derivation without this restriction appears in Appendix A). In the $x$-$z$ plane the slope of $\vec{v}_o$ is the ratio of the $x$ and $z$ components of the vector: $v_{o,x}/v_{o,z}$. The shear necessary in the $x$ direction to make the viewing direction perpendicular to the constant-$z$ slices equals the negative of this slope (because if the slope is zero no shear is necessary). A similar argument holds for

the shear in the Y direction. Thus the shear coefficients are:

$$s_x = -\frac{v_{o,x}}{v_{o,z}} = \frac{m_{22}m_{13} - m_{12}m_{23}}{m_{11}m_{22} - m_{21}m_{12}}$$

$$s_y = -\frac{v_{o,y}}{v_{o,z}} = \frac{m_{11}m_{23} - m_{21}m_{13}}{m_{11}m_{22} - m_{21}m_{12}}$$

We can now write the two factors of the view transformation matrix. The first factor shears the volume:

$$M_{3D} = \begin{bmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Once this transformation has been applied to the volume the transformed voxel slices can be composited together along the $+z$ axis. This operation forms an intermediate image.

The second factor of the viewing matrix describes how to warp the intermediate image into the final image:

$$M_{\text{warp2D}} = M_{\text{view}} \cdot M^{-1}_{\text{shear3D}} = M_{\text{view}} \cdot \begin{bmatrix} 1 & 0 & -s_x & 0 \\ 0 & 1 & -s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix is a general affine warp.

Appendix A contains a more detailed derivation of the affine factorization including many implementation details.

## 3.3 The Perspective Factorization

The shear-warp factorization for perspective viewing transformations also follows from Definition 1. This factorization is new.

Let $M_{\text{view}}$ be a 4x4 perspective viewing transformation matrix that transforms points

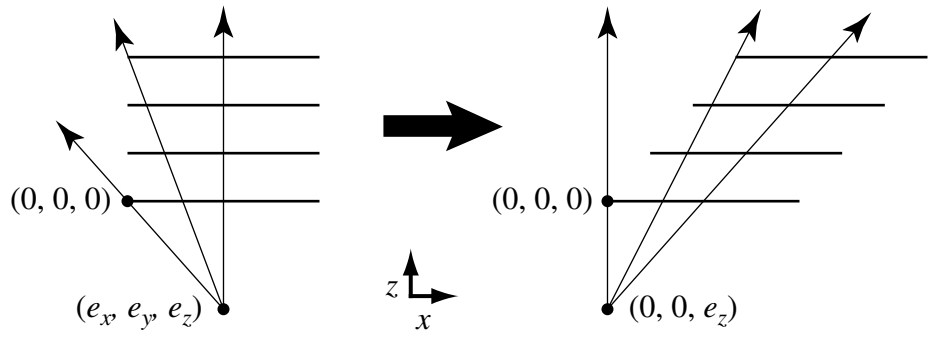



Figure 3.5: The shear transformation in the perspective shear-warp factorization. After the shear the viewing ray from the eyepoint $(e_x, e_y, e_z)$ through the origin of the volume is perpendicular to the slices of the volume.

from object space to image space. We want to factor $M_{\text{view}}$ into two matrices:

$$M_{\text{view}} = M_{\text{warp2D}} \cdot M_{\text{shear3D}}$$

The "shear" factor ($M_{\text{shear}}$) must contain both a shear transformation and a transformation that scales each volume slice in such a way that the viewing direction becomes perpendicular to the slices.

We will choose the shear component of $M_{\text{shear}}$ such that after it is applied to the volume the ray from the eye (the center of projection) through the origin of standard object space becomes perpendicular to the slices of the volume (Figure 3.5). For simplicity we assume that the principal viewing direction is the $+z$ axis of the object coordinate system.

To compute the shear we must first determine the location of the eye in object space ($\vec{e}_o$) given just the viewing transformation matrix.[1] In image space the homogeneous coordinates of the eye are $\vec{e}_i = (0, 0, -1, 0)$. The viewing transformation matrix must transform $\vec{e}_o$ into $\vec{e}_i$:

$$\vec{e}_i = M_{\text{view}} \cdot \vec{e}_o$$

We can solve this system analytically using Cramer's Rule, yielding an expression for each component of the eyepoint expressed as a determinant:

[1] In the affine derivation we started by finding the viewing direction instead of the viewpoint. Alternatively, we could have used the viewpoint by representing it with a point at infinite distance from the volume.

$$e_{o,x} = -\begin{vmatrix} m_{12} & m_{13} & m_{14} \\ m_{22} & m_{23} & m_{24} \\ m_{42} & m_{43} & m_{44} \end{vmatrix} \qquad e_{o,y} = \begin{vmatrix} m_{11} & m_{13} & m_{14} \\ m_{21} & m_{23} & m_{24} \\ m_{41} & m_{43} & m_{44} \end{vmatrix}$$

$$e_{o,z} = -\begin{vmatrix} m_{11} & m_{12} & m_{14} \\ m_{21} & m_{22} & m_{24} \\ m_{41} & m_{42} & m_{44} \end{vmatrix} \qquad e_{o,w} = \begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{41} & m_{42} & m_{43} \end{vmatrix}$$

These equations give the homogeneous coordinates of the eye point in object space. For convenience, define the homogenized coordinates as follows:

$$\begin{aligned} e_x &= e_{o,x}/e_{o,w} \\ e_y &= e_{o,y}/e_{o,w} \\ e_z &= e_{o,z}/e_{o,w} \end{aligned}$$

By inspection (from Figure 3.5) the shear transformation is:

$$M_{\text{sh}} = \begin{bmatrix} 1 & 0 & -\frac{e_x}{e_z} & 0 \\ 0 & 1 & -\frac{e_y}{e_z} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The second component of $M_{\text{shear}}$ is the scale component which scales the slices of the volume so that the sheared viewing rays become parallel. Thus the required transformation is a pure perspective transformation (Figure 3.6). Without loss of generality we arbitrarily choose to scale the voxel slice closest to the viewer (the $z = 0$ slice) by a factor of one. To find the required transformation, consider a particular viewing ray passing through point $\vec{p}$ in the $z = 0$ slice and point $\vec{q}$ in some other slice (Figure 3.6, left side). The perspective transformation must scale the voxel slice containing $\vec{q}$ by $p_x/q_x$ in the $x$ direction and $p_y/q_y$ in the $y$ direction (because if $p_x = q_x$ and $p_y = q_y$ then no scale is necessary). Using the

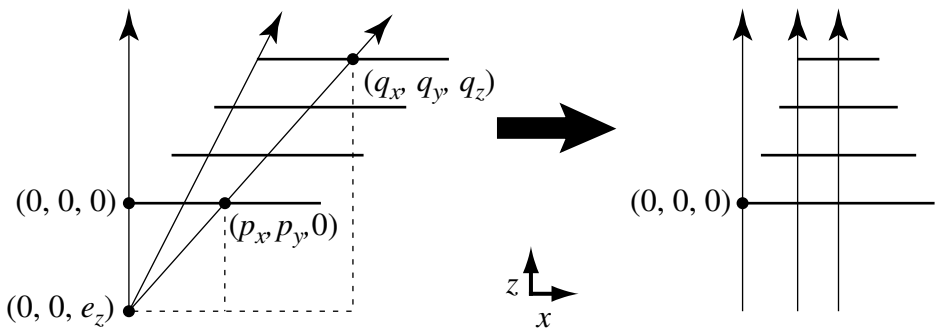



Figure 3.6: The scale transformation in the perspective shear-warp factorization. Each slice is scaled uniformly, but a different scale factor is applied to each slice. After the scale all of the viewing rays are perpendicular to the slices of the volume. This type of transformation is a pure perspective transformation.

similar triangles indicated by the dotted lines,

$$\frac{p_x}{q_x} = \frac{-e_z}{q_z - e_z} = \frac{1}{1 - q_z/e_z}$$

With a different set of similar triangles we can show that $p_y/q_y$ equals the same expression. Thus the required perspective transformation is:

$$M_{\mathrm{s}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{e_z} & 1 \end{bmatrix}$$

The shear factor of the perspective shear-warp factorization is the product of the pure shear and the perspective scaling transformations. After substituting the homogeneous coordinates of the eye location we obtain:

$$M_{\mathrm{shear}} = M_{\mathrm{s}} \cdot M_{\mathrm{sh}} = \begin{bmatrix} 1 & 0 & -\frac{e_{o,x}}{e_{o,z}} & 0 \\ 0 & 1 & -\frac{e_{o,y}}{e_{o,z}} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{e_{o,w}}{e_{o,z}} & 1 \end{bmatrix}$$

This matrix specifies that to transform a particular slice of voxel data in the $z = z_0$ plane from object space to sheared object space the slice must be translated by $(-z_0 e_{o,x}/e_{o,z}, -z_0 e_{o,y}/e_{o,z})$ and then scaled uniformly by $1/(1 - z_0 e_{o,w}/e_{o,z})$. This matrix reduces to the affine shear matrix if the viewing transformation is affine.

The warp transformation equals the view transformation matrix multiplied by the inverse of the shear transformation matrix:

$$M_{\text{warp}} = M_{\text{view}} \cdot M_{\text{shear}}^{-1} = M_{\text{view}} \cdot \begin{bmatrix} 1 & 0 & \frac{e_{o,x}}{e_{o,z}} & 0 \\ 0 & 1 & \frac{e_{o,y}}{e_{o,z}} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{e_{o,w}}{e_{o,z}} & 1 \end{bmatrix}$$

This transformation is a general perspective warp.

We have now completed the derivation of the perspective shear-warp factorization. Refer to Appendix A for a more detailed derivation, including a discussion of how to find the principal viewing axis. In some cases it is impossible to choose a single axis, for example if the perspective projection specifies a wide field of view, since there may be no single slicing direction that is compatible with all viewing rays. In this case the volume must be subdivided, which is discussed with the implementation of the perspective rendering algorithm in Section 4.2.5.

## 3.4 Properties of the Factorization

The projection from the volume to the intermediate image has several geometric properties that simplify the rendering algorithm:

**Property 1:** Scanlines of pixels in the intermediate image are parallel to scanlines of voxels in the volume array.

**Property 2:** All voxels in a given voxel slice are scaled by the same factor.

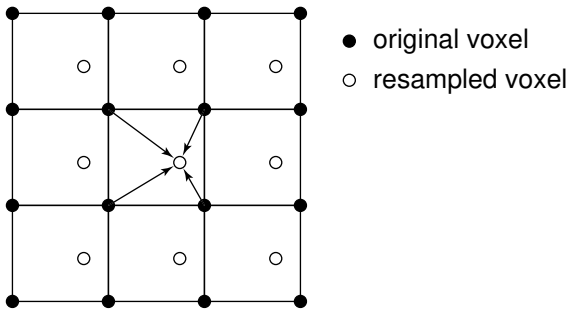



Figure 3.7: In the shear-warp algorithm for parallel projections each slice of the volume is simply translated, so as shown here every voxel in the slice has the same resampling weights.

**Property 3 (parallel projections only):** Every voxel slice has the same scale factor when projected into the intermediate image, and this factor can be chosen arbitrarily. In particular, we can choose a unity scale factor so that for a given voxel scanline there is a one-to-one mapping between voxels and intermediate-image pixels.

Property 1 follows from the fact that the shear matrix contains no rotation component. Property 2 follows from the fact that the shear matrix scales each voxel slice uniformly.

A useful implication of these properties for parallel projections is that every voxel in a given slice of the volume has the same resampling weights (Figure 3.7). Each slice of voxels is simply translated, so one set of resampling weights can be precomputed and then reused for every voxel. This fact eliminates the problems associated with efficient resampling in object-order volume rendering algorithms.

In the next chapter we will use these properties to develop efficient volume rendering algorithms.

## 3.5 Existing Shear-Warp Algorithms

Earlier uses of the shear-warp factorization for parallel projections include three applications: reducing the cost of projection in object-order volume rendering algorithms, reducing the cost of tracing rays in ray casting algorithms, and regularizing communication patterns in parallel volume rendering algorithms for massively-parallel SIMD arrays.

Klein & Kübler [1985] and Ylä-Jääski, Klein & Kübler [1991] describe an object-order rendering algorithm for volume data using the shear-warp factorization. This early work predates modern volume rendering, so the algorithm makes two simplifications that were common at the time: it thresholds each voxel to a binary value, thereby reducing the compositing operation used in volume rendering to the simpler problem of rendering the front-most non-transparent voxel, and it uses point sampling to resample the volume, thereby eliminating the cost of a resampling filter. Apart from these two differences, the algorithm is identical to the volume rendering algorithm for parallel projections presented in Section 3.1.

Ylä-Jääski, Klein & Kübler [1991] and Yagel & Kaufman [1992] independently proposed methods that use the shear-warp factorization to accelerate ray casting algorithms for parallel projections. Both algorithms take advantage of Property 3 of the factorization to reduce the amount of addressing arithmetic required for ray casting. A basic ray caster determines the step from one sample point to the next along a ray by adding a delta vector to the current sample location, converting the new floating-point sample location to an integer array index, and then computing resampling weights and looking up voxel values in the volume array. Alternatively, a discrete line drawing algorithm such as a Bresenham algorithm can be used to eliminate some of the floating point arithmetic. In either case, the addressing arithmetic calculations are a significant overhead in a ray caster (unlike in an object-order algorithm).

However, ray casters that use the shear-warp factorization simplify the computation by casting rays in sheared-object space instead of in image space. Suppose we cast one ray from the center of each intermediate image pixel (Figure 3.8). Then for a given slice of the volume each ray pierces the slice in such a way that the resampling weights are the same for every ray (recall Figure 3.7). Furthermore, if the array indexes (or memory offsets) of the voxels pierced by one ray are stored in a "template," that same template can be reused for every other ray by simply adding an integer offset to each value in the template. The precomputed template can be used to eliminate most of the computation associated with finding sample points along a ray, computing voxel addresses, and computing resampling weights. As in the object-order algorithm, the template-based ray casting algorithm computes a distorted intermediate image that must be warped into the correct final image. Using templates, Yagel reports speedups of roughly 2-3x compared to a brute force algorithm when

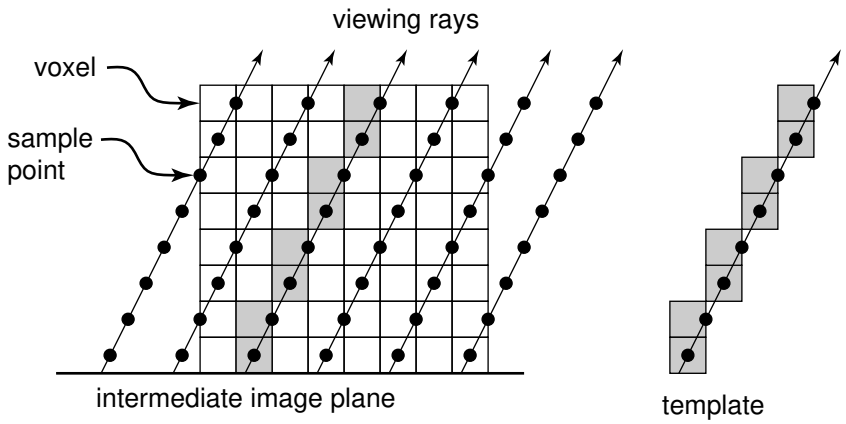



Figure 3.8: Ray casters based on the shear-warp factorization use a precomputed "template" containing resampling weights and memory offsets for the sample points in the volume (after Yagel & Kaufman [1992]).

discrete line drawing and nearest-neighbor sampling are used [Yagel & Kaufman 1992], and 1.3-1.4x when continuous line drawing and trilinear interpolation are used [Yagel & Ciula 1994].

Cameron & Undrill [1992] use the shear-warp factorization in a parallel volume rendering algorithm for a SIMD massively parallel multiprocessor. They also use the object-order rendering algorithm described in Section 3.1. They parallelize the algorithm by assigning each pixel in the intermediate image to a different processor. The volume is loaded into the processor memories in such a way that each voxel in a slice of the volume is assigned to a different processor. The voxels in each slice are shifted via a 2D mesh network to implement the shear and then composited into the intermediate image in parallel. The fact that intermediate image scanlines and voxel scanlines are aligned (Property 1) guarantees that the shift can be performed using the regular communication of a 2D mesh network, and the one-to-one mapping between pixels and voxels (Property 3) ensures that the voxels required to compute a given intermediate image pixel are always available in the local memory or the immediately-adjacent memories in the 2D mesh. The shear-warp factorization therefore enables the algorithm to work on a 2D SIMD array because of the regular communication patterns and the identical processing required for each voxel.

Schröder & Stoll [1992] describe a parallel volume rendering algorithm for a SIMD massively-parallel multiprocessor that independently uses the same rendering algorithm as

Yagel. The algorithm assigns one column of voxels (aligned with the principal viewing axis) to each processor. As the algorithm projects rays from the intermediate image through the volume the partially-computed ray colors and opacities are communicated to the processor containing the next required voxel. This algorithm computes the ray template on-the-fly during rendering. Because the template is the same for every ray the processing for each ray is identical and the algorithm can communicate the partially-accumulated ray values using nearest-neighbor shifts on a 2D mesh network. Just as in Cameron and Undrill's algorithm, the shear-warp factorization makes the communication patterns very regular, although in Schröder and Stoll's algorithm the ray values are communicated instead of the volume data.

To summarize, Klein, Kübler and Ylä-Jääski's object-order algorithm uses the shear-warp factorization to simplify projecting voxels into an image. Their ray casting algorithm and Yagel's template method both use the factorization to reduce the cost of tracing a ray. Cameron and Undrill's algorithm and Schröder and Stoll's algorithm both use the factorization to construct volume renderers with regular communication patterns appropriate for brute-force SIMD parallel implementations.

We extend this work by introducing a factorization for perspective projections and by exploiting the properties of the factorization in new ways to improve the performance of coherence optimizations. These topics are the focus of the next chapter.

## 3.6 Chapter Summary

The shear-warp factorization allows us to incorporate the advantages of ray casting algorithms into an object-order algorithm. In this chapter we presented an existing version of the factorization for parallel projections and a new version for perspective projections. In the next chapter we will combine the factorization with the optimizations from the previous chapter to develop three new volume rendering algorithms. We will exploit the properties of the factorization to maximize the benefit of spatial data structures and early ray termination.

# Chapter 4

# Three Fast Volume Rendering Algorithms

This chapter describes three new volume rendering algorithms using the shear-warp factorization. They are all based on the object-order algorithm presented in Section 3.1 of the previous chapter: translate each slice of the volume, composite the slices together, and warp the result. In this chapter we add data structures that allow the rendering algorithms to exploit coherence. The first new algorithm is optimized for parallel projections and assumes that the opacity transfer function does not change between renderings. The second algorithm supports perspective projections. Both of these algorithms are suitable for interactively manipulating a previously-classified volume's viewing and shading parameters. Finally, the third algorithm is designed for interactively classifying volumes. It allows the opacity transfer function to be modified as well as the viewing and shading parameters, with a moderate performance penalty.

## 4.1 Parallel Projection Rendering Algorithm

### 4.1.1 Overview of the Parallel Projection Algorithm

Property 1 of the shear-warp factorization states that voxel scanlines in the sheared volume are aligned with pixel scanlines in the intermediate image, which means that the volume

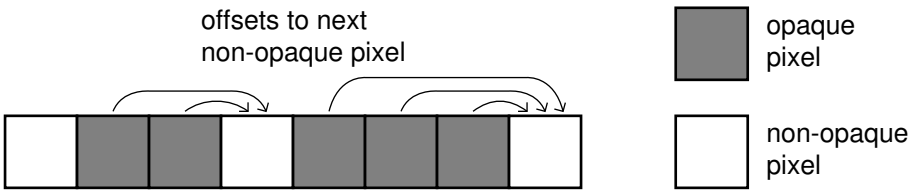



Figure 4.1: Offsets stored with opaque pixels in the intermediate image allow occluded voxels to be skipped efficiently.

and image data structures can be traversed simultaneously in scanline order. Scanline-based coherence data structures such as run-length encoded representations are therefore a natural choice.

The first data structure we use is a precomputed run-length encoding of the voxel scanlines. This data structure allows us to skip over transparent voxels during rendering. A run-length encoding of a scanline consists of a series of runs represented by a run length and a data value for each run [Foley et al. 1990, Ch. 17]. In our data structure the encoded scanlines consist of two types of runs, transparent and non-transparent, defined by a user-specified opacity threshold. We build this data structure during a preprocessing step that scans through the original volume, classifies each voxel (i.e. assigns an opacity), compares the voxel opacities to the threshold, and outputs the run-length encoded representation.

During rendering we use the run-length encoding to determine which parts of the volume are transparent. The rendering algorithm traverses the encoded volume slice-by-slice, just as in the brute-force shear-warp algorithm described earlier. At the beginning of the data for each voxel slice the algorithm uses the shear matrix to compute the translation for the slice. The algorithm then streams through the encoded runs, translates and resamples only the voxels in the non-transparent runs, and composites the resampled voxels into the intermediate image. Using this algorithm the run-length encoded volume can be decoded, sheared, resampled and composited into an image in a single loop without explicitly constructing the undecoded volume or the sheared volume. Furthermore, the algorithm eliminates work in the transparent portions of the volume.

Second, to take advantage of coherence in the image, we use a run-length encoding of the intermediate image. The algorithm constructs this data structure as it computes the intermediate image during rendering. The run-length encoding consists of an offset stored with

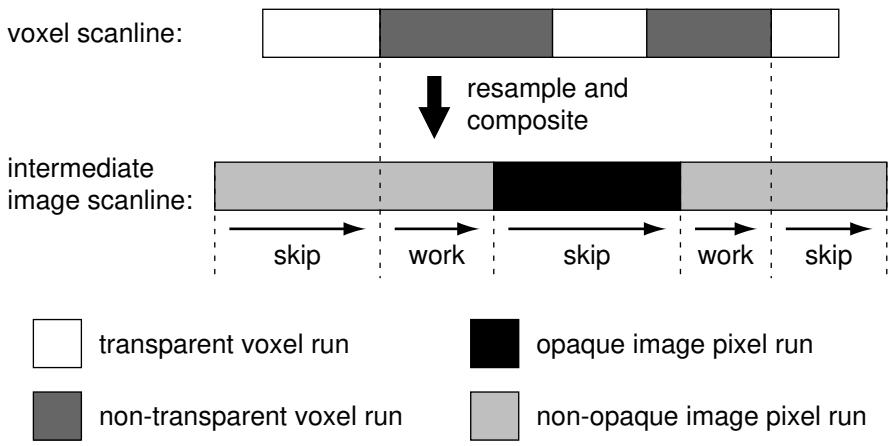



Figure 4.2: Resampling and compositing are performed by streaming through both the voxels and the intermediate image in scanline order, skipping over voxels that are transparent and pixels that are opaque.

each opaque intermediate image pixel (Figure 4.1). The offset points to the next non-opaque pixel in the same scanline. An image pixel is defined to be opaque when its opacity exceeds a user-specified threshold, in which case the corresponding voxels in yet-to-be-processed slices are occluded. The offsets associated with the image pixels are used to skip runs of opaque pixels without examining every pixel. We store an offset with every pixel, not just the first pixel in a run, since it is possible to jump into the middle of a run of opaque pixels after traversing a run of transparent voxels.

These two data structures and Property 1 of the factorization lead to a fast scanline-based rendering algorithm (Figure 4.2). The algorithm streams through both the voxels and the intermediate image in scanline order, using the data structures to avoid work. By marching through the volume and the image simultaneously in scanline order we reduce addressing arithmetic. By using the run-length encoding of the voxel data to skip voxels that are transparent and the run-length encoding of the image to skip voxels that are occluded, we perform work only for voxels that are both non-transparent and visible.

For voxel runs that are not skipped we use a tightly-coded loop that performs shading, resampling and compositing. Properties 2 and 3 of the factorization allow us to simplify the resampling step in this loop since the resampling weights are the same for every voxel in a slice. We use a shading system based on lookup tables to compute the color of each voxel,

as described in Chapter 7.

After the volume has been composited the intermediate image must be warped into the final image. Since the 2D image is smaller than the volume this part of the computation is relatively inexpensive.

### 4.1.2 The Run-Length Encoded Volume

The run-length encoded volume must be constructed in a preprocessing step before rendering. The preprocessing algorithm traverses the volume in storage order, computes the opacity of each voxel, and then compares each voxel's opacity to a threshold opacity to determine if it is transparent or non-transparent. The algorithm uses the threshold only to determine which voxels have negligible opacity; the unthresholded opacities are retained for rendering. As the preprocessing algorithm classifies voxels it collects them into runs. A run consists of a sequence of contiguous voxels that are all transparent or all non-transparent. The algorithm stores the lengths of the runs along with the data for the non-transparent voxels. The transparent voxels are discarded.

The data structure for the encoded volume consists of three arrays (Figure 4.3): an array of run lengths, an array of non-transparent voxels, and an array of pointers into the first two arrays that is used to find the beginning of each slice of the original volume. In the run length array every other entry contains the length of a run of transparent voxels. The remaining entries contain the lengths of the intervening runs of non-transparent voxels. Each run length is represented as an 8-bit byte. If the length of a run exceeds the maximum representable length (255 in the current implementation) then the run is split into several runs interspersed with zero-length runs of the opposite type. Runs are also split at the end of a voxel scanline to ensure that every voxel scanline begins with a new run.

The voxel array simply contains all of the non-transparent voxels, packed contiguously. The transparent voxels are not stored since they are not needed during rendering and memory consumption can be reduced by omitting them. The run length array makes it possible to determine where transparent voxels must be inserted in the compressed voxel array to reconstruct the volume.

Each voxel in the voxel array contains an opacity and any data required for shading. The

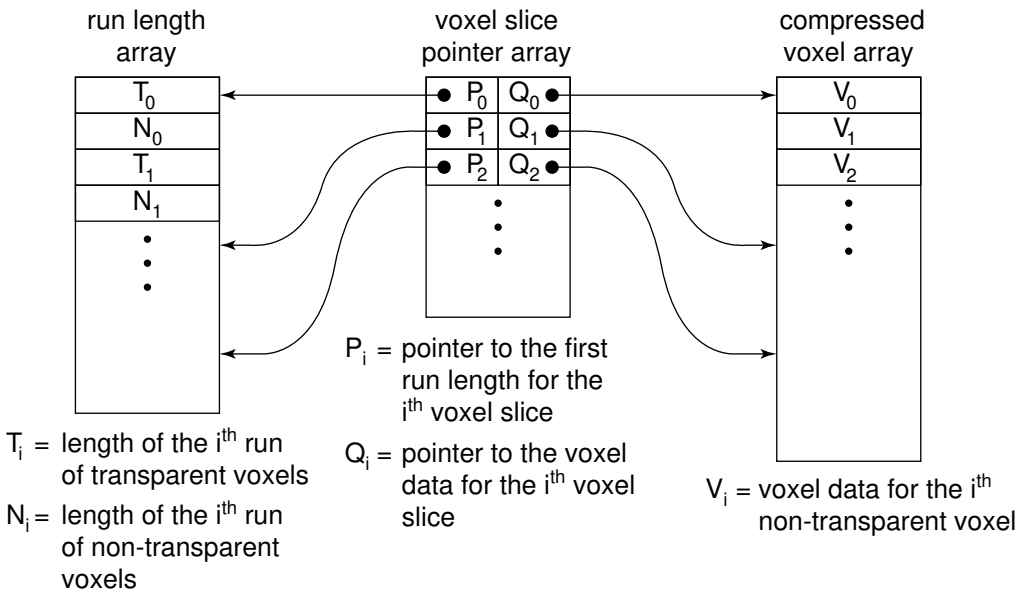



Figure 4.3: The run-length encoded volume consists of three data structures: an array of run lengths (left), an array of pointers that provides random access to voxel slices (middle), and an array containing the non-transparent voxels (right).

shading data will be used by the rendering algorithm to compute the color of the voxel. The color cannot be precomputed since it may depend on the viewpoint and the location of the light sources, both of which might change from frame to frame in a rendering sequence. In the current implementation the opacity data is stored in one byte and the shading data is user-configurable. Typically three bytes are used for shading data (two bytes for a precomputed surface normal and one byte for the original scalar value), for a total voxel size of four bytes per voxel.

The run lengths and the voxel data are stored in separate arrays to avoid byte alignment problems. On some processor architectures particular data types must be aligned to a particular byte boundary in memory. If single-byte run lengths were interspersed with voxels containing larger data types then additional padding bytes would be required.

One difficulty introduced by run-length encoded representations is that random access to the encoded data is not possible. The voxels must be decoded from the beginning of the data structure. The third array in our representation, which contains pointers to the other two arrays, solves this problem. Each entry in the third array points to the first run length

and the first non-transparent voxel for a given slice of the volume, allowing random access to any voxel slice. Random access to individual scanlines, which will be necessary for the parallel rendering algorithm described in Chapter 6, can be supported by including one entry per voxel scanline in this array.

A second difficulty with a run-length encoded volume representation is that the volume can only be traversed in the order that the voxels are encoded. The rendering algorithm may need to use a transposed version of the volume data, depending on the principal axis of the viewing transformation. To solve this problem we precompute three run-length encodings, one for each of the three principal viewing axes. The rendering algorithm chooses the appropriate copy depending on the viewing transformation. It is best to load all three copies into memory simultaneously to avoid page faults when the viewing direction switches to a new principal axis, but maintaining three copies in memory does not impact cache performance since only one copy is needed to render each frame. Furthermore, because transparent voxels are not stored the total size of the encoded volume is typically smaller than the original volume, even with three-fold redundancy.

The combination of the three transposes of the run-length encoded volume form a view-independent spatial data structure that can be precomputed before rendering, provided that the classification function is known. Section 4.3 later in this chapter describes an alternative data structure that does not require specifying the classification function.

### 4.1.3 The Run-Length Encoded Image

The intermediate image is also a run-length encoded data structure: it encodes runs of opaque and non-opaque pixels. However it has different requirements than the run-length encoded volume data structure since the volume can be precomputed whereas the intermediate image must be modified during rendering. We therefore need a data structure that supports dynamic creation of opaque pixel runs and merging of adjacent runs, as well as a fast method to find the end of a run.

This problem is equivalent to the UNION-FIND problem (also called the disjoint-set union problem) [Aho et al. 1974, Cormen et al. 1990]. The problem involves a collection of objects grouped into disjoint sets. In our case the objects are opaque pixels and the sets

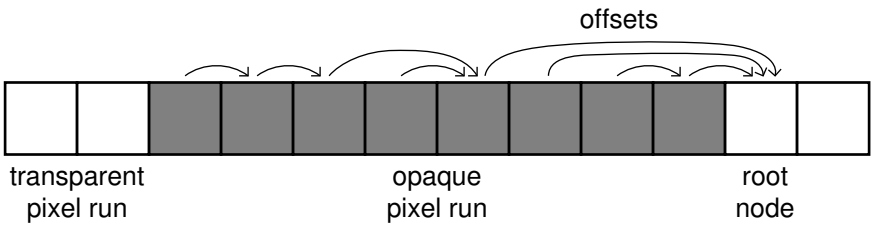



Figure 4.4: A tree data structure to represent an opaque pixel run: each opaque pixel has an offset that points to another pixel in the run or to the end of the run.

are runs of opaque pixels. Each set contains a single "representative" that serves as the name of the set. We will always define the last pixel in a run to be the representative. Three operations are defined: (1) CREATE, which creates a new set containing a single element, (2) UNION, which merges two sets into a new set with a single representative, and (3) FIND, which finds the representative associated with a set, given any element in the set.

A common data structure for representing the sets is a forest of trees. In our algorithm we represent each opaque pixel run by a tree (Figure 4.4). We implement the trees using offsets stored with each pixel in the intermediate image. The intermediate image data structure is a 2D array of pixels. Each pixel contains a color, an opacity, and a relative offset. In the current implementation the color and opacity are stored in single-precision floating point format. The relative offset is a one-byte value containing the number of pixels to skip to reach the pixel's parent node. The root of each tree is the non-opaque pixel at the end of the opaque pixel run (although the non-opaque pixel is not actually part of the run). Each non-opaque pixel has an offset equal to zero.

The rendering algorithm uses the intermediate image data structure as follows. The algorithm initializes the intermediate image by making all of the pixel opacities transparent and all of the offsets zero. During rendering, before the algorithm composites a voxel into a pixel it checks the value of the pixel's offset. If the offset is non-zero then the pixel is opaque, so the algorithm performs a FIND operation to find the end of the run and then skips to it. On the other hand, if the offset is zero then the algorithm completes the compositing operation. After the compositing operation, if the new opacity exceeds the maximum opacity threshold then the algorithm performs a CREATE operation to create a new opaque pixel run, and possibly a UNION operation to merge the run with adjacent runs.

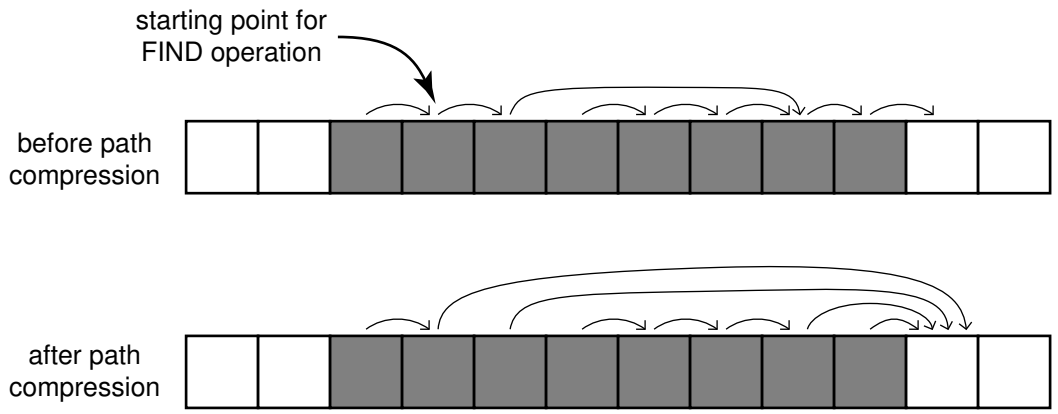



Figure 4.5: After the rendering algorithm traverses a chain of offsets to find the end of an opaque pixel run (top), it traverses the same chain a second time and updates each offset to point to the end of the run (bottom). This optimization is called path compression.

The CREATE operation has a simple implementation: to create a new run when a pixel becomes opaque the algorithm just sets the pixel's offset to one. Furthermore, the UNION operation is free. If there is already a run of opaque pixels to the left of the new run then its final offset already points into the new run. Similarly, the offset in the new run points to the next pixel; if that pixel is opaque then the new run automatically becomes part of the following run.

The algorithm uses the FIND operation to find the end of a run starting from any pixel within the run. This operation can be accomplished by following the chain of offsets up to the first non-opaque pixel, which is the first pixel with an offset of zero. However this implementation is inefficient because the CREATE and UNION operations only produce offsets equal to one, resulting in long chains. There is a very effective optimization called *path compression* that greatly reduces the average length of the chains [Cormen et al. 1990]. The optimization can be implemented as follows: after the algorithm traverses a chain of offsets to find the end of a run, it traverses the same chain a second time and updates each of the offsets to point to the end of the run (Figure 4.5). This optimization is effective because it shortens paths that are later reused.

Without path compression the asymptotic complexity of updating and traversing the intermediate image is $O(m^2)$ operations per scanline, where $m$ is the total number of calls to CREATE, UNION and FIND. However, with path compression the complexity drops to

$O(n_f \log_{(1+n_f/n_c)} n_c)$ where $n_c$ is the number of CREATE operations and $n_f$ is the number of FIND operations [Cormen et al. 1990]. Furthermore, our performance measurements in Chapter 5 show that in a real implementation the cost of traversing opaque pixel runs drops to a negligible fraction of overall execution time.

An additional optimization, called *union by rank*, can be used to further reduce the asymptotic complexity (although for common problem sizes the real execution time is slightly worse). This optimization applies to the UNION operation: when two sets are merged the smaller set should be made to point to the larger set. The size (or rank) of the set is defined to be the number of levels in the tree representing the set. This optimization tends to keep the trees balanced, thereby reducing the maximum number of steps from a leaf node to the root. To add the union-by-rank optimization to our algorithm the rank of each tree must be stored in the pixel located at the root of the tree and the implementation of UNION must be changed.

The combination of path compression and union-by-rank results in an asymptotic complexity of $O(m\ \alpha(m, n_c))$ where $\alpha(m, n_c)$ is the inverse of Ackermann's function—a function that grows so slowly it equals at most four for all reasonable problem sizes. However, the additional cost of the UNION operation results in a higher constant in the cost polynomial for the algorithm. With both optimizations our overall rendering algorithm is 5-15% slower than with path compression alone. Furthermore, for volumes with 256 or 512 voxels per side, a $\log$ term is no worse than $\alpha(m, n_c)$ in the complexity expression: both grow extremely slowly for practical problem sizes. We therefore use only path compression in our current implementation.

The dynamic screen data structure proposed by Reynolds et al. [1987] (described in Section 2.2.2) has the same function as our array of opaque pixel links but uses linked lists with explicit pointers to represent runs of image pixels. In their implementation each image scanline consists of a linked list of opaque and non-opaque runs, initially containing a single non-opaque run. Their rendering algorithm simultaneously scans the linked lists for a run-length encoded voxel scanline and the corresponding image scanline to avoid compositing transparent or occluded voxels, just as in our algorithm.

However, the linked list must be accessed sequentially unlike our array of offsets which can be indexed in random order. As a result their algorithm must traverse every run in an

image scanline even if most of the voxels projecting onto the scanline are transparent. Furthermore, updating the linked list requires using a dynamic memory allocator. The cost of maintaining a pool of free memory blocks adds overhead, and the linked lists are not likely to have good spatial locality. Thus their algorithm may have more computational and memory overhead than ours.

Our algorithm does have somewhat more overhead than early ray termination in a ray caster since our algorithm must traverse the run-length encoded data structures even in occluded regions of the volume. Furthermore, the effectiveness of the opaque pixel links depends on coherence in the opaque regions of the image. The runs of opaque pixels are typically large, so many pixels can be skipped at once and the number of pixels that are individually examined is relatively small. In Chapter 5 we present quantitative results demonstrating that our algorithm compares favorably to a ray caster with early ray termination.

### 4.1.4 Resampling the Volume

The rendering algorithm must resample each voxel slice when it translates the slice into sheared object space. The current implementation uses a bilinear interpolation filter, so two scanlines of voxel data are required to produce one resampled voxel scanline. The rendering algorithm uses a gather-type convolution (backward projection), which requires that the algorithm traverse and decode the two input scanlines simultaneously. As a result, each voxel scanline must be traversed twice (once for each of the resampled scanlines to which it contributes).

An alternative implementation is to use scatter-type convolution (forward projection): the algorithm traverses each input voxel scanline once and splats its contribution into two resampled voxel scanlines. However the partially-computed resampled voxels must be stored in a temporary buffer until they have been completely computed to avoid reordering the compositing and reconstruction filtering steps. Moreover, the temporary buffer must be run-length encoded to retain the benefit of the other coherence data structures.

During resampling the rendering algorithm skips over regions where both input scanlines contain transparent runs. Furthermore, the algorithm uses the opaque pixel links to skip over occluded voxels, so only voxels that are non-transparent and non-occluded are

actually resampled. The cost of resampling can be further reduced since the resampling weights are the same for every voxel in the slice, allowing the algorithm to compute the weights only once per slice.

