Architectures for Real-Time Volume Rendering

A DISSERTATION PRESENTED

BY

HANSPETER PFISTER

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

STATE UNIVERSITY OF NEW YORK

AT STONY BROOK

December 1996

Abstract of the Dissertation

Architectures for Real-Time Volume Rendering

by

Hanspeter Pfister

Doctor of Philosophy

in

Computer Science

State University of New York at Stony Brook

1996

Advisor: Leading Professor Arie Kaufman

In this dissertation we present special-purpose volume rendering architectures that are capable of rendering high-resolution (e.g., 1024^3) datasets at 30 frames per second. The architectures, Cube-3 and Cube-4, exploit a high degree of parallelism to provide real-time parallel and perspective viewing from arbitrary directions. They allow for interactive control of rendering and projection parameters and the direct visualization of dynamically changing volume data.

The underlying algorithm uses lookup-tables for parallel and perspective ray-casting with tri-linear interpolation of samples along rays. The algorithm performs surface normal estimation from cached interpolated samples. Shading and classification are part of the pipeline and do not require any preprocessing. The algorithm allows for a pipelined implementation of ray-casting where each voxel is accessed exactly once per projection.

All architectures presented in this dissertation use a high-bandwidth vector memory system. A linear skewing of the address space allows for conflict-free access to scanlines of the volume data. Cube-3 implements ray-parallel ray-casting where data along viewing rays are processed in parallel. However, the architecture requires a global communication network which ultimately limits the performance and the scalability of Cube-3.

Cube-4 operates on data of several neighboring rays. Depending on how the algorithm proceeds we call this approach beam- or slice-parallel. The beam-parallel approach follows a group of rays in viewing direction, whereas the slice-parallel approach operates on data slices parallel to a face of the volume data. We developed an innovative parallel dataflow scheme that requires no global communication except at the pixel level. This combines the benefits of very high memory bandwidth, modularity, and scalability, a result which has not been achieved before. Possible hardware implementations are PCI boards for 256^3, VME boards for 512^3, and multiple boards for 1024^3 datasets, all with 16-bit voxels and 30 projections per second.

We have simulated the algorithm and architectures and have implemented a working prototype of the Cube-4 slice-parallel hardware on the Teramac configurable custom computer at Hewlett-Packard Labs. Our results indicate true real-time performance for high-resolution datasets and linear scalability of performance with the number of processing pipelines.

Contents

List of Tables **xi**

List of Figures **xii**

Acknowledgements **xix**

1 Introduction **1**

1.1 Design Objectives 4

1.2 Design Approach 5

1.3 Contributions of this Dissertation 6

1.4 Dissertation Outline 7

2 Preliminaries **10**

2.1 Volume Rendering Basics 10

2.2 Image-order Algorithms 12

2.3 Object-order Algorithms 14

2.4 Domain Methods 17

2.5 Processing Requirements 20

2.6 Acceleration Techniques 21

2.7 Summary 26

3 Related Work **28**

3.1 General-Purpose Supercomputers 28

3.2 Graphics Accelerators 29

3.2.1 The PIXAR and PIXAR II Image Computer 30

3.2.2 AT&T Pixel Machine 900 30

3.2.3 The Princeton Engine 31

3.2.4 Pixel-Planes 5 31

3.2.5 PixelFlow 33

3.2.6 SGI Reality Engine 34

3.2.7 Comparison 36

3.3 Volume Rendering Accelerators 38

3.3.1 The PARCUM System 38

3.3.2 The Voxel Processor 40

3.3.3 The Cube Project 41

3.3.4 VOGUE 43

3.3.5 VIRIM 45

3.3.6 Comparison 47

3.4 Summary 48

4 The Ray-Casting Algorithm **51**

4.1 Ray-Casting Basics 52

4.1.1 Data Traversal 53

4.1.2 Interpolation 54

4.1.3 Gradient Estimation 55

4.1.4 Shading 56

4.1.5 Classification and Compositing 58

4.2 Ray-Casting Bottlenecks 62

4.3 Summary 67

5 Fully Pipelined Ray-Casting **70**

5.1 Template Based Ray-Casting 71

5.1.1 Template Generation 72

5.1.2 The Base-Plane Image 74

5.1.3 Resampling the Volume 77

5.1.4 Warping the Base-Plane Image 79

5.2 ABC Gradient Estimation 80

5.2.1 6-Neighborhood Gradient 81

5.2.2 26-Neighborhood Gradient 83

5.2.3 10-Neighborhood Gradient 83

5.2.4 12-Neighborhood Gradient 85

5.2.5 Proof of Equivalence 86

5.3 Summary 87

6 Parallel Architectures **90**

6.1 Memory System Basics 91

6.2 Memory Interleaving 92

6.3 Memory Systems for Volume Rendering 94

6.3.1 Block Memory 96

6.3.2 Vector Memory 98

6.4 Skewed Memory Organization 99

6.5 Memory Partitioning for $m < n$ 101

6.6 Pipelining and Unit Replication 103

6.7 Parallel Pipelined Ray-Casting 104

6.8 Summary 107

7 Cube-3: The Ray-Parallel Architecture **108**

7.1 Cube-3 Ray-Parallel Dataflow 109

7.1.1 Sheared Tri-Linear Interpolation 110

7.1.2 ABC Gradient Estimation 116

7.2 Cube-3 Analysis 117

7.2.1 Tri-Linear Interpolation Comparison 118

7.2.2 ABC Gradient Estimation Comparison 119

7.3 Cube-3 Architecture 122

7.4 Problems with Cube-3 124

7.5 Summary 125

8 Cube-4: The Beam-Parallel Architecture **127**

8.1 Un-skewed Beam-Parallel Dataflow 128

8.2 Skewed Beam-Parallel Dataflow 132

8.2.1 Tri-Linear Interpolation 134

8.2.2 ABC Gradient Estimation 135

8.2.3 Compositing 136

8.2.4 Steps in X and Y 137

8.2.5 Projection in Reverse Major Direction 138

8.2.6 Perspective Projection 138

8.3 The Cube-4 Beam-Parallel Architecture 139

8.4 Performance and Scalability 142

8.5 Summary 145

9 Cube-4: The Slice-Parallel Architecture **147**

9.1 Cube-4 Slice-Parallel Dataflow 148

9.1.1 Tri-Linear Interpolation 149

9.1.2 ABC Gradient Estimation 155

9.1.3 Shading and Classification 161
9.1.4 Compositing 162
9.2 Extensions for $m < n$ 164
9.3 Perspective Projections 165
9.4 Cube-4 Slice-Parallel Architecture 168
9.5 Simulations and Prototyping 170
9.6 Performance Analysis 174
9.7 Summary 176

10 Conclusions **177**
10.1 Future Directions 180

Bibliography **183**

List of Tables

1 *Comparison of Graphics Accelerators.* 36

2 *Comparison of Volume Rendering Accelerators.* 47

3 *Volume Rendering Accelerator Hardware.* 48

4 *Compositing costs for a ray with n samples.* 67

5 *Cube-4 Control Information.* 140

6 *Memory frequencies for $p = 30$ frames per second.* 142

7 *Valid choices for gradient estimation in Z between the ahead and behind slice depending on the non-major components d_x and d_y of the viewing vector.* 159

8 *Timing schedule for two pipelines (0 and 1) of the 1D convolution SFG.* 167

9 *VHDL simulation results: Rendering performance as a function of the number of rendering pipelines.* 170

List of Figures

1 *Volume renderings of a MRI scan of a human head.* 2
2 *Experimental Design Approach.* 5
3 *Dissertation Outline.* 8
4 *Ray-Casting Algorithm.* 13
5 *Splatting Algorithm.* 15
6 *Shear-Warp Algorithm.* 16
7 *Frequency Domain Rendering. (FT = Fourier Transform, IFT = Inverse Fourier Transform)* 18
8 *Compression Domain Rendering.* 19
9 *Pixel-Planes 5 Architecture.* 32
10 *The PixelFlow Architecture.* 33
11 *Planar Texture Resampling.* 35
12 *PARCUM System Architecture.* 39
13 *Voxel Processor Architecture.* 40
14 *VOGUE Architecture.* 44
15 *VIRIM Architecture.* 45
16 *Graphical Conventions.* 52
17 *Coordinate Systems.* 53
18 *Basic Ray-Casting Routine.* 54

19 *Vectors for Local Illumination Models.* 57
20 *Cells with more than one resampling location.* 64
21 *Overlapping cells during gradient computation.* 65
22 *Schematic diagram of the ray-casting pipeline and its performance bottlenecks. Each stage either requires random access to the volume memory, excessive arithmetic processing, or both.* . 68
23 *Discrete rays are cast along cells from the base-plane into the dataset.* 71
24 *Template Generation Pseudo-Code.* 72
25 *A pre-computed template stores discrete ray steps and interpolation weights for sample points along the rays.* 73
26 *Perspective ray-casting requires 2D templates of size n^2 each.* . 74
27 *Varying size of the base-plane depending on the viewing angle.* 75
28 *Switch of the base-plane for viewing angle $\alpha > 45^0$.* 76
29 *Sampling of the base-plane image for perspective projections.* . 76
30 *Sampling distance and interpolation method along rays.* 77
31 *ABC Gradient Estimation.* 81
32 *6-Neighborhood Gradient.* 82
33 *26-Neighborhood Gradient.* 83
34 *10-Neighborhood Gradient.* 84
35 *12-Neighborhood Gradient.* 85
36 *Proof of Equivalence.* 86
37 *Fully Pipelined Ray-Casting Algorithm.* 88
38 *Low-order interleaved memory system with m memory modules. Memory address a is assigned to memory module k according to $k = a \bmod m$.* 93

39 *Assignment of voxel addresses [zyx] in volume space for a 4×4×4 dataset. Numbers at grid points indicate the memory module where the voxel is stored.* 95
40 *Assignment of voxels to memory modules in an 8-way low-order interleaved volume memory.* 97
41 *3D skewed memory organization for a $4 \times 4 \times 4$ dataset. a) Dataset stored in $m = 4$ memory modules. b) Dataset stored in $m = 2$ memory modules. Thick lines indicate slice boundaries inside the memory.* 100
42 *Beams and the partitioning of beams into partial beams in volume space.* 102
43 *Processor Pipeline for Ray-Casting.* 103
44 *Parallel Ray-Casting Pipelines.* 104
45 *Three different approaches to parallel ray-casting. Shaded voxels are processed simultaneously. The dashed arrows indicate in what direction the algorithm proceeds during subsequent timesteps.* 105
46 *The Cube-3 (a) and Cube-4 (b) Basic Architectures.* 106
47 *Cube-3 Dataflow.* 110
48 *Problems with Discrete Ray Interpolation.* 111
49 *Sheared Tri-Linear Interpolation.* 112
50 *Variable ray offsets in major direction and maximum offset calculation.* 114
51 *Maximum Offset Calculation.* 115
52 *Dataset rendered with sheared tri-linear interpolation (left) and the difference image to traditional volume rendering (right) for 45° rotation angle. This is the worst case for sheared tri-linear interpolation.* 119

53 *Sheared Interpolation Error in Percent.* 120
54 *Average Error Magnitude for ABC Gradient Estimations Compared to the Analytic Normal.* 121
55 *Error magnitude of comparing surface normals of 10- (Top) and 26-neighborhood gradients (Bottom) to the true analytic normal of the voxelized sphere. Notice the jump of regions of high error for the 26-neighborhood gradient between* 45° *and* 50° *rotation angle. Dark:* $0^0 \leq |e| < 8.5^0$*, Medium:* $8.5^0 \leq |e| < 20^0$*, Light:* $20^0 \leq |e| < 31.5^0$*, White:* $|e| \geq 31.5^0$*. Rotation angles (left to right):* $30^0, 35^0, 40^0, 45^0, 50^0, 55^0, 60^0$. 122
56 *Cube-3 System Overview.* 123
57 *Un-skewed Dataflow Example.* 129
58 *Skewed Beam-Parallel Dataflow.* 133
59 *3D skewing. Numbers indicate* x *coordinate of voxels stored in module number* $k = (x + y + z)\text{mod}4$. 133
60 *12-neighborhood Gradient Correction.* 138
61 *a) Divergence of rays during perspective projection. b) Diverging perspective rays with unused beam inbetween.* 139
62 *Cube-4 Beam-Parallel Architecture.* 141
63 *Processing frequency,* f_p*, vs. number of units per pipeline stage,* $\frac{n}{m}$*, for three dataset resolutions. Rendering performance is* $p = 30$ *frames per second.* 144
64 *Bi-linear neighborhoods a) in volume space and b) in skewed memory space.* 150
65 *a) SFG for bi-linear interpolation. Edges labeled with* D *indicate that the first operation must precede the second operation by* D *clock cycles. b) Each node performs a linear interpolation.* . . 150

66 *Bilinear Interpolation Schedule (Part I)* 152
67 *Bilinear Interpolation Schedule (Part II)* 153
68 *SFG for tri-linear interpolation and $n = m = 4$. Edges labeled with n indicate that the first operation must precede the second operation by n clock cycles.* 155
69 *Gradient neighborhood inside a slice of interpolated samples a) unskewed and b) skewed.* 156
70 *a) SFG to compute central differences between samples for the X gradient. b) Nodes either forward the inputs or compute their difference.* 157
71 *a) SFG to compute central differences between samples for the Y gradient. Edges labeled with D indicate that the first operation must precede the second operation by D clock cycles. b) Nodes either forward the inputs or compute their difference.* 158
72 *Different choices for gradient correction in Z depending on the non-major component d_y of the viewing vector. a) $d_y > 0$ b) $d_y < 0$* 158
73 *Four possible choices among samples in the ahead or behind slice during gradient estimation in Z for the samples of the current slice.* 159
74 *a) SFG to compute and select corrected samples inside each slice for the Z gradient. Edges labeled with D indicate that the first operation must precede the second operation by D clock cycles. b) Nodes are either linear interpolators or multiplexors to select valid corrected samples depending on the viewing direction.* . . . 160

75 *SFG for ABC Gradient Estimation. Edges labeled with n or $2n$ indicate that the first operation must precede the second operation by n or $2n$ clock cycles, respectively.* 161
76 *Compositing Neighborhood.* 163
77 *a) SFG for compositing. b) Operation of a compositing node. C stands for front-to-back or back-to-front compositing. α is the current sample opacity, b is the current sample value, a is the partially accumulated result, and a' is the new result of the current compositing operation.* 164
78 *Beam Extension.* 165
79 *a) SFG for 1D convolution with a 1×3 filter. b) Operation of each convolution node.* 167
80 *a) SFG for 2D convolution with a 3×3 filter. Edges labeled with D indicate that the first operation must precede the second operation by D clock cycles. b) Operation of each convolution node.* 168
81 *Cube-4 slice-parallel architecture. Bold lines indicate all data connections of the rendering pipeline in the center. (CFB = Cubic Frame Buffer, TRILIN = Tri-Linear Interpolation Unit, Compos = Compositing Unit.)* 169
82 *a) A four-board Teramac system. b) A Teramac multichip module (MCM) with 27 custom FPGAs routed through 39 signal layers.* 172
83 *Different volume renderings of a 128^3 CT lobster dataset generated by the Cube-4 prototype on the Teramac. Each image took 0.6 seconds at 0.96 MHz processing frequency.* 172

84 *Volume rendering images of* 128^3 *datasets produced by the Cube-4 prototype on Teramac. Each image took 1.5 seconds at 0.25 MHz processing frequency. a) Hippocampal pyramidal cell. b) Volume-sampled mechanical part. c) Bullfrog ganglion cell. d) CT head,* 45° *rotated. e) Volume sampled sphere-flake. f) MRI brain. Images g) through i) show the effect of different transfer functions on a simulated high-potential iron protein.* 173

85 *Theoretical rendering performance of Cube-4 as a function of the number of rendering pipelines. We show graphs for different dataset sizes (*$\bigcirc = 256^3, \Lambda = 512^3, \triangle = 1024^3$*). Solid lines indicate 33 MHz processing frequency, and dashed lines indicate 100 MHz processing frequency.* 175

Acknowledgements

"We act as though comfort and luxury were the chief requirements of life, when all that we need to make us happy is something to be enthusiastic about."

Albert Einstein

I sincerely thank my advisor, Leading Professor Arie Kaufman, for his encouragement and guidance that made this thesis possible. His enthusiasm, support, and continuing interest in my work have been particularly invaluable. I thank the members of my dissertation committee, Professor Tzi-cker Chiueh, Professor Amitabh Varshney, and Dr. Bengt-Olaf Schneider, for their time and encouragement. I am also indebted to Professor David Smith who served on previous exam committees and provided guidance in architecture and VLSI related questions. Many thanks to the Stony Brook faculty, in particular Steve Skiena, David Warren, and Anita Wasilewska, for their support and friendship. Several people read drafts of this dissertation and provided a valuable outside perspective; my sincere thanks to Ute Appenzeller, Ingmar Bitter, Barthold Lichtenbelt, and Shigeru Muraki.