We will discuss filter quality issues in Section 5.1.2.

### 4.1.5 Warping the Intermediate Image

The image computed by the compositing phase of the algorithm is an oblique projection of the volume. In general, an arbitrary affine warp must be applied to the intermediate image to produce the correct final image. The second factor of the shear-warp factorization determines the necessary transformation (Section 3.2).

Implementation of the 2D image warp is straightforward (see the surveys by Heckbert [1986] and Wolberg [1990]). The current implementation uses a one-pass gather (backward projection) algorithm with a bilinear filter.

### 4.1.6 Opacity Correction

In our algorithm the resolution of the volume determines the sampling rate that is used to evaluate the volumetric compositing equation (Equation 1.4). As a result the distance between samples along a viewing ray is constant in object space, but varies in image space depending on the current view transformation. This fact leads to a problem. Recall from Section 1.1.2 that the opacity at a sample point is defined as:

$$\alpha_i = 1 - e^{-\phi_i \Delta x}$$

where $\Delta x$ is the sample spacing in image space. Since the sample spacing changes with viewpoint so must the opacities, and thus the precomputed opacities stored in each voxel must be corrected for a particular viewing direction.

Figure 4.6 illustrates the problem. Consider a cube-shaped volume filled with identical low-opacity voxels. If a viewer looks straight at the center of one face of the volume the distance between sample points in image space along the viewing ray equals the length of one side of a voxel. If the viewer rotates the volume 45 degrees around its center axis then

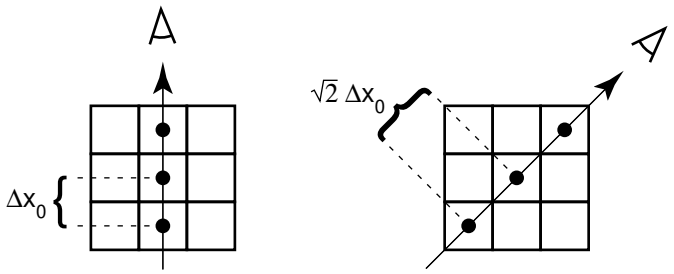



Figure 4.6: The spacing between samples on a viewing ray depends on the viewing direction, so voxel opacities must be corrected to account for the sample spacing.

the number of samples stays the same but the distance between sample points in image space increases (unlike with most ray casters, which maintain the same sample spacing regardless of the viewing direction). The attenuation along the ray should also increase, hence the need for opacity correction. Other object-order volume rendering algorithms must also correct opacities for the viewing angle [Laur & Hanrahan 1991].

Let $\Delta x_0$ be the width of a voxel and assume that the opacity computed by the opacity transfer function and stored with a voxel ($\alpha_{\text{stored}}$) uses $\Delta x_0$ as the sample spacing:

$$\alpha_{\text{stored}} = 1 - e^{-\phi \Delta x_0}$$

Then for some other sample spacing $\Delta x$ the corrected opacity can be computed as follows:

$$\begin{aligned} \alpha_{\text{corrected}} &= 1 - e^{-\phi \Delta x} \\ &= 1 - \left[e^{-\phi \Delta x_0}\right]^{\frac{\Delta x}{\Delta x_0}} \\ &= 1 - \left[1 - \alpha_{\text{stored}}\right]^{\frac{\Delta x}{\Delta x_0}} \end{aligned}$$

The corrected opacity is a function of the stored opacity and the sample spacing ratio $\Delta x / \Delta x_0$, but the function is the same for every voxel. Figure 4.7 shows a plot of the correction function for several values of the sample spacing ratio. Without opacity correction a cube-shaped voxel appears 30% more transparent than it should when viewed from a 45 degree angle, a difference that is visually significant.

The rendering algorithm performs opacity correction on each voxel just before resampling the voxel. The correction can be implemented efficiently with a precomputed lookup

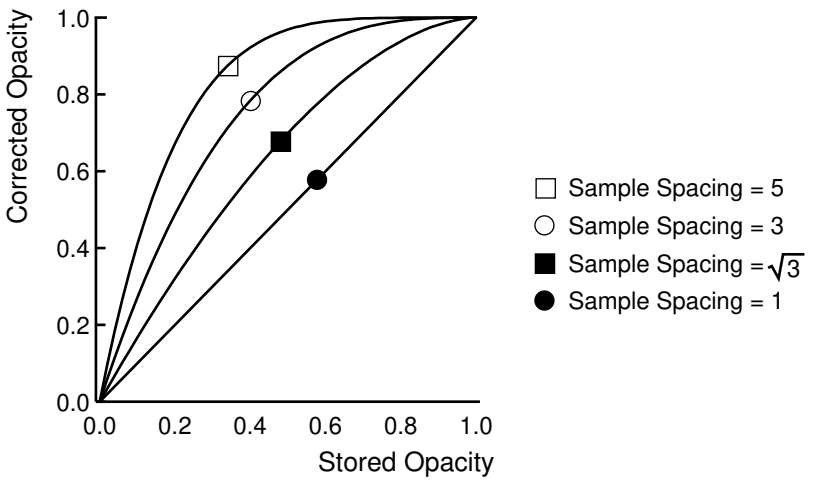



Figure 4.7: Plot of the opacity correction function. Unless the volume has been scaled non-uniformly, the sample spacing lies between 1 and $\sqrt{3}$ (corresponding to viewing angles of 0 and 45 degrees in each direction relative to the voxel slices). Plots for larger sample spacings are shown only to emphasize the shape of the curve.

table that maps uncorrected opacities to corrected opacities. The table must be recomputed whenever the viewpoint changes, but the same table can be used for every voxel.

### 4.1.7 Implementation of the Parallel Projection Algorithm

Figure 4.8 is a pseudo-code description of the shear-warp volume rendering algorithm for parallel projections. The *RenderVolume* routine is the highest-level routine. It first calls *Factor* to compute the shear and warp factors of the view transformation matrix, and *InitOpacityTable* to precompute the lookup table for opacity correction. The call to *Clear* initializes the intermediate image, *TmpImage*. Then the loop on line 5 composites each slice of the volume into the intermediate image in front-to-back order by calling *CompositeSlice*. Finally, *RenderVolume* warps and displays the image.

*CompositeSlice* implements the core of the rendering algorithm. It begins by initializing *voxel_ptr*, an array containing pointers to two adjacent scanlines of run-length encoded voxel data. These two scanlines will be filtered to produce one scanline of resampled voxels (aligned with the intermediate image grid). The routine *FindVoxelRuns* uses the voxel slice pointer array in the run-length encoded volume to find the first two voxel scanlines in

```
1   procedure RenderVolume()
        Factor(ViewMatrix);
        InitOpacityTable();
        Clear(TmpImage);
5       for (k = 1 to VolumeSize[Z])
            CompositeSlice(k);
        Image = Warp(TmpImage);
        Display(Image);
    end
10  procedure CompositeSlice(k)
        voxel_ptr[1..2] = FindVoxelRuns(k, 2);
        for (v = 1 to VolumeSize[Y])
            (im_x, im_y) = (XTranslate(k), YTranslate(k) + v);
            while (not EndOfScanline(voxel_ptr))
15              if (IsOpaque(TmpImage[im_x][im_y]))
                    SkipPixelRun(im_x, im_y, voxel_ptr);
                else
                    run_length = min(voxel_ptr[1].run_length, voxel_ptr[2].run_length);
                    if (IsTransparent(voxel_ptr[1]) and IsTransparent(voxel_ptr[2]))
20                      SkipVoxelRun(im_x, voxel_ptr, run_length);
                    else
                        ProcessVoxelRun(im_x, im_y, voxel_ptr, run_length);
                    endif
                endif
25          end
        end
    end
    procedure SkipVoxelRun(im_x, voxel_ptr, run_length)
        AdvanceVoxelPtr(voxel_ptr, run_length);
30      im_x = im_x + run_length;
    end
```

Figure 4.8: Pseudo-code for the parallel projection algorithm *(continued on next page)*.

```
    procedure SkipPixelRun(im_x, im_y, voxel_ptr)
        run_length = 0;
        while (TmpImage[im_x + run_length][im_y].offset > 0)
35          run_length = run_length + TmpImage[im_x + run_length][im_y].offset;
        end
        path_length = 0;
        while (TmpImage[im_x + path_length][im_y].offset > 0)
            offset = TmpImage[im_x + path_length].offset;
40          TmpImage[im_x + path_length].offset = run_length - path_length;
            path_length = path_length + offset;
        end
        AdvanceVoxelPtr(voxel_ptr, run_length);
        im_x = im_x + run_length;
45  end

    procedure ProcessVoxelRun(im_x, im_y, voxel_ptr, run_length)
        for (i = 1 to run_length)
            if (IsOpaque(TmpImage[im_x][im_y]))
                return;
50          (color, alpha) = Sample(voxel_ptr);
            CompositeVoxel(im_x, im_y, color, alpha);
            AdvanceVoxelPtr(voxel_ptr, 1);
            im_x = im_x + 1;
        end
55  end

    procedure CompositeVoxel(im_x, im_y, color, alpha)
        if (not IsTransparent(alpha))
            TmpImage[im_x][im_y] = TmpImage[im_x][im_y] over (color, alpha);
            if (IsOpaque(TmpImage[im_x][im_y]))
60              TmpImage[im_x][im_y].offset = 1;
        endif
    end
```

Figure 4.8 (cont.): Pseudo-code for the parallel projection algorithm.

a particular slice of the volume.

The outer loop iterates over the scanlines of the voxel slice. Line 13 computes the coordinates of the top-left intermediate image pixel that the current pair of voxel scanlines projects onto. Then the inner loop resamples and composites runs of voxel data onto the intermediate-image scanline until it reaches the ends of the voxel scanlines. The body of the loop contains three cases: skip a run of opaque pixels, skip a run of transparent voxels, or do useful work. If the current intermediate image pixel is opaque then the loop calls *SkipPixelRun* to skip over a run of opaque pixels and the corresponding voxels. Otherwise, the algorithm computes *run_length*, the number of voxels until the end of the next voxel run in either of the two voxel scanlines. If both of the current runs are transparent then *CompositeSlice* calls *SkipVoxelRun* to skip over the voxels; otherwise, it calls *ProcessVoxels* to resample and composite the voxels into the intermediate image.

Once the inner loop terminates, the pointers in *voxel_ptr* have each advanced by one voxel scanline and thus they are ready for computing the next intermediate image scanline. The outer loop continues processing scanlines until all of the voxels runs in the slice have been traversed.

*SkipVoxelRun* is the routine that skips over a run of transparent voxels. It simply calls *AdvanceVoxelPtr* to advance the pointers for the two voxel scanlines and then advances the current position in the intermediate image scanline.

*SkipPixelRun* is the routine that skips over a run of opaque pixels. The loop on line 34 follows the opaque pixel offsets until the next non-opaque pixel (which must have an offset equal to zero). The loop on line 38 performs path compression as described in Section 4.1.3. Finally, *SkipPixelRun* advances the voxel pointers and the current pixel index (*im_x*) by the length of the opaque pixel run.

*ProcessVoxelRun* is the routine that resamples and composites a run of voxels. The loop iterates over the sample points in the resampled voxel scanline. First it checks if the sample point projects to an opaque pixel, in which case the beginning of an opaque pixel run has been reached and the routine returns to *CompositeSlice*. Next, *ProcessVoxelRun* computes the value of the volume at a sample point by calling *Sample*. This routine loads four voxels (two from each of the voxel scanlines), performs opacity correction and shading, multiplies the colors and opacities by the precomputed resampling filter weights, and returns

the resampled voxel's color and opacity. *Sample* uses the information in the run-length encoded voxel data structure to avoid processing any of the four voxels that may be transparent. Next, the *CompositeVoxel* routine composites the resampled voxel into the intermediate image. Finally, the loop advances to the next voxel sample point.

*CompositeVoxel* simply uses the "over" operator to composite a voxel into the intermediate image. It only composites a voxel if its resampled opacity exceeds the opacity threshold that was used to create the run-length encoded volume. The procedure also creates a new opaque pixel run if the intermediate image pixel becomes opaque.

The algorithm we have presented in this section uses the shear-warp factorization to take advantage of optimizations from both image-order and object-order algorithms, thereby benefiting from the performance advantages of both. In the next section we will generalize the algorithm to support perspective projections.

## 4.2 Perspective Projection Rendering Algorithm

Most volume rendering research has focused on parallel projection algorithms. However, perspective projections provide additional cues for resolving depth ambiguities [Novins et al. 1990] and are essential to correctly compute occlusions in such applications as radiation treatment planning, which requires computing an accurate image from the viewpoint of a radiation source [Levoy et al. 1990]. Perspective projections present a problem because the viewing rays diverge so it is difficult to sample the volume correctly. Two types of solutions have been proposed for perspective volume rendering using ray-casters: as the distance along a ray increases the ray can be split into multiple rays [Novins et al. 1990], or each sample point can sample a larger portion of the volume using a mip-map [Levoy & Whitaker 1990, Sakas & Gerth 1991]. A mip-map is a hierarchical data structure that contains prefiltered copies of sampled data (the volume in this case) at several resolutions, so it is possible to compute one sample covering an arbitrarily large portion of the volume in constant time. The application of a mip-map to 3D arrays is a straightforward generalization of the 2D mip-maps introduced for texture filtering applications by Williams [1983]. The object-order splatting algorithm can also handle perspective, but the resampling filter

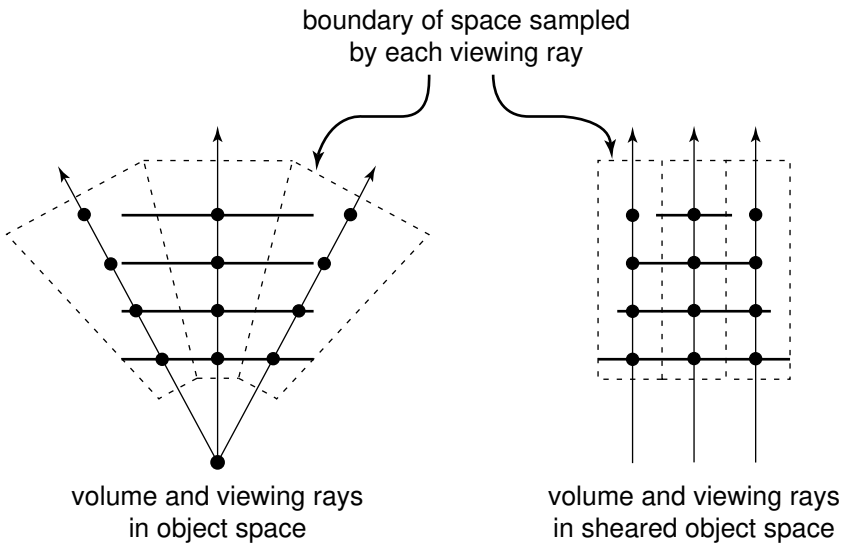



Figure 4.9: Sampling the volume in the perspective projection algorithm: The left diagram shows the sample locations and filter footprint boundaries in object space. The right diagram shows the sample locations in sheared object space.

changes for every voxel [Westover 1990]. This section describes a new algorithm for perspective volume rendering based on the shear-warp factorization.

### 4.2.1 Overview of the Perspective Projection Algorithm

The shear-warp factorization provides a simple and efficient solution to the sampling problem for perspective projections, as we saw in Section 3.1. The rendering algorithm transforms each slice of the volume to sheared object space by a translation and a uniform scale, and the algorithm then resamples and composites the slices together. These steps are equivalent to a ray-casting algorithm in which rays are cast to uniformly sample the first slice of volume data, and as each ray hits subsequent (more distant) slices a larger portion of the slice is sampled (Figure 4.9). The key point is that within each slice the sampling rate is uniform (Property 2 of the factorization), so the filter is relatively easy to implement.

The perspective algorithm is nearly identical to the parallel projection algorithm. The only substantial difference is that each voxel must be scaled as well as translated during resampling. We always choose a factorization of the viewing transformation in which the slice closest to the viewer is scaled by a factor of one and more distant slices are decimated,

so no slice is ever enlarged.

### 4.2.2 Resampling the Volume

To resample the voxel slices we use a box reconstruction filter and a box low-pass filter, an appropriate combination for both decimation and unity scaling. In the case of unity scaling the two filter widths are identical and their convolution reduces to the bilinear interpolation filter used in the parallel projection algorithm.

The implementation of the resampling loop differs from the parallel projection algorithm because the filter footprint may cover more than two voxel scanlines, so a variable number of voxel scanlines must be traversed to produce a given intermediate scanline. Furthermore, the voxel scanlines cannot be traversed at the same rate as the image scanlines and the resampling weights must be recomputed for each sample point.

### 4.2.3 Opacity Correction

Opacity correction is more difficult for perspective projections than for parallel projections because not every voxel requires the same correction and it is expensive to compute the corrections individually (an exponentiation would be required for each voxel). However, two observations allow the calculation to be simplified. First, every sample point along a viewing ray requires the same correction since the sample spacing is constant along the ray (Figure 4.9). Second, the sample spacing for adjacent rays is nearly constant. It is therefore possible to use a set of opacity correction lookup tables precomputed for a subset of the rays and to interpolate between tables for the remaining rays.

An algorithm to correct the voxel opacities works as follows. First, we precompute several opacity correction tables using the formula derived in Section 4.1.6 with different values of $\Delta x$ (the sample spacing along the viewing ray). Each table maps the opacity stored in a voxel to an opacity corrected for the particular value of $\Delta x$. We organize the tables as a 2D array indexed by the sample spacing and the uncorrected opacity:

$$\alpha_{\mathrm{corrected}} = \mathrm{OpacityTable}[\Delta x][\alpha_{\mathrm{stored}}]$$

Once the viewing transformation has been fixed but before rendering begins our algorithm calculates $\Delta x$ for the viewing ray emanating from each intermediate image pixel and stores the value with the pixel. The algorithm computes $\Delta x$ exactly for a subset of the pixels, and the values for the remaining pixels are interpolated from exact values at nearby pixels. Then during rendering when a voxel is about to be projected onto an intermediate image pixel the voxel sampling routine (*Sample*) uses the value of $\Delta x$ from the pixel to select a precomputed opacity correction table. Suppose $\Delta x$ is in between two of the values used for the precomputed tables, $\Delta x_1$ and $\Delta x_2$. Then the algorithm computes the corrected opacity ($\alpha_{\text{corrected}}$) from the opacity stored with the voxel ($\alpha_{\text{stored}}$) and the opacity correction tables by indexing the two tables with the stored opacity and linearly interpolating the results:

$$\begin{aligned} \alpha_1 &= \text{OpacityTable}[\Delta x_1][\alpha_{\text{stored}}] \\ \alpha_2 &= \text{OpacityTable}[\Delta x_2][\alpha_{\text{stored}}] \\ W &= \frac{\Delta x_2 - \Delta x}{\Delta x_2 - \Delta x_1} \\ \alpha_{\text{corrected}} &= \alpha_1 \cdot W + \alpha_2 \cdot (1 - W) \end{aligned}$$

This algorithm is fast and allows us to trade off memory (to store additional tables) for accuracy if necessary.

### 4.2.4 Implementation of the Perspective Projection Algorithm

Figure 4.10 is a pseudo-code description of the shear-warp volume rendering algorithm for perspective projections. The figure only includes routines that differ from the parallel projection algorithm.

The perspective projection algorithm uses a modified version of the *CompositeSlice* procedure that can decimate voxel slices before compositing them. The *voxel_ptr* array contains pointers to all of the voxel scanlines that contribute to a single intermediate image scanline. The maximum number of scanlines that can contribute equals the size of the filter footprint (*footprint*). In the parallel projection algorithm the size of the footprint is always two.

The outer loop of *CompositeSlice* iterates over the intermediate image scanlines onto

```
1   procedure CompositeSlice(k)
        footprint = 1 + ceiling(1 / ScaleFactor(k));
        voxel_ptr[1..footprint] = FindVoxelRuns(k, footprint);
        for (v = 1 to VolumeSize[Y] × ScaleFactor(k))
5           (im_x, im_y) = (XTranslate(k), YTranslate(k) + v);
            while (not EndOfScanline(voxel_ptr))
                if (IsOpaque(TmpImage[im_x][im_y]))
                    SkipPixelRun(im_x, im_y, voxel_ptr);
                else
10                  (color, alpha) = DecimateArea(im_x, im_y, voxel_ptr, footprint);
                    CompositeVoxel(im_x, im_y, color, alpha);
                    im_x = im_x + 1;
                endif
            end
15          AdvanceVoxelPtr(voxel_ptr, (footprint–2) × VolumeSize[X]);
        end
    end

    procedure DecimateArea(im_x, im_y, voxel_ptr, footprint)
        (color, alpha) = (0, 0);
20      for (j = 1 to footprint)
            voxels_left = footprint;
            while (voxels_left > 0)
                run_length = min(voxel_ptr[j].run_length, voxels_left);
                if (IsTransparent(voxel_ptr[j]))
25                  AdvanceVoxelPtr(voxel_ptr[j], run_length);
                else
                    AccumulateVoxelRun(im_x, im_y, voxel_ptr[j], run_length, color, alpha);
                voxels_left = voxels_left – run_length;
            end
30      end
        return(color, alpha);
    end
```

Figure 4.10: Pseudo-code for the perspective projection algorithm. This figure only includes changes to the parallel projection algorithm in Figure 4.8 *(continued on next page)*.

```
    procedure AccumulateVoxelRun(im_x, im_y, voxel_ptr, run_length, color, alpha)
        for (c = 1 to run_length)
35          (vox_color, vox_alpha) = LoadVoxel(voxel_ptr);
            weight = ComputeWeight(im_x, im_y, voxel_ptr);
            (color, alpha) = (color, alpha) + (vox_color, vox_alpha) × weight;
            AdvanceVoxelPtr(voxel_ptr, 1);
        end
40  end
```

Figure 4.10 (cont.): Pseudo-code for the perspective projection algorithm.

which the voxel slice projects. The inner loop (line 6) iterates over intermediate image pixels until the ends of the voxel scanlines are reached. The algorithm skips runs of opaque pixels using *SkipPixelRun* just like the parallel projection algorithm. When the loop finds a non-opaque pixel it calls *DecimateArea* to filter the voxels that project onto the pixel. It then composites the resampled voxel value into the pixel and proceeds to the next pixel. When the loop terminates, the call to *AdvanceVoxelPtr* advances all of the voxel pointers to the beginnings of the scanlines required for the next intermediate image scanline. If the reciprocal of the scale factor is not an integer then the filter footprint may decrease by one for some iterations of the loops, but the details are not shown here.

*DecimateArea* loops over the set of voxels that project onto one intermediate image pixel and sums their contributions. The outer loop iterates over voxel scanlines. For each scanline, the inner loop skips over transparent voxel runs and calls *AccumulateVoxelRun* to composite the non-transparent runs. *AccumulateVoxelRun* steps through each voxel in the run, first calling *LoadVoxel* to perform opacity correction and shading, then calling *ComputeWeight* to calculate the filter weight for a particular voxel and image pixel, and finally adding the contribution of the voxel to the color and opacity accumulators.

Compared to the parallel projection algorithm the perspective projection algorithm has somewhat more control overhead and spends more time calculating resampling weights, but nevertheless the algorithm is efficient compared to other methods. The properties of the shear-warp factorization still allow the algorithm to traverse the volume and the intermediate image scanline-by-scanline while performing work only for visible, non-transparent

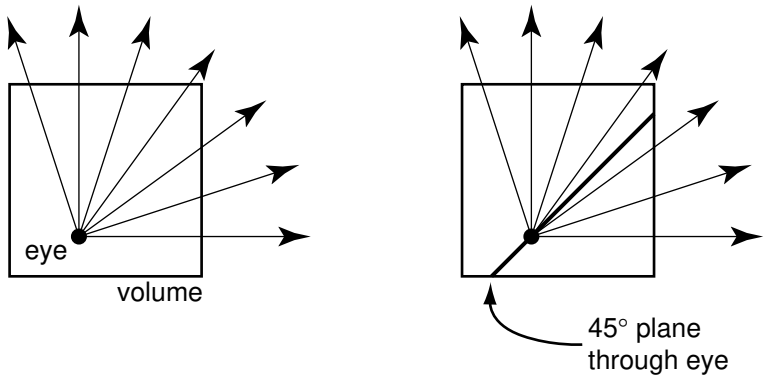



Figure 4.11: For the situation shown on the left there is no single principal viewing axis that is consistent with all viewing rays. The volume must be split along a plane through the eyepoint (the center of projection) at a 45 degree angle to the voxel slices so the subvolumes can be rendered separately, as shown on the right.

voxels, and the resampling filter can be implemented efficiently despite the diverging ray problem.

### 4.2.5 Limitations of the Perspective Projection Algorithm

A potential problem with the perspective projection algorithm is that for some viewing transformations there is no single principal viewing axis that is consistent with all viewing rays. Figure 4.11 shows an example. While it is possible to use the rendering algorithm even if some of the viewing rays intersect the voxel slices at greater than a 45 degree angle, as the rays approach 90 degrees image quality degrades severely. The problem is most likely to occur during a fly-through in which the viewpoint is inside the volume and the field of view approaches 90 degrees or greater.

The solution to the problem is to subdivide the volume by splitting it along planes oriented at 45 degree angles relative to the slices. At most six subvolumes are necessary to render a field of view covering the entire sphere of possible directions (i.e. the volume can be decomposed into six pyramids with their apexes located at the eyepoint). Each subvolume must be rendered separately with the algorithm described above and then the subimages must be tiled to form the complete image. To prevent discontinuities at the seams between tiles each subvolume should contain a few voxels of overlap with the adjacent subvolumes.

The subimages will then overlap slightly, and the overlapping pixels can be averaged to produce a smooth transition. This modification to the rendering algorithm introduces some complexity and has not yet been implemented.

A second problem with the perspective rendering algorithm is that visibility is not correctly computed. Visibility along each viewing ray must be determined before averaging values from the rays together to compute a pixel value, but our algorithm interchanges these two operations. The response $R(i,j)$ of a pinhole-camera sensor spanning a pixel on the image plane is proportional to the incident radiance $L$ integrated over the solid angle $\Omega$ subtended by the pixel:

$$R(i,j) = \int_{\Omega(i,j)} L(i,j,\vec{\omega}) \cos\theta \, d\omega$$

(where $\theta$ is the angle between $\vec{\omega}$ and the image plane normal). The radiance in the integrand can be found by evaluating the volume rendering integral:

$$R(i,j) = \int_{\Omega(i,j)} \int_x e^{-\int \phi_t(x')dx'} \epsilon(x) \cos\theta \, dx \, d\omega$$

Thus the correct way to compute the response is to first integrate along each viewing ray and then integrate over the solid angle. The perspective shear-warp algorithm filters the voxel slices before compositing them together, which is equivalent to integrating over the solid angle before integrating along the rays:

$$R'(i,j) = \int_x e^{-\int \phi_t(x')dx'} \int_{\Omega(i,j)} \epsilon(x) \cos\theta \, d\omega \, dx$$

Thus the order of the integrals has been incorrectly interchanged.

Figure 4.12 illustrates the problem: objects A and B should fully occlude object C. However, the rendering algorithm down-samples the voxel slices before compositing them together. The area of the voxel slice containing object A and its neighborhood are averaged into a single, partially opaque sample. Similarly, object B is averaged into a partially opaque sample. When the two samples are composited together the result is not fully opaque, so object C incorrectly appears partially visible.

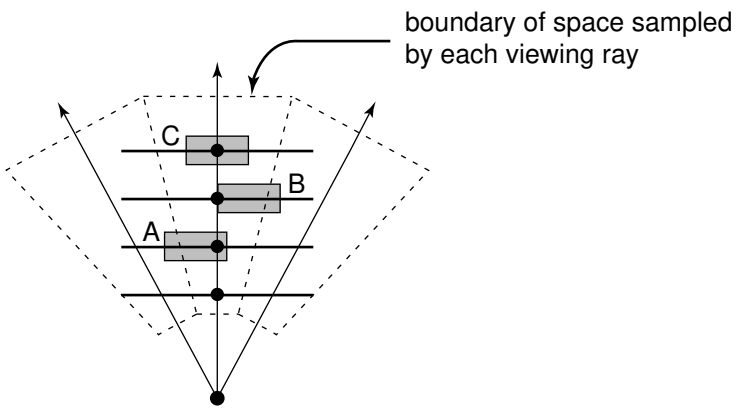



Figure 4.12: Visibility errors can occur in the perspective rendering algorithm because it filters the volume before compositing. In this case object C should be occluded by objects A and B (assuming A and B are fully opaque). However, the rendering algorithm averages objects A and B with the surrounding empty space before compositing the voxel slices together so object C shows through.

This problem can occur in both parallel and perspective renderings, but its likelihood is exacerbated in the perspective case by the divergence of the rays. The problem is fundamental to the shear-warp volume rendering algorithm and any other algorithm that decimates the volume before compositing, such as algorithms employing mip-mapping [Levoy & Whitaker 1990, Sakas & Gerth 1991] and splatting algorithms employing a sheet cache [Westover 1990]. In contrast, the algorithm proposed by Novins et al. [1990] computes visibility more accurately because it casts rays that split into multiple rays as the distance from the viewer increases, so the filter kernel size remains constant along each ray.

In practice the artifacts caused by this problem are small except in cases of very severe perspective distortion.

## 4.3 Fast Classification Algorithm

The run-length encoded data structure used by the previous two algorithms must be recomputed whenever the opacity transfer function changes. This preprocessing time is insignificant if the user wishes to generate many images from a single classified volume, but if the user wishes to interactively adjust the transfer function then the preprocessing step is unacceptably slow. In this section we present a third variation of the shear-warp algorithm that

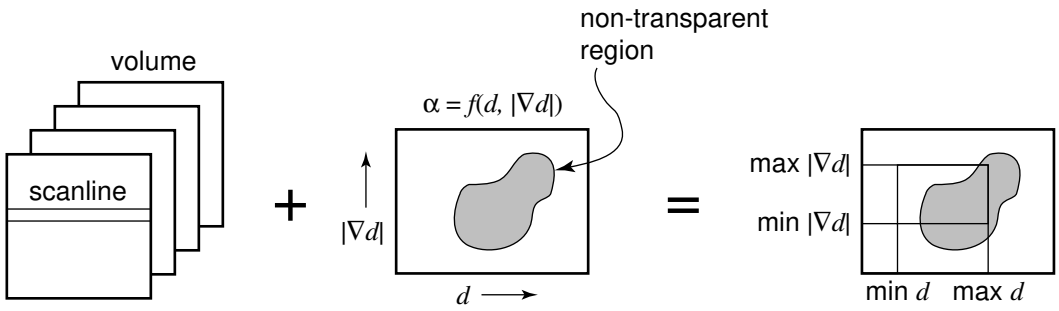



Figure 4.13: Goal of the fast classification algorithm: Given a scanline of voxels, shown on the left, we must determine which voxels are transparent. The classification function $f(d, |\nabla d|)$ shown in the middle of the diagram partitions a two-dimensional feature space into transparent and non-transparent regions. The voxels in the scanline are contained in some bounding box within the feature space, shown on the right. If the bounding box and the non-transparent regions do not intersect then the voxels must be transparent.

evaluates the opacity transfer function during rendering and is only moderately slower than the previous algorithms.

### 4.3.1 Overview of the Fast Classification Algorithm

A run-length encoding of the volume based upon opacity is not an appropriate data structure when the opacity transfer function is not fixed. Instead we apply the algorithms described in Sections 4.1 and 4.2 to unencoded voxel scanlines, but with a new method to determine which portions of each scanline are non-transparent.

We allow the opacity transfer function to be any scalar function of a multi-dimensional scalar domain:

$$\alpha = f(p, q, ...)$$

For example, the opacity might be a function of the scalar field and its gradient magnitude [Levoy 1988]:

$$\alpha = f(d, |\nabla d|)$$

The function $f$ essentially partitions a multi-dimensional feature space into transparent and non-transparent regions, and our goal is to decide quickly which portions of a given scanline contain voxels in the non-transparent regions of the feature space (Figure 4.13).

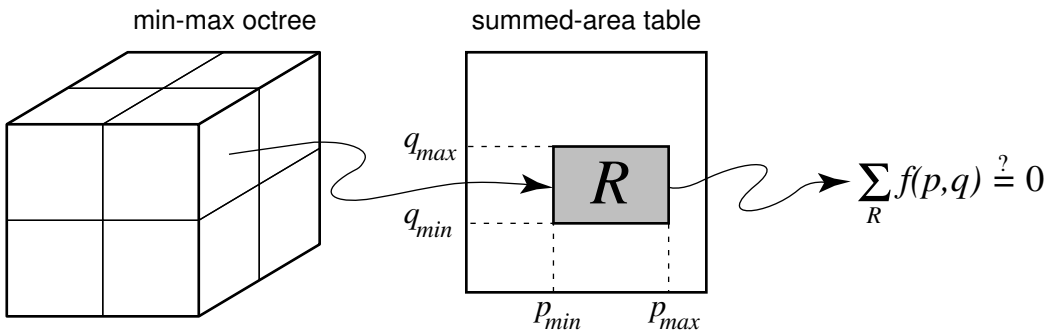



Figure 4.14: Data structures for the fast classification algorithm: A min-max octree is used to determine the range of the parameters $p, q$ of the opacity transfer function $f(p, q)$ in a subcube of the volume. A summed-area table is used to integrate $f$ over that range of $p, q$. If the integral is zero then the subcube contains only transparent voxels.

We solve this problem with the following recursive algorithm which takes advantage of coherence in both the opacity transfer function and the volume data:

**Step 1:** For some block of the volume that contains the current voxel scanline, find the extrema of the parameters of the opacity transfer function ($\min(p)$, $\max(p)$, $\min(q)$, $\max(q)$, ...). These extrema bound a rectangular region of the feature space defined by the domain of $f$.

**Step 2:** Determine if the region is transparent, i.e. $f$ evaluated for all parameter points in the region yields only transparent opacities. If so, then discard the scanline since it must be transparent.

**Step 3:** Subdivide the scanline and repeat this algorithm recursively. If the size of the current scanline portion is below a threshold then render it instead of subdividing.

This algorithm relies on two data structures for efficiency (Figure 4.14). First, Step 1 uses a precomputed min-max octree [Wilhelms & Van Gelder 1990, Wilhelms & Van Gelder 1992]. Each octree node contains the extrema of the parameter values for a subcube of the volume. Second, to implement Step 2 of the algorithm we need to integrate the function $f$ over the region of the feature space found in Step 1. Assume for now that the opacity threshold is zero, so only voxels with opacity equal to zero are transparent. If the integral

of $f$ over the region of feature space covered by the subcube is zero then all voxels in the subcube must be transparent. This integration can be performed in constant time using a multi-dimensional summed-area table [Crow 1984, Glassner 1990a]. The voxels themselves are stored in a third data structure, a simple 3D array.

The overall algorithm for rendering unclassified data sets proceeds as follows. When a volume is first acquired or loaded into memory the algorithm precomputes the min-max octree, which is independent of the opacity transfer function and the viewing parameters. Next, just before rendering begins the algorithm uses the opacity transfer function to compute the summed-area table. This computation is inexpensive provided that the domain of the opacity transfer function is not too large. We then use either the parallel projection or the perspective projection rendering algorithm to render voxels from an unencoded 3D voxel array. We traverse the array scanline-by-scanline as usual. But, for each scanline we use the octree and the summed-area table to determine which portions of the scanline are non-transparent. The algorithm classifies voxels in the non-transparent portions of the scanline using a lookup table and then renders them as in the previous algorithms. The algorithm skips opaque regions of the image just as before. Voxels that are either transparent or occluded are never classified, which reduces the amount of computation.

The main advantage of this algorithm is that the min-max octree can be precomputed before fixing the classification function. Along similar lines, two algorithms that use a spatial data structure for generating isosurfaces from volume data have been proposed: Wilhelms & Van Gelder [1992] use a min-max octree, and Stander & Hart [1994] use an octree containing a Lipschitz constant which bounds the variation of the scalar value. Both of these data structures can be precomputed before the user selects an isosurface threshold. However neither algorithm is appropriate for more general classification functions, which is the advantage of our algorithm.

### 4.3.2 The Min-Max Octree

An octree is a recursive subdivision of space organized as a tree (Figure 4.15). Each node of an octree represents a cubic region of space and has eight children. The children represent a subdivision of the node's space into eight smaller octants. A min-max octree is an octree that

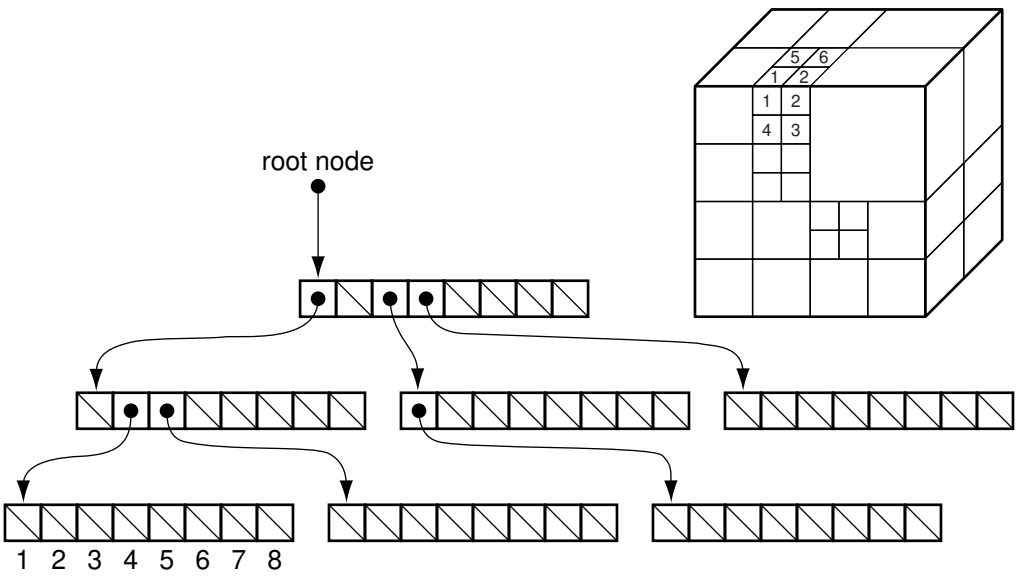



Figure 4.15: Diagram of a min-max octree. Each node contains the minimum and maximum values of each argument to the opacity transfer function over a subcube of the volume.

is used to store the minimum and maximum values of some parameter associated with the region of space represented by each node. The root node contains a minimum and maximum value of the parameter for the entire volume. Each additional level of the tree contains min-max values for smaller regions of the volume, resulting in a more detailed description of parameter variations inside the volume. The smallest octree node size can be chosen to trade off the maximum level of detail versus the amount of time it takes to traverse the octree during rendering. At some point the benefit of increased detail no longer outweighs the time required to traverse the octree nodes while extracting the information.

Each node in the min-max octree contains the following information:

- The minimum and maximum value of each classification parameter over all of the voxels represented by the octree node.

- Storage for caching a result after classifying the octree node. This field indicates whether the voxels represented by the node are all transparent, all non-transparent, or a combination. In the latter case the children of the octree node contain more detailed information.

- A pointer to the children of the octree node. All eight children are stored in a contiguous block of memory so each octree node contains only one pointer that points to the first of its children (as shown in Figure 4.15). Some octree nodes have no children, in which case the pointer is set to NIL. Nodes at the lowest level of the octree contain no pointer field at all (to conserve memory).

The octree does not contain the original volume data. The volume is stored as a separate 3D array of voxels.

An alternative data structure for storing the same information is a forest of min-max binary trees with one tree for each voxel scanline. Binary trees appear to be the natural choice given that we used run-length encoding rather than an octree for the fast rendering algorithm described earlier. We decided against this alternative because binary trees would require eight times as much memory for the same number of levels, and three copies of the data structure would be required (one for each principal viewing axis). The required storage can exceed the size of the volume even in practical cases, whereas the min-max octree is always significantly smaller than the volume.

The algorithm for computing the octree operates as follows. First, the algorithm builds a full octree (an octree in which every node has children except for the nodes in the most detailed level; this data structure is also known as a pyramid). The octree is constructed bottom-up: the algorithm computes the base of the octree by looping over the voxels in the volume to compute the min-max data for the most-detailed octree nodes, and then each node at the next higher level is computed by combining the values from its eight children. This process continues until the root node has been computed.

Second, the algorithm prunes octree nodes for which the difference between the minimum and maximum parameter values falls below a user-defined threshold. For example, if the minimum and maximum values are equal for all parameters in a particular octree node then no additional information is provided by the children of the node, so they can be pruned. Even if the minimum and maximum values are not equal but are nearly so it may be advantageous to remove the children: the more nodes in the tree the longer it takes to traverse the tree during rendering. The user-defined thresholds can be adjusted to optimize the tradeoff between level-of-detail and traversal time.

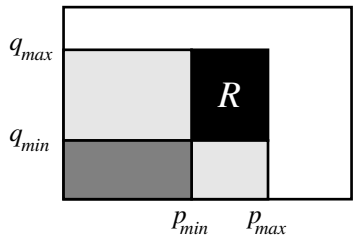



Figure 4.16: Diagram of a 2D summed-area table (after Crow [1984]). The table can be used to integrate a function of two variables over any rectangular region of its domain.

### 4.3.3 The Summed-Area Table

A summed-area table is a convenient data structure for integrating a discrete function over a rectangular region. It was proposed by Crow [1984] for down-sampling texture maps. Figure 4.16 illustrates an example of a 2D summed-area table used to integrate a two-parameter discrete function. Let $f(p, q)$ be a function defined at discrete lattice points on a rectangular domain. The summed-area table for this function is a 2D array of values $S(p, q)$ computed from $f$. Each entry in $S$ is the integral of $f$ over a rectangular region with its bottom-left corner at the origin:

$$S(p, q) = \sum_{p'=0}^{p} \sum_{q'=0}^{q} f(p', q')$$

The integral of $f$ over an arbitrary rectangular region can be found from at most four entries in the table. Consider the region $R$ shown in the figure. $S(p_{\max}, q_{\max})$ gives the integral of $f$ over $R$ plus the region to the left of $R$ and below $R$. The integrals over these two "extra" regions must be subtracted. The integral over the region below $R$ is $S(p_{\max}, q_{\min} - 1)$ and the integral over the region to the left of $R$ is $S(p_{\min} - 1, q_{\max})$. If both of these are subtracted then the integral over the region that is both below and left of $R$ is subtracted twice, so $S(p_{\min} - 1, q_{\min} - 1)$ must be added back. The integral over an arbitrary rectangular region can therefore be computed in constant time using the following formula:

$$\begin{aligned} \sum_{p=p_{\min}}^{p_{\max}} \sum_{q=q_{\min}}^{q_{\max}} f(p, q) \quad = \quad & S(p_{\max}, q_{\max}) - S(p_{\max}, q_{\min} - 1) - \\ & S(p_{\min} - 1, q_{\max}) + S(p_{\min} - 1, q_{\min} - 1) \end{aligned}$$

Similar formulae can be derived for summed-area tables with an arbitrary number of dimensions as described in Glassner [1990a]. An algorithm to compute the table operates as follows: first compute $S(0,0)$, which is equal to $f(0,0)$, and then compute $S(p,q)$ from $f(p,q)$ and entries $S(p-1,q)$, $S(p,q-1)$ and $S(p-1,q-1)$.

In the context of the fast classification volume rendering algorithm, the summed-area table is used to integrate the opacity transfer function over the voxels represented by an octree node. The result of the integral indicates whether or not the voxels are transparent. The algorithm operates as follows. When the user selects a new opacity transfer function $f(p,q,\ldots)$ the algorithm computes a transparency function $f_T(p,q,\ldots)$ by evaluating the opacity transfer function for each $(p,q)$ and comparing the result to the minimum opacity threshold $\alpha_T$:

$$f_T(p,q,\ldots) \equiv \begin{cases} 1 & \text{if } f(p,q,\ldots) > \alpha_T \\ 0 & \text{otherwise} \end{cases}$$

If $f_T$ evaluates to one for a particular point in the parameter space then the function $f$ maps that point to a non-transparent opacity. (Here we have removed the earlier restriction that the opacity threshold must equal zero.)

Next the algorithm computes a summed-area table $S(p,q,\ldots)$ for the transparency function $f_T$. If there are $n$ parameters to the opacity transfer function then the table must have $n$ dimensions. Each table entry contains the number of parameter tuples in a given region of parameter space that map to a non-transparent opacity. Computing the table requires time proportional to the number of entries in the table, which is the size of the domain of $f$.

To determine whether or not the voxels represented by an octree node are transparent, the algorithm uses the summed-area table to integrate the transparency function $f_T$ over the voxels in an octree node. The minimum and maximum values of $p,q,\ldots$ stored in the octree node give the bounds of the region of integration. These bounds are used to index the summed-area table. $2^n$ table lookups are required to compute the integral, where $n$ is the number of parameters to $f_T$. If the integral equals zero then all of the voxels represented by the octree node must be transparent and need not be rendered. Otherwise, a more detailed level of the octree can be used to refine the estimated classification. When the algorithm reaches the lowest level of the octree it classifies the voxels in the potentially-nontransparent octree nodes (using the original opacity transfer function $f$) and then renders them.

This algorithm never discards a non-transparent voxel, but it may cause transparent voxels to be rendered. The run-length encoding used in the previous two algorithms exactly represents the subset of the voxels that are transparent whereas the fast classification algorithm computes a conservative approximation. However, if the volume has a high degree of coherence then the approximation will be good enough to achieve a substantial speedup.

The algorithm also takes advantage of coherence in the opacity transfer function. This type of coherence arises because users often choose transfer functions that partition feature space into a small number of transparent and non-transparent regions, e.g. only the voxels within a certain range of scalar values become non-transparent. During rendering, any octree subcube with parameters lying entirely within one region of the partition can be tested for transparency without visiting more detailed levels of the octree. Thus the algorithm can exploit coherence in the opacity transfer function to classify octree nodes with widely-varying parameter values in constant time.

### 4.3.4 Implementation of the Fast Classification Algorithm

The parallel projection rendering algorithm and the perspective projection rendering algorithm can both be modified to use the data structures described in this section (a min-max octree, a summed-area table, and an unencoded volume array) instead of a run-length encoded volume. In the current implementation each voxel in the volume array contains an explicit value for each parameter to the opacity transfer function as well as any parameters required for shading. Figure 4.17 is a pseudo-code description of the routines that implement the fast classification algorithm. They must be combined with one of the two rendering algorithms described earlier.

The first step during rendering is to compute the summed-area table from the opacity transfer function and to use the table to determine which nodes of the min-max octree represent transparent voxels. The routine *ClassifyOctree* performs these operations. It should be called from the beginning of the *RenderVolume* procedure in either the parallel projection algorithm or the perspective projection algorithm.

*ClassifyOctree* first calls *ComputeSummedAreaTable* to build a summed-area table from the current classification function, as described in Section 4.3.3. Then *ClassifyOctree* calls

```
1   procedure ClassifyOctree
        ComputeSummedAreaTable();
        ClassifyOctreeNode(octree_root);
    end

5   procedure ClassifyOctreeNode(node)
        integral = LookupSummedAreaTable(node.minima, node.maxima);
        if (integral = 0)
            node.status = TRANSPARENT;
        else if (node.children = NIL)
10          node.status = NON_TRANSPARENT;
        else
            node.status = SUBDIVIDE;
            for (i = 1 to 8)
                ClassifyOctreeNode(node.children[i]);
15      endif
    end

    procedure ComputeVoxelRuns(node, node_size, j, k)
        CurrentRunType = NIL;
        CurrentRunIndex = 0;
20      if (node.status = SUBDIVIDE)
            child = 0;
            if ((j mod node_size) > node_size/2)
                child = child + NEXT_J;
            if ((k mod node_size) > node_size/2)
25              child = child + NEXT_K;
            ComputeVoxelRuns(node.children[child], node_size/2, j, k);
            ComputeVoxelRuns(node.children[child + NEXT_I], node_size/2, j, k);
        else
            if (CurrentRunType = node.status)
30              RunLengths[CurrentRunIndex] = RunLengths[CurrentRunIndex] + node_size;
            else
                CurrentRunIndex = CurrentRunIndex + 1;
                RunLengths[CurrentRunIndex] = node_size;
                CurrentRunType = node.status;
35          endif
        endif
        return(RunLengths);
    end
```

Figure 4.17: Pseudo-code for the fast classification algorithm.

*ClassifyOctreeNode* which performs a preorder depth-first traversal of the min-max octree. At each node during the traversal *ClassifyOctreeNode* calls *LookupSummedAreaTable* to integrate the transparency function (derived from the opacity transfer function) as described in Section 4.3.3. If the integral equals zero then the routine marks the node as transparent. If the integral is nonzero and the node has no children then the routine marks the node as potentially non-transparent, meaning that the node will be considered by the rendering routine. The remaining case is that the integral is nonzero but the node is not a leaf node, so *ClassifyOctreeNode* recursively visits the eight children. After the recursion unwinds, the *status* fields in the octree nodes indicate which nodes represent transparent voxels.

The last routine in the fast classification algorithm is *ComputeVoxelRuns* which uses the octree to compute a list of transparent and non-transparent runs for any voxel scanline. The list of runs is stored in an array just like the *RunLengths* array in the run-length encoded volume (Figure 4.3). The rendering routines call *ComputeVoxelRuns* to compute the run lengths on-the-fly and then they use the array of run lengths and a scanline of voxels from the unencoded voxel array instead of the precomputed run-length encoded volume. Otherwise, the rendering algorithms do not change. The parallel projection algorithm in Figure 4.8 can be modified to use the fast classification algorithm by removing the call to *FindVoxelRuns* on line 11 and inserting two calls to *ComputeVoxelRuns* just before line 14 (one for each voxel scanline). The perspective projection algorithm in Figure 4.10 can be modified by removing the call to *FindVoxelRuns* on line 3 and inserting calls to *ComputeVoxelRuns* for each voxel scanline just before line 6. The call to *AdvanceVoxelPtr* on line 15 must also be removed.

*ComputeVoxelRuns* operates as follows. It is a recursive routine that must be called from *CompositeSlice* with the root node of the octree, the number of voxels per side in the root node, and the coordinates of the current voxel scanline ($k$ is the slice number and $j$ is the scanline number). *ComputeVoxelRuns* first examines the *status* field of the current node. This field contains the result computed by *ClassifyOctree*. If the field indicates that the octree node should be subdivided then *ComputeVoxelRuns* computes which two of the eight children of the node intersect the scanline. Since scanlines are always aligned with an axis of the volume, exactly two of the children must intersect and the arithmetic to determine which two is simpler than in a ray caster (lines 21–25). Once the two children have been

determined, *ComputeVoxelRuns* calls itself recursively on each one.

When *ComputeVoxelRuns* finds a node that should not be subdivided it adds a run to the array of run lengths for the current scanline. If the type of the current run and the node match (i.e. if both are transparent or both are non-transparent) then the length of the current run is extended by the size of one side of the octree node (line 30). Otherwise *ComputeVoxelRuns* creates a new run.

When the recursion unwinds the *RunLengths* array contains a list of run lengths indicating which voxels are transparent and which are potentially non-transparent. When the rendering loop in *CompositeSlice* encounters a potentially non-transparent voxel run it must retrieve the voxels from the unencoded voxel array and classify them individually just before they are resampled and composited.

Adjacent voxel scanlines often intersect the same set of octree nodes, so the array of run lengths can sometimes be reused for several adjacent scanlines. If the size of the smallest octree node visited by *ComputeVoxelRuns* is $n_{\min}$ then the next $n_{\min}$ voxel scanlines can reuse the same array of run lengths. *CompositeSlice* keeps track of when to call *ComputeVoxelRuns* and when to reuse the existing run lengths.

### 4.3.5 Limitations of the Fast Classification Algorithm

The fast classification algorithm places two restrictions on the opacity transfer function. First, it must be possible to precompute the parameters to the classification function for each voxel. The function itself need not be specified in advance, but the parameters must be fixed so that the min-max octree can be precomputed. For example, a transfer function that maps the scalar value and the gradient magnitude of the scalar value into an opacity is acceptable because both the scalar value and the gradient magnitude can be precomputed and stored with the voxel. An example of a classification function that does not satisfy this property is a segmentation algorithm based on a 3D flood fill. In order to compute the opacity of a particular voxel, a flood fill function must examine the values of many neighboring voxels. Thus the "parameters" of the function include a non-precomputable subset of the voxels in the volume.

The second restriction is that the cardinality of the domain of the opacity transfer function must not be too large. The cardinality of the domain depends on the number of parameters of the opacity transfer function and the number of possible discrete values for each parameter. The summed-area table must have one entry for every parameter tuple, so if the domain is too large then the table will take excessive time to compute and will occupy a large amount of memory. Classification functions with more than two or three parameters are not likely to be feasible.

Compared to the algorithms based on run-length encoding, the fast-classification algorithm has one primary disadvantage: it requires more memory, which can have an impact on the maximum feasible volume size and on rendering time. The fast classification algorithm uses an uncompressed 3D array of voxels, and each voxel contains precomputed values for each classification parameter and each shading parameter. For example, in the current implementation each voxel contains the scalar value (one byte), the gradient magnitude of the scalar value (one byte), and a precomputed surface normal vector for shading (two bytes). Thus a volume containing $512^3$ voxels occupies 512 Mbytes of memory, too large to fit into main memory on most workstations. A run-length encoded classified volume often occupies much less space, so the previous two rendering algorithms are practical for larger volumes.

Furthermore, because of the size of the voxel array, it is not practical to build three transposed copies of the array. As a result, if the major viewing axis changes then the volume data must be accessed against the memory storage stride and therefore performance degrades. Other volume rendering algorithms are also prone to this problem as we will see in the next chapter. The performance degradation may be acceptably small depending on the speed of the memory system. If memory performance is a problem it can be combated by transposing the 3D array of voxels, but doing so results in a delay during interactive rotation sequences. It is better to use a small range of viewpoints that does not require transposing the volume while modifying the classification function, and then to switch to the run-length encoded representation for rendering with arbitrary viewpoints. The next section describes an efficient algorithm for switching between the two representations.

Another solution is to first eliminate portions of the volume that will never be of interest by classifying the volume with a conservative opacity transfer function. This crudely-classified volume can be stored using the run-length encoded representation. The run-length

encoded volume can then be used in conjunction with the min-max octree and the fast classification algorithm. For example, it is straightforward to choose a transfer function that maps air in a CT scan to transparent opacity, eliminating a large fraction of the volume in many medical data sets. The user could then use the fast classification algorithm in conjunction with a run-length encoded volume to refine the classification function. This hybrid algorithm is a potential area for future research.

## 4.4 Switching Between Modes

We have built a demonstration application based on the algorithms in this thesis that has two modes, one for classification and one for viewing and shading (Figure 4.18). The classification mode uses the min-max octree and summed-area table, and the viewing mode uses the run-length encoded volume. These two modes address different parts of the visualization process outlined in Section 1.1.4.

However, users of the application will want to switch between the two modes at will. Since the min-max octree is independent of all user-selectable classification and rendering options it can be precomputed when the user loads a volume and therefore the data required for the fast classification mode is always available. But when the user wishes to switch from the classification mode to the viewing mode the run-length encoded volume must be recomputed for the current classification function. A fast algorithm to create the data structure is necessary so that the user can switch modes quickly.

The data structures and pseudo-code fragments developed in the previous section can be used to construct such an algorithm. A brute-force algorithm to compute a run-length encoded volume operates by streaming through the unencoded volume data in scanline order, classifying each voxel and collecting the classified voxels into runs. To optimize this algorithm we use the min-max octree and the summed-area table to determine which voxels in a voxel scanline must be transparent before traversing the scanline. Then the transparent runs can be skipped without classifying individual voxels. The routine *ComputeVoxelRuns* from Section 4.3 computes exactly the required information to enable the run-length encoding algorithm (Section 4.1.2) to skip transparent regions of the volume.

One way the rendering algorithms in this chapter can be integrated under a unified user

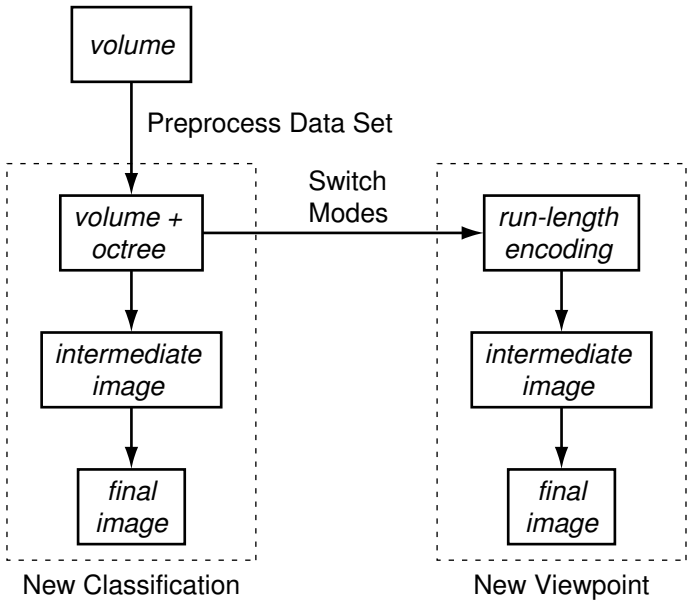



Figure 4.18: A family of volume rendering algorithms: When the user loads a new volume we preprocess the data to produce a min-max octree. The user can then interactively adjust classification parameters using the fast classification algorithm (left side). When the user settles on a classification function we use the min-max octree to compute a run-length encoded volume and the user can interactively adjust the viewing and shading parameters (right side).

interface is to automatically choose the appropriate rendering mode based on the rendering parameters the user adjusts. When the user adjusts a classification parameter the application uses the fast classification algorithm to render the volume and then starts a background process to compute the corresponding run-length encoded volume. If the user continues to adjust the classification parameters then the application interrupts the background process, rerenders the volume using the fast classification algorithm, and restarts the background process with the new parameters. If the user decides to adjust the viewing or shading parameters, the application pauses momentarily until the background process finishes computing the run-length encoded volume and then the application switches to viewing mode.

For very large volumes the fast classification mode is less efficient and the rendering time may be unacceptably slow. A solution is to use the fast classification algorithm to render a small subcube of the volume while using the most-recently computed run-length encoding to render the rest of the volume. The user can then select a subcube of the volume (say, $128^3$ voxels) to focus on during classification, but the rest of the volume would still

be visible to provide context. The application can provide faster feedback using this hybrid algorithm than it can if it rerenders the entire volume using the fast classification algorithm. If the user pauses for a long enough time then the background process can finish recomputing the run-length encoded volume and the main application can rerender the entire volume with the new classification.

## 4.5 Chapter Summary

In this chapter we developed three new volume rendering algorithms based on the shear-warp factorization: a rendering algorithm for parallel projections, a rendering algorithm for perspective projections, and a classification algorithm that can be incorporated into either rendering algorithm.

Unlike previous uses of the shear-warp factorization, the new algorithms exploit the properties of the factorization to combine the advantages of image-order and object-order algorithms. As in other object-order algorithms the new algorithms make efficient use of coherence data structures because they traverse the volume only once and they traverse the data structures in storage order, so there is very little overhead due to addressing arithmetic or inefficient memory access patterns. As in image-order algorithms the resampling filter is accurate and simple to implement, and the new algorithms include the early ray termination optimization.

Furthermore, we introduced coherence data structures for the fast classification algorithm that allow it to determine which portions of the volume are transparent without evaluating the classification function for each voxel. Unlike previous volume rendering algorithms based on spatial data structures the new algorithm allows a user to interactively adjust classification functions and see the results immediately.

This family of algorithms covers all phases of the visualization process outlined in Section 1.1.4. The next chapter examines the performance of the new algorithms and compares them to existing algorithms.

# Chapter 5

# Performance Analysis

Spatial coherence is a well-known means for accelerating volume rendering without degrading image quality. Although speedups of more than an order of magnitude over brute-force algorithms have been reported, the sources of these speedups and the relative merits of competing algorithms are not well-understood. This chapter presents a detailed performance analysis of the shear-warp algorithms and compares them to other coherence-accelerated algorithms.

In the first part of the chapter (Section 5.1) we report on the speed and image quality of the new shear-warp volume rendering algorithms. The results show that the parallel projection algorithm can render $256^3$ voxel medical data sets in less than one second on a 150 MHz R4400-based workstation without sacrificing image quality.

In the second part of the chapter (Section 5.2) we compare the shear-warp algorithms to image-order algorithms, both with and without coherence accelerations. We use an octree-accelerated ray caster as a representative example of image-order algorithms. We first show that the observed computational growth rates of a brute-force volume rendering algorithm, the octree-accelerated ray caster, and the shear-warp algorithm are $n^3$, $n^2 \log_2 n$ and $n^2$ respectively. We then show that the performance of each algorithm is governed not by its complexity, but by the associated constants. We do so by decomposing execution time into categories that distinguish between essential computation, overhead due to the coherence data structures, and memory overhead. This information allows us to pinpoint the differences between the algorithms.

Finally in the third part of the chapter (Section 5.3) we analyze and measure the performance of the shear-warp algorithms with low-coherence data.

From this analysis we make four observations regarding coherence-accelerated volume rendering algorithms:

- The time spent traversing spatial data structures is at least as great as the time spent resampling and compositing voxels.
- Octree-accelerated image-order algorithms spend more time traversing spatial data structures than object-order algorithms because the former must perform analytic geometry calculations (e.g. intersecting rays with axis-aligned boxes). As a result, the shear-warp algorithm is 4-10x faster than the octree-accelerated ray caster.
- Memory overhead is important in brute-force algorithms but not in coherence-accelerated algorithms.
- Coherence-accelerated volume rendering algorithms can be made to degrade gracefully in the presence of low data coherence.

These observations suggest that scanline-order volume rendering algorithms such as the shear-warp algorithm are likely to dominate over other algorithms, at least for software implementations.

## 5.1 Performance of the Shear-Warp Algorithms

### 5.1.1 Speed and Memory Performance

Table 5.1 shows the characteristics of our test data sets, including the sizes and the percentages of non-transparent voxels. The "brain" data set is an MR scan of a human head (Figure 5.1) and the "head" data set is a CT scan of a human head (Figure 5.2). The "brainsmall" and "headsmall" data sets are decimated versions of the larger volumes. The tests use classification functions that produce opaque surfaces and shading functions based on the Phong illumination model implemented with a lookup table. The lookup table maps a precomputed surface normal vector to a color, as described later in Section 7.1. We report rendering times that include the cost of recalculating the entries in the lookup table (which must be performed every time the viewpoint changes).

Table 5.2 lists some additional rendering parameters used in the experiments. All of the results in this chapter were measured on a 150 MHz SGI Indigo2 R4400 workstation without hardware graphics accelerators. This machine has an integer performance rating of 90 SPECint92 and a floating point performance rating of 87 SPECfp92.

The performance results for the three new algorithms are summarized in Table 5.3. The "Fast Classification" timings are for the fast classification algorithm with a parallel projection. The rendering times include all steps required to render from a new viewpoint, including shading, compositing and warping, but the preprocessing step to compute the spatial data structure is not included. The "Avg. Time" field in the table is the average time per

| Data set | Size (voxels) | Memory before classification (Mb) | Non-transparent voxels (%) | Image |
|---|---|---|---|---|
| brainsmall | 128x128x84 | 1.3 | 18 | |
| headsmall | 128x128x113 | 1.8 | 6 | |
| brain | 256x256x167 | 10 | 11 | Fig. 5.1 |
| head | 256x256x225 | 14 | 5 | Fig. 5.2 |
| head (skin) | 256x256x225 | 14 | 11 | Fig. 5.5 |
| engine | 256x256x110 | 6.9 | 8 | Fig. 5.6 |
| abdomen | 257x257x159 | 10 | 11 | Fig. 5.7 |

Table 5.1: Characteristics of the test data sets for the results in this chapter.

Figure 5.1: Volume rendering of an MR scan of a human brain using the shear-warp algorithm with a parallel projection (0.86 sec.).

Figure 5.2: Volume rendering of a CT scan of a human head with a parallel projection (0.80 sec.).

| | |
|---|---|
| Minimum opacity threshold | 0.05 |
| Early ray termination threshold | 0.95 |
| Min-max octree parameter threshold | 4 |
| Smallest min-max octree node size | $4^3$ voxels |
| Image size | $256^2$ pixels |

Table 5.2: Rendering parameters used for the performance tests in this chapter. The minimum opacity threshold determines whether or not a voxel is transparent. Both opacity thresholds are specified on a scale of 0.0 (transparent) to 1.0 (fully opaque). The min-max octree parameter threshold is the largest variation allowed for the parameters of the opacity transfer function in an octree leaf node; nodes with larger variations are subdivided until the size reaches the smallest allowed node size. The octree parameters are described in Section 4.3.2.

| Data set | Avg. Time | Min/Max Time | Memory |
|---|---|---|---|
| Parallel projection algorithm | | | |
| brainsmall | 0.22 sec. | 0.19–0.26 sec. | 4 Mb. |
| headsmall | 0.25 | 0.23–0.28 | 2 |
| brain | 0.86 | 0.69–1.06 | 19 |
| head | 0.87 | 0.77–0.98 | 13 |
| Perspective projection algorithm | | | |
| brainsmall | 0.48 sec. | 0.42–0.54 sec. | 4 Mb. |
| headsmall | 0.57 | 0.53–0.62 | 2 |
| brain | 2.04 | 1.67–2.46 | 19 |
| head | 2.35 | 2.12–2.55 | 13 |
| Fast classification algorithm | | | |
| brainsmall | 0.40 sec. | 0.35–0.47 sec. | 8 Mb. |
| headsmall | 0.54 | 0.49–0.61 | 8 |
| brain | 1.94 | 1.51–2.36 | 46 |
| head | 2.19 | 1.92–2.39 | 61 |

Table 5.3: Rendering time and memory usage on a 150 MHz SGI Indigo2 workstation. Times include shading, resampling, projection and warping. The fast classification times include rendering with a parallel projection. The "Memory" field is the total size of the data structures used by each algorithm; the first two algorithms use a compressed run-length encoded volume while the fast classification algorithm uses an uncompressed voxel array. Table 5.1 describes the data sets.

| Image Size | $64^2$ | $128^2$ | $256^2$ |
|---|---|---|---|
| Warp Time (msec.) | 5.0 | 17 | 64 |
| % Time, $64^3$ volume | 7 | 20 | 48 |
| % Time, $128^3$ volume | | 11 | 31 |
| % Time, $256^3$ volume | | | 9 |

Table 5.4: Execution time for the 2D warp. The time is expressed both in milliseconds and as a percentage of total rendering time for several image and volume sizes. Unless the image is much larger than the volume, the percentage is small.

frame for rendering 360 frames at two-degree-angle increments, and the "Min/Max" times are for the best and worst case angles. The "Memory" field gives the size in megabytes of all data structures. For the first two algorithms the size includes the three run-length encodings of the volume, the image data structures and lookup tables for shading and opacity correction. For the third algorithm the size includes the unencoded voxel array, the octree, the summed-area table, the image data structures, and the lookup tables.

For comparison purposes, we rendered the same data sets with a ray caster that uses early ray termination and a pyramid to exploit object coherence. Because of its lower computational overhead the shear-warp algorithm is 4-10x faster. In Section 5.2 we compare the two algorithms in more detail.

**Cost of the 2D Warp** Table 5.4 shows the time required to warp images of three different sizes and expresses the time as a percentage of total rendering time for three different volume sizes. These results are for a parallel projection of the MR "brain" volume prefiltered to the three resolutions. The implementation of the 2D warp uses bilinear interpolation. These results show that unless the image resolution is much larger than the volume resolution the performance overhead incurred by the warp is a small percentage of overall rendering time. Images with sizes much larger than the volume are not useful since the rendering algorithms sample the volume at object resolution. The volume should always be prescaled to roughly match the desired image size.

**Dependence of Rendering Time on Viewing Angle** Rendering time depends on viewing angle, primarily because the effectiveness of the coherence optimizations may vary with

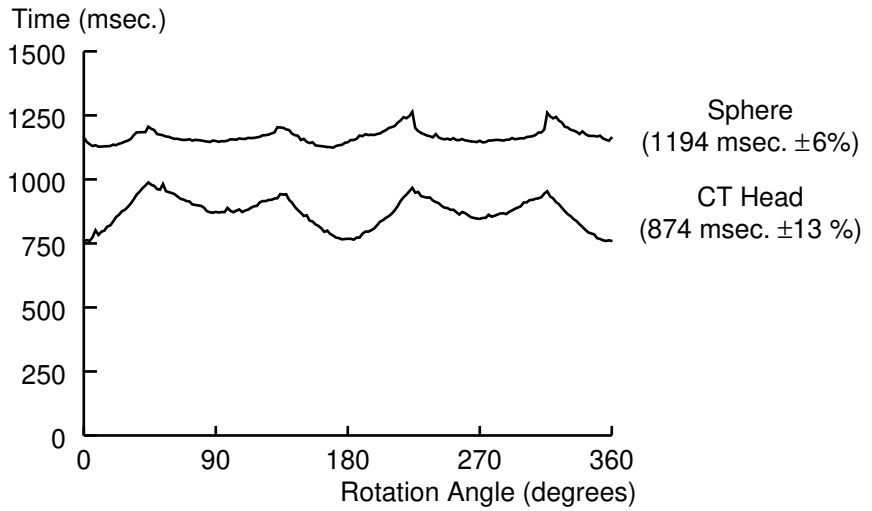



Figure 5.3: Rendering time versus viewing angle for a parallel projection of the CT head data set and a $256^3$ voxel data set containing a sphere.

viewpoint. Figure 5.3 shows the rendering time as a function of viewing angle with the parallel projection algorithm for the $256^3$ CT "head" data set and for a $256^3$ voxel synthetic data set containing a single large sphere. Since the sphere looks identical from all directions the effect of the coherence optimizations does not change with viewpoint, and the rendering time varies by only 6%. For the "head" data set the complexity of the rendered image does change and the variation in rendering time is 13%. There are secondary costs that also vary with viewpoint (for example the size of the intermediate image increases as the rotation angle approaches 45 degrees so the time required to initially clear it increases), but these effects are negligible.

Figure 5.3 also shows that there is no jump in rendering time when the major viewing axis changes. This fact holds true provided the three run-length encoded copies of the volume fit into real memory simultaneously. In the current implementation each copy of the volume contains four bytes per non-transparent voxel and one byte per run. For the 256x256x225 voxel head data set the three run-length encodings total only 9.8 Mbytes. An increase in rendering time would occur only if main memory were not large enough to hold the compressed volume, which would cause paging from the disk.

One might expect cache performance to degrade when the major viewing axis changes,

but the results in Figure 5.3 show that there is no impact on overall performance. This result can be explained by two effects. First, as in other volume rendering algorithms [Nieh & Levoy 1992], the ratio of voxel memory references to executed instructions is low. Even ignoring the complicated control logic for traversing the coherence data structures, the inner loop of the rendering algorithm contains about 100 R4400 machine instructions to resample four voxels and composite the result into the image. Only eight of those instructions load voxel data (four opacities and four surface normal vectors for shading). Second, the volume is always traversed in scanline order so the accesses to the volume data have very good spatial locality. Thus these eight load instructions are likely to hit in the cache.

If all eight voxel load instructions resulted in cache misses then the memory overhead could be non-negligible, since miss penalties of 50-100 cycles are becoming common in modern processors. The fast classification algorithm suffers from this problem if the voxel array is not properly transposed, in which case the algorithm does not traverse the volume in scanline order. For example the fast classification algorithm renders the $256^3$ sphere data set in 2.7 seconds if it is properly transposed versus 4.0 seconds if it is not.

**Cost of Preprocessing and Switching Modes** Figure 5.4 gives a breakdown of the time required to render the brain data set with a parallel projection using the fast classification algorithm (left branch) and the fast rendering algorithm (right branch). The "Switch Modes" arrow shows the time required for all three copies of the run-length encoded volume to be computed from the unencoded volume and the min-max octree once the user has settled on an opacity transfer function.

The diagram also shows the time for preprocessing the volume. This step is independent of both the viewpoint and the classification function. The preprocessing includes computing the gradient of the scalar field, the surface normal vectors (an encoded representation of the gradient), and the min-max octree.

**Rendering Time for Color Images** Compared to gray scale renderings, color renderings take roughly twice as long when using a parallel projection and 1.3x longer when using a perspective projection because of the additional resampling and compositing required for the two extra color channels. The slowdown is worse in the parallel projection algorithm

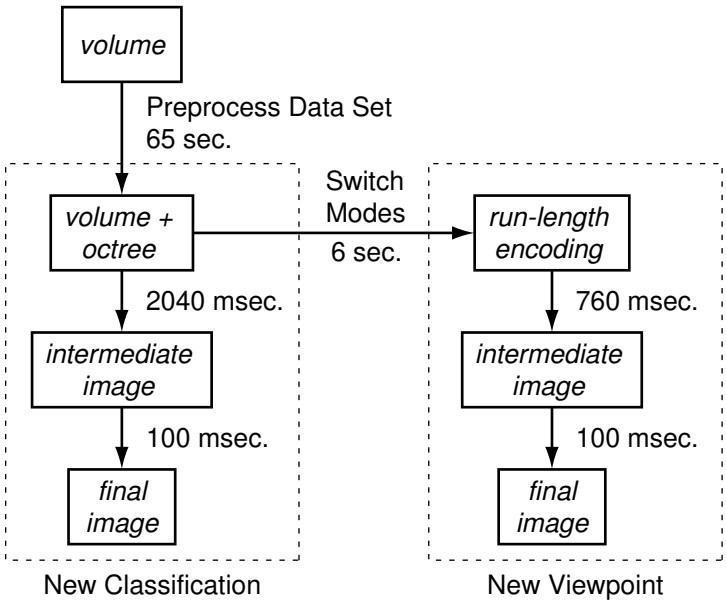



Figure 5.4: Execution time for each stage of rendering the brain data set with a parallel projection. The left side uses the fast classification algorithm and the right side uses the fast rendering algorithm.

because in the perspective projection algorithm a larger percentage of the overall rendering time goes to control overhead and computing resampling weights, and these overheads are not affected by the number of color channels.

Figures 5.5–5.7 show several examples of color volume renderings produced with the parallel projection rendering algorithm. Figure 5.5 is a rendering of the "head" data set classified with semitransparent skin which took 2.8 sec. to render. Compared to Figure 5.2, 630 msec. of the increased time results from using a different classification, 950 msec. of the increase is due to resampling and compositing RGB colors instead of gray scale, and 450 msec. of the increase is due to a more complicated shading function (separate shading lookup tables were used for voxels representing skin and voxels representing bone). Figure 5.6 is a rendering of a 256x256x110 voxel engine block, classified with semi-transparent and opaque surfaces; it took 2.0 sec. to render. Figure 5.7 is a rendering of a 257x257x159 voxel CT scan of a human abdomen, rendered in 2.2 sec. The blood vessels of the subject contain a radio-opaque dye, and the data set was classified to reveal both the dye and bone surfaces. The anatomical structures were segmented manually (a drawing program was used to tag each voxel with an identifier so that the kidneys, blood vessels and bone

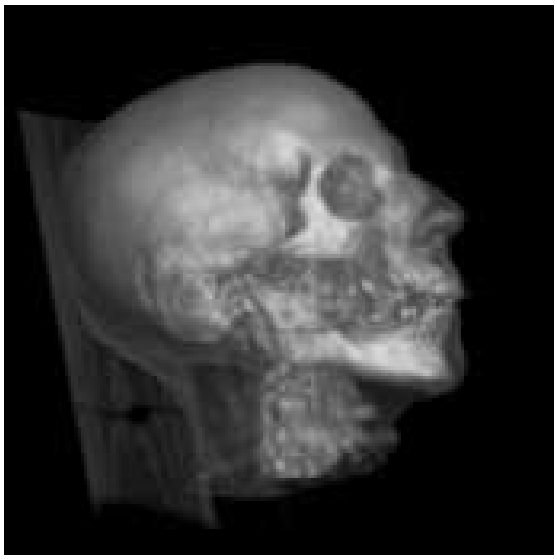


Figure 5.5: Color volume rendering of the CT head data set with a parallel projection, classified with semitransparent skin (2.8 sec.).

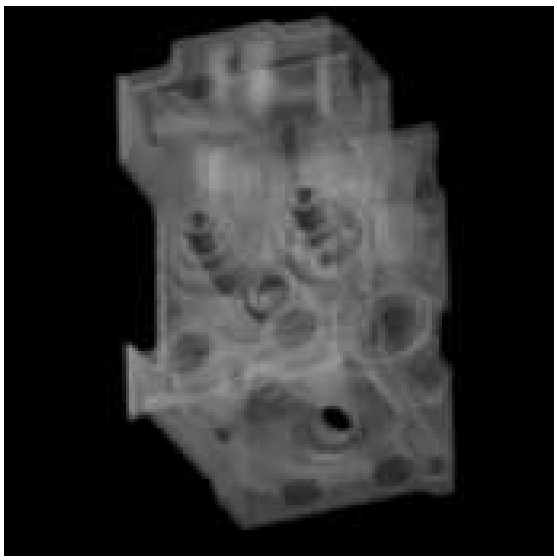


Figure 5.6: Color volume rendering of an engine block with a parallel projection, classified with semitransparent and opaque surfaces (2.0 sec.).

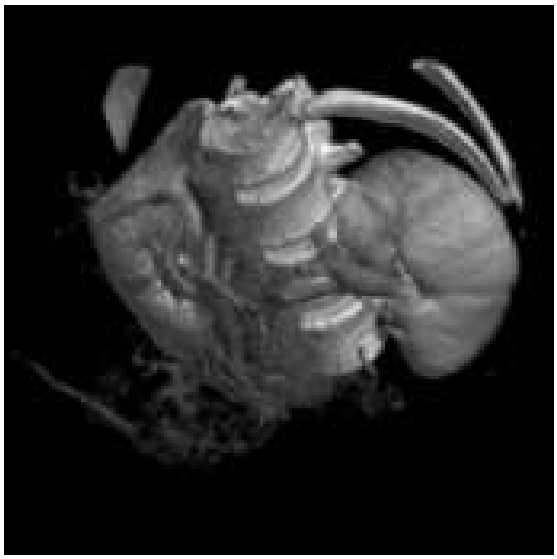


Figure 5.7: Color volume rendering of a CT scan of a human abdomen with a parallel projection (2.2 sec.). The blood vessels contain a radio-opaque dye.