The Visualization Laboratory at Stony Brook was a nurturing and fun research environment. I would like to thank Rick Avila, Dirk Bartz, Ingmar

Bitter, Baoquan Chen, Taosong He, Lichan Hong, Claudio Silva, Lisa Sobierajski, and Sidney Wang for being such a great and inspiring group of people. Many visiting researchers contributed to the lively atmosphere at the lab. I would especially like to thank Issei Fujishiro, Xiaoyang Mao, Shigeru Muraki, and Ikuko Takanashi for their support. I also wish to thank "Paper Queen" Stella Mannino, Kathy Germana, and Betty Knittweis for their smiles, care, and commitment with which they handled all administrative matters. Many thanks to our system administrators, Brian Tria and Pat Tonra, who miraculously kept the computers going.

A lot of people have contributed to the Cube project and to the ideas in this dissertation. Reuven Bakalash, Robert Pacheco, and Roni Yagel helped me getting started. Frank Wessels, on exchange from the University of Twente, the Netherlands, provided many ideas and stimulating discussions that were invaluable for the development of the Cube-4 architecture. Urs Kanus and Michael Meißner, on exchange from the University of Tübingen, Germany, approached the project with unbounded enthusiasm and implemented a working hardware prototype of Cube-4 at Hewlett-Packard Laboratories. I would like to thank Professor Jaap Smit of the University of Twente and Professor Wolfgang Straßer of the University of Tübingen for their support that made these student exchanges possible. I also thank Mohammed Abdelsadek, Kwasi Mireku, and Fayyaz Younas for their evaluation of implementing Cube-4 on DSP processors.

Special thanks to Fred Kitson, Tom Malzbender, and the Teramac design team of Richard Carter, Bruce Culbertson, Philip Kuekes, Greg Snider, and Rick Amerson at Hewlett-Packard Laboratories in Palo Alto, CA. Their support of the Cube project and their excellent technical expertise made the hardware implementation of Cube-4 on Teramac possible.

Life in Stony Brook would never have been this enjoyable without Simon "Momo" Richard and "the family" of Bas Peeters, Leslie "Van Doren" Bryan, Kamla Ahluwalia, Daphnee Yourek, and Jimmy Dee. The inhabitants and visitors at 1466 Stony Brook Road and the Checkmates provided all the friendship, fun, and gossip to make my student days truly memorable. Many thanks to Khalid Abdalla, Virginie Charette, Neil Katz, Marc Laforest, Janice McCarthy, Jimmy Petean, Heberto "cabron" del-Rio, Jean Rousseau, Lori and Tony Scarlatos, Roberta Sullivan, and Leticia Toledo. "Hejödele" to my fellow Swiss friends at Stony Brook, namely "Tante" Ju, Alain Rueff, and Suzie Atanasoski-Rueff. And – of course – a full-hearted "Bescheid" to the Klagenfurt Gang with Charlie Werlen, Martin Rutishauser, Lukas von Känel, Andy Jacob, Leo Felber, and Gunther Lorch.

I am greatly indebted to the following institutions that enabled me to study in the United States: Foreign Student Exchange at the ETH Zürich, U.S. Government Fulbright Agency, Institute of International Education, ABB Switzerland, and the Swiss Academy of Technical Sciences. Many thanks to Professor Alessandro Birolini, Professor Jean Nicoud, and Dr. Laurent Badoux for their help.

Finally, I would like to thank my mother for her unfailing encouragement and support, and Ute, whose love has been the source of my inspiration and my strength.

This work has been supported by the National Science Foundation under grants MIP-8805130, CCR-9205047, MIP-9527694, the Department of Energy under the PICS grant, Japan Radio Corporation, and Hewlett Packard.

Chapter 1

Introduction

Visualization of scientific, engineering or biomedical data is a growing field within computer graphics. In many cases the objects or phenomena being studied are *volumetric*, typically represented as a three-dimensional grid of volume elements, or *voxels*. 3D medical acquisition techniques, such as Magnetic Resonance Imaging (MRI) and Computed Tomography (CT) [SF90a], produce a sequence of 2D cross sections of internal organs that can be reconstructed into a 3D dataset [UH84]. Confocal microscopes are being used to acquire the 3D internal structure of cells in bio-medical applications [BSVN89]. Supercomputer simulations based on 3D finite element models or computational fluid dynamics produce large amounts of volumetric data for a variety of applications, such as aerodynamics, weather forecast, or astrophysics. Another source of volume data are geometrical models that have been synthesized (voxelized) into 3D datasets using techniques of *volume graphics* [KCY93, Wan95].

Volume rendering has become a key technology in the visualization of volumetric data [Kau91]. It encompasses an array of techniques for displaying images directly from the 3D data without intermediate conversion to a surface

representation. Volume rendering allows to fully reveal the internal structure of the 3D data, including amorphous and semi-transparent features. Figure 1 shows a volume rendering of a MRI scan of a human head.

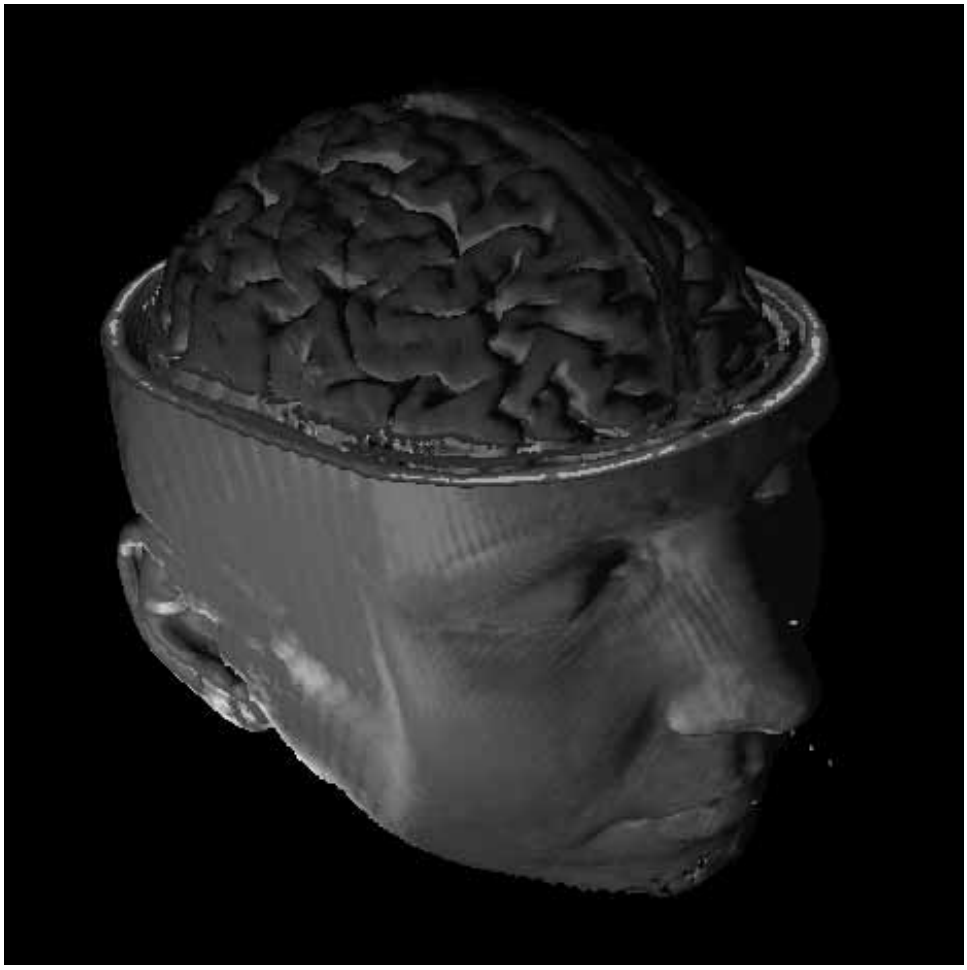

Figure 1: *Volume renderings of a MRI scan of a human head.*

It is essential for the investigation and understanding of 3D datasets to provide real-time frame rates, typically defined to be more than 10 and preferably 30 frames/sec. The real-time rotation of 3D objects in space under user control makes the renderings appear more realistic due to the kinetic depth

effects [SM93]. In many applications volume rendering may also be used to directly view changes of *dynamic* 3D data over time for *4D (spatial-temporal) visualization*. Examples are the real-time analysis of an in-vivo specimen under a confocal microscope, the real-time study of in-situ fluid flow or crack formation in rocks under Computed Micro-Tomography (CMT), or the visualization of a beating heart under real-time 3D ultrasound.

Of particular importance in volume visualization is immediate visual feedback during the interactive change of rendering parameters (e.g., data classification and shading). When exploring new data, the image depends to a large extent on the subjective goals of the user. Real-time visual feedback is essential for the experimentation with different rendering parameters and the exploration of the parameter space.

Real-time volume rendering is extremely compute-intensive. This problem is aggravated by the continuing trend towards larger datasets. The advent of high-resolution sampling devices, faster supercomputers, and more accurate modeling techniques will make high-resolution datasets the norm. However, rendering a high-resolution dataset of 1024^3 16-bit voxels at 30 Hz, for example, requires 2 GBytes of storage, a memory transfer rate of 60 GBytes per second, and approximately 300 billion instructions per second. This is assuming each voxel is accessed exactly once, with a rendering cost of only 10 instructions per voxel. These requirements are several orders of magnitude beyond the capabilities of existing conventional machines.

The high computational requirements of traditional computer graphics led to the development of special-purpose graphics engines, primarily for polygon rendering. Similarly, the special needs of volume rendering, where an image must be computed rapidly and repeatedly from a volume dataset, lends itself to the development of special-purpose volume rendering architectures. A

dedicated accelerator, which separates volume rendering from general-purpose computing, seems to be best suited to provide real-time volume rendering on standard deskside or desktop computers. It may also lead to the direct integration of volume visualization hardware with real-time acquisition devices, in much the same way as fast signal processing hardware became part of today's scanning devices.

1.1 Design Objectives

The main goal of this research is to develop a special-purpose real-time volume visualization architecture for high-resolution datasets that will support 4D volume visualization. We have set the following design objectives based on what we believe to be important features of a real-time volume rendering system:

Real-Time Frame Rates: To create the illusion of smooth motion, the image must be updated a minimum of 24 times per second. The architectures presented in this dissertation aim at achieving projection rates of 30 frames/sec.

4D Visualization: The architecture has to allow for the real-time input of volumetric data. The overall latency of the system should be no more than one frame time.

High-Resolution Datasets: The architecture has to be able to visualize dataset resolutions of $512 \times 512 \times 512$ data samples or higher in real-time. Each sample uses 8- or 16-bits.

Scalability: The design should be modular and the performance should ideally scale almost linearly in the number of modules.

High Image Quality: The images must be of high quality, including surface shading, depth cues, and the provision of transparency. Special care has to be taken to avoid image artifacts such as spatial or temporal aliasing [Wol90].

Interactive Parameter Changes: The algorithm and hardware has to allow for immediate visual feedback during the interactive change of parameters such as shading, data classification, and projection modes.

1.2 Design Approach

The main aspects in the development of any special-purpose hardware are the design of the algorithm and the architecture. Similarly, the design of special-purpose hardware for volume rendering requires an understanding of the relationship between visualization algorithms and hardware structures. However, computer architects occasionally come up with designs that can not be built or are not cost effective. To remedy these problems, we took an experimental approach for this research that combines design, analysis, and prototype implementation, as illustrated in Figure 2 (after [Woo90]).

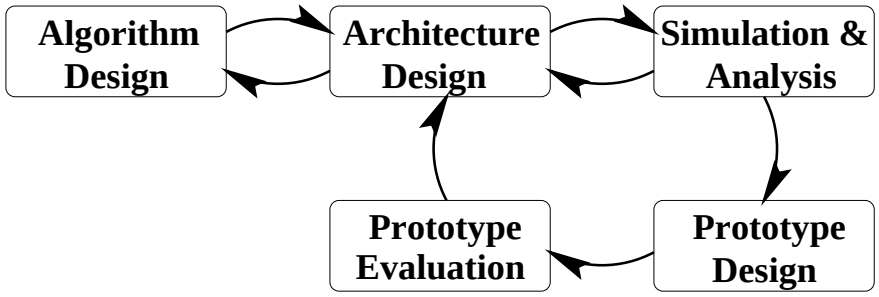

Figure 2: *Experimental Design Approach.*

During algorithm design, we try to recast the basic volume rendering algorithm into a form more suitable for hardware realization. In the architectural domain, this algorithm is reformulated into a datapath structure such as to further improve performance [Kae96]. Both algorithm and architectural design have to be carefully analyzed and simulated to refine the designs and to understand their behavior. Once a sufficiently feasible and simulated solution exists, it is necessary to test it by building an experimental prototype. Only the implementation of real hardware will unveil design errors and performance trade-offs that were not known during analysis and simulation. Performance of the design can be measured on the prototype by rendering real datasets. Finally, we use the experience and performance results from the prototype implementation to further refine the design, possibly by going back to the algorithmic domain. Throughout the project we must keep an eye on the design objectives and try to meet them at all stages of the design process.

1.3 Contributions of this Dissertation

Our research focuses on ray-casting of regular datasets. Ray-casting offers room for algorithmic improvements while still allowing for high image quality. We modified the original ray-casting algorithm to make it better suited for a parallel hardware realization. We developed different parallelizations of the ray-casting algorithm and designed several new real-time volume rendering architectures. During the architectural design we reformulated the algorithm into an efficient memory and datapath structure. We extensively simulated the algorithms and architectures and have implemented a working hardware prototype on the Teramac, a configurable custom computing machine developed by Hewlett-Packard Laboratories.

The main result of our research is Cube-4, a scalable volume rendering architecture that meets all of our design objectives. Cube-4 differs from previous work in that it requires no global communication of data except at the pixel level. Using a specially skewed memory interleaving, we were able to develop a parallel dataflow model that leads to local, fixed-bandwidth interconnections between processing elements. This combines the benefits of very high memory bandwidth, modularity, and scalability, a result which has not been achieved before. Possible hardware implementations of Cube-4 for 30 frames per second projection rates range from an inexpensive PCI board accelerator for 256^3 datasets, to a workstation accelerator board for 512^3 datasets, and to a high-end visualization server for 1024^3 or higher resolutions. The cost-performance ratio of Cube-4 is several orders of magnitude better than existing solutions.

1.4 Dissertation Outline

The material in this dissertation is organized following the design approach outlined in Figure 2. The chapters are roughly grouped according to research phases, as shown in Figure 3.

Chapter 2 provides background and terminology used throughout this dissertation. We describe several approaches to volume rendering and discuss its performance requirements to motivate the design of special purpose hardware. We look at several software acceleration techniques and discuss them in the context of our project goals.

Chapter 3 contains a summary of related research in the field of interactive or real-time volume rendering. We briefly summarize software based techniques on general-purpose multi-processor systems and then discuss several special-purpose graphics and volume rendering accelerators. Up to date, no

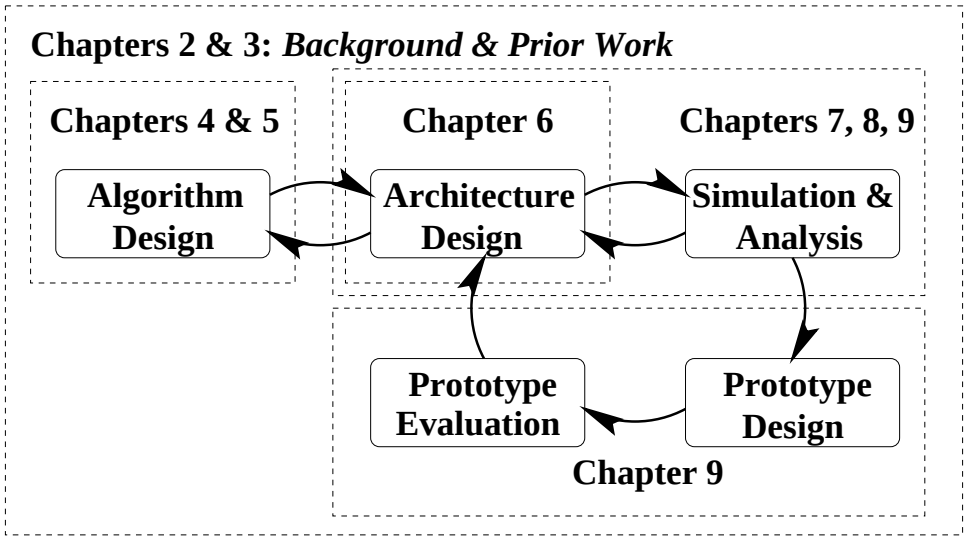

Figure 3: *Dissertation Outline.*

system is capable of delivering the required level of performance as stated in Section 1.1.