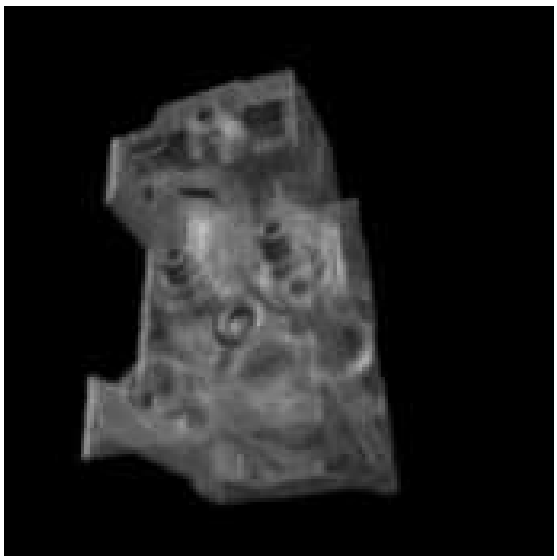


Figure 5.8: Color volume rendering of the engine data set with a perspective projection (2.6 sec.).

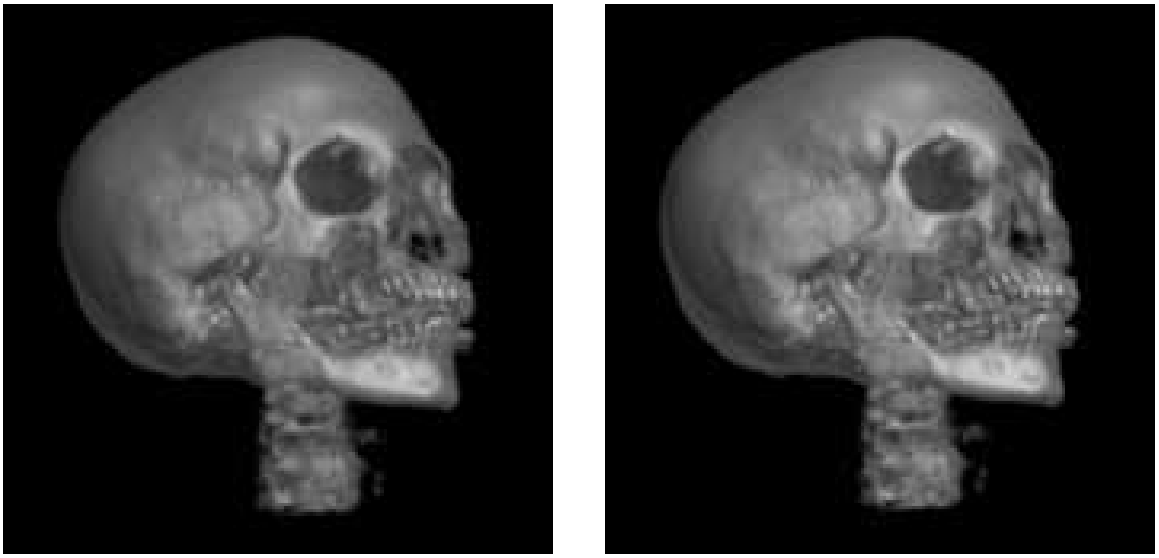


Figure 5.9: Image quality comparison of an image rendered by the shear-warp algorithm (left) with an image rendered by a ray caster using trilinear interpolation (right). The data set is a $256^3$ voxel CT scan of a head, oriented at 45 degrees relative to the axes of the volume.

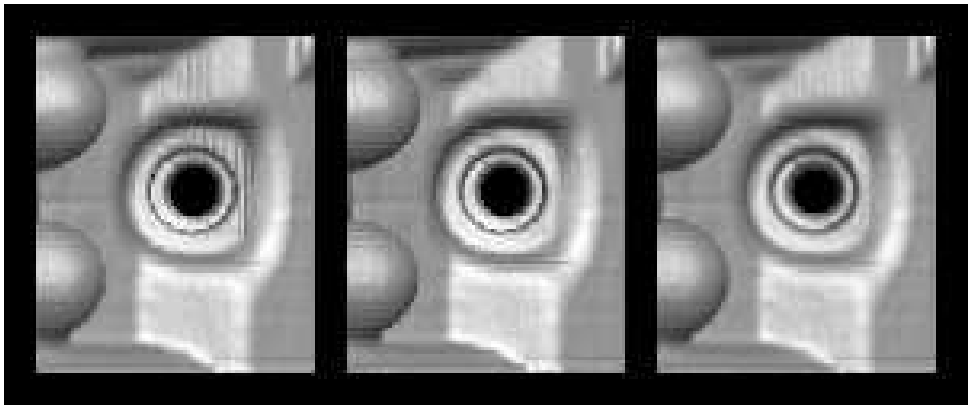
structures could be shaded differently).

Figure 5.8 is a color volume rendering of the engine data set using the perspective projection algorithm. It took 2.6 sec. to compute.

### 5.1.2 Image Quality

Figure 5.9 shows two volume renderings, one computed with the shear-warp algorithm and one computed with a ray caster using trilinear interpolation. The two images are virtually identical.

Nevertheless, the shear-warp algorithm imposes several limitations on the filter used to resample the volume and these limitations can potentially result in image degradation. The first limitation is that two resampling steps are required: the algorithm resamples the volume slices and then resamples the intermediate image to produce the final image. Multiple resampling steps can potentially cause blurring and loss of detail. However in practice this problem is not severe. Even in the high-detail regions of Figure 5.9 there is no noticeable blurring.



(a) (b) (c)

Figure 5.10: Comparison of image quality with bilinear and trilinear filters for a portion of the engine data set. The images have been enlarged by replicating pixels. (a) Bilinear filter with binary classification. (b) Trilinear filter with binary classification. (c) Bilinear filter with smooth classification.

A second limitation is that the shear-warp algorithm uses a 2D rather than a 3D reconstruction filter to resample the volume data. A high-quality filter can be used to accurately reconstruct each voxel slice, but the algorithm does not interpolate data between voxel slices. Aliasing artifacts can appear if the opacity or color attributes of the volume contain high frequencies in the dimension perpendicular to the slices (although if the frequencies exceed the Nyquist rate then these artifacts are unavoidable).

Figure 5.10 shows a case where a trilinear interpolation filter outperforms a bilinear filter. The left image is a rendering produced with the shear-warp algorithm of a portion of a data set that has been classified with extremely sharp ramps to produce high frequencies in the volume's opacity. The viewing angle is set to 45 degrees relative to the axes of the data set—the worst case—and aliasing is apparent. For comparison, the middle image is a rendering produced with a ray caster using trilinear interpolation and otherwise identical rendering parameters; here there is virtually no aliasing. However, with a smoother opacity transfer function these filtering artifacts can be reduced. The right image is a rendering using the shear-warp algorithm and a less-extreme opacity transfer function. Here the aliasing is barely noticeable because the high frequencies in the scalar field have effectively been low-pass filtered by the transfer function.

In practice, if the original volume is band-limited and the opacity transfer function does

not result in a binary classification (so that high frequencies are not introduced) then the bilinear filter produces good results. Synthetic data sets that have not been properly antialiased do not always satisfy these conditions and may result in renderings with artifacts. Real-world sampled data generally does satisfy these conditions as long as binary classification is not used, and as shown in the figures the resulting images are of comparable quality to those produced with trilinear interpolation. Binary classification could be supported by band-limiting the volume after classification, i.e. blurring the opacities.

Finally, a third limitation of the shear-warp rendering algorithms is that the resolution of the input volume fixes the sampling rate during rendering, regardless of the image resolution. If the image resolution is much greater than the volume resolution then the image will be either blurry or blocky, depending upon the filter used for the 2D warp. The quality of the rendering could be improved by supersampling each slice of voxel data as it is transformed into sheared object space. However the increased algorithmic complexity caused by the supersampling would result in reduced performance, and it is impossible to supersample in the dimension perpendicular to the voxel slices without significant changes to the algorithm. A better alternative is to prescale the volume by resampling it to a higher-resolution grid before rendering. A high-quality filter can be used to prescale the volume since the cost would be incurred only once instead of during each rendering. The main disadvantage of prescaling is that the upsampled volume requires more memory.

The volume should also be prescaled if the voxel spacing in the original volume is not isotropic (i.e. the spacing between samples is not the same in all three dimensions). Otherwise the sampling rate in image space will be non-uniform, potentially resulting in either aliasing artifacts or an excessively high voxel sampling rate in some parts of the image. Ray casters do not have this problem because the sample locations along a viewing ray can be distributed uniformly, independently of the volume resolution.

The renderings in this chapter show that the three filtering limitations do not have a significant effect on the quality of still images. However, subtle aliasing artifacts in a still image can become more apparent in movie sequences: pixels scintillate if their brightness changes from frame to frame. We have used the shear-warp algorithm to generate animations from the data sets shown in Figures 5.5–5.7 and we found no objectionable artifacts.

The shear-warp rendering algorithms produce smooth rotation sequences without noticeable aliasing. Furthermore, the image sequences are continuous even when the principal viewing axis changes.

## 5.2 Comparison of Coherence-Accelerated Volume Rendering Algorithms

The shear-warp algorithms achieve substantially improved performance compared to existing volume rendering algorithms. The primary reason for the performance gains is that the shear-warp algorithm traverses the coherence data structures and the volume data in scanline order, resulting in low computational overhead. In contrast, most existing high-quality coherence-accelerated volume rendering algorithms are based on ray casting. In the remainder of this chapter we use an octree-accelerated ray caster as a representative example of existing methods and compare it to the shear-warp algorithms. The comparison illustrates the performance differences between scanline-order algorithms and ray casters.

Our experiments use the optimized ray casting algorithm described by Levoy [1990a] augmented with the template optimization described by Yagel & Kaufman [1992]. The algorithm uses a full octree (a pyramid) to encode a classified volume. Each node in the octree includes a flag indicating whether the node or any of its children contain at least one non-transparent voxel. During ray casting the algorithm uses the information in the octree to determine the largest step that can be taken along a viewing ray before reaching the next non-transparent voxel. When the algorithm finds a non-transparent voxel it enters a standard ray casting loop that steps along the ray at fixed intervals, sampling the volume and compositing the samples together, until a transparent voxel is reached. Then the algorithm again uses the octree to find the next non-transparent voxel. The template optimization is used to reduce the costs of tracing rays and computing resampling weights as described in Section 3.5. The algorithm also uses the early ray termination optimization.

### 5.2.1 Asymptotic Complexity

Brute-force volume rendering algorithms operate by resampling and compositing all of the voxels in a volume. Two examples are the shear-warp algorithm without run-length encoded data structures and the ray casting algorithm without an octree or early ray termination. In these algorithms the work per voxel is constant, so assuming that the volume contains $n^3$ voxels ($n$ voxels per side) and the image contains $n^2$ pixels, and that the ray caster casts one ray per pixel with $n$ samples along the ray, the asymptotic complexity of the brute-force algorithms is $O(n^3)$ operations.

Coherence optimizations reduce the number of voxels processed. For some classes of volumes containing significant coherence, e.g. scenes consisting of voxelized surfaces, these optimizations also reduce the asymptotic complexity. However, accesses to the coherence data structures also have a cost. This section analyzes the complexity of an octree-accelerated ray caster, the shear-warp algorithm with run-length encoded data structures, and the shear-warp algorithm with a min-max octree data structure.

**Complexity of Octree-Accelerated Ray Casting** The octree data structure in the coherence-accelerated ray caster allows the algorithm to avoid shading, resampling and compositing the transparent voxels in the volume. In the worst case all voxels in the volume have a small but non-negligible opacity such that they all contribute to the image. In that case neither the octree nor early ray termination helps, so the worst-case asymptotic complexity of the rendering algorithm is still $O(n^3)$ operations.

However, typical classified volumes have structure that can be exploited with the octree. In practice a common case is a scene consisting of surfaces, some of which may be transparent or fuzzy. For instance, doctors and engineers who use CT and MR scanners to examine physical objects are generally interested in the boundaries between different tissues or materials, so they often choose a classification function that exposes these surfaces. The octree can efficiently encode the large regions of transparent space lying between individual surfaces.

Suppose the classified scene consists of a collection of opaque surfaces. In this case, the number of visible voxels equals the projected surface area of the visible surfaces (at most $n^2$ voxels) multiplied by the average thickness $t$ of the surfaces. The thickness of a voxelized

surface depends only on the support of the band-limiting filter used when creating the volume, not on the resolution of the volume (since the true surface is infinitesimally thin). In the case of acquired data the characteristics of the measurement equipment determine the support of the band-limiting filter.

If the scene contains semi-transparent surfaces then the maximum number of visible voxels increases by a factor $d$ which is the depth complexity of the scene (the average number of surfaces intersected by a viewing ray before the maximum ray opacity threshold is reached). The depth complexity is also independent of the resolution of the volume. Thus in the case of a classified volume representing either opaque or transparent surfaces, the maximum number of visible voxels is $dtn^2$ (where $d = 1$ for strictly opaque surfaces). The ray casting algorithm with an octree processes only the visible voxels, so the asymptotic complexity of resampling and compositing the volume is $O(dtn^2)$ operations.

The octree-accelerated ray caster also incurs costs due to the optimizations. The algorithm implements early ray termination by checking the accumulated ray opacity every time a voxel is composited, so the number of checks is $dtn^2$. Descending the octree to find the intersection of a ray with a surface incurs a cost of $O(\log_2 n)$ operations. If the number of surfaces is much smaller than the number of voxels, $d \ll n$, then the cost to find all surfaces intersecting a ray is $O(d \log_2 n)$ (although if $d \approx n$ then the cost of descending the tree can be amortized over many surfaces and the total work is $O(n)$). Finally, there is an additional cost due to grazing rays which pass near a surface. For these rays the rendering algorithm must still descend the octree in the vicinity of a surface even though the ray does not intersect the surface. It is difficult to analyze this cost rigorously since it depends on how smooth or convoluted the surfaces are (which is related to depth complexity). In practice our measurements have shown that the cost of grazing rays roughly doubles the total cost of casting rays through the octree. We therefore conclude that the overall asymptotic complexity of the optimized ray casting algorithm is:

$$O(dn^2 \log_2 n + dtn^2)$$

This result is a significant improvement over the $O(n^3)$ brute force algorithm.

**Complexity of the Shear-Warp Algorithm** The shear-warp rendering algorithms also process only the non-transparent, non-occluded voxels. Just as for the optimized ray caster, in the case of a scene containing surfaces the cost of resampling and compositing the volume is $O(dtn^2)$ operations. The shear-warp algorithm has four additional costs. It must warp the 2D image, costing $O(n^2)$ operations, and there are costs incurred by the coherence data structures: traversing voxel runs, and performing the UNION-FIND operations associated with the image pixel runs.

The cost of traversing voxel runs is $c_v n^2$ operations, where $n^2$ is the number of voxel scanlines in the volume and $c_v$ is the average number of runs per scanline. Unlike a ray caster the shear-warp rendering algorithm always traverses every run in the volume, although it does not process individual voxels in the transparent runs. If the volume consists of surfaces then the number of runs per scanline is proportional to the number of surfaces (which is also an upper bound on the depth complexity).

The cost of traversing and maintaining the opaque pixel runs depends on the number of times a new run is created ($n_c$) and the number FIND operations ($n_f$) as described in Section 4.1.3. Consider an intermediate image scanline containing $n$ pixels. On average, the number of voxels composited into the image scanline over the course of rendering the entire image equals the average number of surfaces per scanline times the average thickness of each surface, which equals $\frac{1}{2}c_v t$. This number is an upper bound for the number of opaque pixel runs created ($n_c$). An upper bound for the number of FIND operations ($n_f$) is the maximum number of opaque pixel runs times the number of times the image scanline is traversed, which equals $\frac{1}{2}c_v tn$. The cost of the UNION-FIND operations for a single intermediate image scanline is $O(n_f \log_{1+n_f/n_c} n_c)$ [Cormen et al. 1990]. Thus the total cost for all $n$ scanlines in the image is $O(c_v tn^2 \log_{1+n_f/n_c} c_v t)$.

The result of this analysis is the following worst-case asymptotic complexity for the shear-warp algorithm, under the assumption that the scene consists of opaque or semi-transparent surfaces:

$$O([dt + c_v t \log_{1+n_f/n_c} c_v t]n^2)$$

In practice we observe execution time growth rates consistent with this analysis as we saw in Table 5.3: an eight-fold increase in the number of voxels leads to only a four-fold increase in

time for the compositing stage of the algorithm and just under a four-fold increase in overall rendering time.

To place the relative importance of each term into context, for the CT head data set (Figure 5.2) the rough breakdown of computational costs in our implementation is: 40% for resampling and compositing voxels, 30% for traversing voxel runs, 10% for the 2D warp, and 5% for the UNION-FIND operations. These are percentages of computational costs only, ignoring memory overhead. We will analyze the measured results later in this chapter. For this data set, $n_f/n_c = 2.5$, $c_v = 5.8$, $t = 2.4$, and $d \approx 1$ since most surfaces are opaque.

The $n^2$ growth rate of the shear-warp algorithm is better than the $n^2 \log_2 n$ growth rate of the octree-accelerated ray caster. However, an improvement by a factor of $\log_2 n$ does not result in a large performance gain. The more important improvement is that the cost polynomial for the shear-warp algorithm also has lower constant factors. Experimental results later in this chapter validate this claim.

**Complexity of the Fast Classification Algorithm** The fast classification algorithm has the same costs as the basic shear-warp rendering algorithm analyzed in the previous section, plus additional costs to compute the summed-area table and traverse the min-max octree. Computing the summed-area table requires one operation for every combination of arguments to the opacity transfer function. If the function has $p$ parameters each of which can take on $m$ values then the computational complexity for computing the table is $m^p$ operations. Typically the number of parameters $p$ equals one or two, and the value of each parameter is stored in an eight-bit byte ($m = 256$).

The cost for one traversal of the min-max octree (to compute the run lengths for one voxel scanline) is $O(c_v t \log_2 n)$ operations assuming that the number of non-transparent voxels runs per scanline ($c_v$) is much less than $n$, just as we argued for the octree-accelerated ray caster. Since there are $n^2$ voxel scanlines the cost of the octree traversal for all scanlines is $O(c_v t n^2 \log_2 n)$. Adding up all of the terms, the asymptotic complexity of the fast classification algorithm is:

$$O([dt + c_v t \log_{1+n_f/n_c} c_v t]n^2 + c_v t n^2 \log_2 n + m^p)$$

In our implementation the cost of classifying, resampling and compositing the non-transparent voxels dominates (70% of the computation required to compute Figure 5.2). The octree traversal and summed-area-table initialization account for 10% of the computational costs.

### 5.2.2 Experimental Methodology

Operation counts and asymptotic complexities do not give a complete picture of an algorithm's performance. In the case of optimized volume rendering algorithms the constant factors associated with accessing coherence data structures and executing control logic are the dominant influences on performance. The only way to determine these costs is to measure the execution times of real implementations. The disadvantage of comparing algorithms by measuring execution times is that the relative performance depends on the specific implementations and on how carefully the implementations have been tuned. However, we are still able to draw some general conclusions from the results.

**Implementation of the Algorithms** The experiments in this section compare the ray caster and the shear-warp rendering algorithm for parallel projections. The results include timings for both brute-force versions and optimized versions of the algorithms. The optimizations fall into two classes: optimizations that reduce the number of voxels processed, and optimizations that reduce the cost per voxel actually processed. The former category includes early ray termination and coherence optimizations based on spatial data structures. The optimizations in the latter category have been chosen so that the ray caster and the shear-warp algorithm have nearly identical costs per voxel. These optimizations include:

**filter kernel** Different filter kernels have different costs. We use the ray caster to compare the costs of trilinear interpolation and bilinear interpolation. When bilinear interpolation is enabled the ray caster restricts each sample point to lie in the plane of a voxel slice, just like the shear-warp algorithm.

**template-based ray casting** Yagel's template algorithm [Yagel & Kaufman 1992, Yagel & Ciula 1994] (discussed in Section 3.5) uses the shear-warp factorization to reduce the cost of computing sample locations and filter weights in the ray casting algorithm.

With bilinear interpolation and the template optimization in the ray caster the cost of resampling is nearly identical to the cost of resampling in the shear-warp algorithm.

**resampled opacity threshold** When this optimization is enabled, if the opacity of a resampled voxel falls below a user-defined threshold then the renderer does not composite the voxel into the image. Both algorithms use this optimization to reduce the number of unnecessary compositing operations.

The performance results later in this section show that the cost-per-voxel optimizations are important when spatial data structures are not used, but are not nearly as important in the coherence-accelerated algorithms.

To allow for a fair comparison both implementations have been tuned at the source code level for the architecture on which the tests were run (an SGI Indigo2 R4400-based workstation). Except where otherwise noted, we show results for the shear-warp algorithm with run-length encoded data structures. The fast classification algorithm and the octree-accelerated ray casting algorithm are not directly comparable since the latter is not classification-independent.

**Timing Tests** Our experiments measure both the wall clock execution time of each algorithm on a set of benchmark data sets and a breakdown of the execution times into categories. The categories have been chosen to highlight the primary sources of overhead in each algorithm. The categories include:

**resampling and compositing** Time spent resampling voxels and compositing them into the image (or intermediate image). This work is essential to compute the image; all other categories represent overhead.

**looping** Time spent on control overhead (updating loop counters, advancing pointers, etc.) and traversing coherence data structures while searching for the next voxel to process.

**early ray termination** Time spent updating data structures required for early ray termination in the shear-warp algorithm. Early-ray termination has negligible cost in a ray caster.

**fixed costs** Miscellaneous fixed costs (costs that are independent of the size of the volume) required to initialize data structures and precompute lookup tables. For the shear-warp algorithm and the template-based ray caster this category includes the cost of warping the intermediate image into the final image.

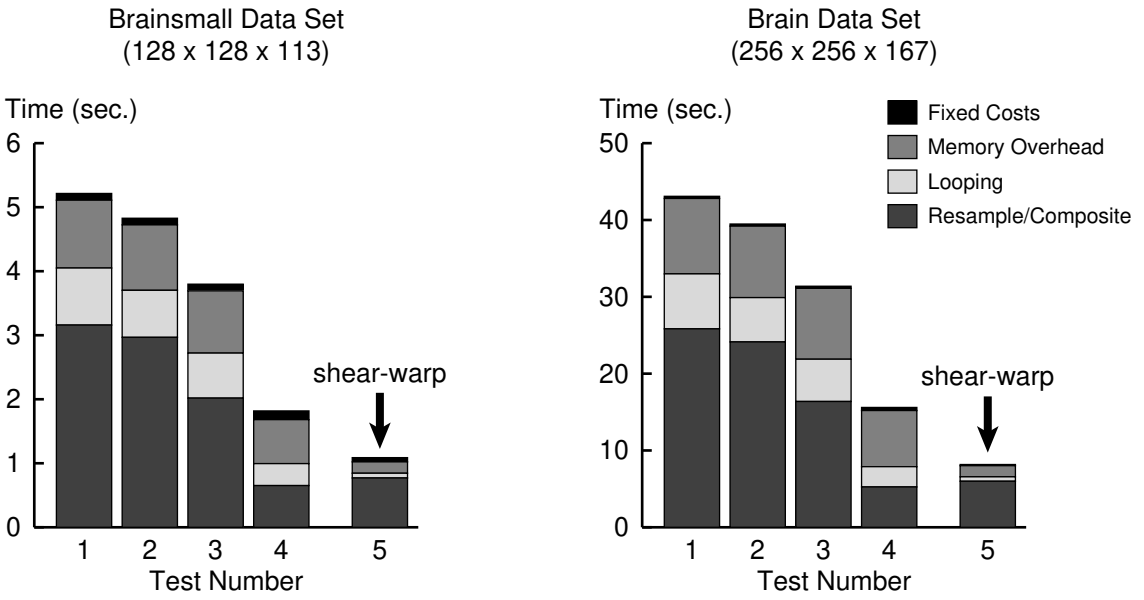
**memory overhead** Stall time associated with memory references. The other categories include computation time and processor stalls due to hardware interlocks (such as instruction dependencies) but not memory stall time.

Most of these cost categories involve operations that are implemented with short code fragments. Interval timers cannot be used to time such short code fragments since the overhead of querying the timer would perturb the results. Instead, the method we use to measure execution times is based on basic-block counting. First, we instrument the volume rendering program's binary executable to count the number of times each basic block is executed (using the Pixie program [Smith 1991]). We then run the instrumented program and use a post-processor to tabulate the number of times each instruction was executed along with the number of cycles the instruction took to execute (including pipeline stalls). Using information in the program's symbol table each instruction can be attributed to an individual source file line, and thus the instruction cycle counts can be grouped into the categories in the list above. Finally, given the speed of the processor, the cycle counts can be converted into execution times.

The basic-block timing method accounts for all costs in the program except for memory stalls. To measure memory overhead we run the uninstrumented program with the same parameters as the instrumented program on an otherwise-idle machine and measure the total execution time of the program with an interval timer. The difference between the total time and the time measured by basic block counting gives the time spent on memory overhead. We have automated the entire process of collecting cycle counts, attributing cycle counts to categories, and computing memory overhead so that data collection is easy and reliable.

### 5.2.3 Comparison of Speedups from Algorithmic Optimizations

Figure 5.11 gives timing breakdowns for the ray caster and the shear-warp algorithm without coherence optimizations or early ray termination. Since the coherence optimizations have




| Test Number | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Renderer | ray caster | ray caster | ray caster | ray caster | shear-warp |
| Opacity threshold | none | 0.05 | 0.05 | 0.05 | 0.05 |
| Filter | trilinear | trilinear | bilinear | bilinear | bilinear |
| Template | no | no | no | yes | N/A |
| Time (sec.) brainsmall | 5.2 | 4.8 | 3.8 | 1.8 | 1.1 |
| Time (sec.) brain | 43 | 39 | 31 | 16 | 8 |

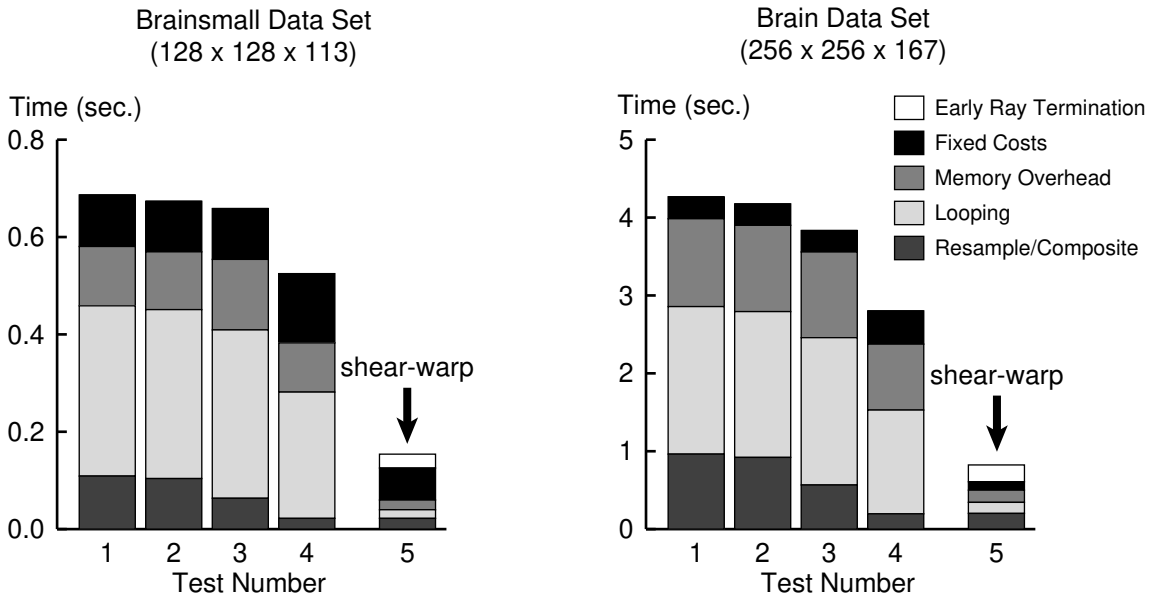
Figure 5.11: Timing breakdowns for the ray casting algorithm and the shear-warp algorithm without coherence optimizations.

been disabled the results are not view-dependent or data-dependent, but timings are shown for two different data set sizes. Each bar is broken down into components corresponding to the cost categories. For the ray caster the figure shows a sequence of measurements starting on the left with the most basic brute-force algorithm and progressively adding the cost-per-voxel optimizations. For the shear-warp algorithm the figure shows a single measurement (Test 5) with all of the cost-per-voxel optimizations.

These results show that when coherence optimizations are not used the dominant computational cost comes from resampling and compositing voxels. As a result the cost-per-voxel optimizations significantly reduce execution time. Using a bilinear interpolation filter rather than a trilinear filter results in a speedup of 1.3x, and the template optimization results in an additional speedup of 2.0x (slightly better than the results reported in Yagel & Ciula [1994]).

With these two optimizations the computation time for the ray caster (Test 4) is within 15% of the computation time for the shear-warp algorithm (Test 5), so the cost per voxel in the two algorithms is nearly identical. This fact will make it easier to isolate the effects of the coherence data structures in the fully-optimized algorithms. However the volume traversal order is different for the two algorithms. The ray caster does not traverse the volume in storage order so it has higher memory overhead than the shear-warp algorithm, resulting in the difference in overall execution time. Memory overhead will be discussed later in this chapter. The looping overhead is also slightly higher in the ray caster, although compared to the total rendering time the difference is not significant enough to merit investigation.

Figure 5.12 gives timing breakdowns for the ray caster and the shear-warp algorithm with both coherence optimizations and early ray termination. These timings were measured for a single representative viewpoint (Figures 5.1 and 5.2). The figure shows the same progression of cost-per-voxel optimizations as in Figure 5.11. These results show that resampling and compositing voxels is no longer the dominant cost. Looping (including traversal of the spatial data structures) is more significant, and in the case of the ray caster it accounts for the overwhelming majority of the computational costs. As a result the cost-per-voxel optimizations make very little difference. On the other hand, comparing the results for the brute-force algorithms and the coherence-accelerated algorithms shows that the coherence data structures provide speedups of roughly an order of magnitude as previously reported in Levoy [1990a].




| Test Number | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Renderer | ray caster | ray caster | ray caster | ray caster | shear-warp |
| Opacity threshold | none | 0.05 | 0.05 | 0.05 | 0.05 |
| Filter | trilinear | trilinear | bilinear | bilinear | bilinear |
| Template | no | no | no | yes | N/A |
| Time (sec.) brainsmall | 0.69 | 0.67 | 0.66 | 0.52 | 0.15 |
| Time (sec.) brain | 4.3 | 4.2 | 3.8 | 2.8 | 0.82 |

Figure 5.12: Timing breakdowns for the ray casting algorithm with an octree and early ray termination, and for the shear-warp algorithm with run-length encoding.

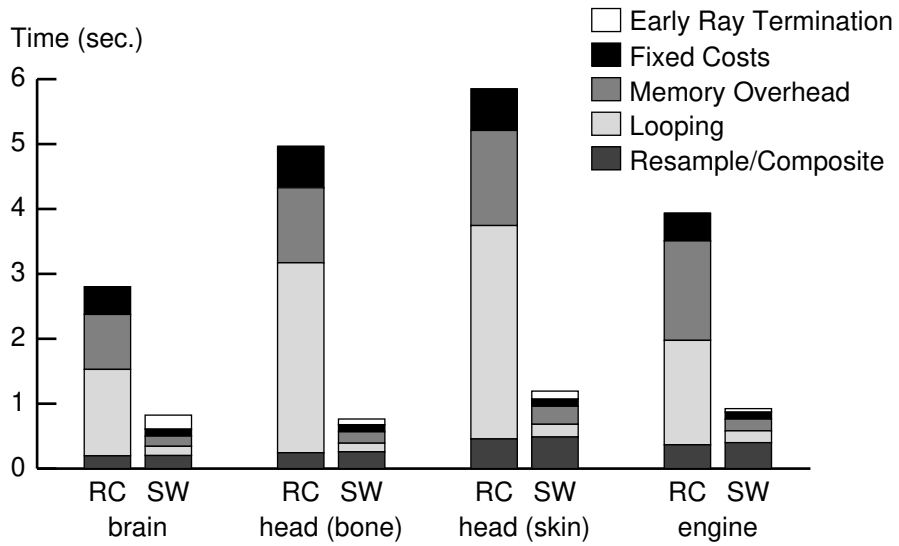



Figure 5.13: Comparison of timing breakdowns for several data sets. The left bar of each pair shows results for the coherence-accelerated ray casting algorithm with the template optimization and bilinear interpolation, and the right bar shows results for the coherence-accelerated shear-warp algorithm. All renderings are gray scale.

Figure 5.12 also shows that the cost of the early ray termination implementations is small. For the ray caster the cost is so small it is difficult to measure reliably and does not show on the scale of the figure. For the shear-warp algorithm the cost is slightly higher. We will discuss this cost in the next section.

Since the coherence optimizations are data-dependent these results depend on the specific data set. Figure 5.13 shows results for the larger MR brain data set and three other data sets: the 256x256x225 CT head data set classified to reveal bone (Figure 5.2), the same data set classified to reveal semitransparent skin and bone (a gray scale version of Figure 5.5), and the 256x256x110 CT engine data set (a gray scale version of Figure 5.6). This figure shows results only for the fully-optimized versions of the ray casting algorithm and the shear-warp algorithm. Again, the cost of processing voxels accounts for a small fraction of rendering time.

Finally, Table 5.5 shows the speedups achieved by the fully-optimized shear-warp algorithm relative to two versions of the octree-accelerated ray caster, one with trilinear interpolation and the other with bilinear interpolation and the template optimization. These results

| Data set | Shear-warp | Ray caster w/ octree and trilirp | | Ray caster w/ octree, bilirp and template | |
|---|---|---|---|---|---|
| | Time (msec.) | Time (msec.) | Shear-warp speedup | Time (msec.) | Shear-warp speedup |
| brainsmall | 154 | 622 | 4.0 | 524 | 3.4 |
| headsmall | 192 | 1201 | 6.3 | 990 | 5.2 |
| brain | 824 | 4124 | 5.0 | 2803 | 3.4 |
| head (bone) | 764 | 9459 | 12.4 | 4962 | 6.5 |
| head (skin) | 1194 | 9100 | 7.6 | 5849 | 4.9 |
| engine | 924 | 7617 | 8.2 | 3935 | 4.3 |

Table 5.5: Speedup of the shear-warp algorithm relative to an octree-accelerated ray caster. The table includes results for two versions of the ray caster: a standard ray caster with trilinear interpolation, and a ray caster with bilinear interpolation and the template optimization.

are for the same data sets used in Figure 5.13. Relative to the standard ray caster the shear-warp algorithm achieves speedups of 4-12x, and relative to the template-based ray caster the speedups are 3-6x.

The results in this section have shown that cost-per-voxel optimizations are not important in coherence-accelerated volume rendering algorithms. The results have also shown that the dominant cost in the brute-force algorithms is resampling and compositing voxels, but in the coherence-accelerated algorithms the overhead of looping and accessing the data structures is more important. The next section compares this overhead for the shear-warp algorithm and the ray caster in more detail.

### 5.2.4 Costs of Coherence Accelerations

Table 5.6 contains timing information in tabular form (but otherwise identical to Figure 5.13) for two of the data sets and shows the percentage of total execution time for each category of overhead.

The most important observation from this data is that the ray caster spends far more time looping than does the shear-warp algorithm. For the ray caster 70% of the time in the "looping" category comes from traversing the octree, and the remaining 30% comes from overheads in the loop that iterates over a span of non-transparent voxels within an octree leaf

| Data Set | brain | | head (skin) | |
|---|---|---|---|---|
| Algorithm | Ray Caster | Shear-Warp | Ray Caster | Shear Warp |
| Total Time (msec.) | 2802 | 824 | 5850 | 1194 |
| Early Ray Termination | 0 (0%) | 216 (26%) | 0 (0%) | 124 (10%) |
| Fixed Costs | 427 (15%) | 108 (13%) | 641 (10%) | 110 (9%) |
| Memory Overhead | 847 (30%) | 157 (19%) | 1465 (25%) | 277 (23%) |
| Looping | 1332 (47%) | 142 (17%) | 3288 (56%) | 196 (16%) |
| Resample/Composite | 197 (7%) | 202 (24%) | 457 (7%) | 487 (40%) |

Table 5.6: Detailed timing breakdowns for coherence-accelerated rendering algorithms (same data as in Figure 5.13). These results are for a ray caster with the template optimization and bilinear interpolation.

node. The source of the overhead in the octree traversal comes from the complexity of computing the intersection of an arbitrarily-oriented ray with the axis-aligned sides of the octree nodes.

Figure 5.14 is a pseudo-code description of the algorithm for tracing a ray through an octree. The algorithm begins by finding the first intersection of the ray with the top-level node of the octree. The loop on line 5 descends to the lowest non-transparent octree node containing the current sample point. The notation "Octree[level, x, y, z]" refers to the octree node at the specified level that contains the specified voxel coordinates. The call to *NextIntersection* calculates the intersection of the ray with the far side of the octree node. The algorithm then renders samples along the ray up to the intersection point. The loop on line 11 ascends back up the octree until the current octree node and the next node intersected by the ray both have the same parent (so that the next call to *RenderSegment* can take the longest possible step). Finally, the algorithm moves to the next octree node along the ray and repeats until the ray exits the volume (or until the accumulated ray opacity reaches the early ray termination threshold, although this test is not shown).

This algorithm contains a substantial amount of control logic (loops and conditionals), and the *NextIntersection* routine must compute the intersection of the ray with each of the three possible cube faces it might hit. Floating point arithmetic must be used to compute the intersection points in order to maintain a consistent sample spacing and to correctly compute resampling weights (unless the rendering algorithm uses a nearest-neighbor filter). These costs are inherent in an octree-accelerated ray caster. In contrast, the shear-warp algorithm

```
1   procedure TraceRay()
        level = top_level;
        (x, y, z) = FirstIntersection();
        while ((x, y, z) ∈ Volume)
5           while (level < bot_level and Octree[level, x, y, z] ≠ transparent)
                level = level + 1;
            end
            (dx, dy, dz) = NextIntersection(level, x, y, z);
            if (Octree[level, x, y, z] ≠ empty)
10              RenderSegment(x, y, z, dx, dy, dz);
            while (level > top_level and
                        Parent(level, x, y, z) ≠ Parent(level, x+dx, y+dy, z+dy))
                level = level – 1;
            end
15          (x, y, z) = (x, y, z) + (dx, dy, dz);
        end
    end
```

Figure 5.14: Pseudo-code for tracing a ray through an octree.

simply loads a run length and increments a pointer to advance to the next non-transparent voxel.

The ray caster has a second extra cost: the cost of tracing rays that graze but do not intersect surfaces in the volume. A ray that passes close to a non-transparent object causes the ray caster to descend to the leaf nodes of the octree, only to determine that the ray does not intersect the object. We have performed measurements indicating that the cost of grazing rays amounts to half of the total cost of looping in the ray caster. The shear-warp algorithm with a run-length encoded volume never wastes time sampling transparent voxels so it does not incur any additional cost for grazing rays.