In Chapter 4, we introduce the basic ray-casting algorithm. A detailed look at its performance bottlenecks reveals the primary issues in real-time ray-casting, which are uniform and regular memory access, parallel processing, and a fully pipelined dataflow.

In Chapter 5, we examine ways of improving the basic algorithm. We describe two important concepts, namely lookup-table based ray-casting and new methods for surface normal estimation using previously interpolated samples. Both of them lead to uniform memory access and a fully pipelined ray-casting algorithm.

In Chapter 6, we discuss several issues related to the design of high-performance parallel architectures. The chapter starts by giving an overview of the design issues for high-bandwidth memory systems. We then give a formal

description of the parallel memory system that is the basis for all architectures presented in this dissertation. We transform the pipelined ray-casting algorithm into the architectural domain and introduce more parallelism. The framework in this chapter leads us to three ways of parallelizing the ray-casting algorithm, called ray-, beam-, and slice-parallel ray-casting. These parallel algorithms are the basis for the three architectures that we describe in the remaining chapters of this dissertation.

In Chapter 7, we describe the Cube-3 architecture, an implementation of ray-parallel ray-casting. We show how to use coherence between neighboring rays to minimize memory access during interpolation. We evaluate the errors introduced by this approach using experiments and measurements on real and synthetic datasets. We describe the Cube-3 architecture and examine its problems due to the global communication of voxel data.

In Chapter 8, we present the Cube-4 beam-parallel dataflow model that eliminates the global communication bottleneck of Cube-3. We show how this dataflow leads to local interconnections between memory and processing elements. We describe the Cube-4 beam-parallel architecture and discuss issues related to an actual hardware implementation of Cube-4.

In Chapter 9, we present the Cube-4 slice-parallel architecture, the main contribution of this dissertation. We use signal flow graphs to formally describe the dataflow during slice-parallel processing. We also present results from our prototype implementation of Cube-4 on the Teramac custom computing machine at HP Labs. The prototype implementation demonstrates the feasibility of Cube-4. Our performance results confirm the theoretical linear increase in rendering performance with increasing number of processing pipelines. Finally, we summarize our work and suggest areas for future research in Chapter 10.

Chapter 2

Preliminaries

In this chapter, we introduce background material and terminology used throughout the rest of this dissertation. We review the fundamental concepts in volume rendering and summarize the basic volume rendering algorithms. We look at the performance requirements of real-time volume rendering under simplified algorithmic assumptions. We then discuss several acceleration techniques for volume rendering and point out their shortcomings in the context of our project goals.

2.1 Volume Rendering Basics

A volumetric dataset consists of information at sample locations in some space. The information may be *scalar* (such as density in a computed tomography (CT) scan), *vector* (such as velocity in a flow field), or a combination (such as energy, density, and momentum in computational fluid dynamics). The space is usually three-dimensional, either consisting of three spatial dimensions or another combination of spatial and frequency dimensions.

In many applications the data is sampled on a *rectilinear grid*, represented as a 3D grid of volume elements, so called *voxels*. Each voxel is a quantum unit of volume that has a value associated with it representing some property of the object or phenomenon. If all the voxels are identical cubes the dataset is said to be *regular*. Examples of regular or rectilinear datasets include 3D reconstructed medical data, simulated datasets from computational fluid dynamics or computed finite element models, to name just a few (see [Kau91, Chapter 7]). Other types of datasets can be classified into *curvilinear grids*, which can be thought of as resulting from a warping of a regular grid, and *unstructured grids*, which consist of arbitrary shaped cells with no particular relation to rectilinear grids [SK90]. In this work we are concerned with the visualization of scalar values represented in a rectilinear three-dimensional grid of voxels.

There are several ways in which the continuous object or phenomenon represented by the volume dataset can be visualized. One method, referred to as *surface rendering*, uses an intermediate geometrical approximation (e.g., polygons, points) to iso-surfaces inside the three-dimensional scalar field [LC87, CLL+88]. The geometrical constructs can be then viewed on conventional computer graphics workstations. However, the coercion process drastically reduces the information content of the data by making a binary decision as to where surfaces lie. This may introduce artifacts [Dür88], the modeling procedure hampers interactive viewing, and internal and amorphous structures cannot be visualized.

Volume rendering refers to techniques for producing 2D images directly from the volume data without intermediate constructs. Volume rendering techniques offer great flexibility in the choice of viewing and rendering parameters and overcome the limitations of surface rendering [Kau91]. For a detailed

discussion of the underlying optical model of volume rendering see [Max95]. Depending on how voxels are processed to generate an image, volume rendering algorithms can be divided into three classes: *image-order algorithms*, *object-order algorithms*, and *domain methods*. We will briefly summarize some representative algorithms in each class and illustrate their advantages and disadvantages. Our presentation is limited to algorithms used for the rendering of scalar rectilinear datasets. Important classes of volume rendering methods for non-regular scalar grids, such as cell projection methods [MHC90, WG91], and for vector data, such as flow-field visualization [CM92], will not be discussed. A more comprehensive summary of volume rendering algorithms can be found in [Elv92, Kau91, SFF91].

2.2 Image-order Algorithms

Image-order algorithms iterate over all pixels of the output image and determine the contributions of voxels towards the pixel currently being composed [TT84, Lev88, Sab88, DH92].

Ray-Casting

Ray-casting is the most commonly used image-order technique. It simulates optical projections of light rays through the dataset, yielding a simple and visually accurate mechanism for volume rendering. Rays are cast from the viewpoint through screen pixels into the volume [Lev88]. The contributions of all voxels along the rays are calculated and used to find the pixel color.

Ray-casting offers very high image quality, including the ability to provide

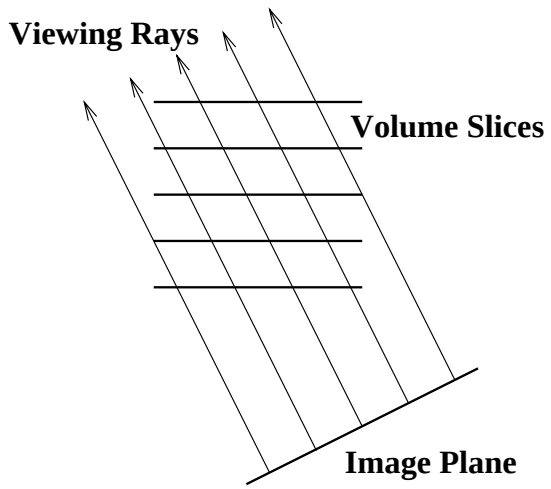

Figure 4: *Ray-Casting Algorithm.*

visual depth cues such as shaded surfaces, shadows, and depth cueing. Unfortunately, these advantages come at the price of high computational requirements. Ray-casters do not access the volume data in storage order because of the arbitrary traversal direction of the viewing rays. This leads to poor spatial locality of data references and requires expensive address arithmetic to calculate the new resampling locations.

Volumetric Ray-Tracing

In contrast to the related surface based ray-tracing techniques [Whi80], ray-casting only uses a local illumination model and does not include effects of ray reflections or refractions. *Volumetric ray-tracing* methods try to overcome this limitation by including global illumination and higher-order light-material interactions.

Yagel et al. [YCK92] propose a *discrete ray-tracing* method that leads to fast implementations because all ray-traversal calculations can be performed

using integer arithmetic. Sobierajski and Kaufman [SK94, Sob94] developed a complete framework for volumetric ray-tracing that includes global illumination effects such as shadows, reflections of the scene in mirrors, light interaction between multiple volumetric and geometric objects, and volumetric rendering effects such as fog and transparency. The results have been incorporated into *VolVis*, a comprehensive volume visualization system developed at SUNY Stony Brook[1] [ASK92, AHH+94].

Volumetric ray-tracing yields the highest image quality of all volume rendering algorithms. However, it is computationally much more expensive than ray-casting due to the recursive generation of multiple rays per pixel.

2.3 Object-order Algorithms

Object-order algorithms iterate over the volume data and determine the contribution of each voxel to the screen pixels. All voxel contributions combined yield the final image.

Splatting

Splatting, introduced by Westover [Wes91], convolves every voxel in object space with a 3D reconstruction filter and accumulates the voxels contribution on the image plane. In practice, the computationally expensive three dimensional convolution is replaced by simpler 2D convolution filters. Lookup tables for the filter weights avoid excessive arithmetic complexity [Wes89]. Using this approximation, the splatting algorithm reduces to projections of object space slices onto the image plane.

[1] VolVis is freely available by ftp. To acquire a copy of send email to volviscs.sunysb.edu.

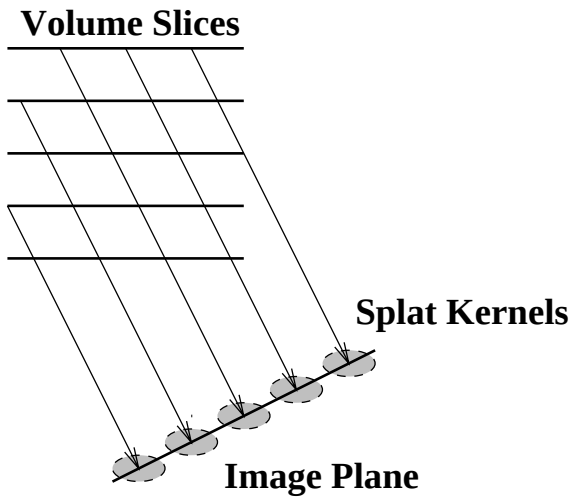

Figure 5: *Splatting Algorithm.*

The reconstruction filter is view-dependent, and its footprint has to be transformed into image space for each change in viewing direction. For perspective projections this footprint is different for each voxel. Because the computation of the footprints and of the filter weights is expensive, Westover proposes to use lookup-tables for rotation-invariant Gaussian resampling filters. The approximations to the ideal reconstruction filter lead to compromises in image quality, typically in the form of blurred edges and reduced image contrast. The main advantage of splatting is the storage-order access to the data. This method, in combination with filter kernel approximations, allows to trade-off image quality for fast algorithm execution.

Volume Shearing

Volume shearing algorithms resample the volume data from object space to the image coordinate space so that the resampled voxels line up on the viewing axis in image space [DCH88, UK88]. The voxels are then composited onto

the viewing plane using ray-casting along axis-aligned viewing rays. The 3D affine transformation between object space and image space can be decomposed into three sequential 1D shear operations [Han90]. Alternatively, the

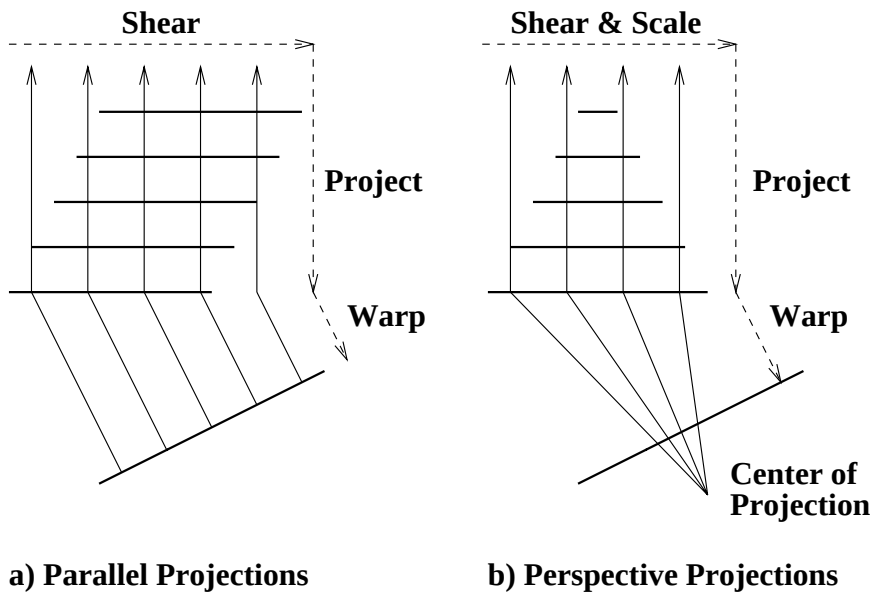

Figure 6: *Shear-Warp Algorithm.*

viewing transformation can be decomposed into a shear and a 2D image warping operation [RGC87, LL94]. Perspective projections require an additional transformation, typically in the form of a scale operation of the sheared data slices [VFR92, LL94].

Volume shearing algorithms are very efficient due to the combination of object order traversal of the data and the scanline-oriented resampling operations. However, multipass 1D resampling algorithms suffer from high memory overhead due to the multiple accesses to each voxel. To avoid view-dependent performance bottlenecks, the volume must be transposed and stored multiple times as the viewpoint changes [CS80, LL94]. Furthermore, current shear-warping algorithms use only a 2D reconstruction filter which may lead to

image artifacts due to high-frequency aliasing in the third dimension. Shear-warping is currently the fastest software volume rendering algorithm, achieving one projection in a few seconds on a regular workstation [LL94] or sub-second display times on a shared-memory parallel multiprocessor machine [Lac95b].

2.4 Domain Methods

The spatial 3D data is transformed into another domain, such as the frequency, compression, or wavelet domain. A projection is generated directly from the domain data.

Frequency Domain Rendering

Frequency domain rendering algorithms are based on the Fourier projection slice theorem [MO74]. This theorem states that a projection of the 3D data volume in a certain view direction can be obtained by extracting a 2D slice perpendicular to that view direction out of the 3D Fourier spectrum and then inverse Fourier transforming it. At run-time only a 2D Fourier transform is needed to obtain the projection, reducing the computational complexity for volume rendering from $O(n^3)$ to $O(n^2 \log n)$. Dunne, Napel, and Rutt [DNR90], Malzbender [Mal93], Malzbender and Kitson [MK93], and Levoy [Lev92] have employed the theorem on volumetric datasets to obtain projections.

A major problem of frequency domain volume rendering is the fact that the obtained projection is a line integral along the view direction which does not exhibit any occlusion effects. The resulting images are X-ray-like in that the attenuation effects are not accounted for. Totsuka and Levoy [TL93] proposed a linear approximation to the exponential attenuation [Sab88] and an

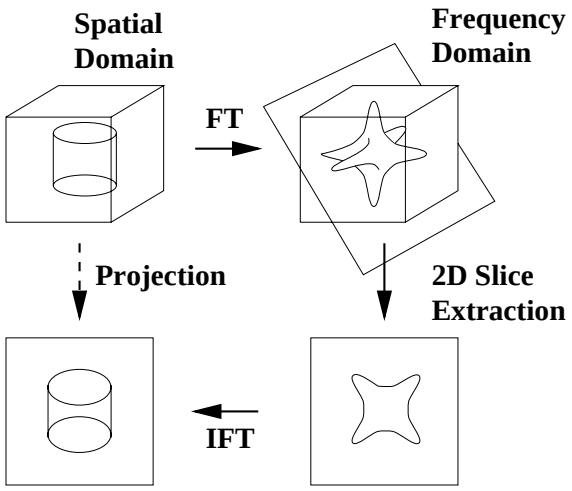

Figure 7: *Frequency Domain Rendering. (FT = Fourier Transform, IFT = Inverse Fourier Transform)*

alternative shading model to fit the computation within the frequency-domain rendering framework. Although their images show improved visual depth cues, the inaccuracy of the linear attenuation model and the lack of occlusion is still noticeable.

Compression Domain Rendering

Effective compression methods are imperative to reduce the cost of storing and transmitting the vast amounts of 3D datasets. *Compression domain rendering* allows to perform volume rendering directly on the compressed datasets without the need for a separate decompression step before rendering. Ning and Hesselink [NH92, NH93] use vector quantization in the spatial domain in order to allow for arbitrary voxel access and directly render out of the compressed domain using regular spatial domain volume rendering algorithms.