Most other coherence-accelerated volume rendering algorithms also trace rays through an octree [Meagher 1982, Wilhelms & Van Gelder 1992, Danskin & Hanrahan 1992, Stander & Hart 1994] or a similar data structure such as a k-d tree (constructed using a median-cut algorithm to partition the volume with axis-aligned planes) [Subramanian & Fussell 1990]. These algorithms all have overheads due to the tree traversal. The fast classification algorithm proposed in Chapter 4 does not suffer from these overheads even though

it uses an octree because the algorithm always traverses the data structure along axis-aligned rays, so the traversal algorithm is more efficient.

Zuiderveld proposes an alternative ray casting algorithm that uses distance transforms instead of a hierarchical data structure [Zuiderveld et al. 1992, Zuiderveld 1995]. During a preprocessing step the algorithm computes the distance from each voxel to the nearest non-transparent voxel. The rendering algorithm uses the distance stored in a voxel at a particular sample point to determine the longest step size before the next interesting voxel. Zuiderveld shows that the computational overhead required to access the distance array and increment the sample location is less than the overhead of traversing an octree.

However, the algorithm has two disadvantages. First, the algorithm requires enough memory to store the entire volume and the distance array. Second, the distance transform only produces a conservative approximation of the distance to the next sample point. A ray caster using the distance transform steps by shorter and shorter distances as it approaches a non-transparent voxel, and at each step it must sample the volume to determine if the adjacent voxels are transparent (similar to the grazing-ray costs in the octree-accelerated ray caster). The shear-warp algorithm is more memory-efficient and spends less time searching for non-transparent voxels.

Returning to the table of timing breakdowns (Table 5.6), we see that the cost of early ray termination in the shear-warp algorithm is small but non-negligible. About 60-80% of the time in the "early ray termination" category is due to traversing the run-length encoded voxel scanlines when skipping occluded voxels. The individual voxels need not be visited, but the algorithm must step through the runs in sequence to reach the next non-occluded voxel. In a ray caster the octree nodes associated with occluded voxels are never visited, so there is no corresponding overhead. The remaining 20-40% of the time is spent traversing and coalescing the opaque image pixel links (Section 4.1.3).

The fixed costs for the ray caster are somewhat higher than for the shear-warp algorithm. Since this category accounts for only 10-15% of overall execution time the differences are not significant, but most of the increase is due to initializing the ray template and performing clipping calculations for each ray cast into the volume.

The final cost category is memory overhead, which we consider in the next section.

### 5.2.5 Memory Overhead

To demonstrate the impact of memory overhead we performed an experiment in which we rotated the CT head volume over a 90 degree range of angles and rendered it with both the ray caster and the shear-warp algorithm. We performed the experiment with both the brute-force and coherence-accelerated algorithms. We chose the viewpoint such that at the initial angle the viewing rays coincided with the direction of the voxel scanlines in the data structures used by the ray caster, and at the final angle the viewing rays were perpendicular to the voxel scanlines.

When the viewing rays line up with the voxel scanlines the ray caster benefits from good spatial locality, meaning that when the algorithm references a voxel the voxels immediately adjacent in memory will soon be referenced. When the viewing rays are perpendicular to the voxel scanlines there is no spatial locality. However, in the experiment we cast the rays in a sequence that maximizes temporal locality, meaning that when the algorithm references a voxel the same voxel may be referenced again soon. The algorithm does this by casting adjacent rays in sequence so that each ray reuses some of the same voxels as the previous ray during resampling.

Spatial and temporal locality improve memory performance because the cost of fetching a cache line from memory can be amortized over several voxel accesses, whereas an algorithm that references memory randomly causes a cache miss for every reference [Hennessy & Patterson 1990]. The shear-warp algorithm always benefits from good spatial locality because it traverses the volume in scanline order regardless of the viewing direction.

Figure 5.15 shows the results of the experiment for the brute force algorithms, and Figure 5.16 shows the results for the coherence accelerated algorithms. The timings were performed on an SGI Indigo2 workstation with a 16 Kbyte direct-mapped primary cache with 16 byte lines and a 1 Mbyte two-way associative secondary cache with 128 byte lines. The long cache lines in the secondary cache result in a significant benefit from spatial locality.

The brute-force shear-warp algorithm has constant memory overhead of only 15% for the entire range of angles. For the brute force ray caster the memory overhead increases by a factor of 2.5x as the viewing angle changes from 0 to 90 degrees, causing a 1.25x increase in rendering time. Because of the long secondary cache line the benefit of spatial locality

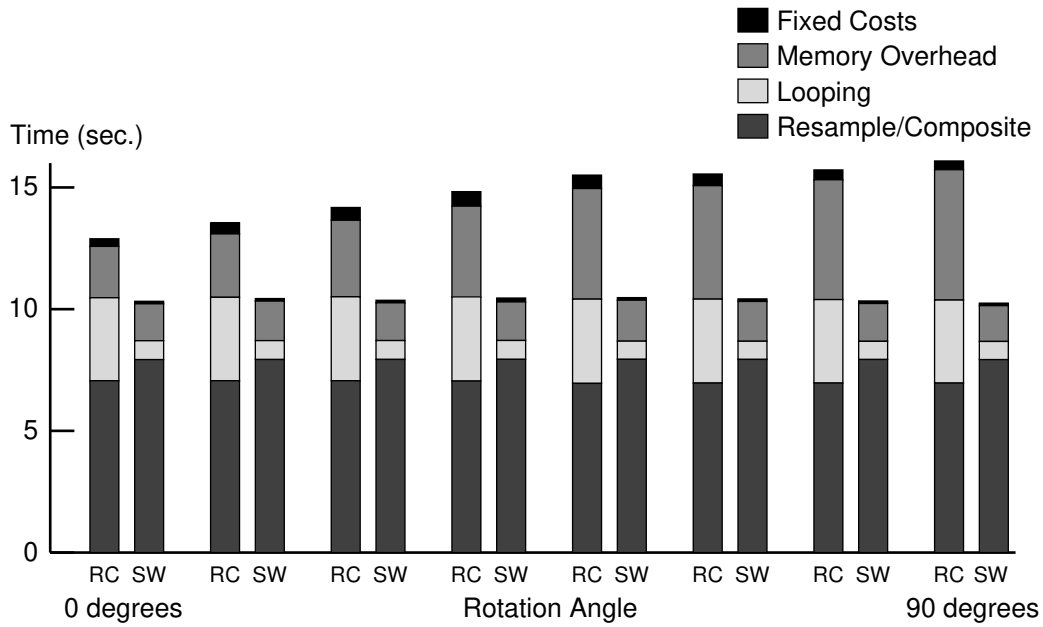



Figure 5.15: Comparison of memory overhead in the ray caster and the shear-warp algorithm without coherence optimizations. The ray caster uses the template optimization and a bilinear interpolation filter. The data set is the 256 x 256 x 225 voxel CT head. As the viewpoint changes over a 90 degree range of angles the memory overhead increases significantly for the ray caster but remains small for the shear-warp algorithm.

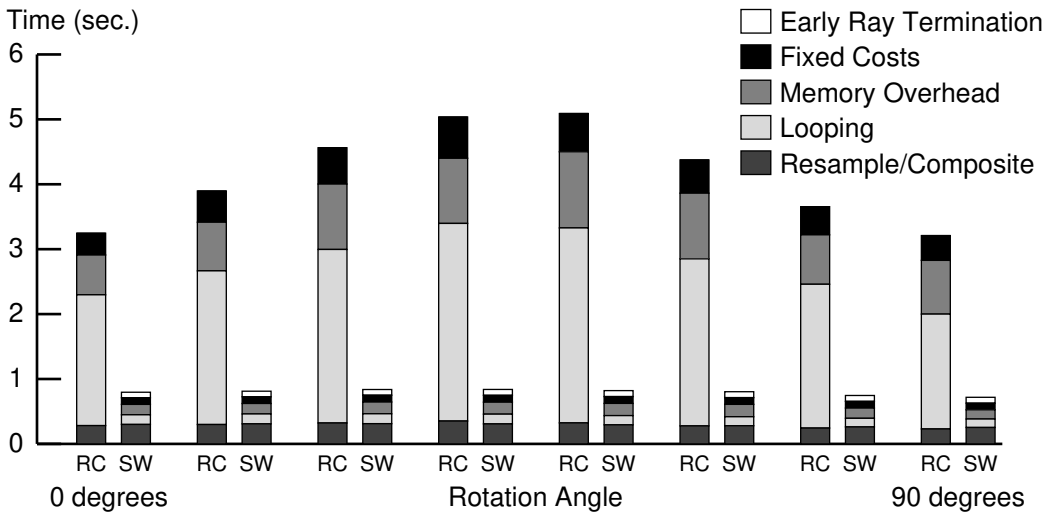



Figure 5.16: Comparison of memory overhead in the coherence-accelerated ray caster and the shear-warp algorithm. Memory overhead does not vary significantly as the viewpoint changes, although the overhead incurred by the octree does change.

is greater than the benefit of temporal locality. However, temporal locality still provides a significant benefit: when the rays are cast in a different order to eliminate the temporal locality the memory overhead increases by 10x relative to the case with optimal spatial locality, and the overall rendering time increases by 2.6x. The current trend in processor design is towards longer cache lines to cope with the increasing gap between processor speeds and memory latencies, so algorithms that benefit from spatial locality are important.

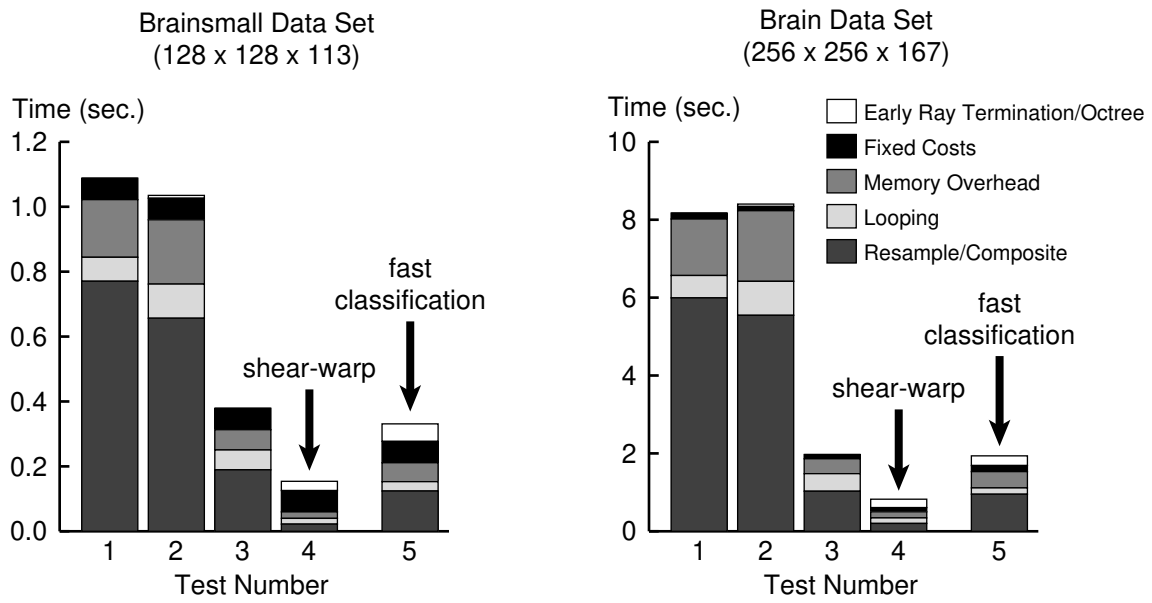
Figure 5.16 shows that for the coherence-accelerated algorithms memory overhead is not so significant: 20% of execution time for both algorithms. Most of the time in these algorithms is spent traversing data structures rather than processing voxels, so loading voxels from memory is a virtually insignificant overhead. The variations in memory overhead are small and are caused by variations in the accesses to the spatial data structures rather than the voxel array. Most likely the good memory performance in the ray caster is due to temporal locality when accessing the octree since many adjacent rays traverse the same octree nodes.

We conclude that when coherence optimizations are disabled or do not work effectively (due to low-coherence data sets) the shear-warp algorithm has lower memory overhead than the ray caster. In practice, the coherence optimizations are effective and both algorithms have excellent memory behavior.

### 5.2.6 Analysis of the Shear-Warp Coherence Data Structures

Figure 5.17 demonstrates the impact of each of the data structures used in the shear-warp algorithms. In the table the row labeled "Samples" gives the number of resampling operations, the row labeled "Time" gives the total time in seconds, and the row labeled "Speedup" gives the speedup relative to the brute-force algorithm (Test 1).

The left column gives results for the brute-force algorithm without early ray termination or a spatial data structure. The second column is for the brute-force algorithm with early ray termination enabled. The optimization has virtually no impact by itself because for this data set most of the transparent voxels are not occluded, so a majority of the voxels must be processed. The third column shows the effect of using a run-length encoded volume without early ray termination. This optimization yields a 3-4x speedup. If early ray termination is




| | | Render Only | | | | Classify |
|---|---|---|---|---|---|---|
| Test Number | | 1 | 2 | 3 | 4 | 5 |
| Spatial Data Structure | | none | none | run-length encoding | run-length encoding | min-max octree |
| Early Ray Termination | | no | yes | no | yes | yes |
| brainsmall | Samples | 1,397,844 | 1,021,395 | 301,138 | 32,495 | 90,391 |
| | Time (sec.) | 1.1 | 1.0 | 0.38 | 0.15 | 0.33 |
| | Speedup | 1.0 | 1.1 | 2.9 | 7.1 | 3.3 |
| brain | Samples | 11,030,183 | 8,556,478 | 1,760,992 | 305,127 | 629,080 |
| | Time (sec.) | 8.2 | 8.4 | 2.0 | 0.82 | 1.94 |
| | Speedup | 1.0 | 0.97 | 4.1 | 9.9 | 4.2 |

Figure 5.17: Impact of individual coherence optimizations in the shear-warp algorithm.

also enabled (Test 4) then an order-of-magnitude speedup is achieved. Early ray termination helps in this case because one-half to one-third of the non-transparent voxels are occluded, so a significant fraction can be culled.

The fifth column shows results for the fast classification algorithm using the min-max octree, the summed-area table and early ray termination. As shown earlier in Table 5.3 this algorithm is about twice as slow as the algorithm based on a precomputed run-length encoded volume. There are two reasons for the increase in time: the renderer samples more voxels, and it performs more work for each sample since the voxels must be classified. The increase in the number of samples occurs because the min-max octree and summed-area table data structures produce a conservative estimate of the regions containing non-transparent voxels.

## 5.3 Low-Coherence Volumes

### 5.3.1 Categories of Volume Data

The classification functions we used in the previous sections resulted in volumes with a high degree of coherence and voxels with relatively high opacity, so both of the optimization techniques in the shear-warp algorithm (coherence acceleration and early ray termination) were effective. However, there are other types of volumes for which the optimizations will fail.

The different types of classified volumes can be categorized along three axes (Figure 5.18):

1. The *amount of coherence in the volume*, which is measured by counting the number of runs in a run-length encoded representation of the classified volume. If there are only a few runs per voxel scanline then the volume contains a lot of coherence, whereas if each voxel scanline has many short runs of transparent voxels interspersed with short runs of non-transparent voxels then the volume has low coherence.

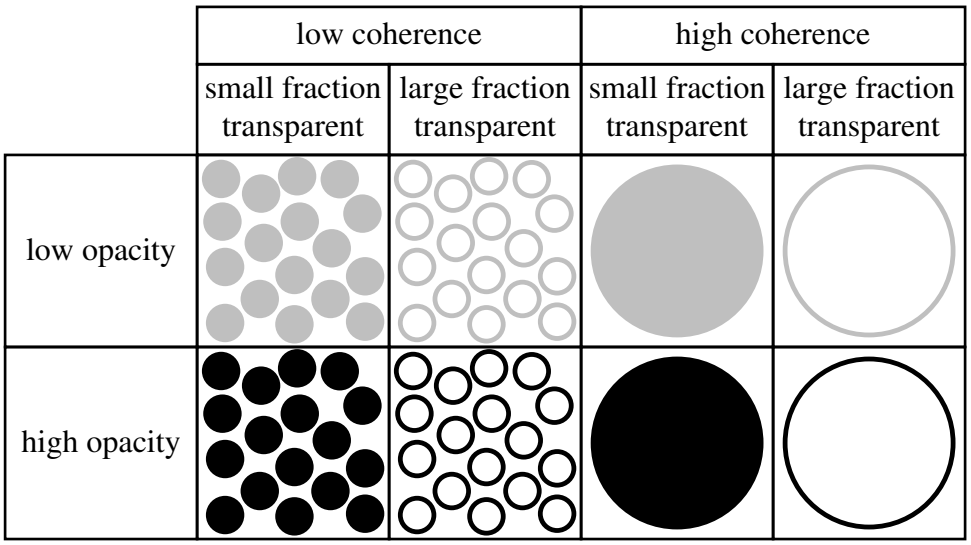



Figure 5.18: Classified volume data can be categorized along three axes: the amount of coherence in the volume (measured by the number of runs in a run-length encoded representation), the fraction of voxels that are transparent, and the average opacity of the non-transparent voxels. The diagram shows schematic cross-sections of volumes at eight points within the categorization.

2. The *fraction of voxels that are transparent*. Given sufficient coherence, these voxels can be skipped efficiently using a spatial data structure.

3. The *average opacity of the non-transparent voxels*. This quantity determines the effectiveness of early ray termination. The higher the average opacity, the earlier each pixel becomes opaque.

This categorization can be used to make first-order predictions about rendering performance. In the remainder of this section we discuss performance qualitatively, and in the next two sections we present some quantitative results for our implementation of the shear-warp algorithm.

There are four performance regimes. In the first regime neither of the optimizations is effective. Early ray termination is ineffective if the average opacity of the non-transparent voxels is low, since few (if any) image pixels become opaque. Spatial data structures may be ineffective for either of two reasons. First, if there is little coherence then the rendering algorithm cannot skip over large regions of the volume, so the spatial data structure cannot reduce the time spent traversing the volume. Second, if the volume does have high coherence but most of the voxels are non-transparent then the rendering algorithm must still traverse most of the volume. These situations correspond to the first three cases in the top row of Figure 5.18. In this regime the rendering time is proportional to the number of voxels in the data set, and the optimized shear-warp algorithm cannot outperform a brute force algorithm.

In the second performance regime early ray termination is ineffective but spatial data structures do improve performance. This regime applies to volumes that have low average opacity, high coherence, and a large fraction of transparent voxels (the last case in the top row of Figure 5.18). The rendering algorithm can use a spatial data structure to efficiently skip over the transparent voxels, so the rendering time depends linearly on the number of non-transparent voxels (which is proportional to the size of our run-length encoded representation). There is also overhead due to traversing the spatial data structure. In the case of the shear-warp algorithm the overhead is proportional to the number of runs.

In the third performance regime spatial data structures are ineffective but early ray termination improves performance. For this regime to apply the average voxel opacity must

be relatively high. Furthermore the side of the volume facing the viewer must be covered with high-opacity voxels so that viewing rays do not penetrate far into the volume before they are terminated. These conditions apply to the first and third cases on the bottom row of Figure 5.18. The second case on the bottom row may be in this performance regime or in the first regime depending on whether or not the spaces between the non-transparent voxels leave "holes" that can be penetrated by viewing rays. For the third regime rendering time depends linearly on the number of non-occluded voxels.

Finally, in the fourth performance regime both optimizations are effective. The last case in the bottom row of Figure 5.18 falls into this regime, as do all of the data sets in the performance tests described earlier (except the CT engine data set which has low average opacity). Rendering time depends linearly on the number of visible voxels (voxels that are neither transparent nor occluded). For this type of volume the number of visible voxels is proportional to the projected surface area of the objects in the volume, so the rendering algorithm only processes $O(n^2)$ voxels.

### 5.3.2 Voxel Throughput

To demonstrate the characteristics of the shear-warp algorithm in each performance regime we conducted two experiments. For both experiments we used the parallel projection algorithm with a run-length encoded volume.

The first experiment measures how rendering time depends on the number of visible voxels. In this experiment the amount of coherence and the number of transparent voxels are constant, but the average voxel opacity varies. This experiment also measures the renderer's voxel throughput.

To perform the experiment we constructed a series of 256x256x256 voxel volumes with constant opacity. Each volume had a different opacity, ranging from 0.0 (entirely transparent) to 1.0 (entirely opaque). Since the volumes had constant opacity all of the run-length encoded data structures contained exactly one non-transparent run per voxel scanline, resulting in a constant (high) amount of coherence and a constant (low) fraction of transparent voxels.

In this experiment the opacity associated with each volume determines the number of

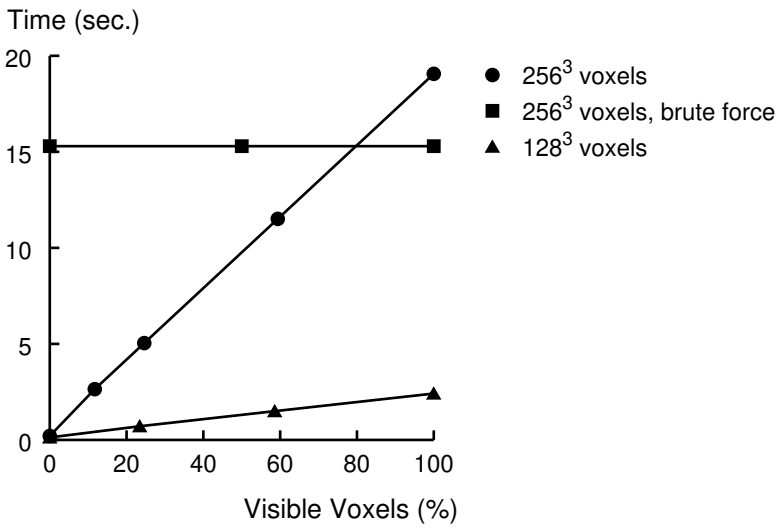



Figure 5.19: Results of the voxel throughput experiment for a $256^3$ voxel volume with the shear-warp algorithm, a $256^3$ voxel volume with a brute-force algorithm, and a $128^3$ voxel volume with the shear-warp algorithm. In each experiment the amount of coherence is held constant while the voxel opacity is varied so that the number of visible voxels changes. The results show that the rendering time using the optimized shear-warp algorithm is proportional to the number of visible voxels (voxels that are neither transparent nor occluded).

visible voxels (since we also hold the minimum opacity threshold constant). Only the front slices of the high-opacity volumes are visible, while all of the voxels are visible in the low-opacity volumes. We expect rendering time to vary linearly with the number of visible voxels since the overhead due to the coherence data structures is constant.

Figure 5.19 shows the results of the experiment. When plotted against the number of visible voxels, the timings for the optimized shear-warp algorithm do in fact lie on a straight line. The slope of the line indicates that the voxel throughput is 880 Kvoxels/sec. (on a 150 MHz R4400 SGI Indigo2).

The figure also shows the rendering time for a brute-force shear-warp algorithm with both early-ray termination and run-length encoding disabled (the same algorithm as we used for Test 5 in Figure 5.11). The crossover point at which the brute-force algorithm matches the optimized algorithm occurs when 80% of the voxels are visible. When all voxels are visible the optimized algorithm is about 25% slower than the brute-force algorithm. The additional cost in the optimized algorithm is due to early-ray termination tests, but the worst-case

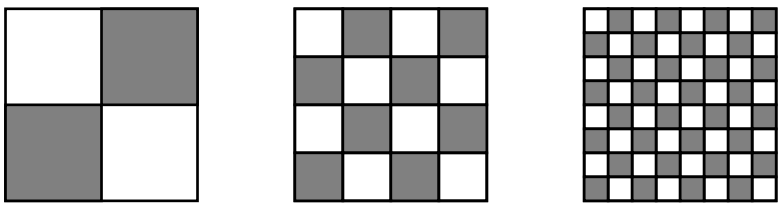


Figure 5.20: Test volumes for the voxel coherence experiment: Each volume in a series contains a checkerboard pattern constructed from blocks half as big as in the previous volume. The white blocks represent transparent voxels and the gray blocks represent voxels with a constant, non-zero opacity. The number of visible voxels is constant but the number of voxel runs increases by a factor of two in each volume.

25% performance degradation is small compared to the order-of-magnitude performance increase for more typical volumes. The only disadvantage of the shear-warp algorithm for low-coherence volumes is that the algorithm requires more memory than a brute-force algorithm since it stores three copies of every non-transparent voxel. However, as we argue later in Section 5.3.4, volumes with high percentages of visible voxels result in cluttered, fuzzy images and are therefore not useful.

### 5.3.3 The Impact of Coherence on Rendering Time

The second experiment measures how rendering time depends on the amount of coherence in the volume data. In this experiment the amount of coherence varies, but the number of transparent voxels is constant. We also keep the average voxel opacity constant, although we repeat the experiment for several opacity values.

To perform the experiment we construct several series of 256x256x256 voxel volumes containing checkerboard patterns (Figure 5.20). Half of the blocks in each checkerboard contain transparent voxels and the other half contain voxels with a constant, non-zero opacity. We use a different opacity for the non-transparent voxels in each series. For a given series, the non-transparent voxels in the different volumes all have the same opacity but we vary the size of the blocks to produce different numbers of voxel runs. For a viewing direction perpendicular to one face of the volume the number of visible voxels is constant over an entire series of volumes since exactly 50% of the voxels in each slice of each volume are transparent.

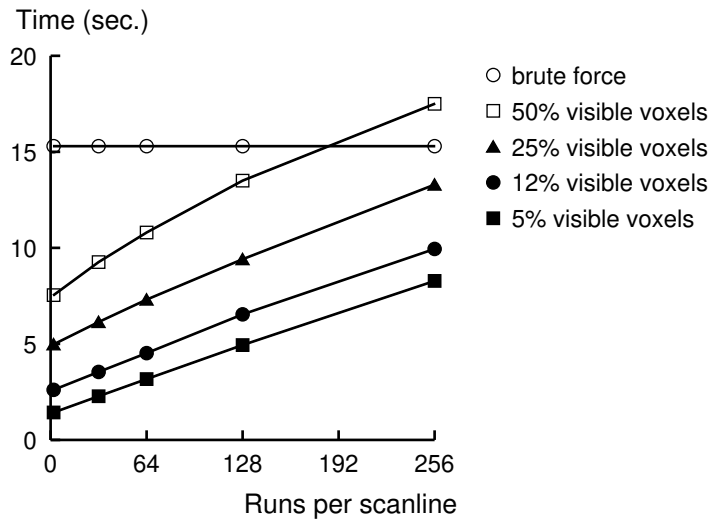



Figure 5.21: Results of the voxel coherence experiment: with a fixed number of visible voxels, the rendering time using the optimized shear-warp algorithm is proportional to the number of voxel runs in the volume. A small number of runs per scanline corresponds to high coherence.

In this experiment the number of blocks in the checkerboard pattern determines the number of voxel runs in the run-length encoded volume. We expect rendering time to vary linearly with the number of runs since the cost of rendering the voxels is the same for every volume in a given series. The number of runs per scanline includes both transparent runs and non-transparent runs, and every scanline has an even number of voxel runs as described in Section 4.1.2 (so a completely-transparent voxel scanline contains two runs, including a zero-length non-transparent run).

Figure 5.21 shows the results of the experiment. As expected, the plot of rendering time versus number of runs yields a straight line. From the slope of the line we deduce that the overhead incurred by traversing the run-length encoded data structures increases at a rate of 400 nsec./run on our test machine.

Typical data sets such as the classified CT and MR scans used earlier in this chapter have 5-15 runs/scanline. We can compute the overhead incurred by the spatial data structures by multiplying the number of runs, the runs per scanline, and the cost per run. The overhead for these data sets ranges from 130 msec. to 180 msec., a small fraction of total rendering time.

For volumes with 40% or less of the voxels visible the optimized algorithm always outperforms the brute-force algorithm, regardless of the number of voxel runs. For a volume with 50% of the voxels visible the crossover point occurs at 192 runs/scanline. In the worst case, with 256 runs/scanline (implying 50% of the voxels visible), the optimized algorithm is only 15% slower than the brute force algorithm. Furthermore, volumes with such large numbers of runs are unlikely to occur in practice: the number of runs is proportional to the depth complexity, and useful classification functions rarely produce volumes with such high depth complexity.

From the two experiments we conclude that even for volumes with worst-case coherence (only one voxel per run) the optimized shear-warp algorithm never performs significantly worse than a performance-tuned brute-force algorithm.

These results also provide us with an empirically-derived first-order cost polynomial for the shear-warp algorithm:

$$T_{\text{render}} = \frac{N_v}{800 \text{ Kvoxels/sec.}} + N_r \cdot 400 \times 10^{-9} \text{ sec.}$$

where $T_{\text{render}}$ is the time to render a volume, $N_v$ is the number of visible voxels (voxels that are neither transparent nor occluded) and $N_r$ is the number of voxel runs in the entire volume. The constants in this equation assume a 150 MHz SGI Indigo2 workstation and that shading is implemented with a gray scale lookup table. Second-order effects such as the cost of the 2D warp have not been included, so this equation is most valid when the rendering time is large.

The cost polynomial is useful for estimating whether or not a volume falls into a category that can be rendered at interactive rates. For example, the CT head data set (Figure 5.2) contains 800,000 non-transparent voxels and about half of them are visible (the average opacity is high and the depth complexity is about two) so $N_v = 400,000$. The number of runs is $N_r = 340,000$. Thus the predicted rendering time is about 0.6 sec. The CT engine data set (Figure 5.6) has about 600,000 voxels, most of which are visible since the average opacity is relatively low, and the number of runs is 300,000. Thus the predicted performance is about 0.9 sec. for a grayscale rendering.

### 5.3.4 The Role of Coherence in Visualization

In this chapter we have shown how the shear-warp algorithm exploits coherence in classified volumes to achieve fast rendering rates. Coherence is also important for producing an informative visualization: an effective classification function emphasizes structure and eliminates clutter.

This statement does not imply that the original volume data has obvious structure or coherence. Visualization is an art that requires choosing appropriate visual representations of the information in the underlying data. Haber & McNabb [1990] and Delmarcelle & Hesselink [1995] propose frameworks for scientific visualization based on three steps: data preprocessing, visualization mapping, and rendering. Visualization mapping is the process of choosing an appropriate "icon" that represents the information in a data set. An icon should efficiently convey the essential information and avoid clutter that obscures the information.

In volume rendering, the icon is the classified volume. The visualization mapping step consists of choosing a classification function and the rendering step consists of transforming the classified volume into an image. Good classification functions expose structure, thereby guaranteeing coherence that we can take advantage of. Examples of structure include opaque or translucent surfaces, fuzzy surfaces, or small, indistinct clouds in a larger structured context.

Volume rendering is often called "direct volume rendering" to emphasize the fact that the volume is not first mapped to polygons, in contrast to surface rendering. Nevertheless, volume rendering does not eliminate the fundamental need for careful visualization mapping. The human visual system cannot extract useful information from a fuzzy, translucent cloud.

Useful classifications functions produce high-coherence classified volumes, precisely the type of volume that can be rendered at interactive rates with the shear-warp algorithm.

## 5.4 Chapter Summary

The performance results in this chapter lead to several general conclusions about coherence-accelerated volume rendering algorithms. Most importantly, the cost of traversing spatial data structures is the primary overhead. The shear-warp algorithm achieves speedups of 4x or more relative to octree-accelerated ray casters by reducing this cost: the ray caster must perform analytic geometry calculations to traverse octree nodes, whereas the shear-warp algorithm simply traverses run-length encoded scanlines in storage order.

We also demonstrated that the performance of coherence-accelerated volume rendering algorithms degrades gracefully for volumes with lower coherence, and that in these cases the scanline-order shear-warp algorithm has better memory performance than a ray caster.

These performance gains come without sacrificing image quality, provided that the volume data is band-limited and properly classified. For these reasons, scanline-order volume rendering algorithms such as the shear-warp algorithm are likely to dominate.

# Chapter 6

# A Multiprocessor Volume Rendering Algorithm

The new volume rendering algorithms described in the previous chapters enable interactive visualization applications on current-generation workstations, but they are not fast enough to provide real-time frame rates (10-30 Hz). Real-time rendering allows smooth user interfaces and instantaneous feedback when exploring a data set. Furthermore, a fast update rate during a rotation sequence provides strong depth cues that can greatly aid the human visual system to understand a data set.

Four approaches have been proposed in the literature to achieve high frame rates for volume rendering. The first approach is to trade off image quality for speed, for instance by using low-resolution versions of the data during user interaction [Laur & Hanrahan 1991]. Such methods allow rough manipulation of the data but the loss of resolution eliminates many perceptual cues. The second approach is to rely on special-purpose hardware designed either for graphics applications or specifically for volume rendering. Recent work in this area includes volume rendering algorithms for the Pixel-Planes 5 graphics architecture [Yoo et al. 1992] and for 3D texture mapping hardware [Cabral et al. 1994], a custom accelerator board for volume rendering [Knittel & Straßer 1994], and the Cube-3 architecture [Pfister et al. 1994]. Designs such as these can deliver high-quality images at high frame rates; however, algorithms that are implemented in hardware lose flexibility.

The third approach is to parallelize a brute-force volume rendering algorithm for a

large multiprocessor. Many volume rendering algorithms have been reported for massively-parallel SIMD multiprocessors [Cameron & Undrill 1992, Schröder & Stoll 1992, Vézina et al. 1992, Hsu 1993, Wittenbrink & Somani 1993]. These architectures perform best when communication patterns are regular and uniform, so the best speedups are achieved with simple brute-force rendering algorithms. Brute-force parallel algorithms for large MIMD machines have also been reported [Ma et al. 1993, Corrie & Mackerras 1993b, Montani et al. 1992]. This solution requires a large number of processors to achieve acceptable frame rates.

The fourth approach is to parallelize a fast rendering algorithm that relies on algorithmic optimizations, such as spatial data structures, to reduce the computational costs [Nieh & Levoy 1992, Neumann 1993]. A MIMD multiprocessor must be used since intricate algorithms with many special cases do not map well onto SIMD arrays. The disadvantages of this approach are that the optimizations are data-dependent and preprocessing is often necessary. However, less-expensive hardware is required to achieve the same frame rates as with the other approaches, so using an optimized algorithm is more cost effective. This is the approach we adopt.

In this chapter we describe a new parallel volume rendering algorithm that performs well due to three factors. First, the computational costs are lower than previously-published parallel algorithms because the algorithm is based on the serial shear-warp rendering algorithms described in Chapter 4. Furthermore, we have designed our parallelization using an image partition that takes full advantage of the optimizations in the rendering algorithm.

Second, we control synchronization and idle waiting costs by using the dynamic load balancing method described by Nieh & Levoy [1992]. Our task partition also minimizes the number of synchronization events, thereby making load balancing easier.

Third, our algorithm has low data communication costs because it is implemented for cache-coherent shared-memory multiprocessors. This type of parallel architecture supports fine-grain low-latency communication, unlike message-passing machines which require expensive operating-system intervention for every message and therefore demand careful data distribution [Neumann 1993, Montani et al. 1992, Corrie & Mackerras 1993b].

We have implemented our algorithm for two different MIMD shared-memory architectures: the Silicon Graphics Challenge multiprocessor, which is a bus-based architecture, and

the Stanford DASH Multiprocessor, a scalable distributed shared-memory multiprocessor. Our implementation on a 16 processor Challenge renders a $256^3$ volume at 13 Hz and a $128^3$ volume at 31 Hz. These rates are comparable to results reported for special-purpose graphics hardware [Yoo et al. 1992] and exceed the best reported results for massively-parallel SIMD machines [Hsu 1993, Wittenbrink & Somani 1993] and much larger MIMD message-passing machines [Neumann 1993]. On a 32-processor configuration of DASH the algorithm renders a $256^3$ volume at 5.9 Hz, six times faster than the results for DASH reported by Nieh & Levoy [1992].

Our performance results lead to several conclusions. First, as others have noted, data redistribution and communication costs do not dominate rendering time on a shared-memory machine [Nieh & Levoy 1992]. Second, we find that cache locality requirements impose a limit on parallelism in volume rendering algorithms. Caches hide latency by exploiting locality of reference, but locality is destroyed if the task size is too small. In our experiments our algorithm approaches this limit, and we believe that for any volume rendering algorithm based on an image partition shared-memory machines with hundreds of processors would be useful only for very large data sets.

Section 6.1 of this chapter describes the multiprocessor algorithm and the design tradeoffs we considered. Section 6.2 describes the architectures of the Challenge and DASH multiprocessors and the details of our implementation for these machines. We then present performance results for the parallel algorithm in Section 6.3 and discuss the limitations on parallelism. Finally we discuss related work and our conclusions.

## 6.1 Multiprocessor Rendering Algorithm

Designing a parallel algorithm for a shared-memory multiprocessor consists of choosing a partition of the computation into tasks for each processor and choosing appropriate synchronization mechanisms. The serial algorithm contains three phases, each of which must be parallelized: computing the shading lookup table, projecting the volume into the intermediate image, and warping the intermediate image. We begin by focusing on the projection of the volume data since that phase dominates the cost of the serial algorithm, and then we consider the issue of data distribution and the complete parallel algorithm.

### 6.1.1 Image and Object Partitions

The two general types of task partitions for parallel volume rendering algorithms are object partitions and image partitions. In an object partition each processor is assigned a specific subset of the volume data [Neumann 1993, Ma et al. 1993]. The processors first resample and composite the voxels assigned to them, and then the partially-computed image pixels must be composited with results from other processors to form the final image pixel values. In contrast, in an image partition each processor's task is to compute a specific portion of the image [Nieh & Levoy 1992, Corrie & Mackerras 1993b]. Each image pixel is computed by only one processor, but the volume data must be moved to different processors as the viewing transformation changes. The choice of an image partition versus an object partition is independent of the choice of an image-order or object-order rendering algorithm.

Neumann [1993] shows that object partitions require less data redistribution than image partitions because only the image is communicated, while the much larger 3D volume is never redistributed. On a message-passing machine the cost of communication is high due to operating-system involvement in every message, so object partitions appear attractive. However, on shared-memory machines the cost of communication may not be dominant [Nieh & Levoy 1992] and therefore other factors are also important. One disadvantage of object partitions is that early ray termination becomes less effective as the number of processors increases. An increasing percentage of the processors are assigned occluded portions of the data and therefore either perform unnecessary computation or have no work to do. Early ray termination can result in 2-3x performance improvements as we saw in Chapter 5, so 50% or more of the processing power of a large parallel machine may be wasted with an object partition.

A second potential disadvantage of object partitions is that increased synchronization is necessary. In an object partition multiple processors must write to the same portion of the image, so the processors must explicitly synchronize these writes to avoid erroneous results. Less synchronization is required in an image partition because only one processor writes to a given portion of the image; the only shared data structure is the volume data, which is read-only and therefore requires no synchronization.

Algorithms that require fine-grain synchronization often suffer from high percentages of

idle waiting time and load-balancing difficulties. For example, in an early attempt to parallelize the shear-warp algorithm we used an object partition in which each processor was assigned a portion of each slice of voxels [Agrawala 1993]. To maintain the required compositing order all of the processors were required to synchronize at a barrier after each slice had been composited into the intermediate image. The result was poor speedup due primarily to idle waiting time at the barriers. Load balancing could not compensate because of the small amount of work between each synchronization event. Challinger [1992] reports results for an object partition on a shared-memory machine that agree with these conclusions.

The binary-swap image compositing algorithm proposed by Ma et al. [1993] also appears to have high synchronization overhead: the image-compositing phase suffers from low speedups. Ma's overall parallel volume rendering algorithm has good speedups because the cost of compositing is far less than the cost of rendering, but the lower cost of rendering in our algorithm makes the cost of compositing more significant.