Chiueh et al. [CHKP94] propose a Fourier transform based compression and volume rendering technique. The slice projection theorem is used on

subcubes in the frequency domain to obtain average subcube intensities on projected subimages. Spatial compositing techniques are used to combine these subimages according to an exponential attenuation model. The result-

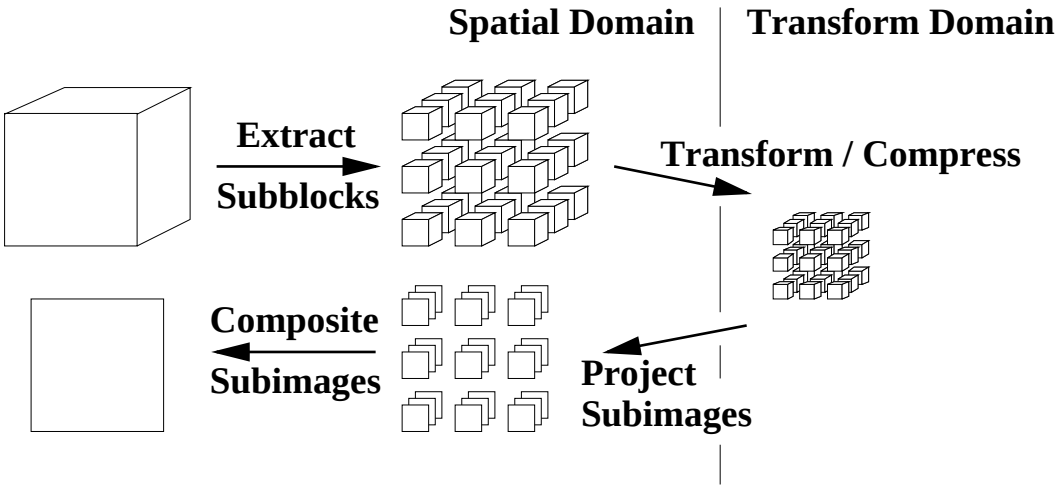

Figure 8: *Compression Domain Rendering.*

ing images show occlusion and improved attenuation effects and are hard to distinguish from those generated by conventional spatial volume rendering techniques.

Using a similar subcube compression approach based on the discrete cosine transform (DCT), Yeo and Liu [YL95] achieve high compression ratios and virtually no loss of image quality when rendering out of the compressed DCT domain. Other researchers have used the hierarchical compression properties of the wavelet transform [Mur93, LG95].

Compression domain methods will become increasingly important with the advent of larger datasets. However, the initial domain transformation is time consuming and does not allow for real-time update of the dataset. Although compression domain methods offer interesting possibilities for the future, at present they seem not well suited for fast, high-quality volume visualization.

2.5 Processing Requirements

The main activities in computing are storage, processing, and communication of data [Wor91]. Volume rendering pushes the limits of current technology in each of these areas. To get a feeling for the processing requirements, we consider an average CT study of $512 \times 512 \times 64$ (2^{24}) voxels that we visualize using a straight forward software implementation of volume rendering. In order to project an image of the data, the volume rendering algorithm has to access each voxel at least once per projection. At the very least, the algorithm needs to compute sample values throughout the volume that are subsequently displayed on the screen. Because these samples may not fall directly on the voxel grid, their value must be estimated from the surrounding voxels using an interpolation technique. The simplest 3D resampling with acceptable image quality is tri-linear interpolation and requires about 13 multiplications and 20 additions per sample [HCN94] (see Section 4.1.2). Finally, as stated in our projects goals, we seek real-time rendering rates of 30 projections per second.

A lower bound of the estimated processing performance would be:

$$2^{24} \text{ samples} \times 30 \text{ instructions} \times 30 \text{ frames/sec}$$

or about 15 billion (15×10^9) instructions per second. To access the 16 MByte dataset 30 times per second requires 30×2^{24} or about 500 MBytes/sec bandwidth between processor and memory. Rendering a high-resolution dataset of 512^3 16-bit voxels at 30 Hz, for example, requires 256 MBytes of storage, a memory transfer rate of 7.6 GBytes per second, and approximately 120 billion instructions per second. These requirements are several orders of magnitude beyond the capabilities of existing conventional machines.

These estimations are based on conservative assumptions. As we will show in Chapter 4, typical volume rendering algorithms access each voxel multiple

times per projection. For example, tri-linear interpolation requires to access eight voxels for each interpolated sample. Furthermore, 30 instructions per sample are not enough for an algorithm that tries to achieve acceptable image quality.

2.6 Acceleration Techniques

Given these performance requirements it becomes clear that a brute-force software implementation of volume rendering requires an excessive amount of processing. It is therefore not surprising that many high-level algorithmic optimizations and data structures have been developed to speed up algorithm execution.

Coherence Optimization

The most powerful acceleration technique is to avoid computations in transparent regions of the volume data. This is commonly referred to as *coherence optimization*. A user-defined classification function encodes non-transparent areas of the data into hierarchical [Mea82, Lev90, DH92, SF90b] or run-length encoded [RGC87, LL94] data structures.

A drawback of hierarchical data structures is that they must be accessed several times per projection, leading to multiple traversals and to redundant computation. Run-length encoding of the data leads to more regular data access and provides lossless data compression. The resulting data reduction and regular object-order data access is used in current implementations of shear-warp algorithms, which store three copies of the run-length encoded dataset, one for each major viewing direction.

An inherent problem of coherence optimization is that the performance is

data- and classification-dependent. Published performance results are typically achieved with an almost iso-surface like classification, where only 5% of the voxels are non-transparent [LL94]. Mostly transparent data or datasets with a wide range of values, such as temperature distributions, lead to much degraded performance.

Furthermore, any change of the data or the classification function requires a time consuming pre-processing step to recompute the coherence data structures (over 70 seconds for shear-warp rendering [LL94]). However, pre-processing inhibits data classification with immediate visual feedback, one of our project goals for interactive volume rendering. Furthermore, it does not allow for 4D visualization with direct data input.

Space-Leaping

Empty regions of the volume data may be skipped using pre-computed spatial data structures. In contrast to coherence optimization, the skipping of empty space does not have to take the opacity classification of the data into account. Levoy [Lev90] uses an octree data structure to skip empty subvolumes of the data during ray-casting. A *bounding box* around relevant information in the volume data allows to reduce computation by clipping the rays against the bounding box. Avila et al. [ASK92] use convex polyhedral shapes as bounding volumes and available graphics hardware to efficiently handle ray clipping.

The *proximity cloud* method [Sra94, ZKV92] pre-computes a distance function to indicate the radial distance from each voxel in the data to the closest voxel above a certain threshold. The result is stored as a distance volume that can be used during ray-casting. If a ray encounters a proximity voxel containing value l it can safely take an l-step leap forward without missing

small features of the data. Yagel et al. [YCK92] use a fast *discrete line algorithm* (e.g., an integer based 26-connected discrete line algorithm) in the empty space and a slower but more accurate line traversal (e.g., a 6-connected floating point continuous line algorithm) in the vicinity and interior of objects.

Although empty space skipping is a powerful technique, it requires a preprocessing step. The achievable speedup is data dependent. However, space-leaping is preferable over coherence optimization because it is independent of data classification.

Early Ray Termination

Early ray termination is a widely used method to speed up ray-casting [Lev90, AK90, DH92]. The accumulation of new samples along a ray is terminated as soon as their contribution towards the currently computed pixel becomes minimal. Typically, the ray is terminated as soon as the accumulated ray opacity reaches a certain threshold since any further samples along the ray would be occluded. Using this technique, Levoy reports a two-fold speedup for ray-casting of medical datasets [Lev90]. More general methods terminate rays according to a probability that increases with increasing accumulated ray opacity [AK90, DH92], or decrease the sampling rate as the optical distance to the viewer increases [DH92].

Reynolds et al. [RGC87] and Lacroute and Levoy [LL94] use early ray termination in object-order shear-warp algorithms. They observe that transformed and resampled object space voxel scanlines line up with pixel scanlines of the rotated image. Each voxel and image scanline is traversed simultaneously, and samples are accumulated onto corresponding pixels. The run-length encoding of voxel and pixel scanlines allows to skip empty voxels and occluded pixels. A warp of the 2D image results in the final projection.

However, similar to coherence optimizations, early ray termination only proves effective for classification functions that yield high opacity values throughout the data. The performance is dataset and classification dependent. The opacity and run-length encoding of the volume in object-order shear-warp algorithms requires a time consuming pre-processing step.

Pre-calculation of View-Independent Data

Pre-calculation of view-independent data is a general technique that is widely used by most acceleration methods. View independent attributes, such as data classification, normals, colors, or other shading information, are evaluated at the grid points of the volume in a pre-processing step. The pre-computed values are interpolated during rendering to obtain values at the non-integer sample positions.

Although substantial speedups can be achieved using pre-computed data, the storage overhead per voxel is significant. Typically, the additional data increases the storage requirements by a factor of three to four of the original volume data. This overhead becomes prohibitive for high resolution datasets. A more fundamental problem is that classification and shading are in general non-linear operations. Interpolation of the pre-computed values to non-integer points degrades the achievable image quality when compared to the evaluation of the non-linear functions at the sample locations [Neu93b, Ben95]. Furthermore, pre-classified data does not allow for interactive classification, and pre-shaded data does not allow for changes of highlights when the observer moves. Finally, the pre-processing inhibits the visualization of dynamically changing data.

Subsampling

Some volume rendering acceleration techniques trade off image quality for speed. Most of these techniques use a variation of *subsampling*, which avoids excessive computation by calculating less samples in areas of low data variance.

In image-order algorithms subsampling is easily achieved by casting less rays from the image plane, calculating less samples along each ray, or by lower cutoff thresholds for accumulated opacity and early ray termination. Levoy [Lev90] uses a technique called *adaptive sampling* during ray-casting, where rays are cast only for a subset of screen pixels. If the variance of neighboring samples is above a certain threshold, more rays are cast by recursive subdivision of these screen regions. Pixels in low variance areas are interpolated from neighboring sampled pixels.

Danskin and Hanrahan [DH92] use a technique called *β-acceleration* for ray-casting, which lowers the sampling rate in areas where the volume is homogeneous. They also provide error bounds to control the error of the resulting image. Information about regions of low variation is pre-computed and stored in an octree data structure. Adaptive sampling in combination with error bounds have also been used in object-order algorithms. Hierarchical splatting [LH91] uses a pyramidal data structure to store average opacity values as well as estimated error bounds. When large errors are acceptable, rendering is performed at coarser resolution levels of the pyramid.

Subsampling potentially misses or blurs small features of the data, which is in general not acceptable for scientific visualization. However, subsampling proves to be a powerful method for rapid image generation in combination with *progressive refinement* [BFGS86]. A crude image is rendered while the user changes viewing parameters, and progressively better images are computed as soon as the user interaction stops.

2.7 Summary

This chapter introduced the background information on volume rendering and several different volume rendering algorithms. The algorithm that offers the highest image quality and the biggest opportunity for algorithmic acceleration is ray-casting. The main drawback of image-order algorithms are the non-regular and viewing angle dependent accesses to the volume data.

We have shown that a brute-force sequential implementation of volume rendering exceeds the processing capabilities of current conventional systems, even for small datasets. Our project goals, the real-time rendering of high-resolution datasets, lead to processing requirements that are several orders of magnitude beyond the capabilities of current general-purpose systems.

We have described the main acceleration techniques that are commonly used in software implementations of volume rendering. However, all of these software acceleration techniques require pre-computation, additional data storage, or trade off image quality for speed. This is in direct conflict with our project goals. To provide direct data input from acquisition devices does not allow for pre-processing. Furthermore, data duplication is prohibitive for real-time rendering of high-resolution datasets.

Throughout this chapter we have also seen that current shear-warp algorithms use 2D resampling of the dataset and combine many acceleration techniques that require time consuming pre-processing and data duplication. However, shear-warp rendering leads to the currently fastest software implementation of volume rendering, mainly due to the regular, object-order data access. In Chapter 6 we describe a parallel memory system that supports object-order access to scanlines of voxel data from any major viewing direction. We present three parallel ray-casting algorithms that use this memory

organization to efficiently stream through the data in object-order. In contrast to object-order volume rendering, we combine this high-bandwidth data access with an image-order, fully pipelined ray-casting algorithm, which is described in Chapter 5.

In the following chapter, we review the most relevant previous work in interactive or real-time volume rendering. We will see that none of the current systems satisfies all of our project goals.

Chapter 3

Related Work

In this chapter we survey the most relevant previous work that tries to achieve real-time frame rates for volume rendering. We also describe the Cube project at SUNY Stony Brook, which provided the framework for this research.

3.1 General-Purpose Supercomputers

Many researchers have implemented volume rendering algorithms on large general-purpose multiprocessors. Advantages of this approach are the flexible programming environment and the ability to integrate simulation and visualization on the same machine. Large-scale parallelism on *Single Instruction Multiple Data* (SIMD) computers [Fly72] has typically been used for parallel implementations of volume rendering [CU92, Hsu93]. More recent parallel volume rendering algorithms seem to favor *Multiple Instruction Multiple Data* (MIMD) machines with shared or distributed memory [CM93, MPHK93, MPS92, Neu93a].

The state-of-the-art in volume rendering performance on general-purpose

supercomputers is in the range of one to 10 frames per second for low-resolution datasets [NL92, MPHK94, VFR92]. One of the best performances has been achieved by a parallel implementation of the shear-warp algorithm [Lac95b]. Parallel color projections of 256^3 datasets can be generated at 5 frames per second on a 16-processor MIMD shared-memory SGI Challenge

A fundamental problem of general-purpose multiprocessors is that it requires expensive machines to achieve acceptable frame rates. Due to their high cost, these machines are shared by many users, typically through network connections. Sharing and partitioning of the machine among several users further limits their performance. Furthermore, the lack of direct frame buffer access prohibits real-time output rates. It seems infeasible that our project goals could be achieved on the current generation of parallel supercomputers.

3.2 Graphics Accelerators

Several researchers have implemented volume rendering algorithms on special-purpose graphics systems. These high-performance accelerators are typically used as external graphics engines to offload graphics computations from a general-purpose host computer. The machines in this class contain high-speed hardware implementations of what is known as the *pipeline of raster graphics rendering* [FvDFH90]. This pipeline can be divided into geometry operations and scan-conversion or rasterization of the geometric primitives.

Most of the machines in the following survey are experimental high-performance systems that were never commercially available. We include them because of their important contribution towards fast volume rendering. However, the following selection of machines is not comprehensive, and it merely serves to outline the historic development of volume rendering applications on

special-purpose graphics machines.

3.2.1 The PIXAR and PIXAR II Image Computer

The PIXAR machines were designed in 1984 to meet the requirements of image generation for motion picture special effects and animation. All computation as well as flow-control of pixels is performed by *channel processors* (Chaps). A Chap is a microcoded parallel vector processor, operating in a SIMD fashion by executing each instruction on four operands simultaneously [LP84]. The PIXAR Image Computer has one to three Chaps and 24 to 192 MBytes of image memory. PIXAR II can contain one or two Chaps and up to 128 MBytes of image memory.

Drebin et al. [DCH88] successfully implemented volume shearing with multipass resampling on the PIXAR machine. The object-order scanline approach leads to regular communication patterns that are well suited for the SIMD vector processors. The time to render a volumetric dataset depends on the number of pre-processed auxiliary opacity and color volumes. It varies from a few minutes to one hour for a 256^3 dataset.

3.2.2 AT&T Pixel Machine 900

The Pixel Machine 900 was completed in 1989 by AT&T Bell Laboratories. The machine incorporates two forms of parallelism: a coarse-grain operating *geometry pipeline* and a fine-grain parallel array of *pixel nodes*. The frame buffer is distributed among the array of pixel nodes, and pixels are interleaved into the processor array.

Volume rendering has been implemented on the architecture using the ray-casting algorithm. A Pixel Machine with 18 geometry pipelines, 64 pixel nodes,

and a 32 MBytes distributed frame buffer renders a $256 \times 256 \times 60$ volume dataset in 10 seconds [SF91].

3.2.3 The Princeton Engine

The Princeton engine is a SIMD real-time video system simulator [CPB+88]. It uses a linear array of up to 2048 processors. Each processor contains a 16-bit ALU, a 64-bit register set, a 16-bit communication port, and up to 320 kBytes of memory. The processors are interconnected with a 16-bit bus that supports left/right circular shifts and broadcast of data.

Kaba et al. [KMS+92] implemented volume shearing using multi-pass resampling for terrain data. The volume data is mapped onto the hardware with the X axis corresponding to the linear processor array and YZ slices of data stored in every processor. Projections of a 256^3 dataset are performed at 4 frames per second using 1024 processors. Schröder and Stoll use a 1024 processor Princeton Engine to achieve 30 frames per second for 128^3 datasets [SS92]. The algorithm assigns a column of voxel data to each processor. Using a lookup-table based ray-casting technique [YK92], partially finished results of each ray are communicated using nearest-neighbor shifts.

3.2.4 Pixel-Planes 5

Pixel-Planes 5 is a heterogeneous multiprocessor architecture for graphics using both MIMD and SIMD parallelism [FPE+89]. The global system structure is shown in Figure 9. Several *graphics processors* and *renderers* are connected to a high-speed *ring network* and a conventional *frame buffer*. The graphics processor are MIMD geometry engines and perform display-list traversal, clipping, and lighting calculations. The renderers use custom built enhanced

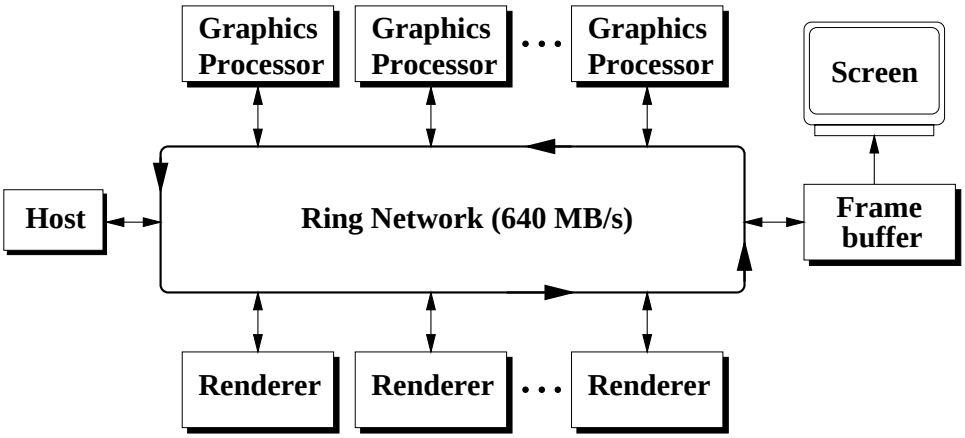

Figure 9: *Pixel-Planes 5 Architecture.*

memory chips, upgraded from the Pixel-Planes 4 system [FP81]. Each enhanced memory ASIC contains a 2D array of 128^2 one-bit ALUs for arithmetic and logical operations at each pixel, taking the parallel rasterization organization to the extreme. The renderer implements many other features such as shadows, textures, transparencies and anti-aliasing [FGH+85]. The processors and the frame buffer are connected by a token ring network that can support a total bandwidth of 640 MBytes/sec.

Several volume rendering algorithms, such as ray-casting and splatting, were implemented on Pixel-Planes 5 [YNF+92]. A machine with 20 graphics processors and eight renderers is capable of rendering $128 \times 128 \times 56$ datasets at 20 frames per second, and $192 \times 192 \times 128$ datasets at 11 frames per second. However, measurements on Pixel-Planes 5 showed that the bandwidth of the ring network limits the performance and scalability of the system [MEP92].

3.2.5 PixelFlow

PixelFlow is currently under development at the University of North Carolina at Chapel Hill [MEP92]. PixelFlow departs from the conventional raster graphics rendering paradigm and explores a form of parallelism called *image composition*. Geometric primitives are assigned to different renderers. The images of all renderers are then composed over a composition network. The advantages of image composition are linear scalability of performance with increasing number of renderers. The disadvantages are the high bandwidth requirements on the composition network, since every pixel must be transferred multiple times for every frame.

Figure 10 shows an overview of the proposed PixelFlow system. There are

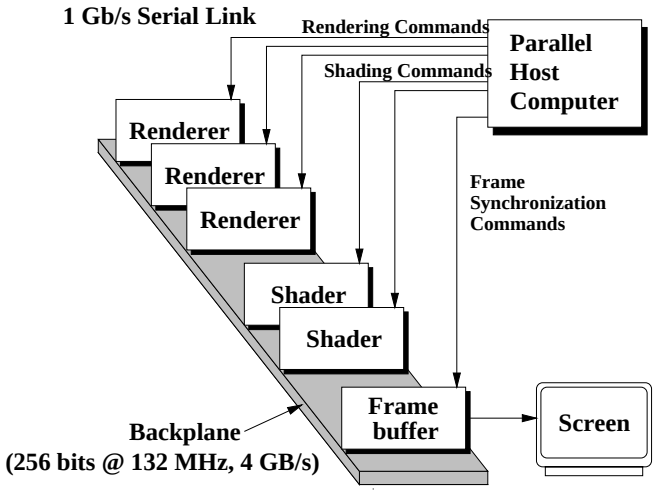

Figure 10: *The PixelFlow Architecture.*

three different types of boards in PixelFlow. *Renderers* contain an i860XP general purpose processor as geometry engine which is connected to a 128^2 SIMD array of enhanced image memory. The enhanced memory cells are similar to the ones in Pixel-Planes 5 and are mainly used for rasterization. *Shaders*

are similar to the renderers, with the addition of a texture memory that can be mapped onto the SIMD image memory array [DWS88]. *Frame buffer* boards contain a double-buffered Video RAM frame buffer. Composition of images is performed over the composition network, a 256-bit wide pipelined bus operating at 132 MHz for a net bandwidth of 4 GBytes/sec.

Cullip et al. [CN93] propose to use the texture mapping hardware of PixelFlow for volume rendering using *planar texture resampling*, a technique presented in the next section. Preliminary calculations estimate the performance for 256^3 datasets of a 43 processor card PixelFlow system at over 10 frames per second.

3.2.6 SGI Reality Engine

The SGI Reality Engine supports texture mapping of polygonal scenes through 3D texture memory on one or more *Raster Manager* boards. Each Raster Manager contains up to 16 MBytes of texture memory. Higher performance is achieved through the use of multiple Raster Managers with their own independent copy of texture data.

Texture mapping hardware can be used for volume rendering applying a method called *planar texture resampling* [CN93, HCN94]. The resampling locations inside the volume are constrained to lie in planes aligned with the image-space or object-space axis (see Figure 11). Textured polygons are used to resample the volume by evaluating textures for each sample plane at the locations that project to a pixel. Sample planes aligned in image space must be clipped to the boundaries of the volume to ensure valid texture coordinates. Clipping of sample planes aligned in object space is not necessary, but the sampling rate along rays varies for each pixel due to the perspective divergence of rays and the viewing angle dependence of sample planes. The resulting

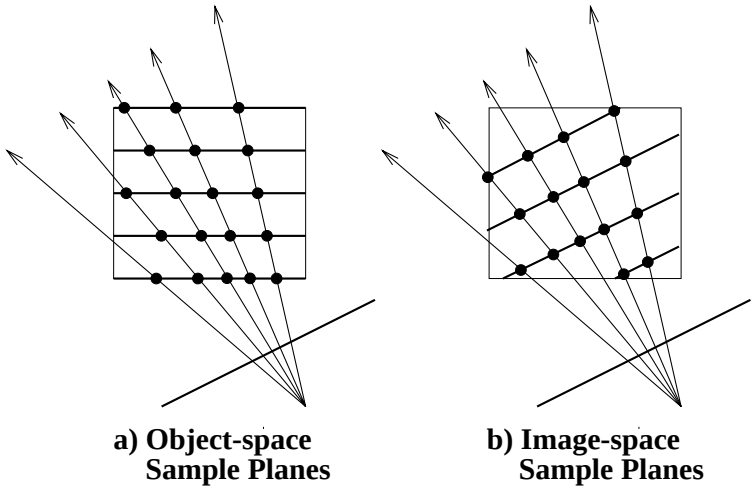

Figure 11: *Planar Texture Resampling.*

texture images are combined using compositing or weighted summation.

Hemminger et. al [HCN94] report a series of results on a Reality Engine 2 with one or two Raster Managers. The best result was achieved when rendering a $256 \times 256 \times 32$ volume at 12 frames per second. Cabral et al. [CCF94] use a four Raster Manager Reality Engine Onyx to render a $512 \times 512 \times 64$ dataset at 10 frames per second.

However, the SGI Reality Engine hardware is expensive and does not support estimation of surface normals that are required for high-quality shading or classification. Furthermore, each Raster Manager contains an independent texture unit with no provisions for communication to other Raster Manager boards. This leads to data duplication in texture memory and makes the hardware not scalable in performance.

Machine	Year	Performance	Algorithm
Pixar	1984	256^3, > 60 s/fr	volume shearing
Pixel Machine	1989	$256 \times 256 \times 60$, 10 s/fr	ray-casting
Princeton Engine	1992	256^3, 4 fr/sec 128^3, 30 fr/sec	volume shearing
Pixel-Planes 5	1992	$128 \times 128 \times 56$, 20 fr/sec $192 \times 192 \times 128$, 11 fr/sec	ray-casting
PixelFlow	1993	256^3, > 10 fr/sec (estimated)	planar texture resampling
SGI Reality Engine	1994	$256 \times 256 \times 32$, 12 fr/sec $512 \times 512 \times 64$, 10 fr/sec	planar texture resampling

Table 1: *Comparison of Graphics Accelerators.*

3.2.7 Comparison

Table 1 outlines the historic development of the use of graphics accelerators for volume rendering. Machines built before 1993 are not able to render small size datasets of 256^3 voxels at interactive rates. They contain parallel implementations of the traditional graphics pipeline, where most of the hardware resources are spent on video processing or polygon rasterization. This high degree of specialization makes them not well suited for direct volume rendering applications. Furthermore, their cost and size does not allow to integrate them into desktop or deskside systems.

More recently, the use of texture mapping hardware holds great promise to achieve interactive rates for volume rendering. Newly developed machines, such as the SGI Reality Engine and PixelFlow, dedicate large resources to texturing hardware, which is likely to become a common feature of graphics systems in the future. Market economics dictate that an expensive resource such as texture memory is used for more than just texturing polygons. Volume

rendering seems to be an application that naturally fits into the framework of 3D texture hardware.

But the application of current texture hardware to volume rendering suffers from several limitations. The texture memory does not allow to calculate gradients for directional shading of the samples. The view-dependent sampling of the volume on planes aligned in object-space leads to motion aliasing. Additional image artifacts may be introduced through scaling of values to prevent arithmetic overflows in the texture hardware. Furthermore, high-resolution datasets do not fit into the texture memory, which is currently limited to 16 MBytes. One solution is to partition larger datasets into smaller pieces. However, the limited texture memory bandwidth inhibits fast reloading of the partitioned data, which makes interactive viewing of larger datasets impossible. The only approach to higher performance is the addition of more texturing boards (Raster Managers in the Reality Engine or Shaders in PixelFlow) with duplication of the volume dataset in each of them. But data duplication becomes infeasible for higher resolution datasets, and it limits the scalability of the approach. Finally, current texture hardware has no provision for real-time data input from acquisition devices.

The application of texture hardware to volume rendering is still in its infancy. In lack of better solutions, planar texture resampling is a valid approach for fast, low quality volume rendering of low-resolution datasets. However, we believe that the development of special-purpose volume rendering hardware may influence the design of next generation texturing systems. The lessons learned from the development of hardware for high-quality real-time volume rendering should be easily applicable to other applications that require large 3D discrete data storage. In Chapter 10 we show that the contributions of

this dissertation successfully address the shortcomings of current texture systems. This may lead to the development of high-performance texture hardware that integrates volume rendering and polygonal texturing more naturally than current solutions.

3.3 Volume Rendering Accelerators

The best volume rendering performance on large general-purpose supercomputers or special-purpose texture mapping hardware is below 10 frames/sec for 256^3 volumes. Furthermore, almost all of the software based methods on parallel machines use several of the acceleration methods discussed in Chapter 2, requiring pre-processing, additional storage of auxiliary data, and a trade-off between image quality and speed. In view of the limitations of software based methods, it is no surprise that several researchers have undertaken the development of special-purpose volume rendering hardware [Kau91, SF91, SFF91].

3.3.1 The PARCUM System

The PARCUM system (Processing ARchitecture based on CUbic Memory) has been developed by Jackel [Jac85] at the Technical University of Berlin. The core of the system is a specially organized 3D memory called the *Memory Cube*. Figure 12 gives an overview of the system. PARCUM has been designed for 512^3 datasets. The memory cube allows to simultaneously read and write so-called *Macro Volume Elements* (MVEs) consisting of 64 (4^3) voxels [JS88]. The memory is physically divided into 64 memory modules with a capacity of 2 MBytes each. Every voxel of a MVE is assigned to a different memory module in an interleaved fashion in order to access all 64 voxels of an MVE in parallel. Because of the interleaving scheme, voxels at the output of the memory have

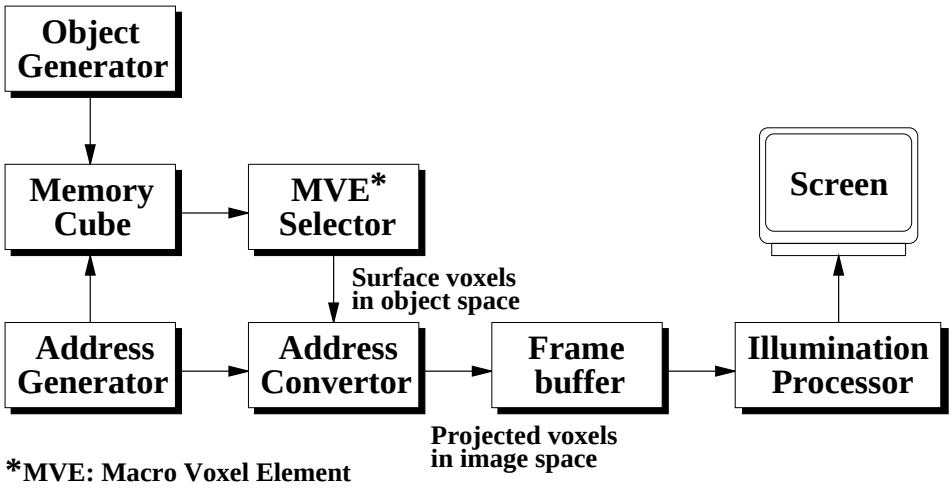

Figure 12: *PARCUM System Architecture.*

to be aligned to their correct position inside a data word. This alignment is performed with cyclic shift operations using network interconnected barrel shifters [Jac88].

The *Object Generator* generates volume objects inside the memory cube. This volume data is projected using parallel ray-casting. A reference path inside the plane of parallel rays is computed by the *Address Generator*. Macro voxels are accessed along the reference path. The sequence of macro voxels is processed inside the *MVE Selector*. The MVE selector selects visible surface voxels inside each MVE using a process called voxel reduction [JS88]. The surface voxels are projected onto a user-specified projection plane inside the *Address Convertor*. Each voxel is transformed by the viewing matrix and then projected onto a Z-buffer [DCH88]. The depth information of the projected voxels is used for image-based shading inside an *Illumination Processor*.

A hardware implementation of a 512^3 PARCUM memory system has been completed while the other parts of the architecture have been emulated on a MC68020 micro-processor. This experimental system generates an image of

512^3 datasets in about one minute and of 256^3 datasets in about 40 seconds.

3.3.2 The Voxel Processor

The Voxel Processor architecture, introduced by Goldwasser and Reynolds [GR83, Gol84] and conceptually shown in Figure 13, uses an octant-wise recursive subdivision of object space and a hierarchical tree of both rendering and display processors. Each subcube of volume data is assigned to a render-

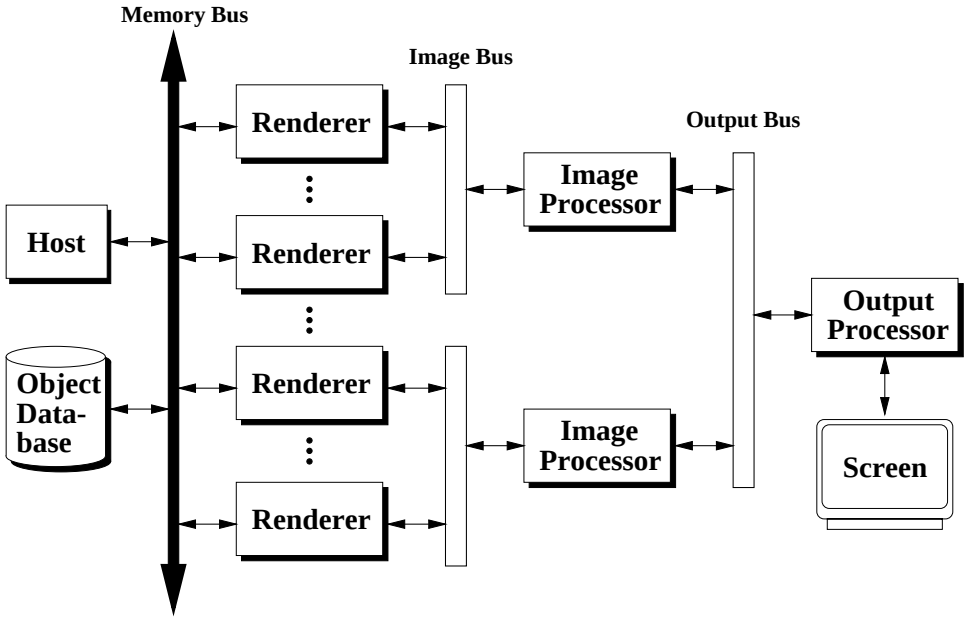

Figure 13: *Voxel Processor Architecture.*

ing processor in the first level of the processor tree. Processors render their subcubes by traversing them in back-to-front order. The subsequent image processing stages merge the subcube renderings in back-to-front order into the final image.

The Voxel Processor was designed for 256^3 datasets. The volume data is stored on a *host computer* that manages the object database. A rich software

environment on the host enables direct editing, thresholding, and segmentation of volume data [GRB+85]. The host subdivides the data into 64^3 subcubes and distributes them to 64 *rendering processors* over the *memory bus*. The processors independently render their subcubes by traversing them in back-to-front order and mapping each voxel onto image space using the 3D painters algorithm [FGR85].