We therefore choose an image partition, which requires no synchronization until the entire volume has been composited.

### 6.1.2 Task Shape

There are several choices for the unit of work in an image partition: individual pixels, scanlines of pixels [Challinger 1992, Yoo et al. 1991], or rectangular tiles [Nieh & Levoy 1992]. Generally, tile-shaped tasks are best in order to maximize temporal cache locality [Nieh & Levoy 1992, Corrie & Mackerras 1993b]. However, two factors unique to the run-length encoded data structures in the shear-warp algorithm lead us to partition the intermediate image into groups of contiguous scanlines. First, the run-length encoded volume data structure makes it difficult to find the voxels associated with an image tile. Finding the data associated with an arbitrary voxel requires decoding the run-length encoding for all of the voxels preceding it. In a scanline-based task partition we can precompute pointers to the beginning of each scanline of voxel data to eliminate all decoding overhead, as mentioned in Section 4.1.2. In a tile-based task partition pointers would be required for every voxel since in an object-order rendering algorithm an image tile can potentially map to any voxel (depending on the view transformation).

The second reason for choosing scanline-shaped tasks instead of tiles is to maximize spatial locality in both the intermediate image and the run-length encoded volume data. If the compression ratio in the run-length encoded volume is high then several scanlines of voxels may fit in a single cache line and it is advantageous to process a contiguous group of scanlines as a unit. The optimal number of scanlines in a task depends upon the cache line size and will be discussed further in Section 6.3.

### 6.1.3 Load Balancing

Given that the fundamental unit of work is a group of contiguous scanlines of the intermediate image, we must choose how to assign the scanlines to processors while minimizing load imbalances. We consider three options: a static contiguous partition, a static interleaved partition, and a dynamic partition.

In a *static contiguous partition* the image is divided into large fixed blocks of scanlines, one block per processor. Such a scheme works well only if the execution time for each block can be accurately predicted so that each processor has the same amount of work. However, view-dependent variations in the effectiveness of the coherence optimizations makes it difficult to predict performance. Our experiments with a simple measure of work, the number of nonzero voxels in a scanline, showed it to be an effective predictor of execution time [Agrawala 1993].

In a *static interleaved partition* the image is divided into small blocks of scanlines and the blocks are distributed in round-robin order to the processors. This scheme introduces a tradeoff between load balancing and memory performance. If the image is divided into a small number of large tasks then it is likely that some processors will be assigned more expensive tasks than others and the load balance will be poor. On the other hand, if the image is divided into a large number of small tasks then spatial locality is reduced and memory overhead rises.

In a *dynamic partition* tasks may be reassigned to different processors during rendering to optimize the load balance. This method results in the best load balance, but care must be taken to avoid overhead caused by redistributing tasks. The method we consider employs a distributed task queue and dynamic task stealing [Nieh & Levoy 1992]. The image

is first partitioned using one of the static partitioning methods and the tasks are placed on task queues associated with each processor. If a processor completes all of its tasks then it may "steal" a task from one of the other queues. Distributed queues are used rather than a single centralized queue in order to avoid a bottleneck.

We implemented all of the above methods and found that dynamic task stealing with a moderately interleaved initial task distribution performed best, as our performance results will show. The interleaved partition increases the probability of a good initial load balance, and dynamic task stealing compensates in cases where the initial partition is not optimal.

### 6.1.4 Data Distribution

Explicit data distribution is a difficult problem when an image partition is used because the portion of the volume required by a particular processor depends on the viewpoint. One solution is to replicate the data on every processor's memory, but this design severely limits the maximum size of the volume. However, explicit data distribution is not necessary on a shared memory machine with caches. Because all of the processors share the same address space, data in any memory can be accessed transparently by any processor. The performance penalty for remote accesses is small provided the cache hit rate is sufficiently high and there are no memory hot spots. To avoid hot spotting we use a round-robin distribution of all large data structures to the different physical memories. On a shared-memory machine the hardware caches automatically replicate the subset of the data required by multiple processors. The memory performance of the algorithm will be analyzed in the results section.

### 6.1.5 Overall Algorithm

The pseudo-code in Figure 6.1 shows the flow of control for each processor in the parallel algorithm. First, each processor selects a portion of a lookup table used for shading voxels (parallelized with a static contiguous task partition) and then waits at a barrier for the other processors to finish. Next, each processor computes a portion of the intermediate image (image_scans) by choosing a task from one of the task queues (FindTask()). Each task entails

```
1   procedure MultiRenderVolume()
        ComputeShadingLookupTable(); { static contiguous partition }
        barrier
        while (not done)
5           image_scans = FindTask(); { dynamic partition }
            if (image_scans ≠ NIL)
                Clear(image_scans);
                foreach (voxel_slice from front to back)
                    voxel_scans = Intersect(image_scans, voxel_slice);
10                  image_scans = image_scans over Resample(Shade(voxel_scans)));
                end
            else
                done = 1;
            endif
15      end
        barrier
        image = Warp(); { static interleaved partition }
        barrier
        Display(image);
20  end
```

Figure 6.1: Pseudo-code for the multiprocessor algorithm.

looping through the slices of the volume, finding the voxel scanlines (voxel_scans) that intersect the assigned portion of the intermediate image, and then resampling and compositing the voxel scanlines. When all of the tasks have been completed each processor synchronizes at the second barrier, warps a portion of the image (parallelized with a static interleaved partition consisting of rectangular tiles of the final image), and synchronizes again. Then the image is displayed.

Apart from the three barriers, the only synchronization required is a set of locks to govern access to the task queues in the FindTask() function. Each queue has its own lock, and processors request tasks from their own queue except when stealing, so the contention for the locks is negligible.

| Architecture | SGI Challenge | Stanford DASH |
|---|---|---|
| Nodes | 16 | 32 |
| Processor | 150 MHz R4400 | 33 MHz R3000 |
| 1st Level D-Cache | 16 Kbytes | 64 Kbytes |
| 2nd Level D-Cache | 1 Mbytes | 256 Kbytes |
| 2nd Level Line Size | 128 bytes | 16 bytes |
| Interconnect Type | Global Bus | 2D Mesh |
| Interconnect Bandwidth | 1.2 Gbytes/sec. | 960 Mbytes/sec. |

Table 6.1: Characteristics of the parallel architectures used for the performance measurements. Both machines are shared-memory multiprocessors with hardware cache-coherence.

## 6.2 Implementation

### 6.2.1 Hardware Architectures

We have implemented the algorithm for two architectures, the Silicon Graphics Challenge and the Stanford DASH Multiprocessor. Table 6.1 lists the characteristics of the machines.

The SGI Challenge is a bus-based symmetric shared-memory multiprocessor. A global bus allows any processor to access any memory bank with the same latency. DASH is an experimental prototype of a scalable distributed shared-memory architecture. The machine consists of processor clusters connected by a 2D mesh network (Figure 6.2). Each cluster contains four CPUs and a memory module on a local bus. The processor caches are kept consistent by the hardware using a distributed directory-based protocol [Lenoski et al. 1993]. The mesh network in this design is scalable, whereas in a bus-based design the bus saturates if too many processors are attached.

From the parallel programmer's perspective, both architectures implement a shared-memory model. However, the memory hierarchies are significantly different. The secondary cache on the Challenge has a long cache line size (128 bytes) so that programs with good spatial locality benefit from a prefetching effect which helps to mask the latency of main memory accesses. The short cache lines on DASH lessen the benefit of spatial locality. The caches on DASH are also much smaller. Finally, since DASH is a distributed shared-memory machine the memory latency increases if an access must be serviced by a remote memory.

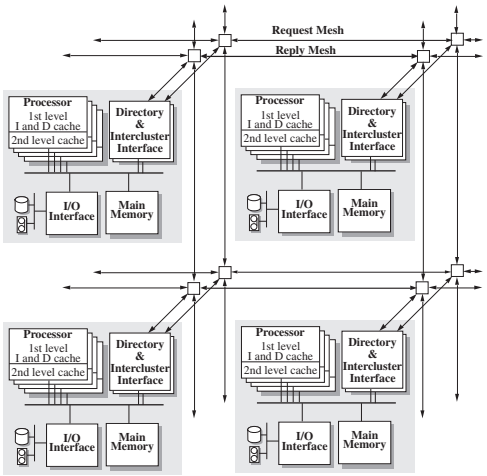



Figure 6.2: Block diagram of a 2 x 2 DASH system (courtesy of Daniel Lenoski, Computer Systems Laboratory, Stanford University).

### 6.2.2 Software Implementation

We implemented the algorithm using the C language and the ANL parallel programming macro package [Boyle et al. 1987]. Because of the shared-memory programming model the parallel code is almost identical to the original serial code. The only additions are functions to compute task partitions and a few calls to the synchronization primitives. The same source code runs without modification on both the Challenge and DASH architectures and is portable to other shared-memory machines.

All of the large data structures, including the run-length encoded volume, the intermediate image, the final image, and the shading lookup table are allocated in the shared address space. In order to avoid hot spotting, on DASH we distribute the shared data structures to the different memories by assigning pages (4096 byte blocks) to memories in round-robin fashion. On the Challenge the memory system is eight-way interleaved, so round-robin distribution of the data structures is performed automatically at cache-line granularity.

The memory requirements of our implementation are four bytes per non-transparent voxel (including a precomputed opacity and normal vector for shading) and twelve bytes per pixel for a gray scale intermediate image (including an intensity and an opacity stored in floating point format, and storage for the run lengths). Three copies of the volume data are stored in memory, one for each possible transpose, as in the serial algorithm.

We shade the voxels using the Phong illumination model implemented with an 8192-entry lookup table (further described in Chapter 7). The table is recomputed at the start of each frame to allow view-dependent directional lighting effects.

The task sizes for the compositing phase and the warping phase of the algorithm are parameters which can be changed at run-time. The task sizes must be adjusted to find the optimal tradeoff between load balancing and memory overhead on each architecture.

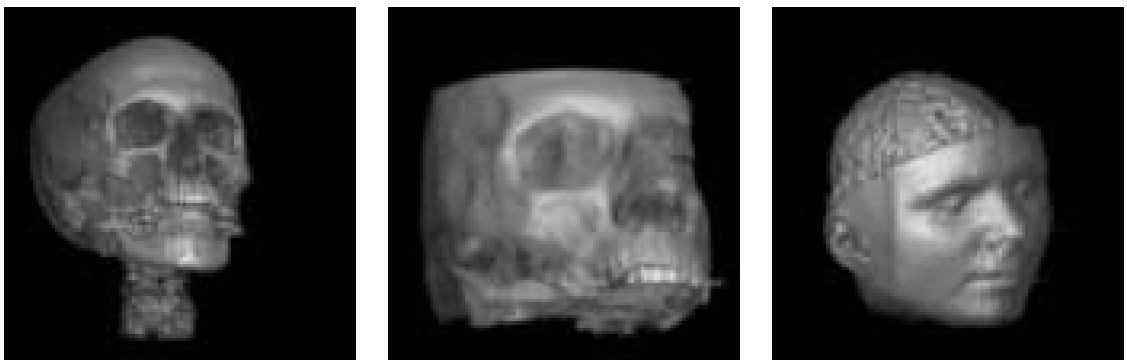
Our current implementation only supports parallel projections and preclassified volumes. The perspective version of the rendering algorithm and the fast classification algorithm can be parallelized using the same method described here.

## 6.3 Results

We tested our algorithm using two data sets: a 256x256x225 voxel CT scan of a human head and a 128x128x109 voxel MR scan of a head. These are the "head" and "brainsmall" data sets described in Table 5.1. Figure 6.3 shows renderings of the test data sets. The images are gray scale and contain $256^2$ pixels (although the rendering times are relatively independent of the image size, just as in the serial algorithm). On the Challenge we parallelized the image warp using tiles with 8x256 pixels and on DASH we used 16x16 tiles. We used an early ray termination opacity cutoff of 95%. The total memory usage of the parallel algorithm is the same as the serial algorithm (Table 5.3). Figure 6.4 illustrates a graphical interface that allows the user to rotate the volume in real time by dragging on the rendered image.

### 6.3.1 Rendering Rates

Tables 6.2 and 6.3 give our performance results on the Challenge and on DASH. These results do not include image display and preprocessing to compute the run-length encoded volume. On the Challenge our display routine (implemented with Tcl/Tk [Ousterhout 1994] and GL [Silicon Graphics 1991]) requires 3 msec. to display a $256^2$ image. The preprocessing time is 65 sec. on a single processor for the $256^3$ volume. The frame rates in the tables are averages for a 180-frame animation with a two-degree rotation of the volume between



(a) (b) (c)

Figure 6.3: Test data sets for the multiprocessor algorithm: (a) CT head data set (256 x 256 x 225 voxels) (b) subvolume extracted from the CT head data set (128 x 128 x 128 voxels) (c) MR brain data set (128 x 128 x 109 voxels).

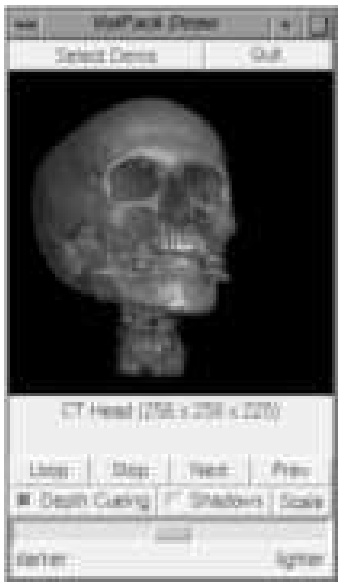


Figure 6.4: The interactive user interface for the multiprocessor algorithm. The user interface allows the volume to be rotated in real time by direct manipulation with a mouse. Sliders (not shown) allow interactive adjustment of shading and lighting parameters.

| Data set | Processors | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| CT Head (128x128x128) | | | | | |
| Frame Rate (Hz) | 2.5 | 5.3 | 9.9 | 18 | 31 |
| Task Size (scanlines) | 256 | 8 | 8 | 8 | 8 |
| CT Head (256x256x225) | | | | | |
| Frame Rate (Hz) | 1.1 | 2.2 | 4.3 | 7.5 | 13 |
| Task Size (scanlines) | 512 | 20 | 20 | 20 | 13 |

Table 6.2: Rendering rates and tasks sizes on the SGI Challenge. The task size is the number of intermediate image scanlines per task for the projection stage of the parallel algorithm.

| Data set | Processors | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 |
| MRI Brain (128x128x109) | | | | | | |
| Frame Rate (Hz) | 1.2 | 2.4 | 4.4 | 7.0 | 12 | 18 |
| Task Size (scanlines) | 32 | 20 | 12 | 12 | 5 | 3 |
| CT Head (256x256x225) | | | | | | |
| Frame Rate (Hz) | 0.33 | 0.67 | 1.3 | 2.1 | 3.4 | 5.9 |
| Task Size (scanlines) | 32 | 20 | 12 | 12 | 8 | 3 |

Table 6.3: Rendering rates and task sizes on the Stanford DASH Multiprocessor.

each frame. The fastest rates are achieved with the Challenge (which has more total computational power than DASH): 13 Hz for the $256^3$ volume.

The speedup curves for the algorithm are not ideal, as will be discussed later in this section. Nevertheless, these rendering rates are among the fastest reported. In comparison to the results reported by Nieh & Levoy [1992] on the DASH architecture, our parallel algorithm is faster by a factor of six for a 32-processor machine. It is also competitive with results for much more expensive massively parallel processors and special-purpose hardware. The multiprocessor shear-warp algorithm achieves these results only for data sets with high coherence, but as we argued earlier in Section 5.3.4 classification functions that produce low coherence also produce poor visualizations.

Color renderings take roughly twice as long as gray scale renderings, just as in the serial algorithm. We rerendered the color images shown in Figure 5.5–5.7 on DASH with 32 processors. The CT head (256 x 256 x 225 voxels) was rendered at a rate of 2.0 Hz, the CT engine (256 x 256 x 110 voxels) was rendered at a rate of 2.8 Hz, and the CT abdomen (256

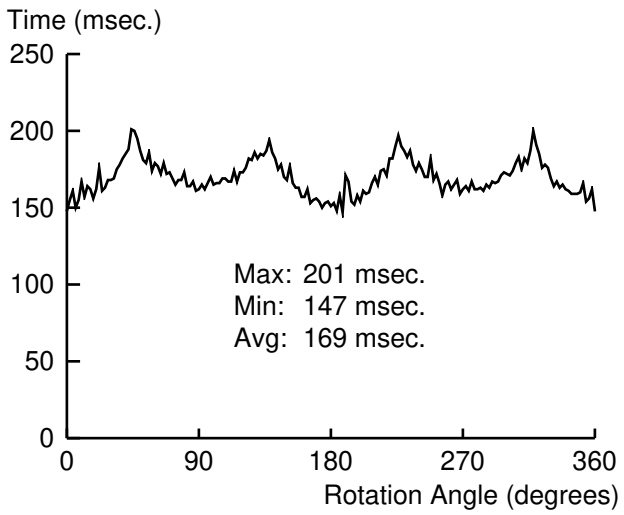



Figure 6.5: Rendering time for the $256^3$ gray scale head on DASH with 32 processors as the viewing angle changes.

x 256 x 159 voxels) was rendered at a rate of 2.4 Hz.

Figure 6.5 shows how the rendering time varies with viewing angle during a continuous 360-degree rotation sequence. The variations are due to data-dependent changes in image complexity. There is no discontinuous jump in rendering time when the volume is rotated past a 45-degree point, even though the renderer must switch to a different transposed copy of the run-length encoded volume. Neither is there any visual discontinuity in the rendered images. The rendering time does not change if the rotation angle between frames is increased, indicating that the performance results do not depend on cache coherence between frames.

### 6.3.2 Performance Limits on the Challenge Multiprocessor

Figure 6.6 shows the speedup curve for the Challenge multiprocessor. While the algorithm achieves high frame rates, the speedup for 16 processors is only about 12 (an efficiency of 75%). There are two potential sources of overhead leading to the non-linear speedup: memory stall time caused by cache misses, and processor idle time caused by load imbalances.

Figure 6.7 illustrates how the task size can be adjusted to minimize the total overhead by balancing the tradeoff between memory stall time and processor idle time. The figure

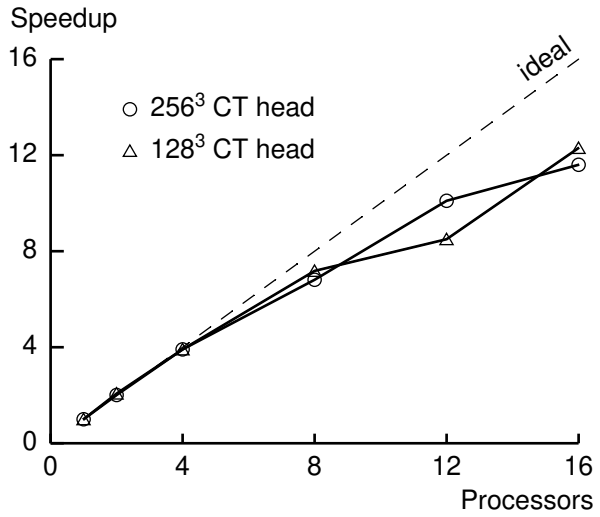



Figure 6.6: Speedup curve on the Challenge multiprocessor.

shows a breakdown of the overheads for a fixed number of processors (16) as the task size is varied. The total length of each bar represents the sum of the time spent by all of the processors to render a single frame (wall clock time would be found by dividing the length of the bar by the number of processors). Each bar is broken down into sections representing computation time, memory stall time, and idle time (synchronization overhead). Idle time was measured using high-resolution interval timers at each barrier, computation time was measured using a cycle-counting profiler (Pixie [Smith 1991]), and memory stalls account for the remaining time. The figure shows that small tasks result in larger memory overhead while large tasks result in poor load balancing.

The optimal task size depends on the data set and the number of processors (see Table 6.2). Figure 6.8 shows the magnitudes of the overheads for the $256^3$ CT head data set with the optimal task size on a range of machine sizes. There are two observations to make from the figure. First, with optimal task sizes the memory overhead for the 16 processor run is only 9% more than the overhead for the uniprocessor algorithm. This result indicates that data redistribution and communication costs do not dominate rendering time.

The second observation is that the overhead due to idle time does increase. But from Figure 6.7 we know that the load balancing algorithm is not the problem since the idle time can be reduced to a negligible amount by reducing the task size. In fact, the decreasing efficiency

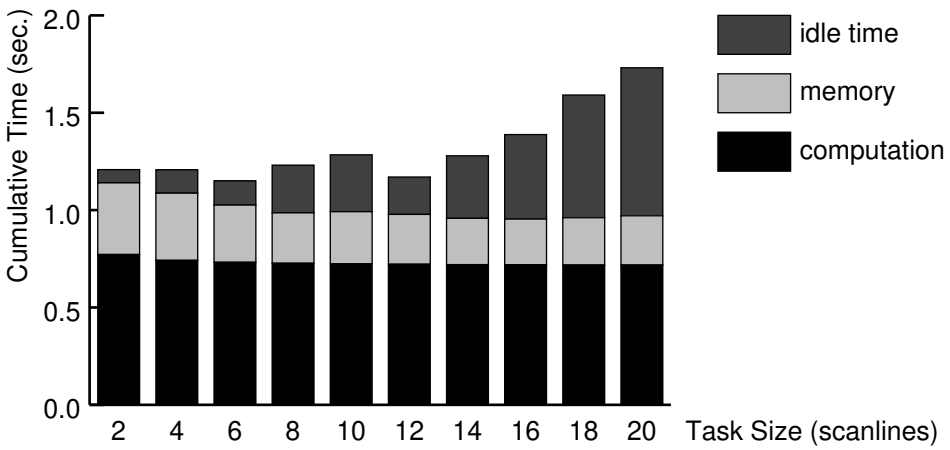



Figure 6.7: Breakdown of execution time into computation cycles, memory overhead, and idle time due to synchronization on a 16-processor Challenge as the task size varies. The time is cumulative over all processors. There is a tradeoff between the two types of overhead.

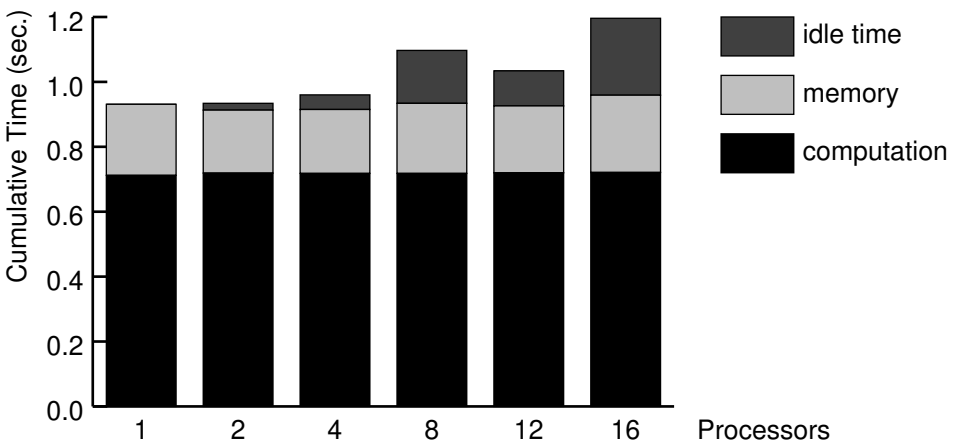



Figure 6.8: Breakdown of execution time into computation cycles, memory overhead, and idle time on the Challenge as the number of processors varies.

is due to a limit on the available parallelism in our task partitioning algorithm. As more processors are added the intermediate image must be partitioned into a larger number of tasks to achieve good load balancing, but as a result the tasks are smaller and memory overheads rise. At some point there is no net gain from making the tasks smaller. Beyond that point, since the tasks are not made smaller the overhead due to load imbalances increases as more processors are used, and eventually the speedup curve becomes flat. Therefore, cache locality requirements impose a limit on the maximum useful number of parallel tasks.

We suspect that other algorithms for volume rendering on shared-memory machines have similar limits. For example, Nieh's parallel ray caster benefits from good temporal locality since adjacent rays access data from the same cache lines [Nieh & Levoy 1992, Singh et al. 1994]. The algorithm uses an image partition, so as the number of parallel tasks increases the size of the image tiles assigned to a processor must decrease. If a large number of processors is used the likely result is a degradation in temporal locality and an increase in memory overhead.

However, our goal is to achieve interactive rendering rates, and on the Challenge our algorithm achieves that goal for moderately-large data sets. The limit on available parallelism is less of a problem for larger data sets since they can be divided into more tasks, so larger machines are useful with our algorithm provided that larger data sets are rendered. Another factor affecting the available parallelism is the architecture of the memory system, which determines the minimum practical task size. DASH, which we consider next, has a much smaller optimal task size and therefore a larger number of processors can be used. We will discuss memory performance in more detail later in this section.

### 6.3.3 Performance Limits on the DASH Multiprocessor

Figure 6.9 shows the speedup curve for the DASH multiprocessor. Compared to the results for the Challenge, the efficiency on DASH is much lower (60% for 16 processors, and about 50% for 32 processors). The source of the overhead, shown in Figure 6.10, is also different: memory stall time is the dominant overhead. In the rest of this section we will show that on DASH the memory overhead increases due to an increase in remote memory references,

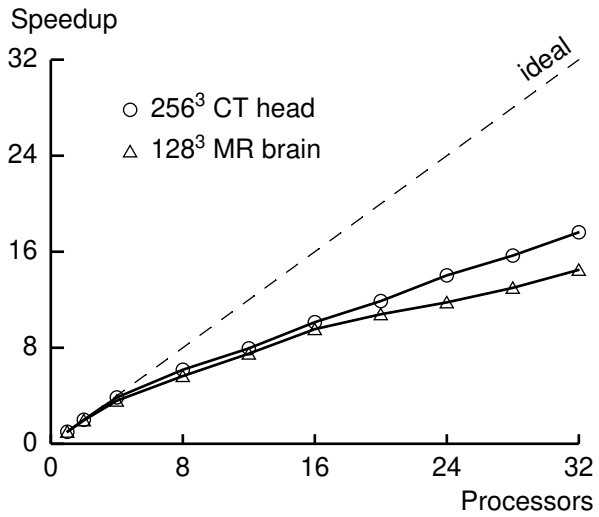



Figure 6.9: Speedup curve on the DASH multiprocessor.

indicating that the caches are not completely hiding the latency of memory. In the next section we will analyze the cache behavior in more detail and show that the performance on the Challenge is better because the caches are larger, so the working set fits, and the cache line size is longer, so the benefit of spatial locality is greater.

The first step to understanding the increased memory overhead on DASH is to recall the structure of the memory hierarchy: the machine consists of clusters of four processors connected by a bus to a local memory, and accesses to memory on remote clusters require communication via a 2D mesh network. The latency for remote accesses is roughly four times longer than for local accesses. This fact explains the jump in memory overhead from the four-processor run to the eight-processor run: the average latency for memory accesses rises abruptly. As the number of processors increases further, a larger fraction of the data structures required by a particular processor reside on remote memory and so the average latency increases even further. We used a hardware performance monitor to count the number of remote references and found that it does increase slowly: when the number of processors is doubled from 16 to 32 the number of remote memory references increases by about 10%. To prove that the memory overhead is due to an increasing fraction of remote references and not to an increasing total number of memory references, we moved all data structures to remote clusters. In this experiment the memory latency is almost constant since almost

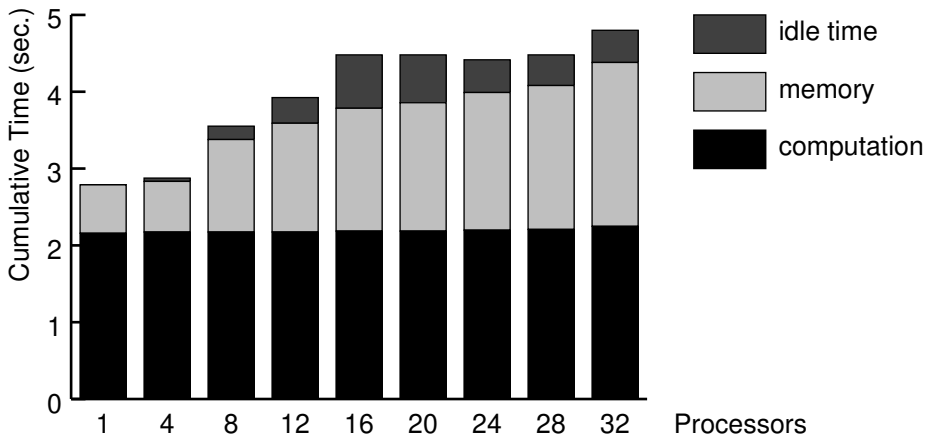



Figure 6.10: Breakdown of execution time into computation cycles, memory overhead, and idle time on DASH as the number of processors varies.

every reference is remote. As expected, we found that the memory overhead also became nearly constant (the algorithm achieved efficiencies of 85-90% for 16 processors).

Unlike our results on the Challenge, the memory overhead cannot be significantly reduced by increasing the task size. The reason is that on DASH the cache line size is much shorter (16 bytes instead of 128 bytes on the Challenge), so spatial locality is less important. The optimal task sizes shown in Table 6.3 are significantly smaller than for the Challenge. However, because the optimal task sizes are smaller, more parallelism is available and more processors can be used.

From this line of reasoning we conclude that the sub-linear speedup on DASH is caused by the caches not completely hiding the latency of remote memory accesses, and by the increasing average latency as the data structures are spread over a larger number of processors. Next, we consider the reasons for sub-optimal cache performance.

### 6.3.4 Memory Performance

Nieh & Levoy [1992] report that memory overhead for an optimized ray caster on shared-memory multiprocessors such as DASH is a small fraction of execution time, partly due to the effectiveness of the caches and partly due to the high ratio of computation to memory

accesses. Our algorithm eliminates a considerable fraction of the computation time associated with traversing data structures, so we might expect the impact of memory overhead to be more important. To investigate further, and to explain the different results on the two machines, we used a memory system simulation tool [Martonosi et al. 1992] to qualitatively understand the memory behavior. The simulator does not model all of the details of the memory system so it cannot be used to predict miss rates on the real machines, but it does allow us to determine which data structures miss most often and the dominant causes of the misses.

With the cache parameters adjusted to roughly approximate the caches at the cluster level on DASH, the simulator indicates two dominant causes of cache misses. The first cause is communication (invalidation misses) resulting from the redistribution of the intermediate image and to a lesser extent the shading lookup table. These misses result from inherent communication in the parallel algorithm. The intermediate image is redistributed during the 2D warp since each processor is responsible for assembling rectangular pieces of the final image from pieces of the intermediate image computed by other processors. It is difficult to reduce this cost since a warp inherently requires moving pixels to different parts of the image. The communication cost slowly increases as the number of processors increases because smaller fractions of the intermediate image and the shading table are computed on the same processor as they are used.

When the cache line size is increased to 128 bytes (to match the line size on the Challenge) the simulator reports that the number of misses due to communication drops by more than a factor of five. This result indicates that our algorithm has significant spatial locality. Therefore, on architectures with long cache lines the cost of communication misses can be masked by the caches.

The second source of misses on DASH is from capacity misses, which occur when the cache is not large enough to capture the working set. The second-level caches on DASH (256 Kbytes) are small by today's standards. Increasing the cache size in the simulation to 1 Mbyte per processor (to match the second-level cache size on the Challenge) causes a significant drop in the number of capacity misses, so they should have a smaller effect on the Challenge architecture. The simulator also shows that as the number of processors increases the number of capacity misses decreases due to the increased amount of cache in

the machine.

To summarize the analysis for DASH, when the increasing cost of communication and the decreasing cost of capacity misses are combined the net result is that the total number of remote memory references increases slowly as the number of processors increases. This result from the simulator matches the increase in remote references observed with the hardware performance monitor. The memory overhead increases at a slower rate than the processing power, so the speedup continues to rise as more processors are used, but the efficiency goes down. In comparison to Nieh's parallel ray caster [Nieh & Levoy 1992], our algorithm traverses the volume data in a different order that results in a reduction in temporal locality and a larger working set.

On the Challenge, the combination of larger caches (which can capture a larger working set) and longer cache lines (which capitalize on spatial locality to reduce the number of misses) result in good memory performance. The current trend in processor design is towards increasing memory latency with longer cache lines to amortize that latency over many references. Scanline-based algorithms with good spatial locality, such as the shear-warp algorithm, can effectively make use of the prefetching effect of long cache lines.

As a final experiment to determine the impact of volume data redistribution, we eliminated the cost of redistribution by replicating the $128^3$ MR volume on all of the clusters of DASH. The rendering time decreased by less than 5% with two processors and by less than 1% with 12 or more processors, indicating that volume redistribution costs are not the limiting performance factor on shared-memory multiprocessors. The hardware handles communication and caching of the volume data automatically with no performance penalty. Unfortunately the caches are less effective at hiding the latency of inherent communication in the algorithm, i.e. redistribution of the intermediate image, so the memory overhead is non-negligible despite the low cost of accessing volume data.

### 6.3.5 Load Balancing

Table 6.4 compares the performance of the different task partitioning algorithms we implemented. The load imbalance is measured by timing the resampling and projection computation (between the first and second barriers in Figure 6.1) on each processor and then

| Task Assignment Algorithm | Load Imbalance | Rendering Time (msec.) |
|---|---|---|
| round-robin | | |
| stealing | 7% | 149 |
| no stealing | 18% | 160 |
| contiguous | | |
| stealing | 32% | 187 |
| no stealing | 169% | 342 |

Table 6.4: Comparison of load balancing algorithms for the $256^3$ head on DASH with 32 processors. Round-robin task assignment with dynamic stealing performs best.

computing the average time and the maximum deviation from the average time. The numbers reported in the table are the maximum deviations as a percentage of the average time. A value of 0% indicates that all processors take the same amount of time, and a value of 100% indicates that at least one processor takes 100% longer than the average time.

The best results are obtained with a round-robin assignment and stealing (dynamic repartitioning); the load imbalance is almost negligible. When stealing is disabled the load balance increases significantly, although overall rendering time increases by only 7%. This result indicates that the initial round-robin assignment is close to optimal, but stealing is easy enough to implement on a shared-memory machine that even a small increase in performance makes it worthwhile. The contiguous partition is a sub-optimal initial partition, so stealing makes a much bigger difference. But even with stealing, the contiguous partition is worse than the round-robin partition. We theorize that because the round-robin partition reduces the number of tasks that are stolen there is somewhat more temporal locality and fewer cache misses.

### 6.3.6 Related Work

We have argued that an image partition coupled with hardware caching results in low memory overhead. Most volume rendering algorithms for message-passing machines use object

partitions to minimize communication costs [Neumann 1993]. However, Corrie & Mackerras [1993b] propose an algorithm with an image partition that uses software caching to exploit data reference locality on a message-passing machine. The target application is non-interactive rendering of very large data sets, so the granularity of communication is large (data blocks of 8 Kbytes or larger) and the overhead of software caching is reasonable. But for moderately-sized data sets and real-time rendering rates the granularity of communication is necessarily smaller and the overhead of software-controlled caching is likely to be high. Furthermore, the method does not reduce the increase in message traffic incurred if a dynamic load balancing algorithm is used. In contrast, the hardware caches on a shared-memory multiprocessor automatically hide most of the cost of communication in our algorithm.

Our results corroborate the conclusions of Nieh & Levoy [1992] and Singh et al. [1994], both of which demonstrate the success of image partitions on shared-memory multiprocessors.

## 6.4 Chapter Summary

The multiprocessor shear-warp algorithm is capable of rendering $256^3$ volumes at rates exceeding 10 Hz on readily-available hardware. We have achieved these speeds by using an optimized rendering algorithm with low computational costs, and an image partition which can take full advantage of the optimizations and which has lower synchronization costs than object partitions. The memory performance of the algorithm is good because we use cache-coherent shared-memory architectures. On the Challenge, which has a long cache line, the spatial locality in our task partition results in negligible memory overhead. Even on DASH, with a much shorter cache line and small caches, the memory overhead is acceptable. Shared-memory architectures work well for volume rendering because they efficiently support fine-grain communication, and have the added benefit of a simple programming model: the parallel code is virtually identical to the serial code.

One disadvantage of the algorithm described in this chapter is the preprocessing step required to run-length encode the volume whenever the user reclassifies the volume (i.e.

reassigns voxel opacities). However, the same parallelization can be used for the fast-classification algorithm, allowing real-time classification and rendering. The perspective projection algorithm can also be parallelized in the same way. Neither of these extensions has yet been implemented, but we expect a factor of 2-3x slowdown compared to the results presented here.

Another interesting area for future work is a more detailed analysis of the memory performance of volume rendering algorithms for shared-memory multiprocessors. Unfortunately, such studies are difficult because the shared-memory model encourages implicit communication through shared variables rather than explicit calls to message-passing routines. While it takes little effort to produce a correct, efficient program, it is difficult to analyze the source of performance bottlenecks deep in the memory system.

In the coming years, small-to-medium-scale shared-memory multiprocessors based on fast microprocessors are likely to become commonplace. The parallel shear-warp algorithm is well-suited to this type of architecture and enables a cost-effective solution for real-time volume rendering.

# Chapter 7

# Extensions

Data visualization is a process that requires experimentation. To support this experimentation a visualization application must provide flexible tools such as user-defined shading functions and techniques for focusing attention on a particular region of interest. The more control the user has, the more useful the application will be for a wide range of visualization problems.

The preceding chapters discuss the core component of a volume rendering application: fast, flexible rendering algorithms for volume data. This chapter discusses a collection of extensions to the rendering algorithms that provide a variety of visualization tools. Section 7.1 describes a shading algorithm, i.e. a method for computing the color of each voxel in a volume. The algorithm uses a technique proposed by Abram & Whitted [1990] to combine the flexibility of a shade-tree language [Cook 1984] with the speed of a lookup-table implementation.

Section 7.2 describes a fast depth cueing algorithm. Depth cueing is a technique for simulating fog in a rendered scene. The further an object is from the viewer the more the fog obscures it, thereby helping a viewer to distinguish between foreground and background objects. We present a new algorithm that uses a factorization of the depth cueing function to reduce the cost of computing the distance between the viewer and each voxel.

Section 7.3 describes a new algorithm for creating volume renderings with shadows. It uses the shear-warp rendering algorithm both to compute the illumination at each voxel and to render the image efficiently. Furthermore, it is a one-pass algorithm that requires

significantly less memory than traditional two-pass methods that require a 3D buffer as large as the volume to store intermediate results.

Section 7.4 describes an extension of the shear-warp rendering algorithm for data sets containing mixtures of volume data and geometric data. Mixed data sets are useful for applications in which some of the objects to be rendered are analytically defined. The algorithm presented in this section has not yet been implemented.

Finally, Section 7.5 describes how to add support for arbitrary clipping planes to the shear-warp rendering algorithm. Clipping planes allow a user to render cut-away views of the interior of a volume or to focus attention on a subset of a volume. This extension has not been implemented either.

## 7.1 Flexible Shading With Lookup Tables

### 7.1.1 Shade Trees and Lookup Tables

Shading languages such as RenderMan [Upstill 1990] are widely used in photo-realistic rendering systems because they allow the user to specify a rich variety of shading functions. In the context of a visualization system a flexible shading language can be used to tailor the shading function for a specific data set. For example, a user might specify a function to highlight important aspects of the data or to produce appropriate perceptual cues. Languages that have been used for shading in volume rendering systems include a graphical network editor [Abram & Whitted 1990] for building shade trees [Cook 1984], the C programming language [Montine 1990], a specialized language called Vexpr that can be translated into C [Palmer et al. 1993], and an extension of RenderMan tailored for volume data [Corrie & Mackerras 1993a].

However, flexibility often comes at the price of slower execution. Most shading systems that support a general shading language operate by compiling a shading language expression into machine code or an intermediate language and then evaluating the compiled function for every rendered voxel. Compiler optimizations help to reduce the cost of evaluating the function [Hanrahan & Lawson 1990] but the cost increases as the number of operations in

the expression increases. Unless the shading expression is trivial, this solution is too computationally expensive for interactive applications.

An optimization commonly used to reduce the cost of a shading function is to tabulate values of the function in a precomputed lookup table. The cost of evaluating a tabulated function depends only on the number of parameters to the function, not on the number of operations in the original function.

Implementing a shading function with a lookup table has two potential problems. First, there is a tradeoff between the size of the table and the quality of the results since the shading function must be discretized. In practice this problem is less severe for volume visualization applications because volume data is typically noisy, unlike analytically-defined polygonal data, so the banding artifacts caused by quantization errors are less noticeable. Thus we can represent a shading function with a smaller table than would be necessary for polygon-based applications.

The second problem with lookup table implementations is the combinatorial explosion that results as the number of parameters to the shading function increases. A shading function with a large number of parameters requires a lookup table with many dimensions, resulting in excessive memory consumption and high computational costs whenever the table must be recomputed.

We adopt the following solution: instead of using a single multi-dimensional table, we precompute several tables with low dimensionality and at run time we evaluate a simple expression to combine values from the individual tables.

We will represent a shading function by a tree called a *shade tree* [Cook 1984]. Each leaf node in the tree represents a constant or a voxel parameter, each internal node represents an operator that combines its children into a single value, and the root node represents the final value produced by the shading function. Before rendering the volume we precompute lookup tables for selected subtrees of the shade tree. The result is a collapsed shade tree in which many of the nodes have been replaced by the lookup tables so the cost to evaluate the tree at rendering time is small. This method was proposed by Abram & Whitted [1990] and used for volume rendering by Westover [1990].

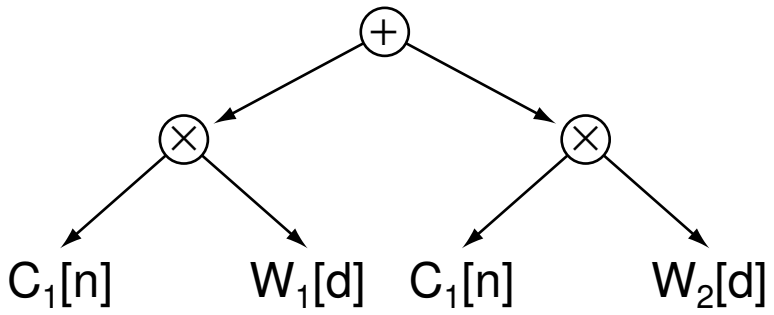



Figure 7.1: An example shade tree for the multiple-material shading model with two materials. Each leaf node is a lookup table indexed by either the scalar value $d$ or the normal vector $\mathbf{n}$ stored with a voxel.

### 7.1.2 Implementation of Shading Functions

This section illustrates our implementation of shading functions with a specific example from our volume rendering system. The implementation described here is also the shader we used in our performance tests in Chapter 5.

We have used the combination of shade trees and lookup tables to implement the multiple-material shading model proposed by Drebin et al. [1988]. This shading model assumes that each voxel consists of a mixture of several material types, for instance skin, bone and fat in a medical data set. We use a lookup table indexed by the scalar value of a voxel to produce a set of weights indicating the fraction of each material type in the voxel. Next, we use one lookup table per material type to produce the color of that material. Finally, to compute the color of a voxel the shader looks up the color of each material type present in the voxel and then combines the colors in proportion to the weights.

Figure 7.1 shows a shade tree for the multiple-material shading model with two materials. The leaf nodes labeled $C_i$ are the lookup tables containing colors for each material and the leaf nodes labeled $W_i$ are the lookup tables containing material weights. To shade a voxel the shader looks up one value from each table, multiplies each color by its associated weight, and adds all of the weighted colors. Our implementation supports an arbitrary number of materials.

The lookup tables containing material weights are defined by the user. The lookup tables for computing the color of each material contain values precomputed with the Phong

illumination model [Foley et al. 1990]. These tables are indexed by the surface normal vector associated with the voxel, which we will describe below. The equation used to compute each color is:

$$L = L_a + \sum_i L_i k_d(\mathbf{n} \cdot \mathbf{l_i}) + L_i k_s(\mathbf{n} \cdot \mathbf{h_i})^s$$

where:

$L$ is the radiance reflected by the material from the light sources to the viewer. The value of the color $C$ used in the volumetric compositing equation (Equation 1.4) equals $L$.

$\mathbf{n}$ is the normalized surface normal vector at the voxel.

$\mathbf{l_i}$ is a unit vector specifying the direction to light source $i$.

$\mathbf{h_i}$ is a unit vector equal to the surface normal that results in the maximum specular highlight for light source $i$:

$$\mathbf{h_i} = \frac{\mathbf{l_i} + \mathbf{v}}{|\mathbf{l_i} + \mathbf{v}|}$$

where $\mathbf{v}$ is a unit vector that specifies the direction from the voxel to the viewer.

$L_i$ is the radiance emitted by light source $i$.

$L_a$ is the radiance reflected by the material from ambient light.

$k_d$ is the diffuse reflection coefficient for the material.

$k_s$ is the specular reflection coefficient for the material.

$s$ is the shinyness exponent for the material.

For color images this equation is evaluated once for each color channel to produce an RGB lookup table. Each material type has a different set of material parameters ($L_a$, $k_d$, $k_s$ and $s$).

We make the assumption that all of the light sources are infinitely distant so that the vectors $\mathbf{l_i}$ are constant for every voxel. For parallel projections the viewer is also infinitely distant, so $\mathbf{v}$ and hence $\mathbf{h_i}$ are also constant. For perspective projections we can define $\mathbf{v}$ to be the direction of the central viewing ray; this definition will result in inaccurate specular

highlights but the approximation may be acceptable unless the viewer is close to or inside the volume. Under these assumptions the value of $L$ and hence the value of $C_i$ in each of the subtrees depends on only one variable, the surface normal vector $\mathbf{n}$. Thus each subtree can be tabulated in a one-dimensional lookup table indexed by the surface normal vector.

The surface normal at any point in the volume is defined to be a unit vector parallel to the local gradient of the voxel scalar value $d(x, y, z)$:

$$\mathbf{n}(x, y, z) = \frac{\nabla d(x, y, z)}{|\nabla d(x, y, z)|}$$

The gradient is approximated using the central difference gradient operator:

$$\begin{aligned} \nabla d(x, y, z) = & \quad \frac{1}{2}\left[d(x+1, y, z) - d(x-1, y, z)\right]\mathbf{i} \\ + & \quad \frac{1}{2}\left[d(x, y+1, z) - d(x, y-1, z)\right]\mathbf{j} \\ + & \quad \frac{1}{2}\left[d(x, y, z+1) - d(x, y, z-1)\right]\mathbf{k} \end{aligned}$$

The surface normal vectors are precomputed and stored with the voxels. Instead of storing all three components of the normal as floating point numbers we use an encoded representation that requires less storage and provides a convenient integer index for the lookup tables [Glassner 1990b, Fletcher & Robertson 1993]. The general method for encoding normal vectors works as follows:

1. Choose a set of unit vectors that are approximately uniformly distributed over all directions.

2. Assign a unique integer code to each vector in the set.

3. Compute the gradient at each voxel and find the closest unit vector in the predetermined set of unit vectors. Represent the normal by the integer code for the chosen unit vector.

The set of unit vectors in our implementation is found by normalizing the solutions of the equation:

$$|n_x| + |n_y| + |n_z| = 1$$

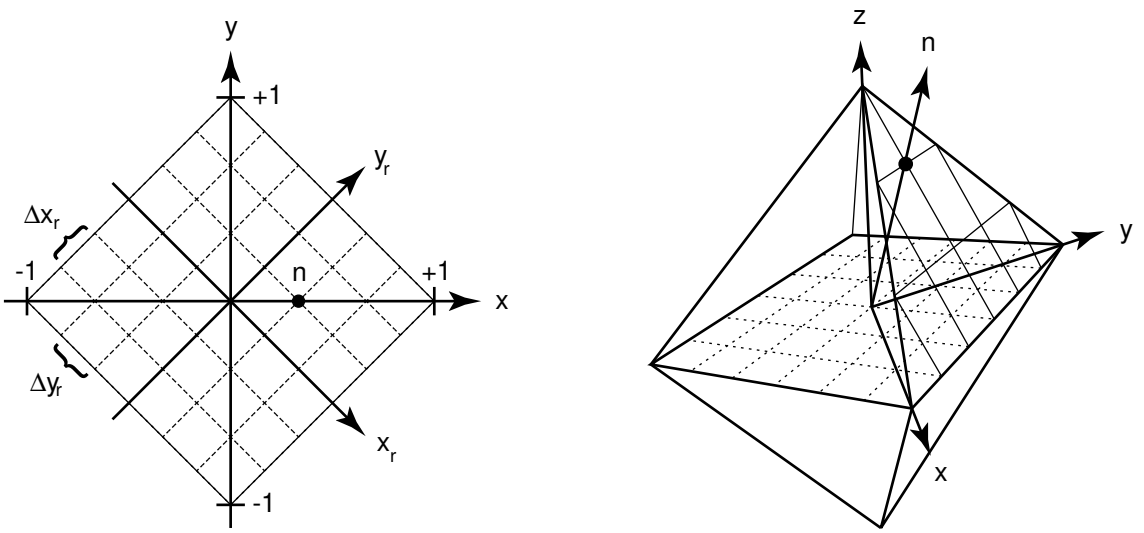



Figure 7.2: Grid locations for quantizing normal vectors. The grid (left) forms the base of a pyramid (right) with equation $|n_x| + |n_y| + |n_z| = 1$. Each vector is constrained to pass through the origin and a point on the surface of the pyramid that projects along the $z$ axis onto a grid point.

where $n_x$ and $n_y$ are constrained to locations on a grid centered at the origin and rotated 45 degrees, as shown in Figure 7.2. The solutions are of the form:

$$\begin{aligned} n_x &= \frac{y_r + x_r}{\sqrt{2}} \\ n_y &= \frac{y_r - x_r}{\sqrt{2}} \\ n_z &= \pm(1 - \frac{|y_r + x_r| + |y_r - x_r|}{\sqrt{2}})) \end{aligned}$$

where

$$\begin{aligned} x_r &\equiv x_i \Delta x_r \\ y_r &\equiv y_i \Delta y_r \end{aligned}$$

for integers $x_i$, $y_i$ and grid spacings $\Delta x_r$, $\Delta y_r$ (shown in the figure). Each vector can be encoded by concatenating $x_i$, $y_i$, and a bit indicating the sign of $n_z$. This particular definition for the set of quantized normal vectors allows for easy conversion between the encoded and unencoded representations and has a nearly-uniform distribution of directions. We have

| Shader Implementation | Shading Time (sec.) | Total Time (sec.) | Normalized Time |
|---|---|---|---|
| No Lookup Tables | 13.3 | 14.5 | 5.2 |
| Lookup Tables | 2.3 | 3.4 | 1.2 |
| Lookup Tables w/ Inlined Procedures | 1.6 | 2.8 | 1.0 |

Table 7.1: Performance results for three implementations of the shading function in Figure 7.1 to produce the image in Figure 5.5: evaluation without lookup tables, evaluation with lookup tables, and evaluation with lookup tables and an inlined shading procedure.

found that 13 bits (6 bits for each of $x_i$ and $y_i$ and 1 bit for $n_z$) is sufficient to avoid objectionable quantization errors.

The lookup tables for the color of each material ($C_i$) are recomputed whenever the user modifies any of the constants in the lighting model, such as the viewing direction or the material parameters. Each table consists of 8192 entries (the total number of surface normal vectors), so in large volumes the cost of computing the tables is significantly smaller than the cost of shading every rendered voxel.

Table 7.1 compares the cost of three implementations of the shade tree shown in Figure 7.1 for the case of two materials, one light source, and a color (RGB) image. In the first implementation, for every voxel the renderer calls a shading procedure that evaluates the entire tree without using lookup tables for $C_i$. In the second implementation the shading procedure uses the lookup tables. In the third implementation the lookup-table shading procedure is inlined into the rendering procedure to eliminate the overhead of a procedure call (including register spills due to the procedure calling conventions). The first column is the time for shading alone, the second column is the total time to render an image, and the third column is the total time normalized by the time to compute the image with the fastest implementation.

For the case of the shading expression in this example the lookup table implementations are five times faster than the implementation without precomputation, and the inlined procedure is about 20% faster than the non-inlined procedure. This example shows that even with a fairly simple shading model the computational costs can be very high in a procedural implementation, but a lookup-table implementation can reduce the cost to acceptable levels.

### 7.1.3 General Shade Trees

The current implementation allows shade trees of the form shown in Figure 7.1: a sum-of-products expression with an arbitrary number of product terms and one-dimensional lookup tables at the leaves. The implementation uses predefined macros that can be inlined into the rendering code. Even with this limited model many shading functions can be specified by changing the contents of the lookup tables. More general shading expressions could be supported in a straightforward manner by using a shading language system with a compiler and a dynamic linking facility as described in [Corrie & Mackerras 1993a]. The user would specify hints to indicate which subtrees should be tabulated (as proposed by [Abram & Whitted 1990]).

An area for future research is to develop automatic partitioning algorithms that can choose an optimal set of subtrees for tabulation. An optimal set minimizes the number of computations in the collapsed tree and the total size of the tables. The compiler should produce two procedures: a procedure for precomputing the lookup tables, and a procedure for rendering a volume with an inlined shading function implementing the collapsed shade tree (along the same lines as the specializing shaders of Guenter et al. [1995]). Such a system could provide the flexibility of existing shading languages with nearly the same performance as hand-optimized hard-wired shading systems.

## 7.2 Fast Depth Cueing

### 7.2.1 Depth Cueing

Depth cueing is a technique for simulating atmospheric attenuation of light, for instance due to uniform absorbing, non-scattering fog. The fog causes objects far from the viewer to appear darker than foreground objects. In visualization applications the technique can be used to emphasize depth relationships between different parts of a data set. Figure 7.3 shows two volume renderings with and without depth cueing.

Depth cueing can be implemented in a straightforward manner by computing the distance from the viewer (the depth) for each rendered voxel and attenuating the voxel's color

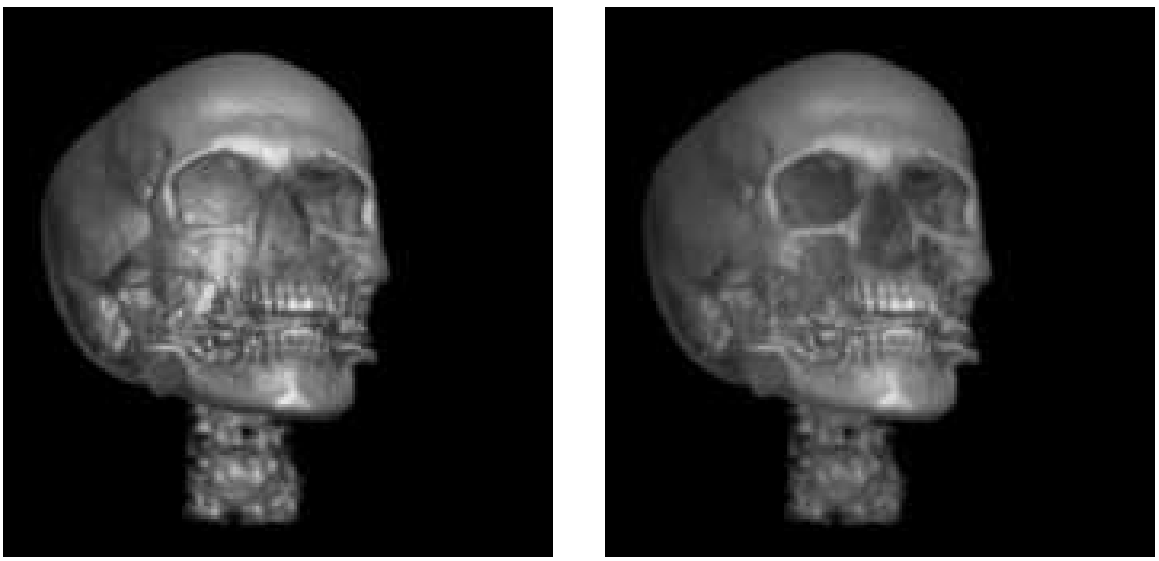


Figure 7.3: The left image is a rendering of a CT data set without depth cueing, and on the right is the same rendering with depth cueing. The more distant surfaces are darker in the depth cued image.

by a function of the depth. One standard depth cueing function is an exponential decay function:

$$T_{\text{fog}}(d) = T_0 e^{-\rho d}$$

where $T_{\text{fog}}(d)$ is the fraction of light transmitted after traveling a distance $d$ through the fog, and $T_0$ and $\rho$ are user-defined constants. If the color of a voxel before depth cueing is $C$ then after depth cueing the color is $C \cdot T_{\text{fog}}(d)$. To avoid the cost of evaluating the exponential during rendering, the depth cueing function can be precomputed and stored in a lookup table.

The computational cost of depth cueing is a table lookup and a multiply for each rendered voxel plus the cost of computing the voxel depths. Either the depths can be computed from the viewing transformation matrix for every rendered voxel, or the depths can be computed incrementally by adding a delta every time the rendering algorithm moves to a new voxel. The incremental approach is usually less expensive: for a parallel projection the depth delta is constant, so a single addition per voxel is required. For a perspective projection a divide is also necessary.

However, in the shear-warp algorithm and other object-order algorithms that use spatial data structures the depth must be updated not only for rendered voxels but also every time a

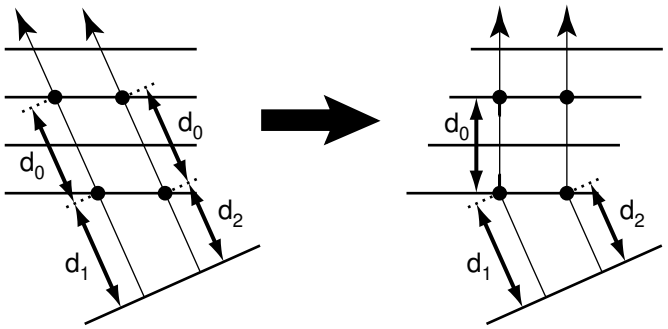



Figure 7.4: Factoring the depth cueing function: The distance from a voxel to the image can be split into two pieces, the distance from the voxel to the intermediate image and the distance from the intermediate image to the image. For a parallel projection the distance to the intermediate image is constant for every voxel in a slice.

run of transparent voxels or opaque pixels is skipped. The cost is therefore proportional to the number of rendered voxels plus the number of runs. When depth cueing is implemented using incremental calculation of the depth with floating-point arithmetic the time to render the depth cued image in Figure 7.3 increases by 20% compared to rendering without depth cueing.

### 7.2.2 Factoring the Depth Cueing Function

In this section we develop a faster depth cueing algorithm using a factorization of the depth cueing function. As shown on the left side of Figure 7.4, the distance $d$ from a voxel to the image plane can be broken into two pieces: the distance $d_0$ from the voxel to the front voxel slice, plus the distance $d_i$ ($i = 1, 2, \ldots$) from the front slice to the image. The exponential depth cueing function can be written as a product of two factors involving these two distances as follows:

$$T_{\text{fog}}(d) = T_0 e^{-\rho d} = e^{-\rho d_0} \cdot T_0 e^{-\rho d_i}$$

The right side of the figure shows that for a parallel projection the distance $d_0$ is constant for every voxel in a slice. As a result, depth cueing can be performed using a two-step process:

1. During rendering, multiply the color of every rendered voxel in a given slice by a constant depth cueing factor. The factor is:

$$T_{\text{slice}} = e^{-\rho d_0}$$

2. After all of the voxel slices have been composited together, multiply the color of every pixel in the intermediate image by a second depth cueing factor which must be computed individually for every pixel:

$$T_{\text{pixel}} = T_0 e^{-\rho d_i}$$

The first step of the algorithm accounts for light attenuation as the voxels are projected onto the plane of the intermediate image (which is coincident with the front voxel slice), and the second step accounts for light attenuation from the intermediate image plane to the final image plane.

The advantage of the algorithm is that the depths of individual voxels need not be computed. During the projection stage of the algorithm $T_{\text{slice}}$ must be computed only once per slice. The cost to apply the depth cueing factor is a single multiply for each rendered voxel. During the warping stage the cost of applying the second depth cueing factor is proportional to the number of pixels in the intermediate image. The cost per pixel is one add (to incrementally compute the depth of the pixel), one table lookup (to retrieve the value of the depth cueing function), and one multiply. The 2D intermediate image is generally much smaller than the volume so the cost of applying the second depth cueing factor is also small.

For perspective projections the value of $d_0$ is not constant for every voxel in a slice. However, it can be computed inexpensively from the sample spacing used for opacity correction (see Section 4.2.3):

$$d_0 = k \cdot \Delta x$$

where $k$ is the voxel slice number (counting from zero for the front slice) and $\Delta x$ is the distance between slices along a viewing ray in image space. $\Delta x$ must be stored with each intermediate image pixel for the opacity correction algorithm. Thus the depth cueing factor can be computed using one multiply and one table lookup per rendered voxel, in addition

| Algorithm | Depth Cueing Time (msec.) | Total Time (msec.) | Normalized Time |
|---|---|---|---|
| No Depth Cueing | 0 | 780 | 1.00 |
| Fast Depth Cueing | 40 | 820 | 1.05 |
| Standard Depth Cueing | 160 | 940 | 1.21 |

Table 7.2: Performance results for fast depth cueing compared to straightforward depth cueing and rendering without depth cueing. The first column is the time for depth cueing alone, the second column is the total time to render an image, and the third column is the total time normalized by the time to compute the image without depth cueing. These times are for rendering the data set in Figure 7.3 with a parallel projection.

to the work already required for opacity correction.

### 7.2.3 Implementation of Fast Depth Cueing

Table 7.2 shows the performance of the fast depth cueing algorithm compared to the standard depth cueing algorithm with incremental calculation of the voxel depths, and the rendering algorithm without depth cueing. The overhead of the fast depth cueing algorithm is only 5%, compared to 20% for the standard algorithm. Although the performance gain is not large compared to the benefit of coherence accelerations and fast shading algorithms, the simplicity of the fast depth cueing algorithm and the modest performance boost make it worth implementing.

A potential pitfall in the implementation of the fast depth cueing algorithm is that the intermediate image pixels must be represented using floating point numbers rather than a fixed-point representation with a small dynamic range. A large dynamic range is necessary because the first factor of the depth cueing equation sometimes "over attenuates" the color of a voxel, possibly resulting in a very small value which will later be restored to a higher value by the second depth cueing factor. This case occurs if the distance from the intermediate image to the final image ($d_i$) is negative for any intermediate image pixels. If an eight-bit integer representation is used for intermediate image pixels then the intermediate intensities may be incorrectly rounded to zero. We use a floating point representation both to eliminate this problem and to reduce round-off errors while compositing voxels into the intermediate image.

## 7.3 Rendering Shadows with a 2D Shadow Buffer

### 7.3.1 Algorithms for Rendering Shadows

The volume rendering equation (Equation 1.3) and the local illumination models described earlier in this chapter do not model occlusions between the voxels and the light sources. As a result, images produced using these methods do not contain shadows. Shadows are useful in certain visualization applications because they help the user to perceive depth relationships, and with properly-arranged lighting shadows can enhance small details by improving contrast.

Shadows can be incorporated into the basic volume rendering model by adding a second pass to the rendering algorithm: during the first pass through the volume the algorithm computes the illumination reaching each voxel, and during the second pass the algorithm generates an image. An additional pass is necessary for each light source if there is more than one light. The original version of this algorithm is due to [Kajiya & Von Herzen 1984] who used it in a ray caster. First, for each light source their algorithm casts rays through the volume from the light source and accumulates opacity along the rays. The accumulated opacity at each sample point along a light ray determines the illumination reaching that point in the volume. These illumination values (or equivalently, the accumulated opacities) are stored in a 3D array with the same dimensions as the volume. The array is called a *shadow buffer*. During the final rendering pass the shader uses the illumination values from the shadow buffer to compute the color of each voxel. Levoy [1989] also describes an implementation of this algorithm.

The primary disadvantages of the algorithm are that it requires a 3D shadow buffer as big as the volume to store the illumination values and it requires multiple passes through the volume. However, Grant [1992] describes an alternative method for computing the illumination using a much smaller 2D shadow buffer. This algorithm performs a single pass through the volume, computing illumination and the image simultaneously.

Grant's algorithm operates by sweeping the 2D shadow buffer through the volume in a direction that visits each voxel in depth-sorted order with respect to both the light source and the viewer, as shown in Figure 7.5. As in Kajiya and Von Herzen's algorithm the buffer contains illumination values. The algorithm sweeps the 2D shadow buffer through space until it

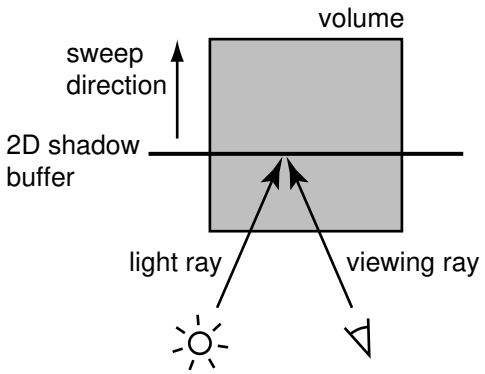



Figure 7.5: Diagram of Grant's shadow mask sweep algorithm [Grant 1992]. The 2D shadow buffer stores the illumination reaching the current slice of voxels. The rendering algorithm sweeps the buffer through the volume and updates the illumination values at the same time as the image is computed.

reaches a new slice of voxels. It uses the illumination values in the buffer to shade the voxels and then projects the voxels into the image along the viewing direction. The algorithm then projects the voxels into the shadow buffer along the light direction and updates the illumination in the buffer. Because the traversal order is depth-sorted with respect to both the light and the viewer, the information necessary for shading a voxel is always present in the depth buffer at the required time.

In order for the algorithm to work properly the volume must be traversed in front-to-back order with respect to the light source. If the light source and the viewer are both on the same side of the volume than the traversal order will also be front-to-back with respect to the viewer, as shown in the figure. If the angle between the light direction and the view direction is greater than 90 degrees then the traversal order may be back-to-front with respect to the viewer, which would make it impossible to implement early-ray termination during image formation.

Our shadow rendering algorithm combines the 2D shadow buffer with the shear-warp volume rendering algorithm. The algorithm sweeps the shadow buffer through sheared object space parallel to the slices of the sheared volume. The algorithm uses two shear-warp factorizations simultaneously: one for the viewing transformation matrix, and one for the light projection matrix. Each voxel slice must be translated and resampled twice, first to composite it into the intermediate image for the image formation computation and then to

```
1   procedure RenderVolumeWithShadows()
        view_factors = Factor(ViewMatrix);
        light_factors = Factor(LightMatrix);

        Clear(TmpImage);
5       Clear(ShadowBuffer);

        foreach (voxel_slice from front to back)
            light_attenuation = Resample(ShadowBuffer, light_factors);
            TmpImage = TmpImage over Resample(Shade(voxel_slice, light_attenuation), view_factors);

            ShadowBuffer = ShadowBuffer over Resample(voxel_slice, light_factors);
10      end

        Image = Warp(TmpImage);
        Display(Image);
    end
```

Figure 7.6: Pseudo-code for the shadow rendering algorithm.

composite it into the shadow buffer for the illumination computation. The algorithm uses a run-length encoded representation for the shadow buffer (the same data structure as for the intermediate image) so that the illumination computation can take advantage of coherence in the shadow buffer: voxels that are already in shadow need not by composited into the shadow buffer.

### 7.3.2 Implementation of the Shadow Rendering Algorithm

Figure 7.6 is a pseudo-code description of the shadow rendering algorithm for an infinitely-distant light source. First, the algorithm factors the viewing transformation matrix and the lighting transformation matrix (which transforms the light direction vector from the coordinate system of the light source into the image coordinate system). The rendering algorithm cannot proceed unless the principal axis is the same for both factorizations, i.e. both factorizations shear the same set of slices of the volume. We discuss this limitation later in this section.

Next, the algorithm clears the intermediate image (TmpImage) and the shadow buffer (ShadowBuffer). The shadow buffer is used to store the accumulated opacity of the voxels

between the light source and the voxel slice currently being rendered; the illumination can be easily computed from this opacity.

The algorithm then loops through the slices in front-to-back order with respect to the light source. Inside the loop the first step is to determine how much light reaches each voxel. The algorithm determines the opacity between a voxel and the light source (light_attenuation) by sampling a value from the shadow buffer. Because the algorithm is based on the shear-warp factorization, the transformation from the shadow buffer to the voxel slice only requires a translation. The factored lighting transformation (light_factors) determines how much the shadow buffer must be translated. Resampling is required since the translation distance might not be an integer (but no scaling or rotation is necessary). Next the algorithm uses the light attenuation to shade each voxel. Finally the algorithm resamples the shaded voxels and composites them into the intermediate image, which completes the image formation step.

The remainder of the voxel processing loop handles the illumination step. The algorithm resamples the voxel slice using the factored lighting transformation and composites the result into the shadow buffer. When the compositing loop completes, the algorithm warps the intermediate image and displays the result as in the standard shear-warp rendering algorithm.

Either the parallel projection algorithm or the perspective projection algorithm can be used in the image generation step inside the rendering loop. Similarly, the parallel projection algorithm can be used in the illumination step for infinitely-distant light sources or the perspective projection algorithm can be used for finite-distance light sources. In the case of finite-distance lights the transformation from the shadow buffer to the voxel slice includes a uniform scale as well as a translation (since the shadow buffer must be scaled to larger sizes as the distance from the light source increases), so the resampling step on line 7 becomes more expensive than for the infinite-distance case. A potential problem occurs if the size of the volume is comparable to or larger than the distance to the light source, since the scale factor may then become large enough to cause distortion. The current implementation does not support finite-distance light sources.

Multiple light sources can be supported by the algorithm provided that the light directions are all consistent with the same principal axis and a separate shadow buffer is used for

each light source.

All shadow rendering algorithms for volume data have a problem caused by surfaces shadowing themselves. Voxelized surfaces in band-limited volumes are several voxels thick, so voxels on the outer edge of a surface cast shadows on the partially-visible voxels inside the same surface. As a result the surfaces appear darker than expected, and self-shadowing can also introduce severe aliasing artifacts. We solve this problem by translating the shadow buffer a few voxels away from the light source during the image formation step [Williams 1978, Reeves et al. 1987, Levoy 1989].

Implementation of the shadow bias in the shear-warp shadow rendering algorithm is straightforward: in the compositing loop, instead of compositing the same slice of voxels (slice number $k$ for example) into both the intermediate image and the shadow buffer, the algorithm composites slice number $k$ into the intermediate image and slice number $k-\Delta k$ into the shadow buffer where $\Delta k$ is the shadow bias. The shift by $\Delta k$ slices in the illumination pass translates the slice along the direction of the light rays, as required. The shadow bias reduces the accuracy of shadows produced by small objects, but it also reduces the aliasing artifacts associated with self-shadowing.

The shadow rendering algorithm cannot be parallelized using the task partition in Chapter 6 because of the additional ordering constraints introduced by interleaving the illumination and image formation steps.

### 7.3.3 Performance of the Shadow Rendering Algorithm

Figure 7.7 shows an image generated with the shear-warp shadow rendering algorithm. The data set is the 256x256x225 voxel CT head data set. The image took 1.6 sec. to render, which is 2x longer than the image without shadows. This slowdown is typical since each voxel slice must be resampled and composited twice, although the volume must only be traversed once. The memory required by the algorithm is eight bytes per shadow buffer pixel (for the opacity and opaque pixel links), in addition to the memory required for the standard algorithm (see Table 5.3).

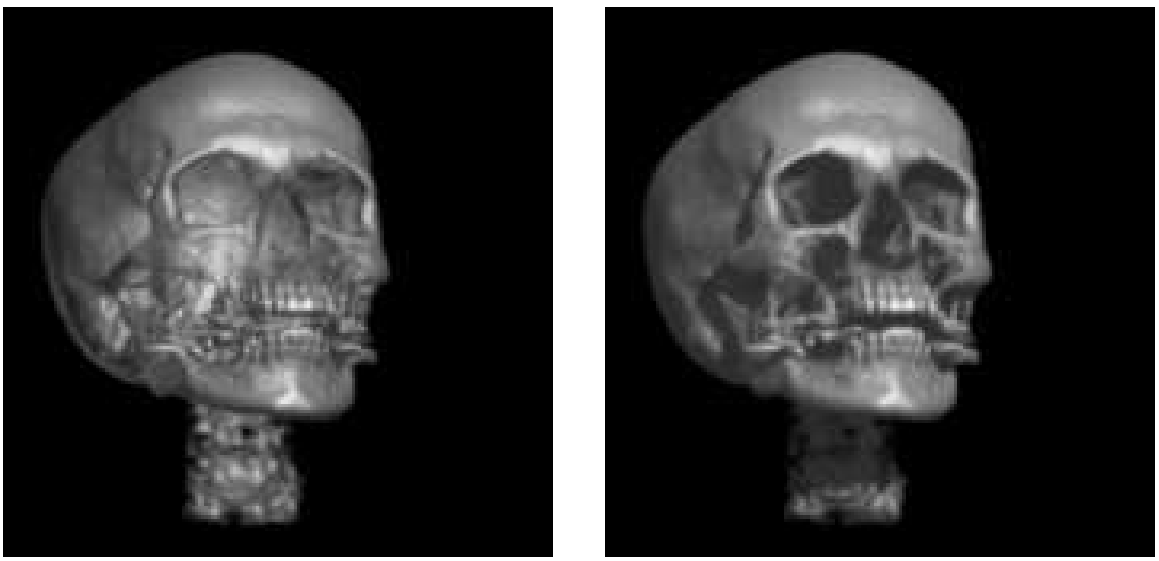


Figure 7.7: Volume rendering of a CT scan with the standard shear-warp rendering algorithm (left) and the shadow rendering algorithm (right). These renderings are not depth cued.

Compared to Grant's algorithm, the shear-warp shadow rendering algorithm is more efficient because filtering the values in the shadow buffer is less expensive. In Grant's algorithm, to compute the illumination required for shading a pixel fragment the algorithm must project the pixel fragment into the shadow buffer and the projected area must be sampled. This procedure may require decimating an arbitrarily-large area of the shadow buffer. Standard sampling acceleration techniques such as summed-area tables [Crow 1984] and mip-maps [Williams 1983] cannot be used since they require precomputation, whereas the shadow buffer is constantly modified during rendering. In the shear-warp algorithm the shear-warp factorization produces a simple mapping from voxels to shadow buffer pixels. For infinite-distance light sources the transformation is simply a translation, so filtering is trivial, and for finite-distance light sources the factorization guarantees that the shadow buffer will only be scaled to larger sizes (which results in less-expensive resampling than for the case of decimation).

Compared to a 3D shadow buffer algorithm the shear-warp shadow rendering algorithm places restrictions on the viewing direction and the light direction. As noted above, the shear-warp factorization for the view transformation matrix and the lighting transformation matrix must have the same principal axis. This requirement leads to cases that cannot be

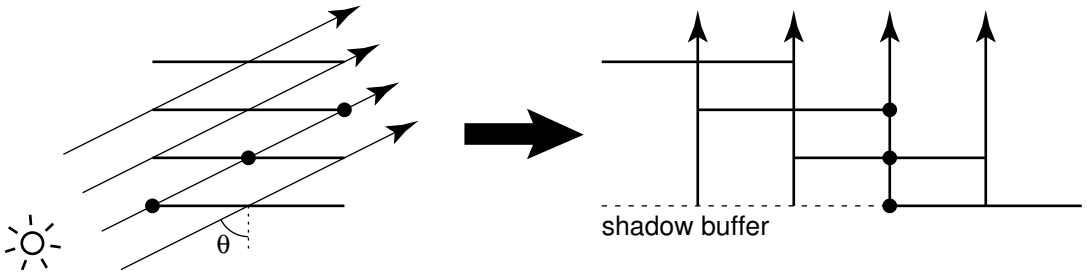



Figure 7.8: Limitations on the volume sampling rate in the shadow rendering algorithm: As the angle between the light vector and the principal viewing axis increases the volume shear increases. As a result the distance between samples along a viewing ray increases.

rendered. For example, if the view direction is perpendicular to one face of the volume and the light direction is perpendicular to another face then there is no consistent choice for the principal axis.

Furthermore, for a given viewing direction and a moving light source, as the light direction approaches an unrenderable orientation the image quality degrades. A larger angle between the light direction and the principal viewing axis corresponds to a larger shear in the shear-warp factorization. A larger shear is equivalent to a lower sampling rate along a viewing ray in a ray casting algorithm (Figure 7.8). As the sampling rate decreases below the Nyquist rate for the volume, aliasing artifacts appear in the shadow buffer and carry over into the rendered image.

The maximum acceptable angle between the light source and the principal viewing axis depends on the volume. For the CT head data set shown in Figure 7.7, the classified bone surfaces are roughly 2-3 voxels thick so the distance between image-space samples must not drop below about two voxels. This estimate agrees with our experiments: we achieve high-quality renderings of the data set with light-direction angles up to 60 degrees, which corresponds to a sample spacing of two voxels along a light ray (Figure 7.9). This restriction specifies the maximum angle between the light vector and the principal viewing axis. The maximum angle between the viewing direction and the principal viewing axis is 45 degrees, so for this volume the maximum angle between the light source and the viewing direction may range between 15 degrees and 105 degrees, depending on the viewing direction.