The resulting 128^2 images contain the visible voxel values and their image space z-values. The renderers place their images in local frame buffers for use by the eight *image processors*, which combine eight input images to one 256^2 image. The eight 256^2 images are merged to the final 512^2 picture by the *output processor*. Once the image is in the frame buffer, a post-processor performs simple image-space shading [GR85] and pseudo-coloring. The entire rendering process is pipelined and uses double-buffering at each stage, so that at any time four frames are processed in the machine.

A small prototype operating on 64^3 4-bit voxels has been implemented and incorporated into a Physician's workstation [GRB+85]. The predicted speed for 256^3 volumes is 25 frames/sec, but the final machine was never built [SF91].

3.3.3 The Cube Project

We are conducting our research as part of the Cube project for hardware accelerated volume visualization at the State University of New York at Stony Brook. The Cube project pioneered several hardware architectures for volume rendering as well as a host of algorithms and software systems.

Cube-1, a first generation hardware prototype, is based on a specially organized *Cubic Frame Buffer* (CFB) [KB88], which has also been used in all subsequent generations of the Cube architecture. It uses a simple linear memory skewing that we discuss in Chapter 7. The skewed memory allows for

simultaneous access to all voxels of a *beam*, which is a ray parallel to a main axis of the dataset.

Cube-1 allows for orthonormal projections of the data using a *multiple-write bus* (VMWB) [GL82] which locates the first opaque voxel along beams. Successive beams of voxels most parallel to the viewing direction are placed into the processing units of the VMWB. A bit-wise comparison of the depth index of each voxel determines the voxel that is closest to the observer. Depth clipping of beams is performed by a similar mechanism. The projection time is proportional to $\log n$ for beams of length n. Shading is performed on the depth and value of the projected voxels by the lookup-table based *congradient shading* method [CKBB90].

A fully operational hardware prototype of Cube-1 consists of 16 boards and confirms the feasibility of the skewed memory scheme. Cube-1 is capable of generating orthonormal projections for 16^3 datasets [KB88]. *Cube-2* is a single-chip VLSI implementation of this prototype. A 14,000 transistor Cube-2 chip has been fabricated.

A different approach is taken in the *Flipping Cube* architecture [Yag91b]. Efficient access to the volume data requires a reorganization of the volume memory after a change of the major viewing direction through shearing or so-called *flip* operations. Only two flips are necessary to change the major axis of the volume. One of them can be implemented by an address mapping mechanism, and the second one by means of shear operations on a barrel shifter. The orthonormal projection and shading mechanisms are the same as for Cube-1. The estimated time for flipping a 512^3 volume by 90° is 0.5 seconds.

To allow for arbitrary parallel projections, an extension to Cube-1 has been proposed by Bakalash et al. [BKPP92]. An arbitrary plane of parallel

projection rays is accessed beam by beam from the memory system. The memory modules are connected to the processors by the so-called *conveyor*, a modified barrel shifter based on crossbar switches [CB92]. The conveyor is able to shift an entire beam of voxels by m places in one shift cycle. Shifted beams are then stored in a 2D skewed memory, where columns and rows can be read and written conflict free [KBC91]. From there the viewing rays are read and aligned by a second conveyor that shifts the voxels to the nodes of a tree of projection units. This extended Cube-1 architecture has a predicted performance of 16 frames/sec for 512^3 datasets using 512 processors [BKPP92].

3.3.4 VOGUE

VOGUE (previously called VERVE [Kni93]) is a compact and scalable image order ray-casting unit which provides interactive rendering speed at moderate hardware costs [KS94, Kni94]. The basic unit consists of the interleaved volume memory and four VLSI chips as shown in Figure 14. It provides arbitrary perspective projections (e.g., for walk-throughs), Phong shading, a freely movable point light source, depth-cueing, and non-binary classification using both opacity and color transfer functions. After having obtained all ray parameters from the host, the *address generator* sequentially generates all ray points. For each of them, up to seven sets of eight addresses are passed to the memory system. The *volume memory*, which has a maximum capacity of 256 MBytes for 512^3 16-bit voxels, consists of eight independent memory banks and delivers eight voxels per access. The *interpolator and gradient estimator* performs the tri-linear reconstruction and computes the gradient and gradient magnitude. Sample value and gradient magnitude then address several lookup tables to yield the shading parameters (color and specular coefficient) and the

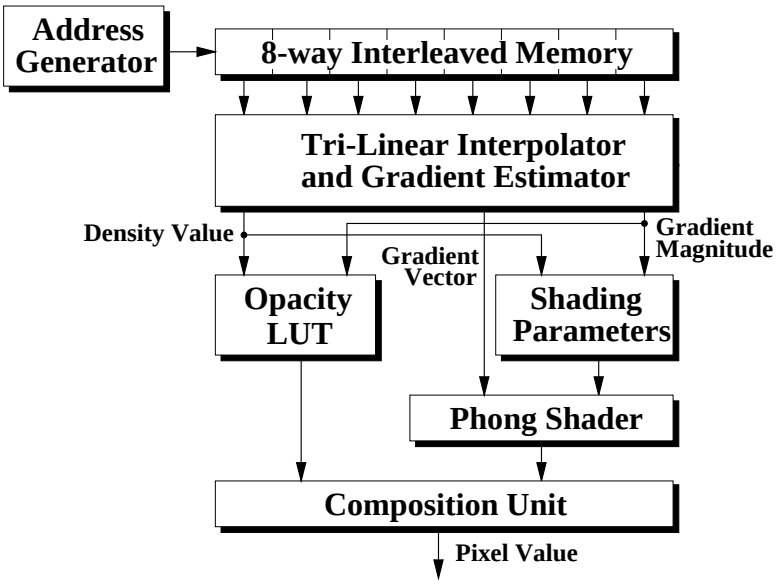

Figure 14: *VOGUE Architecture.*

sample opacity. A *Phong shader* implements the unrestricted Phong illumination model (non-parallel light, perspective projection) for a single point light source, and performs depth-cueing. The *compositing unit* sums up the intensities of all points on a ray and passes the pixel color to the frame-buffer.

Real-time performance can be achieved by multiple parallel units. The dataset is divided into subvolumes that are distributed across different units. Each unit processes a given ray as long as it traverses through its own subvolume. On exit, the properties defining the ray at this point are sent to the neighboring unit. Up to eight units can be connected by a bus with a bandwidth of 640 MBytes/sec. 64 units, organized in a $4 \times 4 \times 4$ mesh, require a ring-connected cubic network with a total transfer rate of about 5.2 GBytes/sec.

Estimated performance of one VOGUE module is 2.5 frames/sec for 256^3

datasets using the fastest rendering mode with lower-quality gradient estimation. Near real-time rates of 20 frames/sec can be achieved using eight modules for 256^3 datasets and 64 modules for 512^3 datasets. Higher quality gradient estimation slows this performance down to 2 frames/sec [HMK+95].

3.3.5 VIRIM

VIRIM is an object-order volume rendering engine currently being assembled and tested at the University of Mannheim [GPR+94]. The hardware of VIRIM is organized into modules with four double-height VME boards. Each modules contains a *geometry unit* for volume rotation and resampling, and a *ray-casting unit* for the final image generation (see Figure 15). One geometry unit can

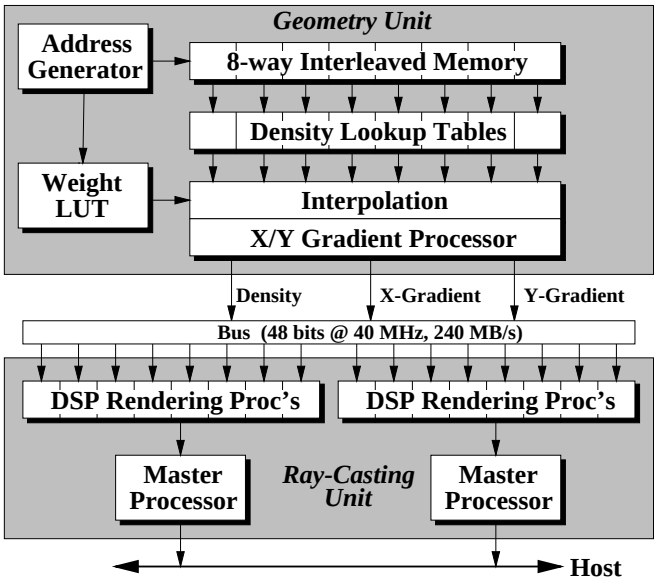

Figure 15: *VIRIM Architecture.*

generate 26 to 36 million transformed locations per second using 3D resampling. The maximum dataset size for 16-bit voxels is 256^3 for the currently

used 4 Mbit DRAMs, and $512 \times 512 \times 256$ voxels for 16 Mbit DRAMs. A dedicated 8-way interleaved memory system inside each geometry unit allows conflict-free data access to eight voxels. The maximum read-out rate is 640 MBytes/sec using commercial DRAMs.

Rotation of the dataset is performed by weighted interpolation among the eight voxel neighbors around the resampling location. Before interpolation, the voxel values are mapped onto density values using a *density lookup table* (LUT). Address generation hardware generates the positions of the sampling points. The interpolation weights are pre-calculated and stored in lookup tables. Different interpolation filters like tri-linear interpolation or Gaussian filters can be used in order to improve the resampling quality. Only the X- and Y-component of the gradient are estimated using a 2D Sobel gradient operator. The sample density and the X- and Y-gradient values are transmitted from the geometry unit to the ray-casting unit over a bus with a peak transfer rate of 240 MBytes/sec.

Using the sample and gradient values of the rotated dataset, the ray-casting unit generates the final image. In order to allow maximum flexibility, VIRIM uses programmable *digital signal processors* (DSPs). The VIRIM system implements the Heidelberg ray-tracing algorithm [MMSE91], which allows for shadowing using two light sources at 0^0 and 45^0 with respect to the viewing direction. The DSPs compute the interaction of light and material for each sample. One or more scanlines of the resulting projection are calculated per DSP and stored in local memory. A local *master processor* on the ray-casting board collects all scan lines of the final image from the DSP memories and transfers the results to the host system.

One VIRIM module with four boards achieves 2.5 Hz frame rates for $256 \times 256 \times 128$ datasets. Four modules (16 boards) achieve 10 Hz for the same

Machine	Year	Performance	Algorithm
PARCUM	1985	512^3, 60 sec/fr	first opaque
Voxel Processor	1985	256^3, 25 fr/sec	first opaque
Cube	1986	512^3, 16 fr/sec	first opaque
VOGUE	1993	256^3, 2.5 fr/sec	ray-casting
VIRIM	1994	$256 \times 256 \times 128$, 2.5 fr/sec	ray-casting

Table 2: *Comparison of Volume Rendering Accelerators.*

dataset size, and eight modules (32 boards) achieve 10 Hz for 256^3 datasets [HMK+95].

3.3.6 Comparison

Table 2 lists the rendering performances and the projection algorithms of the volume rendering accelerators that have been presented in this chapter. There is a striking performance gap between older designs, that use first opaque projection, and the more recent ray-casting architectures. The higher performance of early machines can be explained by the simple rendering and shading methods that were common at the time. Projection was reduced to the problem of displaying the front-most opaque voxels. Zero-order point sampling was used instead of more expensive resampling filters. None of these assumptions are valid anymore in current volume rendering applications. Consequently, the performance numbers of architectures before and after 1993 in Table 2 should not be compared.

Table 3 shows a more conclusive comparison of the memory, processor, and communication hardware that is used by the different architectures. All architectures use a form of memory interleaving in order to increase the data throughput. We discuss the issues of memory organization in Chapter 6. The

Machine	Memory Organization	Processors	Memory-Processor Connection
PARCUM	64-way interleaved	5 processors	network connected barrel shifters
Voxel Processor	on host	64 renderers 9 image proc.	memory bus
Cube-1	skewed	512 processors	barrel shifters
VOGUE	8-way interleaved	4 ASICs per board	ring-connected cubic network
VIRIM	8-way interleaved	4 boards with 32 DSPs	global bus

Table 3: *Volume Rendering Accelerator Hardware.*

amount of hardware required by the different machines varies according to performance levels. At both ends of the scale are Cube-1, with high-performance massive parallelism (512 processors), and VOGUE, with a low-performance compact implementation (4 ASICs). The right-most column of Table 3 shows how the volume memory and the parallel processors are connected. In general, all architectures use a *global communication network* between memory modules and processors. We discuss the issues related to the global communication of voxel data in Chapter 6.

3.4 Summary

General-purpose parallel supercomputers offer a flexible programming environment and the integration of simulation and visualization on the same machine. However, inter-processor communication and I/O are still major bottlenecks

and severely limit volume rendering performance. To achieve acceptable performance requires the combination of many software acceleration techniques using pre-computation, data duplication, and lower image quality. The biggest problem is that supercomputers are very expensive – few people have them. The machines are typically accessible only over (slow) network connections. Sharing of the machine with other users and the lack of distributed framebuffers further inhibit real-time performance. Our research ultimately aims at providing real-time volume rendering performance on the desktop at an affordable price.

Special-purpose graphics architectures can be roughly divided into systems that dedicate most of their resources to video or polygon processing, such as the Princeton Engine and Pixel-Planes 5, and systems that contain large amounts of texture memory, such as PixelFlow and the SGI Reality Engine. The first class of machines implements the traditional graphics or video pipeline in hardware, which has only limited value for the visualization of volumetric datasets. Consequently, the volume rendering performance of these systems is very low compared to the size and the price of the hardware.

Texture memory offers a more natural integration of discrete 3D datasets into traditional graphics accelerators. The availability of texture systems is increasing with the advent of 3D graphics in PC systems. However, current texture systems do not fully address the needs of volume rendering applications. The lack of gradient estimation and directional shading prohibits high-quality rendering. Higher-resolution datasets have to be partitioned, and each smaller subset is subsequently being rendered. The slow texture download rates prohibit fast rendering times for these datasets. Most importantly, current texture memory approaches are not scalable, and they require the duplication of the dataset in each texture sub-system.

Early on, the high data storage and communication requirements of volume rendering have led to the development of special-purpose volume rendering accelerators. Systems that were developed before 1993 use simplified resampling and projection methods that are not acceptable in current volume visualization applications. The more current approaches, VOGUE and VIRIM, use ray-casting and image- or object-order traversal of the dataset, respectively. However, none of the architectures achieves real-time frame rates at an acceptable hardware cost. Furthermore, none of the approaches is scalable. They either require duplication of the dataset at each processing node or large, global interprocessor communication networks.

The main result of this dissertation is Cube-4, a special-purpose volume rendering architecture that implements high-quality ray-casting. Cube-4 leads to a modular implementation with linear scalability of performance. The cost-performance ratio of Cube-4 surpasses the state-of-the-art on supercomputers, special-purpose graphics systems, and on other special-purpose volume rendering systems.

In the following chapter we discuss the basic ray-casting algorithm and identify its performance bottlenecks. And in Chapter 5 we present several algorithmic modifications that lead to an efficient implementation of ray-casting in hardware.

Chapter 4

The Ray-Casting Algorithm

As we have seen in Chapter 2, image-order ray-casting offers high image quality and a simple implementation in software. However, it suffers from computational expenses due to the arbitrary accesses to the dataset. In this chapter we look in detail at the basic ray-casting algorithm and carefully identify its performance bottlenecks. In the next chapter we present modifications of this basic algorithm that eliminate the costly random and non-uniform accesses to the volume memory.

The discussion in this chapter expose the following areas of improvement that are at the focus of our attention throughout the remainder of this dissertation: uniform data traversal, the use of coherency between samples of neighboring rays, caching of intermediate results, and fully pipelined dataflow.

Figure 16 shows the graphical conventions we use for the figures in this dissertation. We use squares to indicate voxels and solid lines along rays to indicate resampling locations. The voxel neighborhood that is used for the interpolation at the non-integer resampling location is called a *cell*, indicated by thick lines. In the 2D drawing of Figure 16, a cell consists of four voxels.

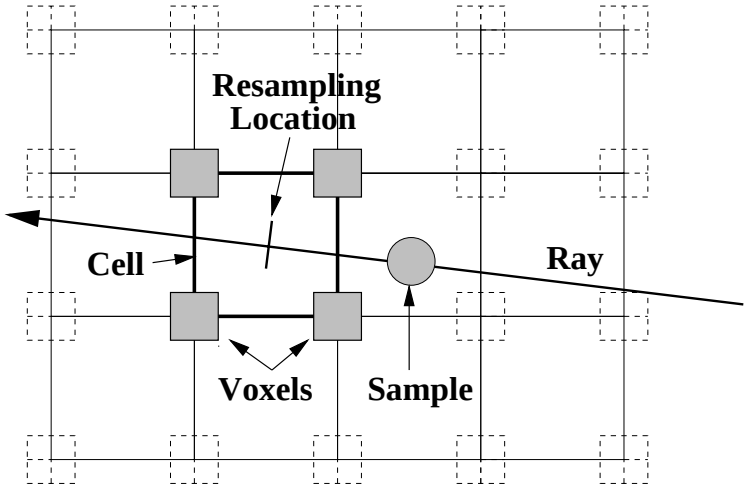

Figure 16: *Graphical Conventions.*

In 3D it may consist of eight or more voxels, depending on the support of the interpolation filter. The results of interpolation are called sample values or *samples* and are indicated by a circle.