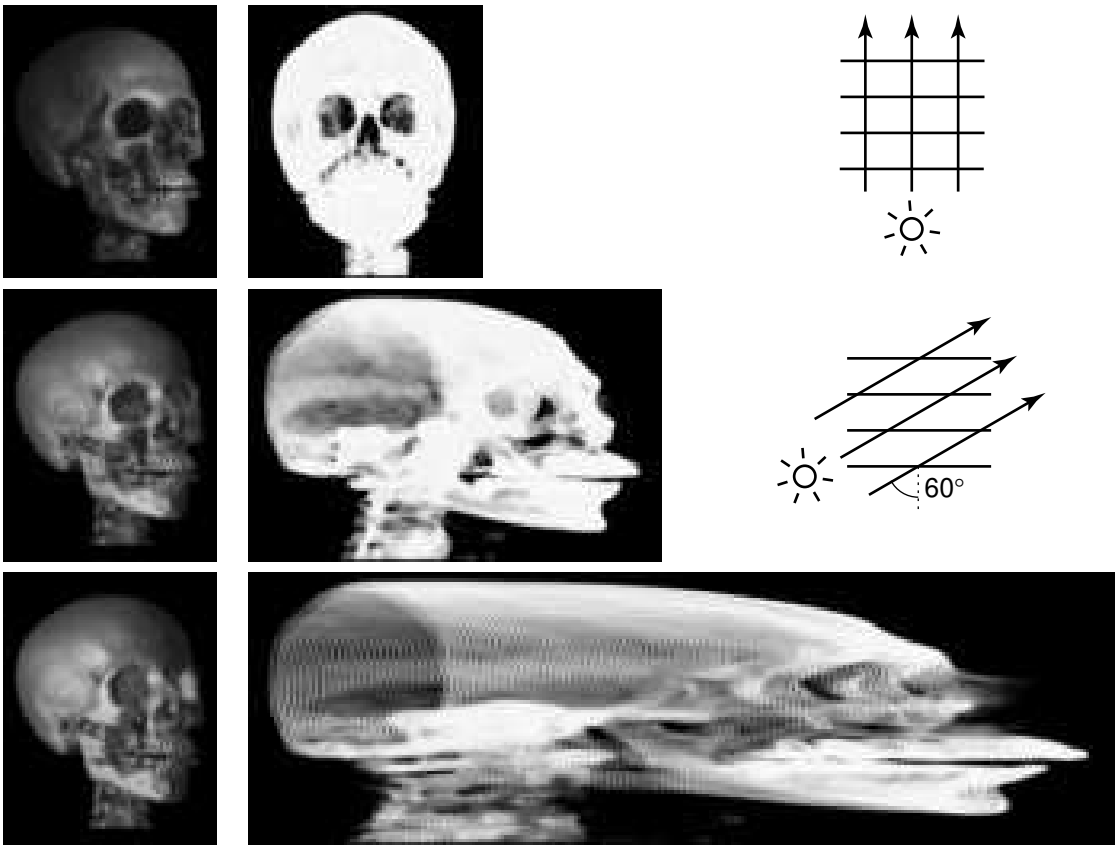



Figure 7.9: Aliasing in the shadow buffer: The first column of images shows three renderings of the CT head data set using the shadow algorithm. The viewing direction is constant for all three images but the angle between the light direction and the principal viewing axis varies (0 degrees for the top row, 60 degrees for the middle row, 78 degrees for the bottom row). The second column of images shows the contents of the shadow buffer after rendering half of the voxel slices in the volume. A large angle results in a large shear between adjacent voxel slices, so as the light angle approaches 90 degrees artifacts appear: in the bottom rendering the shadow behind the eye socket in the center of the image has stripes. The white stripes in the shadow buffer come from slices of the volume that have been translated far enough relative to each other to reveal gaps. This problem cannot be corrected unless a 3D resampling filter is used. However, up to an angle of 60 degrees there are no severe aliasing artifacts.

The angle restriction limits the usefulness of the shear-warp shadow algorithm for rendering arbitrary scenes. However, in a visualization application the user chooses the location of light sources to achieve an informative visualization rather than to simulate a real-world environment, so restrictions may be acceptable. Furthermore the speed of the algorithm makes it attractive for an interactive visualization application.

## 7.4 Rendering Mixtures of Volume Data and Polygons

Visualization applications sometimes require rendering mixtures of volume data and polygons. An example application area is radiation treatment planning: a radiation beam is defined analytically and therefore can be represented by a polygon mesh, whereas patient anatomy is most easily described via a scanned volume [Levoy et al. 1990]. Existing methods for rendering such mixed data sets include scan-converting the polygons into a volumetric representation that can be rendered with a conventional volume renderer [Wang & Kaufman 1994] and using a ray tracing algorithm adapted for both types of primitives [Levoy 1990b].

The shear-warp volume rendering algorithm can be combined with a modified version of a scanline polygon rendering algorithm to support mixed data sets directly. The algorithm has not been implemented, but in this section we describe a possible design.

The algorithm operates by alternating between two phases: rendering a slice of volume data, and rendering the portion of the polygon database lying between two voxel slices (Figure 7.10). The polygon rendering algorithm is a modified version of a scanline algorithm which renders each polygon incrementally, so the renderer can suspend work on a polygon after scan-converting the portion lying between two voxel slices. The algorithm also requires special cases to properly render polygons that intersect non-transparent voxels.

The standard scanline algorithm for rendering a polygonal scene works as follows [Foley et al. 1990]. The algorithm computes the image one scanline at a time using two data structures: an edge table and an active edge table. The edge table is an array of buckets, one bucket per scanline, each containing a list of polygon edges that start on a particular scanline. The active edge table is a list containing the edges that cross the scanline currently being rendered. The algorithm processes each scanline in turn by updating the active edge

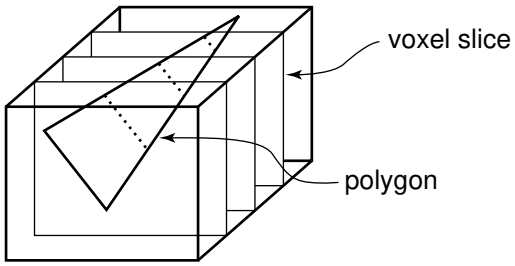



Figure 7.10: Mixtures of volume data and polygons can be rendered using an algorithm that alternates between two phases. During one phase the algorithm renders a slice of voxel data. During the alternate phase the algorithm incrementally scan-converts the portion of each polygon lying between the current voxel slice and the next.

table (adding newly active edges from the edge table and deleting completed edges) and then painting the pixels between pairs of active edges from the same polygon. To compute the intersection of each edge with the current scanline the algorithm uses the intersection with the previous scanline and the slope of the edge, both of which are stored with the edge. Visibility can be computed using the z-buffer algorithm or the painter's algorithm.

The algorithm for rendering mixed data sets processes the volume slice-by-slice, as in the standard shear-warp algorithm. Each slice of voxels represents a slab of the volume that intersects a subset of the polygons. The rendering algorithm first shades and resamples the voxel slice, then scan converts the intersecting portions of the polygons, and finally composites the resampled voxels and the scan-converted polygons into the intermediate image. The polygons are scan converted using a scanline algorithm with two modifications: the algorithm uses an additional active edge table to sort the polygons by the first intersecting voxel slice, and the endpoints of a horizontal pixel span for a particular polygon may be bounded by either the polygon edges or by the intersections of the polygon with the current voxel slab.

Two important issues must be addressed to produce high-quality images. First, the polygon scan conversion algorithm must be designed to avoid aliasing artifacts. The fixed sampling rate in the intermediate image may not be high enough to avoid aliasing, but the problem can be reduced by supersampling the polygons during scan conversion.

Second, visibility must be correctly computed when a polygon intersects a semi-transparent voxel. One approximate solution is to divide the voxel into pieces based on the

location of the intersection and then to composite contributions from the pieces and from the polygon into the image in front-to-back order [Levoy 1990b].

We expect this algorithm could achieve rendering times close to the rendering time of the standard shear-warp algorithm plus the time to render the polygons with a standard scanline algorithm implemented in software.

## 7.5 Clipping Planes

Clipping planes are commonly used in volume visualization applications to provide cut-away views of interior structures in a data set or to remove extraneous information while focusing on a particular part of a data set. Clipping planes with arbitrary orientations can be added to the shear-warp volume rendering algorithm by using a 3D scan conversion algorithm to convert the clipping planes into bounds for the rendering loops.

The 3D scan conversion algorithm can use data structures similar to the mixed-data-set rendering algorithm: an edge table and active edge tables to keep track of which clipping planes intersect the current voxel scanline. The slopes of the clipping planes can be used to calculate the intersection points incrementally as the algorithm iterates through the voxel scanlines. The intersection points determine the bounds for the loop that iterates over voxels in a voxel scanline.

Voxels that intersect or abut a clipping plane must be treated specially. In particular, to avoid aliasing the clipping algorithm must not cause voxel opacities to drop abruptly to zero. A suitable solution is to attenuate voxels as a function of their distance from the cutting plane using a smooth ramp a few voxels wide. It is also useful to let the user apply a special shading function to voxels near a cutting plane. Levoy et al. [1990] identifies several possibilities, including texture-mapping the original voxel values onto the cutting plane or recomputing voxel gradients after applying the cutting plane.

We have not yet implemented clipping planes. With clipping planes enabled the number of rendered voxels may decrease, resulting in improved rendering performance. The fast rendering algorithm must still traverse all of the voxel runs in the run-length encoded volume regardless of the location of the clipping planes, but as we saw in Chapter 5 this cost accounts for only a small fraction of the overall rendering time.

## 7.6 Chapter Summary

This chapter has discussed a number of extensions to the shear-warp volume rendering algorithm. These extensions show that the basic shear-warp algorithm can support a wide range of visualization options, including flexible user-defined shaders, fast depth cueing, shadows, mixed data sets and clipping planes. The implementations described in this chapter also show that complex shading and lighting effects can be achieved without sacrificing the fast rendering performance provided by the shear-warp algorithm.

# Chapter 8

# VolPack: A Volume Rendering Library

This chapter describes the design of an experimental software library called VolPack. The library provides a high-performance implementation of the algorithms in this thesis with a simple, flexible programmer's interface.

A fundamental problem when designing a volume rendering package is that rendering speed and flexibility are conflicting goals. The fastest rendering speed can be achieved by hardwiring all options, including the layout of a voxel and the functions for shading and classification, and carefully hand-tuning the code for a particular hardware platform. The most flexible system can be achieved by providing many rendering options and allowing the user to change any of them at run time.

Existing volume rendering packages generally fall at one of the two extreme ends of this spectrum. Montine [1990] proposes a flexible architecture for a volume rendering library based on user-defined callback functions for shading and classification. Alternatively, a general class of shading functions can be supported by letting the user specify a shading language procedure that is compiled and dynamically linked at run time [Corrie & Mackerras 1993a]. These approaches maximize flexibility but make it difficult to optimize specific cases since the renderer and the callback functions are developed and compiled separately and must adhere to a predefined interface. Rigid interfaces often lead to performance bottlenecks.

A number of existing systems take the opposite approach by providing a set of visualization "tools" with powerful but fixed functionality [Brittain et al. 1990, Avila et al. 1994].

The only way to extend such systems, for instance to add a new shading algorithm or to support a new voxel data type, is to modify the source code for an existing tool or to write a new tool. Typically the systems are designed with multiple layers of abstraction and modules with well-defined interfaces to facilitate adding new functionality, but extending the system still requires substantial effort. On the other hand, this architecture allows each tool to be fully optimized.

Finally, a number of commercial visualization systems (including AVS [Upson et al. 1989], Data Explorer [Lucas et al. 1992] and IRIS Explorer) are based on a data flow model: the system contains a collection of data processing and visualization modules that may be connected in a data flow network. Each module in a data flow system is independent and has a general interface to maximize reusability, so the system cannot optimize computation across modules. As a result, one can either implement a single monolithic volume rendering module with a fixed set of options and optimal rendering speed, or a flexible collection of interoperable shading, classification and rendering modules with significantly lower performance. However, there are no intermediate points in the flexibility/performance spectrum.

VolPack was designed to provide as flexible an interface as possible without significantly sacrificing performance. Specifically, the goals include avoiding hardwired formats for the volume, the shading function, and the classification function. Another goal is to provide a simple application programmer interface (API).

## 8.1 System Architecture

VolPack consists of a library of software routines for volume rendering and a specification of the interface between an application program and the library routines.

The software library includes a collection of core routines for specifying parameters and rendering a volume, plus a collection of utility routines that implement common shading and classification algorithms. These routines are designed to be building blocks for creating an application. The library does not specifically support a graphical user interface but can be combined with general user-interface toolkits.

The interface between the application program and the volume rendering library is the most critical part of the design since the interface impacts flexibility and performance. The

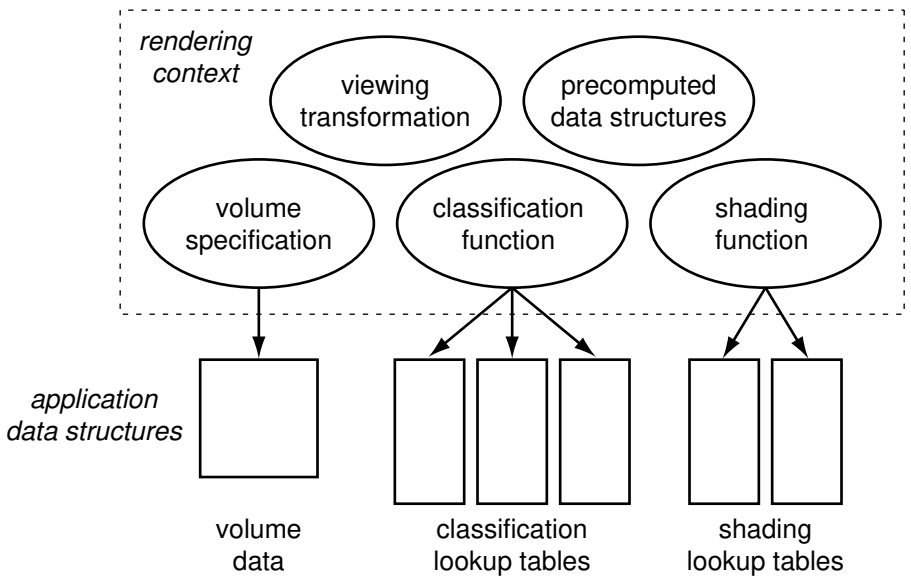



Figure 8.1: Schematic diagram of a VolPack rendering context. The context encapsulates user-specified parameters and internal precomputed data structures, as well as pointers to external application-specific data structures.

two aspects we focus on in this chapter are volume representation (i.e. the specific layout of the data in a volume) and specification of the shading and classification functions.

A volume rendering application built on top of VolPack first calls several library routines to set rendering parameters and then calls a rendering routine. The rendering parameters and all of the internal state required to render a volume are stored in a data structure called a *rendering context* (Figure 8.1). All of the library routines require a context as one of the parameters, except for the routine to create a new context.

Most of the routines in VolPack operate by transforming the state in a rendering context, a model inspired by the OpenGL interface [OpenGL Architecture Review Board et al. 1993]. The library routines allow the programmer to set options in the context, and the options remain in effect until they are disabled by the programmer. Most of the options have reasonable defaults, simplifying the programmer's job. The context also provides a place for routines to store precomputed data structures (e.g. a min-max octree or a run-length encoded representation of a volume).

The specific information stored in the rendering context is not visible to the programmer and is not defined by the VolPack interface. The context abstraction allows the library

to store information in the best format for use during rendering without burdening the programmer with a complicated interface. All necessary state is stored in the context rather than in global variables internal to the library so that the library code is reentrant.

The rendering context abstracts the internal implementation of the library's data structures in a way similar to object-oriented systems. However, in contrast to such systems VolPack defines only a single type of object (the rendering context) that contains all necessary state. Partitioning the state of a volume rendering system into a deep hierarchy of objects with well-defined interfaces has been shown to result in poor performance since the interfaces lead to overhead and hinder optimization [Zuiderveld & Viergever 1994]. Encapsulating the entire library in an object for inclusion in a larger system may be practical, but the potential benefits of the object-oriented model are dubious for the internal implementation.

## 8.2 Volume Representation

The volume rendering algorithms described in this thesis apply to volumes sampled on a regular grid and stored as a 3D array of samples, so VolPack does not currently support other data representations. However, the library does allow an arbitrary format for the voxels in the 3D array.

The application must define a volume by specifying its dimensions (width, height, and depth), a pointer to an array containing the voxel data, and a description of the layout of a voxel. Each voxel may contain a sequence of fixed-size fields. Every voxel in a volume must have identical fields, but the application defines all aspects of the layout including the number of fields, the sizes of the fields, and the interpretation of each field. For example, the representation can support voxels containing the following types of information:

- 8-bit, 16-bit or 32-bit scalar values
- multiple scalar values per voxel, e.g. data acquired using different scanning technologies
- precomputed data in addition to the original scalar values, e.g. normal vectors and gradients

In an application the layout of a voxel can be described using a compound data type such as a C-language structure or a Pascal record. The programmer then describes the voxel layout to the library using a set of routines provided by VolPack.

In the internal implementation these language constructs cannot be used since the layout of a voxel is not known when the library is compiled. Instead, the size of each field (in bytes) and the offset of each field from the beginning of the voxel are stored in an array in the rendering context. Each field can be identified by its ordinal index in the array. Library routines that must access the data in a voxel, such as classification and shading functions, take a field index as an argument and use the field offset array to find the location of the field within the voxel.

This design allows a single library routine to render volumes with arbitrary voxel layouts. It is possible to write shading and classification functions that are independent of a specific voxel layout. It is also possible to define the format of a voxel at run time based on user input or information in the header of a data file. The overhead of the extra level of indirection is small: to compute the address of a voxel field the renderer need only add the field offset to the address of the voxel.

## 8.3 Classification and Shading Functions

VolPack uses the flexible lookup-table shading algorithm described in Section 7.1 for both voxel classification and shading. The application specifies a shade tree and a set of lookup tables for each of the classification and shading functions. The speed of the method was demonstrated in Section 7.1.2.

In the current implementation of VolPack the shading tree is limited to a sum-of-products expression with lookup tables at the leaves, and the classification tree is limited to a product of the values from the lookup tables. Even with these restrictions a wide variety of shading functions can be implemented since an application can initialize the tables arbitrarily.

The method extends to more general shading functions as described in Section 7.1.3. To achieve optimal performance with user-defined shading functions we envision a system that operates as follows. The shade tree can be translated automatically into a C-language macro that implements the shading function. The translator can also provide a list of input values

required by the function, such as values from particular voxel fields. The system can then dynamically compile and link a specialized rendering routine with the shading macro inlined. The list of input values can be used to tailor the rendering routine so that computation that is not required can be omitted. For instance, if the user wishes to generate a maximum intensity projection instead of a true volume rendering then the shader will not reference the opacity of a voxel, so the code to compute and resample opacities can be omitted from the renderer.

VolPack currently contains predefined macros implementing a small number of shading functions. Specialized renderers for each of these shaders are generated by inlining these shading functions at compile time. We have not yet implemented the dynamic compilation and linking facility but this technique has been used by others. Corrie's implementation of data shaders [Corrie & Mackerras 1993a] uses dynamic linking to support shaders defined at run time and implemented via callback functions. We propose going one step further by inlining the shader in the renderer, customizing the renderer for the shader, and then compiling and linking the optimized renderer.

## 8.4 Viewing Model

VolPack uses four-by-four homogeneous transformation matrices to specify viewing transformations, giving the application writer the flexibility to construct viewing transformations with any 3D viewing model. For convenience VolPack also provides routines to construct matrices using the viewing models defined in the OpenGL standard [OpenGL Architecture Review Board et al. 1993] and the PHIGS standard [Foley et al. 1990, Chapter 6].

## 8.5 Functionality Provided by VolPack

The previous sections have described the architecture of VolPack and the techniques used to provide a flexible, efficient programmer's interface. This section briefly describes the categories of routines provided by the library:

**context manipulation** Routines to create and destroy a rendering context and to retrieve state variables from a context.

**volume specification** Routines to define the size and layout of a volume and to attach an array of volume data to a context.

**classification** Routines to define a classification function (including the lookup tables and the shade tree) and to precompute data structures for fast classification (i.e. a min-max octree) and fast rendering (i.e. a run-length encoded volume).

**shading** Routines to define a shading function. A higher-level routine to build a shading lookup table for the commonly-used Phong illumination model is included for convenience.

**view transformation** Routines to construct a four-by-four viewing transformation matrix. The current transformation matrix for a volume is stored in the rendering context.

**image specification** Routines to define the size of the rendered image, the format of the pixels, and an array for storing the image data.

**rendering** Routines to invoke one of the volume rendering algorithms and to set options specific to a renderer. The options are stored in the rendering context.

**utilities** Routines for file I/O, volume reformatting (extracting and inserting subvolumes, resampling, and transposing), and linear algebra.

Given these building blocks, a simple volume rendering application can be written using 30-50 lines of C code.

## 8.6 Chapter Summary

In this chapter we have discussed approaches to designing a volume rendering library interface that strikes a balance between speed and flexibility. VolPack uses a procedural interface and encapsulates all internal state in a rendering context rather than using an object-oriented

interface or a data flow model. VolPack's high-level interface eliminates the overhead associated with a large collection of independent objects or modules.

VolPack's volume representation supports volumes with arbitrary data defined on a regular grid. The flexible voxel format does not impact performance.

We have proposed a shading and classification system based on rendering routines that are automatically specialized for a particular shade tree through dynamic compilation and linking. This method supports a very general class of shading functions and provides good performance. Rendering functions that have been hand-tuned for a particular set of shading parameters and a specific voxel layout are likely to perform marginally better, but with a great loss in flexibility.

The current version of VolPack only partially implements the architecture described in this chapter, but the existing interface requires little modification to incorporate all of these features. Combined with the shear-warp rendering algorithms, VolPack provides fast rendering with substantial flexibility.

# Chapter 9

# Conclusions

## 9.1 Final Summary

Many volume rendering algorithms have been proposed in the last decade, yet volume rendering has not become a widely-used visualization technique because of the lack of interactive systems. By a system we mean an algorithm combined with the hardware it runs on. Existing algorithms are too slow to provide interactive response on a workstation, and multiprocessors with sufficient computational power are too expensive for general-purpose use. Interactive rendering rates are essential because the process of visualizing volume data requires experimentation.

This thesis describes a family of new volume rendering algorithms that delivers interactive rendering rates on current-generation workstations. The algorithms improve on existing methods by combining the advantages of two classes of algorithms: image-order ray casting algorithms and object-order splatting algorithms. Like other object-order algorithms, the new shear-warp algorithms stream through the volume data in storage order so they incur little overhead due to addressing arithmetic and data structure traversal. In contrast, these overheads constitute a substantial fraction of rendering time in image-order coherence-accelerated algorithms.

The shear-warp algorithms also benefit from synchronized scanline-order access to both the volume data and the image, making it possible to use coherence in the image to implement early ray termination. Most previous uses of early ray termination have been restricted

to ray casting algorithms. Furthermore, implementing efficient, high-quality resampling filters in the shear-warp algorithms is as straightforward as in image-order algorithms.

The technique that allows us to traverse both the volume and the image in storage order is the shear-warp factorization: a factorization of the viewing matrix into a 3D shear, a projection into a distorted 2D intermediate image, and a 2D warp that transforms the intermediate image into the correct final image. We showed how to use the factorization and coherence data structures based on run-length encoding to develop a parallel-projection volume rendering algorithm for classified volumes. We then showed how to extend the factorization to perspective projections and developed a fast perspective volume rendering algorithm. Finally, we developed coherence data structures based on a min-max octree and a summed-area table that, when combined with either rendering algorithm, allows interactive classification of volume data. These three algorithms complement each other by addressing different phases of the visualization process.

Performance tests show that on current desktop workstations the new parallel projection algorithm can render $256^3$ medical data sets in less than one second and $128^3$ data sets at over 5 frames per second. These speeds are over five times faster than previously reported results for a ray caster with an octree. The fast classification algorithm can classify and render a $128^3$ volume in under a second. None of these algorithms use any specialized hardware so the rendering rates will continue to improve as faster workstations appear on the market, and the algorithms are portable to any general-purpose machine. Rendering speed no longer limits the usefulness of volume rendering.

We have also parallelized the rendering algorithms for shared-memory multiprocessors. On a 16-processor SGI Challenge the parallel projection algorithm renders $256^3$ data sets at over 10 frames per second and $128^3$ data sets at over 25 frames per second. Inexpensive small- to medium-scale shared-memory multiprocessors based on fast microprocessors are likely to become common in the coming years, and the multiprocessor shear-warp volume rendering algorithm is well-suited to this type of architecture.

Finally, we have demonstrated that the different shear-warp algorithms can be integrated into a practical interactive volume rendering system. We have presented an efficient method for precomputing the necessary data structures when switching between the rendering algorithms, and we have presented extensions for rapidly rendering a variety of visual cues:

depth cueing, shadows, clipping planes, and lookup-table based shading functions. These visual cues help to produce informative, easily-interpreted visualizations.

The rendering algorithms presented in this thesis have several limitations. Most important is that the speed of the algorithms depends on the degree of coherence in the volume. Classified volumes containing large transparent regions or large opaque regions result in fast rendering rates. The algorithms exhibit slower performance if a substantial fraction of the volume consists of low-opacity voxels. However, such data sets typically result in fuzzy, indistinct images that do not convey useful information. Even if the original volume contains little apparent coherence, in our experience properly-classified volumes expose inherent structure in the volume. Thus substantial performance gains can be achieved by exploiting coherence.

The shear-warp algorithms also restrict the choice of resampling filter since 3D filter kernels cannot be supported. In practice the algorithms produce high-quality images provided the volume data is alias-free, so image quality is not a major limitation. The fast classification algorithm also places restrictions on the classification function. The algorithm supports multi-feature classification based on user-specified transfer functions.

## 9.2 Future Directions for Performance Improvements

Our performance measurements lead to two important observations that impact future work on high-performance volume visualization algorithms. First, we have found that object-order scanline-based algorithms are inherently more efficient than image-order algorithms. Streaming through the volume in storage order will always be faster than traversing the volume in an arbitrary view-dependent direction, and we have shown that in coherence-accelerated algorithms the performance difference between these two alternatives is significant. For this reason scanline-order algorithms are likely to dominate volume rendering in the future.

Scanline-order algorithms can yield performance gains in other volume-processing operations as well. For example, one application is to combine a series of range images (acquired with a laser range scanner) into a 3D volumetric model called an *occupancy grid*. One existing method operates by shooting rays from the range image pixels into a 3D

grid and storing occupancy information at the lattice points [Connolly 1984]. However, a scanline-order algorithm based on the shear-warp factorization would be faster than a ray casting algorithm.

Even for a volume sampled on a non-rectangular grid, such as a curvilinear grid, we speculate that the fastest rendering algorithm may be an object-order algorithm that first resamples the volume to a rectangular grid and then uses a scanline-order rendering algorithm.

The second important observation is that there is no single performance bottleneck remaining in the shear-warp volume rendering algorithm. The rendering time is divided roughly equally between essential computation (shading, resampling and compositing voxels), overhead due to the coherence data structures and control logic, and memory overhead. It will therefore be difficult to further improve the algorithm. Remembering Amdahl's Law, we observe that optimizations that improve on any one of these areas will not have a major impact on overall execution time. Optimizations that reduce the cost per voxel, more efficient coherence data structures, and faster memory systems will not individually reduce rendering time by more than about 10-20%.

The most likely approach for improving performance is to trade off image quality for speed by further reducing the number of voxels processed, thereby potentially reducing all three cost categories. Techniques such as adaptive sampling [Levoy 1990a] and subsampling the volume [Laur & Hanrahan 1991] have been proposed and could be applied to the shear-warp algorithm. Approaches such as these or techniques based on rendering compressed volume data [Ning & Hesselink 1993, Muraki 1994] may also be effective ways of handling very large data sets, an important topic not addressed in this thesis.

## 9.3 Hardware Support for Volume Rendering

Our performance results also have implications for hardware designers. Specialized instruction sets for graphics appear not to be useful for volume rendering. This class of instructions is designed for accelerating inner loops that operate on pixels. The instructions perform operations in parallel on several pixels or color channels packed into one machine word. The processor includes special hardware to perform the packing and unpacking operations, and to perform arithmetic on the independent components in parallel. Such instructions may

be useful for brute-force resampling and compositing loops, but we have shown that only a small fraction of the time in coherence-accelerated volume rendering algorithms is spent processing voxel data. By Amdahl's Law the overall speedup achievable with graphics instructions is negligible.

In a broader context, we speculate that many graphics algorithms can be accelerated by using high-level algorithmic optimizations and data structures that have the side effect of eliminating opportunities to use special-purpose graphics instructions. Unless the graphics instructions provide better speedups, the software solution is the preferred alternative.

High-performance hardware-accelerated volume rendering has recently become feasible with the advent of 3D texture mapping hardware in high-end graphics workstations [Cabral et al. 1994]. A volume can be treated as a 3D solid texture. To render the volume an application generates a sequence of polygons that slice through the volume and are parallel to the plane of the image. The graphics hardware then shades the polygons using the volume as a texture and composites the polygons together. Classification can be performed in hardware by mapping the voxel values through a lookup table containing the opacity transfer function.

Using this algorithm, current-generation high-end graphics workstations can achieve sub-second rendering rates and can support interactive classification. However the hardware is expensive and cannot support sophisticated shading or classification. The only type of shading available is a mapping from the original scalar value to a color (implemented via a hardware lookup table). The Phong illumination model, which requires the gradient of the scalar value, cannot be implemented. Assuming that the cost of the hardware drops in the future, this lack of flexibility is the chief short-coming of 3D texture mapping hardware used for volume rendering. Nevertheless, for some specialized applications supported by the hardware this approach may be an effective solution.

However, a second argument against the widespread use of brute-force graphics hardware for volume rendering is the large performance difference between brute-force and optimized algorithms. For example, compared to the brute-force ray caster, the optimized shear-warp algorithm is 25-50x faster (as shown in Chapter 5). Hardware solutions require large amounts of hardware just to match this speedup. This situation is different than for

the case of polygon rendering hardware since the speedups achievable with software optimizations are not as high. Polygon rendering algorithms that compute visibility prior to rasterization (such as Watkins' scanline algorithm and Warnock's area-subdivision algorithm [Foley et al. 1990, Ch. 15]) can reduce the cost of rasterization, but even if we ignore the costs of computing visibility the maximum possible speedup equals the depth complexity of the scene. A rule of thumb for the average depth complexity in typical polygonal scenes is three to four [Molnar et al. 1994], so the speedup necessary to make hardware solutions attractive is much lower for polygons than it is for volumes. Thus even though polygon rendering hardware is becoming widely available, it is less likely that hardware for volume rendering will become cost effective.

Special-purpose hardware implementations of a specific algorithm will always be faster than a software-only implementation, but fully-custom hardware is too expensive for general use. We believe the most cost-effective platform for volume rendering consists of simple, fast, general-purpose microprocessors combined with optimized software algorithms. Inexpensive small-scale shared-memory multiprocessors built from general-purpose microprocessors will continue this trend.

## 9.4 Interactive Volume Rendering

A fast volume rendering system is the first step towards the widespread use of volume rendering. The next step is to create effective user interfaces that provide control over the many parameters required by the rendering algorithm without overwhelming the user. Choosing classification functions that expose the information in a volume and tuning shading parameters to highlight that information are complicated tasks.

In the past, much research has been devoted to the automatic choice of parameters. In particular, many algorithms have been proposed for automatic classification by segmenting volume data (e.g. distinguishing individual organs and tissue types in a medical data set). With a slow rendering system it is not practical to require a user to choose parameters interactively, so an automatic algorithm is attractive. Unfortunately, with the exception of methods for certain restricted applications (such as classification of high-contrast structures in CT data), automatic algorithms are not robust and do not produce reliable results.

Fast volume rendering enables a new approach to processing and visualizing volume data. Instead of using complicated, unreliable batch algorithms, an application can provide a set of simple controls that can be adjusted by the user. As long as the user can immediately see the result while adjusting a control, the feedback loop created by the rendering system and the user creates a much more robust and stable system than an automatic batch algorithm.

As an example, medical CT data is often classified by specifying a "window": a target data value and a width indicating a tolerance around the target value. The target value can be adjusted to select a tissue type with a particular CT value, and the window width excludes tissues with different values. A user interface with two controls tied to the window value and width, combined with a fast volume rendering system, would allow a user to rapidly select tissues of interest.

A fast volume rendering system also enables new applications. Morphing of 3D volumetric models is now computationally feasible [Lerios et al. 1995]. 3D morphs are superior to traditional 2D morphs because occlusions and lighting changes can be accurately rendered in the intermediate frames, but a fast rendering system is necessary to make the technique practical. Another new application is simplification of complex geometry via 3D scan conversion into a volumetric model, which is a useful method for manipulating and visualizing complex scenes [Wang & Kaufman 1994]. In the near future, real-time visualization in the operating room using open-magnet MR scanners and fast volume rendering may become practical.

In this thesis we have provided appropriate algorithms for the rendering system. The next step is to create simple yet powerful user interfaces that harness the power of volume rendering.

# Appendix A

# Mathematics of the Shear-Warp Factorization

This appendix contains complete derivations of the shear-warp factorization for rendering a volume with either a parallel or a perspective projection. The derivations assume that we are given only an arbitrary four-by-four viewing transformation matrix and the dimensions of the volume to be rendered. The results of the derivations are the shear and warp matrix factors.

## A.1 Coordinate Systems and Definitions

We will use the four coordinate systems illustrated in Figure A.1: object coordinates, standard object coordinates, sheared object coordinates (also called intermediate image coordinates), and image coordinates. All are right-handed coordinate systems.

The object coordinate system is the natural coordinate system of the volume data. The origin is located at one corner of the volume. The unit distance along each axis equals the length of one voxel along that axis. The axes are labeled $x_o$, $y_o$ and $z_o$.

We form the standard object coordinate system by permuting the axes of the object coordinate system so that the principal viewing axis becomes the third coordinate axis. The principal viewing axis is the object-space axis that is most parallel to the viewing direction. The standard object coordinate system axes are labeled $i$, $j$ and $k$, where $k$ is the principal

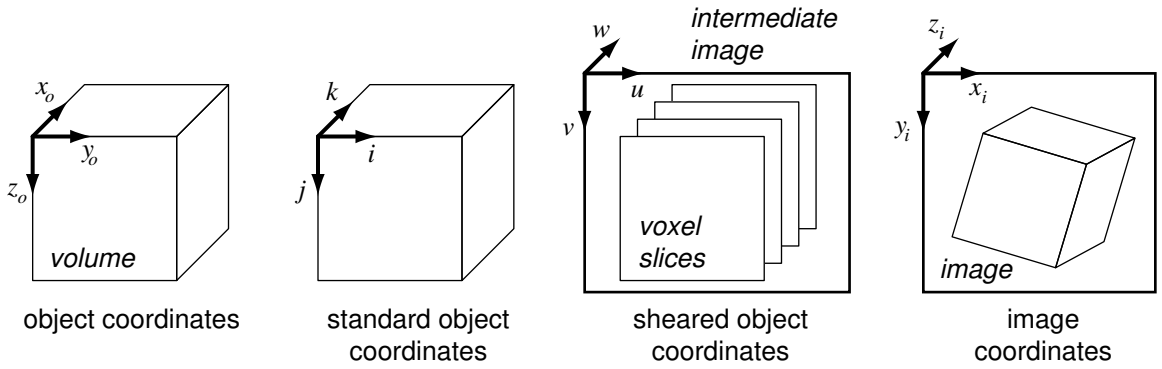



Figure A.1: Coordinate systems used in the derivation of the shear-warp factorization.

viewing axis.

We form the sheared object coordinate system by shearing the standard object coordinate system with the shear matrix from the shear-warp factorization. The sheared object coordinate system is also the coordinate system of the intermediate image. The origin is located at the upper-left corner of the intermediate image. The axes are labeled $u$, $v$ and $w$.

The image coordinate system is the coordinate system of the final image. The warp matrix of the shear-warp factorization transforms sheared-object coordinates into image coordinates, and the original viewing transformation matrix transforms object coordinates into image coordinates. The origin of the image coordinate system is located at the upper-left corner of the image. The axes are labeled $x_i$, $y_i$ and $z_i$.

In the following derivations $\vec{v}$ represents a vector and $v_x$ represents a component of $\vec{v}$. The viewing transformation matrix is a four-by-four matrix $M_{\text{view}}$ that transforms homogeneous points from object space to image space:

$$\begin{bmatrix} x_i \\ y_i \\ z_i \\ w_i \end{bmatrix} = M_{\text{view}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

All vectors are column vectors.

## A.2 The Affine Factorization

In this section we derive a factorization for affine viewing transformations. Affine transformations include uniform and non-uniform scales, translations and shears used in parallel projections. The goal of the derivation is to factor an arbitrary affine viewing transformation matrix $M_{\text{view}}$ as follows:

$$M_{\text{view}} = M_{\text{warp}} \cdot M_{\text{shear}}$$

$M_{\text{shear}}$ is a shear matrix and $M_{\text{warp}}$ is an affine transformation matrix. The matrix factors must be chosen such that we can project a volume from 3D to 2D between the shear and warp transformations and still produce a correct image.

### A.2.1 Finding the Principle Viewing Axis

We define the principal viewing axis as the object-space axis that forms the smallest angle with the viewing direction vector.[1] In image space the viewing direction vector is:

$$\vec{v}_i = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Let $\vec{v}_o$ be the viewing direction vector transformed to object space. It obeys the linear system of equations:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} v_{o,x} \\ v_{o,y} \\ v_{o,z} \end{bmatrix}$$

where the $m_{ij}$ are elements of the viewing transformation matrix $M_{\text{view}}$. Only the upper-left 3x3 submatrix of $M_{\text{view}}$ is necessary since $\vec{v}_i$ and $\vec{v}_o$ are vectors. In the remainder of this section let $M_{\text{view,3x3}}$ be the 3x3 submatrix.

[1] This definition is not equivalent to choosing the normal vector for the set of voxel slices most perpendicular to the viewing direction. If the viewing transformation includes non-uniform scaling then for some viewing directions this alternative definition results in larger (sub-optimal) shear coefficients.

By Cramer's Rule the solution to the linear system is:

$$v_{o,x} = \frac{\begin{vmatrix} 0 & m_{12} & m_{13} \\ 0 & m_{22} & m_{23} \\ 1 & m_{32} & m_{33} \end{vmatrix}}{|M_{\text{view,3x3}}|} = \frac{m_{12}m_{23} - m_{22}m_{13}}{|M_{\text{view,3x3}}|}$$

$$v_{o,y} = \frac{\begin{vmatrix} m_{11} & 0 & m_{13} \\ m_{21} & 0 & m_{23} \\ m_{31} & 1 & m_{33} \end{vmatrix}}{|M_{\text{view,3x3}}|} = \frac{m_{21}m_{13} - m_{11}m_{23}}{|M_{\text{view,3x3}}|}$$

$$v_{o,z} = \frac{\begin{vmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ m_{31} & m_{32} & 1 \end{vmatrix}}{|M_{\text{view,3x3}}|} = \frac{m_{11}m_{22} - m_{21}m_{12}}{|M_{\text{view,3x3}}|}$$

Since the denominator is the same for all three components of $\vec{v}_o$ it can be eliminated, yielding:

$$\vec{v}_o = \begin{bmatrix} m_{12}m_{23} - m_{22}m_{13} \\ m_{21}m_{13} - m_{11}m_{23} \\ m_{11}m_{22} - m_{21}m_{12} \end{bmatrix} \tag{A.1}$$

The cosine of the angle between the viewing direction and each object-space axis is proportional to the dot product of $\vec{v}_o$ with each of the object-space unit vectors. The largest dot product corresponds to the smallest angle. Thus we find the principal viewing axis by computing:

$$c = \max(|v_{o,x}|, |v_{o,y}|, |v_{o,z}|) \tag{A.2}$$

If $c = |v_{o,x}|$ then the principal viewing axis is the $x_o$ axis. If $c = |v_{o,y}|$ then the principal viewing axis is the $y_o$ axis. Otherwise, the principal viewing axis is the $z_o$ axis.

### A.2.2 Transformation to Standard Object Coordinates

The shear-warp rendering algorithm operates by resampling and compositing the set of voxel slices that is perpendicular to the principal viewing axis. To eliminate special cases for each of the three axes in the remainder of the derivation we first transform the volume into standard object coordinates. If the principal viewing axis is the $x_o$ axis then the permutation matrix is:

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$