4.1 Ray-Casting Basics

The volume data is defined as a set of voxels on a 3D array of grid points. In this dissertation, voxels are assumed to be the scalar values defined on the grid points. This is in contrast to an alternative definition, where a voxel is interpreted as a small unit cube of volume with a constant value. We label the axes of the voxel array coordinate system as X, Y, and Z (see Figure 17). These coordinates define what is called *object space*. The *image space* coordinate system is the coordinate system of the projected image. It consists of the axes I and J of the image to be generated, also called the *view-plane*. The image-space coordinate K indicates the depth from the image towards the object. The coordinates (I, J) of the projected image are mapped

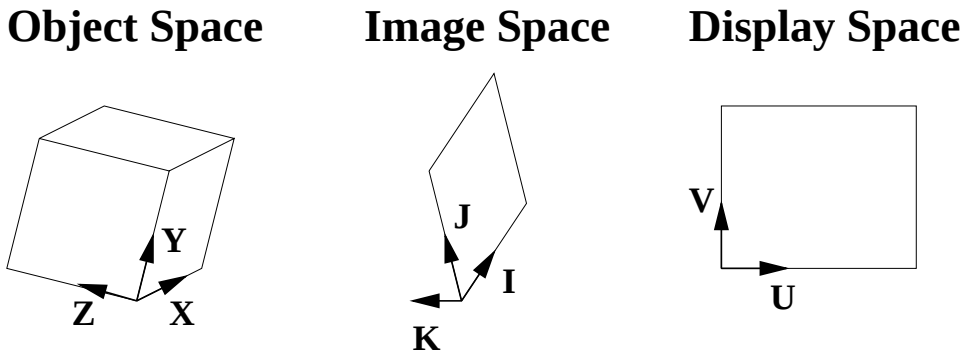

Figure 17: *Coordinate Systems.*

onto *display space* coordinates (U, V) by a display transformation that defines viewport, window size, and position of the final image on the screen.

The program segment shown in Figure 18 outlines the basic ray-casting operations. Rays are cast from each pixel of the view-plane into the volume data. At locations along each ray a *sample value* and a *surface normal* approximation are calculated using values of surrounding voxels. Using the sample value and normal, a sample opacity is assigned and a local shading model is applied. Finally, the ray samples along the ray are composited into a pixel value for the final image [Lev88]. We now look at each of these steps in more detail.

4.1.1 Data Traversal

The first step of the ray-casting algorithm is to generate addresses of *resampling locations* along each ray. In general, this involves a coordinate transformation from image space to object space. The transformed resampling locations in object space are most likely not positioned on voxel locations, which requires *interpolation* from surrounding voxels to estimate sample values at non-integer positions.

```
for  i = 1 to  imageWidth do
    for  j = 1 to  imageHeight do
        prevColor = backgroundColor;
        for  k = 1 to  rayLength do
            // Calculate next resampling location
            rayPoint = TraverseData (viewingDirection);

            // Access voxel data for interpolation
            sample = InterpolateVoxelData (rayPoint);

            // Access voxel data for normal approximation
            normal = EstimateGradient (rayPoint);

            // Shading can be arbitrarily complex
            color = Shade (sample, normal, lightSources, ... );

            // Classification may only depend on Sample value
            opacity = Classify (sample, normal, transferFct);

            // Back-to-front or front-to-back
            intensity = Composite (color, prevColor, opacity);
        end for
        DisplayPixel (intensity);
    end for
end for
```

Figure 18: *Basic Ray-Casting Routine.*

4.1.2 Interpolation

Interpolation involves a convolution of voxel values with an interpolation filter. There is a wealth of literature that deals with the theory and application of appropriate interpolation filters in computer graphics [Gla95, Wol90] and volume visualization [Neu93b, Ben95]. In practice, due to the prohibitive computational cost of higher order filters, the most commonly used filters for ray-casting are *nearest neighbor interpolation* and linear interpolation in three

dimensions, also called *tri-linear interpolation*.

Suppose the relative 3D coordinate of a sample location within a cell with respect to the corner voxel closest to the origin is $\langle a, b, c \rangle$ and the values of corner voxels of the cell are P_{ijk}, where i, j, k = 0 or 1. The interpolated data value associated with the sample location, P_{abc}, is computed through tri-linear interpolation as follows:

$$\begin{aligned} P_{abc} &= P_{000}\,(1-a)(1-b)(1-c) + P_{100}\,a(1-b)(1-c) + \\ &\quad P_{010}\,(1-a)b(1-c) + P_{001}\,(1-a)(1-b)c + \\ &\quad P_{101}\,a(1-b)c + P_{011}\,(1-a)bc + \\ &\quad P_{111}\,abc + P_{110}\,ab(1-c). \end{aligned}$$

4.1.3 Gradient Estimation

To approximate the surface normals necessary for shading and classification requires the computation of a *gradient*. Given a continuous function $f(x, y, z)$, the gradient ∇f is defined as the partial derivative of the function with respect to all three coordinate directions:

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right). \tag{1}$$

Due to the sampled nature of volumetric data the computation of this continuous gradient has to be approximated using discrete *gradient filters*. The convolution of the input voxels with the filter kernel yields one component of the gradient vector. Most of these filters are straight forward 3D extensions of the corresponding two-dimensional edge detection operators, such as the Prewitt, Laplacian, or Zucker-Hummel operators [ZH81]. As an example, Equation 2 shows the filter coefficients for the Sobel operator [Sob95], one of

the most widely used gradient filters for volume rendering.

$$G_x = \begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix} \begin{bmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{bmatrix} \begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix} \tag{2}$$

G_x is the $3 \times 3 \times 3$ convolution kernel for the X-component of the gradient vector. The convolution kernels for G_y and G_z can simply be obtained by rotation and alignment with the positive Y and Z axes, respectively. An analysis of several gradient filters for volume rendering has been published by Goss [Gos94] and Bentum [Ben95].

In practice, and due to computational considerations, most ray-casting algorithms use the *gray-level gradient*, which is computed by local differences between voxel or sample values in all three dimensions [HB86]. The gradient at voxel position (x, y, z) is computed as follows:

$$\nabla f(x,y,z) = \begin{bmatrix} \nabla f_x \\ \nabla f_y \\ \nabla f_z \end{bmatrix} = \frac{1}{2} \begin{bmatrix} f(x+1,y,z) - f(x-1,y,z) \\ f(x,y+1,z) - f(x,y-1,z) \\ f(x,y,z+1) - f(x,y,z-1) \end{bmatrix}. \tag{3}$$

The factor $\frac{1}{2}$ is often omitted, since the length of the gradient vector is normalized before shading calculations.

4.1.4 Shading

Local illumination models use the basic vectors shown in Figure 19. The equation to compute the reflected intensity at each sample is:

$$I = I_a + \sum_l I_l k_d (N \cdot L_l) + I_l k_s (N \cdot R_l)^s, \tag{4}$$

where:

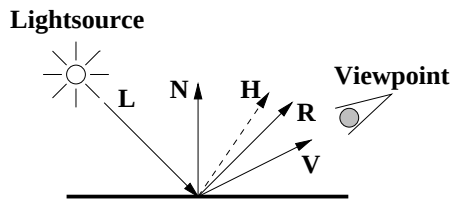

Figure 19: *Vectors for Local Illumination Models.*

I is the reflected intensity by the sample from the light source towards the viewer. I is used as the source term S of each sample in the following explanation of compositing (Equation 9).

I_a is the reflected intensity due to ambient light.

I_l is the intensity emitted by light source l.

k_d is the diffuse reflection coefficient for the material.

k_s is the specular reflection coefficient for the material.

s is the specular exponent for the material.

N is the normalized surface normal at the sample location.

L_l is the normalized vector to light source l.

R_l is the normalized reflected light vector of light source l.

The evaluation of Equation 4, including specular highlight effects due to k_s and s, corresponds to the *Phong* illumination model [Pho75]. An alternative formulation of the Phong model [Bli77] avoids the computation of the reflection vector R by using the halfway vector H between the viewing vector V and the surface normal N (see Figure 19). This is an accurate approximation if the light sources and the viewpoint are assumed to be at infinity. For

$k_s = 0$ the illumination model reduces to pure diffuse or Lambertian reflection [FvDFH90]. Higher-order shading models, which include the physical effects of light-material interaction [CT82], are computationally too expensive to be considered for volume rendering.

4.1.5 Classification and Compositing

Ray-casting algorithms rely on the low-albedo approximation to how the volume data generates, scatters, or occludes light [Bli82, KH84]. Effects of the light interaction at resampling location are integrated along the viewing rays according to the following equation:

$$I(a,b) = \int_a^b s(x)\, e^{-\int_a^x \alpha(t)dt}\, dx, \tag{5}$$

where

$I(a,b)$ is the intensity from a ray passing through the dataset between points a and b;

$s(x)$ is the source term, giving the light added per unit length along the ray, including self-emission and reflected light; and

$\alpha(t)$ is the absorption coefficient, corresponding to the attenuation of light per unit length along the ray due to scattering or extinction.

The mapping which assigns a value for optical properties like α or s to each sample is called a *transfer function*. The transfer function for α is called *opacity transfer function*, typically a continuously varying function $\alpha(f)$ of the scalar sample value. Often it is also useful to include the gradient ∇f as an additional parameter of the opacity transfer function $\alpha(f, \nabla f)$. This approach has been

widely used in the visualization of bone or other tissues in medical datasets or for the iso-surface visualization of electron density maps [Lev88].

The source term s can also be specified as a transfer function $s(f)$ of the scalar f. The simplest source term is direction independent, representing the glow of a hot gas [Max95]. It may have red, green, and blue components, with their associated *color transfer functions* $s_{red}(f)$, $s_{green}(f)$, and $s_{blue}(f)$. More sophisticated models include *shading* effects, where the source term at position x is calculated as $s(x) = I(x)$. $I(x)$ is the illumination at position x as computed by Equation 4.

To get the discrete formulation of Equation 5 we consider a ray where the intensities S_i and opacities α_i have been computed at at resampling locations i. Using a Riemann sum approximation to the integral we can write Equation 5 as:

$$I(a,b) = \sum_{x=a}^{b} S_x \; e^{-\sum_{t=a}^{x-1} \alpha_t}, \tag{6}$$

which may be reformulated to:

$$I(a,b) = \sum_{x=a}^{b} S_x \; \prod_{t=a}^{x-1} e^{-\alpha_t}. \tag{7}$$

Assuming that α_t is in the range $[0,1]$, we can approximate the exponential term by its Taylor series expansion as $e^{-\alpha_t} \simeq (1-\alpha_t)$. This leads to the following equation for the final intensity along a ray:

$$I(a,b) = \sum_{x=a}^{b} S_x \; \prod_{t=a}^{x-1} (1-\alpha_t). \tag{8}$$

This derivation follows [Lev88, Max95, Sab88, Wit93]. Intuitively, Equation 8 says that the intensity I at the current resampling location is the emission S_x multiplied by the transparency $(1-\alpha_t)$ accumulated so far. The final intensity $I(a,b)$ is the sum of all intensities along the ray between points a and b.

Compositing is the recursive evaluation of Equation 8. It was first introduced in the context of digital image compositing, where it was formulated using the *"over" operator* [PD84]:

$$\begin{aligned} I(0, n-1) &= \sum_{x=0}^{n-1} S_x \prod_{t=0}^{x-1} (1-\alpha_t) \\ &= S_0 + S_1(1-\alpha_0) + S_2(1-\alpha_0)(1-\alpha_1) + \cdots \\ &\qquad + S_{n-1}(1-\alpha_0)\cdots(1-\alpha_{n-2}) \\ &= S_0 \text{ over } S_1 \text{ over } S_2 \text{ over } \cdots S_{n-1}. \end{aligned} \tag{9}$$

Because of the associativity of the "over" operator, the composition of four sample source intensities S_i can be computed in three different ways [WH94, Wit93]:

$$\begin{aligned} &\text{Front-to-back:} && I = (((S_1 \text{ over } S_2) \text{ over } S_3) \text{ over } S_4) \\ &\text{Back-to-front:} && I = (S_1 \text{ over } (S_2 \text{ over } (S_3 \text{ over } S_4))) \\ &\text{Binary tree:} && I = ((S_1 \text{ over } S_2) \text{ over } (S_3 \text{ over } S_4)) \end{aligned}$$

The *front-to-back* or *back-to-front* formulations are used in sequential ray-casting algorithms. The last formulation as a *binary tree* is especially useful for parallel ray-casting algorithms, where partial results of segments along the ray can be computed in different processors [PKC94, Hsu93, WS93]. The final composition of the partial results yields the same image as sequential compositing along the ray.

Compositing is expressed algorithmically using recursion. The front-to-back formulation is:

$$\begin{aligned} I_{out} &= I_{in} + S_i\, \alpha_i\, (1-\alpha_{in}) \\ \alpha_{out} &= \alpha_{in} + \alpha_i\, (1-\alpha_{in}), \end{aligned} \tag{10}$$

where I_{out}, α_{out} indicate the results of the current iteration, I_{in}, α_{in} the accumulated results of the previous iterations, and S_i and α_i the sample intensity and opacity values at the current resampling location. Substituting *transparency* T_i for $(1 - \alpha_i)$ we get [Wit93]:

$$\begin{aligned} I_{out} &= I_{in} + S_i\ \alpha_i\ T_{in} \\ T_{out} &= T_{in}\ T_i. \end{aligned} \tag{11}$$

Notice that the formulation in Equation 11 with transparencies T_i is more efficient than the traditional formulation with opacities α_i in Equation 10. It is therefore an advantage to to store transfer function tables with transparencies instead of opacities.

When compositing back-to-front, the incremental opacities of Equation 10 do not need to be maintained:

$$I_{out} = S_i\ \alpha_i + I_{in}\ (1 - \alpha_i). \tag{12}$$

Since opacities do not need to be updated, the complexity of back-to-front compositing is independent of the use of opacities or transparencies.

All nodes in binary tree composition take four inputs: I_L, I_R are the accumulated intensities, and α_L, α_R are the accumulated opacities from the left and right children, respectively. With these definitions, binary tree composition can be expressed as:

$$\begin{aligned} I_{out} &= I_L + (1 - \alpha_L)\ I_R \\ \alpha_{out} &= \alpha_L + (1 - \alpha_L)\ \alpha_R, \end{aligned} \tag{13}$$

where the leave nodes at the first level of the tree compute $I_L = S_i\ \alpha_i$ and $\alpha_R = \alpha_i$.

There are several alternatives to volumetric compositing that have proven useful. In *X-ray* or *weighted sum* projections, the value of the pixel equals the sum of the intensities. *Maximum Intensity Projections* (MIP) project the maximum intensity along the ray into a pixel. Other simplifications includes *first opaque projection*, where a sample is either fully opaque or fully transparent.

4.2 Ray-Casting Bottlenecks

Although conceptually simple, ray-casting is computationally expensive. The main disadvantage is that the volume data is not accessed in storage order due to the arbitrary direction of the viewing rays. We now discuss other bottlenecks that inhibit real-time frame rates.

Expensive Address Arithmetic

The transformation from image space to object space coordinates is achieved by using the homogeneous *viewing transformation matrix* M_v:

$$\begin{bmatrix} i_i \\ j_i \\ k_i \\ w_i \end{bmatrix} = M_v \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}. \tag{14}$$

M_v defines an affine transformation that involves rotation, translation, and scaling. It has to be calculated only once per projection. For each ray cast from image space, M_v is used to determine the intersection of the ray with the volume data. Alternatively, hierarchical data structures or hardware acceleration is used to determine the intersection of the ray with an internal object bounding box to skip empty space [SK94] (see Chapter 2).

The viewing vector is transformed to object space. We call the distance between resampling locations along the the transformed viewing vector the *sampling distance* δ, and $\frac{1}{\delta}$ is the corresponding *sampling frequency* per unit length. Starting from the first intersection point, δ is added to the each resampling location along the ray. The new floating-point sample location is converted into an integer array index into the volume data. The floating-point or fixed-point portion determines the weights for the interpolation inside the cell. Alternatively, a 3D variation of Bresenham's algorithm modified for non-integer endpoints [KS86, Kau91] can be used to eliminate some of the floating-point arithmetic. In either case, the required address arithmetic is substantial, and the computational cost is much higher than in object-order algorithms.

Non-Uniform Sampling

Due to the arbitrary ray direction, the volume data is not sampled uniformly. As Figure 20 illustrates, cells may contain more than one resampling location. If the sampling distance δ along the ray is small, such as for ray r_i in the figure, consecutive samples fall into the same cell and have to be interpolated using the same voxels. For non-orthogonal viewing directions this may happen even if δ has unit length, and the situation is aggravated for higher sampling frequencies. One possible solution to this problem is to use *caching* of cell voxels between the interpolation of consecutive samples along the ray.

However, caching is almost impossible if samples of neighboring rays map onto the same cell, such as for rays r_j and r_k in Figure 20. If the samples that share a cell are computed on different processors in multi-processor machines, this situation may also lead to expensive interprocessor communication.

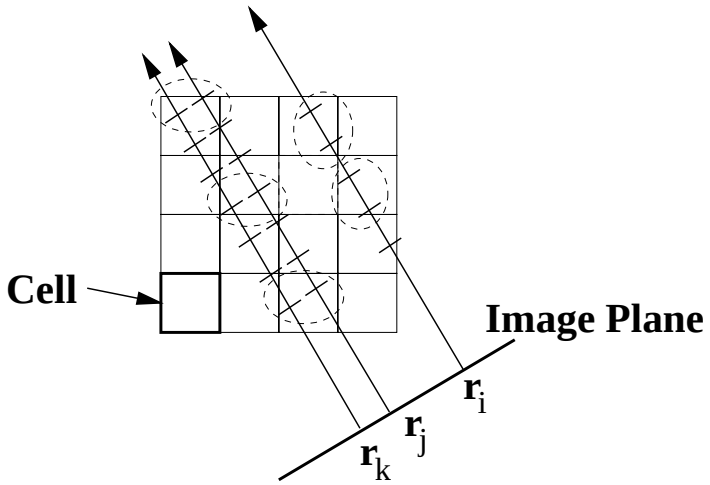

Figure 20: *Cells with more than one resampling location.*

Interpolation

The implementation of tri-linear interpolation as defined in Equation 1 requires 13 multiplications and 20 additions for each resampling location. This arithmetic overhead combined with the memory access to fetch the voxels makes interpolation one of the most time consuming operations during volume rendering.

Gradient Estimation

The gradients on voxel positions are independent of the viewing direction and can therefore be pre-computed. Nearest neighbor sampling can be used to estimate the gradient at the sample position during rendering by using the gradient at the voxel position closest to the sample location. A more accurate method performs a tri-linear interpolation of the gradient at the sample position using the gradients of surrounding voxels.

Storing three floating-point components of the gradient vector at each voxel

leads to high storage overhead. This method becomes prohibitive for medium to high resolution datasets. Consequently, it is better to store an integer encoding of the quantized gradient vector [FR93, Gla90]. However, the non-linear shading function is very sensitive to quantization errors of the gradients. To avoid any additional storage overhead, gradients are computed during rendering in high-quality ray-casting algorithms. However, as we show in the next section, gradient computation leads to redundant data access.

Overlapping Cells

In general, gradient estimation for samples involves a larger voxel neighborhood than interpolation. Using the 3D Sobel gradient filter of Equation 2 requires access to a $3 \times 3 \times 3$ neighborhood of voxels. Suppose that all cells outlined in Figure 21a are used for the gradient computation. Neighboring

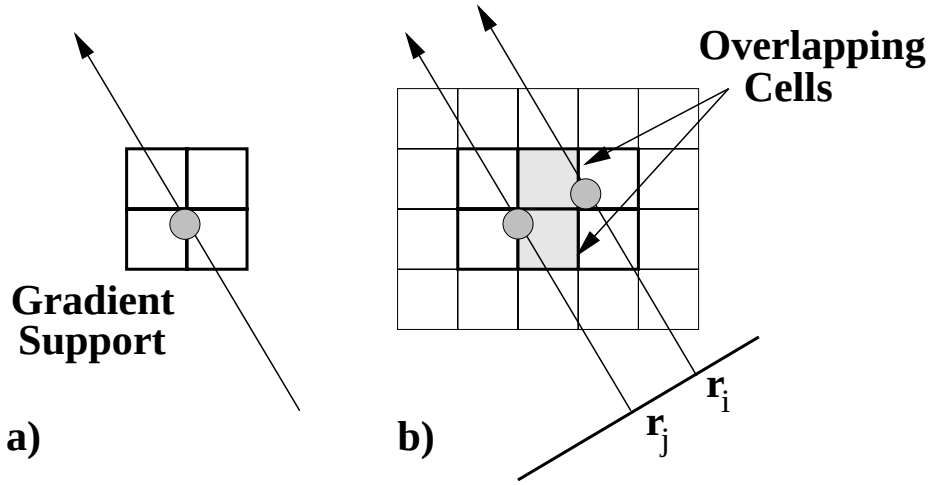

Figure 21: *Overlapping cells during gradient computation.*

cells overlap for gradient estimation of samples along a single ray or from

different rays (for example, rays r_i and r_j in Figure 21b). This leads to unnecessary memory accesses and redundant computation. Notice that even uniform resampling with one sample location per cell can not prevent this overlap.

Shading

Shading involves the application of a local illumination model to each sample along the ray. It is without doubt one of the most expensive computations of ray-casting in terms of arithmetic complexity. To get an approximation to the surface normal N requires normalization of the gradient vector ∇f:

$$N(x, y, z) = \frac{\nabla f(x, y, z)}{|\nabla f(x, y, z)|}. \tag{15}$$

A straight forward implementation of this normalization requires an expensive square root and divide operation, although many optimizations have been developed [Gla90].

Each change of viewing angle or light source position requires the normalization of vectors and the evaluation of Equation 4 at each resampling location. To get color images, the illumination model has to be computed for each color channel. However, shading does not require additional memory access to voxels because it is a local operation on the samples.

Compositing

The cost of compositing varies between the three different composition orders. Front-to-back compositing has different costs depending on whether opacities or transparencies are used. Updating of opacities involves the computation of $\alpha_{in} + \alpha_i\ (1 - \alpha_{in})$ at the cost of one addition and one multiplication if $(1 - \alpha_{in})$ is saved during intensity calculations. Using transparencies, the update involves

	Updates	Multiplies	Adds	
			Opacity	Transparency
Front-to-back:	$n-2$	$3n-3$	$4n-6$	$2n-2$
Back-to-front:	none	$2n-1$	$2n-2$	$2n-2$

Table 4: *Compositing costs for a ray with n samples.*

only one multiplication for the calculation of $T_{in}\ T_i$. Table 4 summarizes the cost of the different approaches for n samples [Wit93].

4.3 Summary

Figure 22 shows a schematic diagram of the ray-casting algorithm and the performance bottlenecks we discussed in this chapter. The dataflow shown in the figure corresponds to the pseudo-code of Figure 18.

Ray-casting, as shown in Figure 22, can be viewed as a set of pipelined processing steps. *Pipelining* is an important concept in hardware design and for the design of efficient parallel algorithms with local communication. A pipeline consists of a sequence of so called *stages* through which a computation and data flow. New data is input at the start of the pipeline while other data is being processed throughout the pipeline. We call an algorithm *fully pipelined* if all data transfers occurs between consecutive stages of the pipeline. However, as we have outlined in this chapter, traditional ray-casting is not fully pipelined due to the frequent and mostly random accesses to memory at various stages of the algorithm.

Based on the observations in this chapter, we can identify several areas of improvement to the basic ray-casting algorithm:

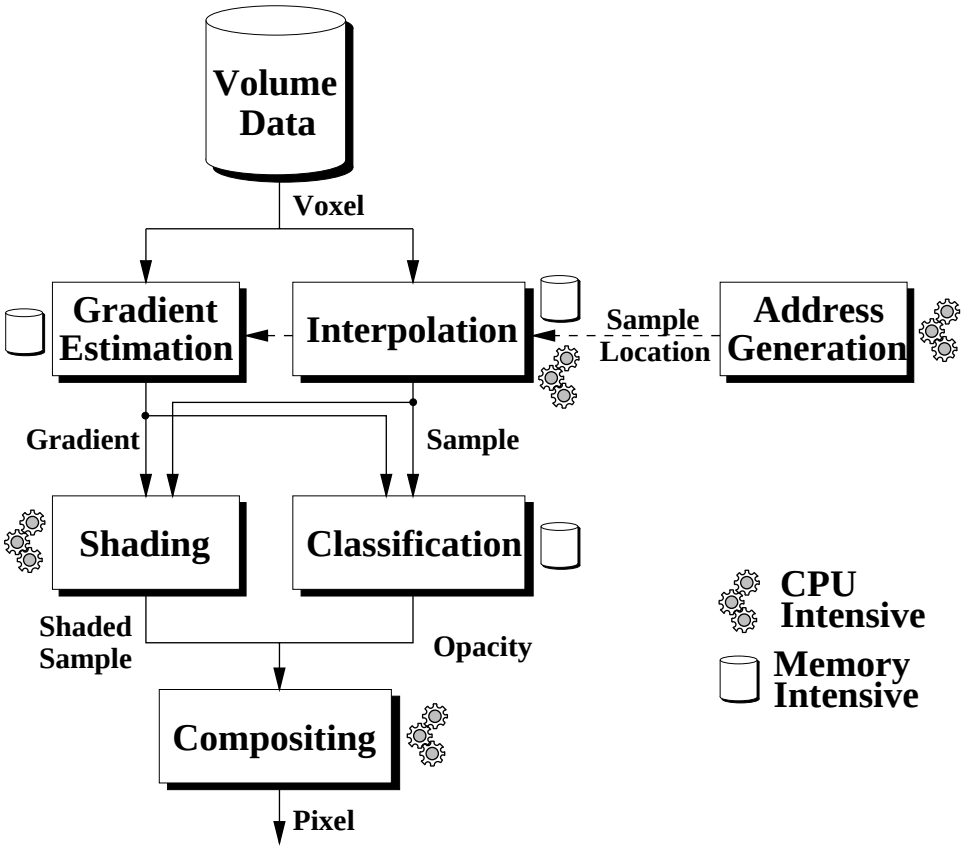

Figure 22: *Schematic diagram of the ray-casting pipeline and its performance bottlenecks. Each stage either requires random access to the volume memory, excessive arithmetic processing, or both.*

Uniform resampling: A regular data access pattern into the dataset leads to simpler address arithmetic and better spatial and temporal locality.

Use of coherence: By using the coherence between rays we avoid multiple memory accesses to voxels of overlapping cells.

Caching of intermediate results: Buffering of intermediate results leads to savings in arithmetic complexity and much reduced memory access.

Fully pipelined dataflow: A fully pipelined dataflow leads to ray-casting implementations where data flows strictly between computational stages. Because voxels are accessed only at the beginning of the pipeline, this leads to much reduced memory traffic.

Parallelism: The computational demands of real-time ray-casting require to use a high degree of parallelism, both for the memory system and for the arithmetic processing units.

Some of these improvements are of strictly algorithmic nature, and they are discussed in the following chapter. Other improvements require to think about architectural issues, and we discuss them in Chapter 6. However, due to the nature of the design process, most design decisions are not based on either algorithmic or architectural considerations, but on the combination of both. Consequently, the division between algorithm and architecture in the following two chapters becomes difficult and should be regarded as somewhat artificial.

Chapter 5

Fully Pipelined Ray-Casting

In this chapter we describe the fully pipelined algorithm for ray-casting that has been implemented in the Cube-3 and Cube-4 hardware architectures. We present several important extensions to the traditional ray-casting algorithm that have been developed as an attempt to simplify both hardware and software complexity.

Beginning with the first section, we discuss the use of templates to generate discrete rays from the description of the viewing parameters. This approach guarantees that each voxel inside the dataset is accessed exactly once per projection. Although template-based ray-casting is not new and has been well documented in the literature [YK92, SS92], we carefully explain all aspects and details of the algorithm that are relevant to the architectures presented in Chapters 7 through 9.

Section 5.2 describes a novel way of gradient estimation using interpolated samples of neighboring rays. Using this approach avoids multiple access to voxels of overlapping cell neighborhoods. The caching of interpolated samples avoids redundant computation and greatly improves the computational

efficiency.

Both improvements will lead to a fully pipelined version of ray-casting, discussed in Section 5.3, that dramatically reduces accesses to the voxel data and leads to an efficient implementation in hardware.

5.1 Template Based Ray-Casting

To simplify the address arithmetic and to get a mapping of ray-samples onto cells that is one-to-one, we use a template based ray-casting technique that was first introduced by Yagel and Kaufman [YK92] and independently developed for a SIMD massively-parallel processor by Schröder and Stoll [SS92]. Figure 23 shows an example for parallel projection. Instead of casting rays in

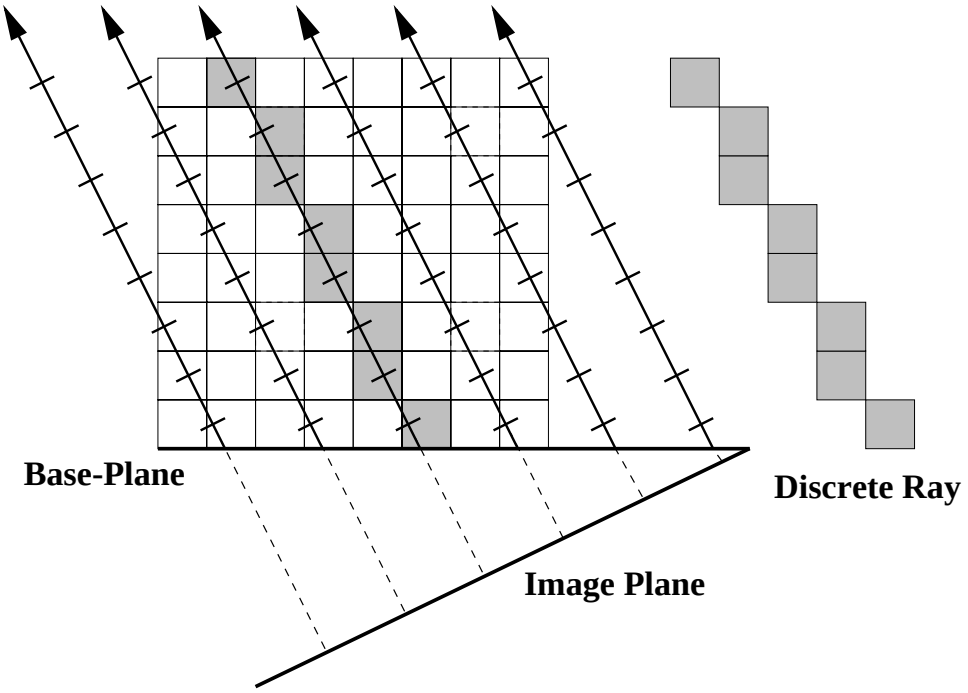

Figure 23: *Discrete rays are cast along cells from the base-plane into the dataset.*

```
v⃗ := o⃗v⃗ := v⃗0;
n := 0;
while ( v⃗ != v⃗1 ) do
Ray_Template[n] := ⌊v⃗ + 0.5⌋ − o⃗v⃗;
   o⃗v⃗ := ⌊v⃗ + 0.5⌋;
   n++;
   v⃗ += Δ⃗v;
end while
```

Figure 24: *Template Generation Pseudo-Code.*

image space, rays are sent into the dataset from each pixel on the *base-plane*, which is the face of the volume memory that is most perpendicular to the viewing direction. The resulting base-plane image is then projected onto the image plane using a 3D transformation and a 2D image resampling operation.

The cells that are pierced by a ray lie on so-called *discrete rays*. The discrete rays are pre-computed using a 3D variation of Bresenham's line drawing algorithm [KS86, Kau91]. This algorithm guarantees constant stepping with unit increment in major viewing direction so that all discrete rays perfectly tile the volume without holes or overlaps (see the two discrete rays in Figure 23). The uniform sampling of the volume by the discrete rays allows to store the path of one discrete ray in a lookup table or *template* that can be reused for every other ray.

5.1.1 Template Generation

The pseudo-code in Figure 24 shows the simple routine to generate templates containing the integer *steps* of the discrete rays. The code generates a template for the discrete ray starting at voxel position $\overrightarrow{v0}$ on the base-plane and ending at voxel $\overrightarrow{v1}$. $\overrightarrow{\Delta v}$ is the floating point vector in viewing direction whose

vector component in major viewing direction has been normalized to one. The variable $\vec{v}$ holds the floating point coordinates of the current sample location, and $\overrightarrow{ov}$ contains the truncated integer coordinates of the previous sample location. *Ray_Template* is a one-dimensional array of step vectors. The steps that are stored at index n are equal to $\lfloor \vec{v} + 0.5 \rfloor - \overrightarrow{ov}$. When sampling in the same cell, this vector will be zero in all dimensions.

Figure 25 shows an example template using a 2D drawing. Since the dis-

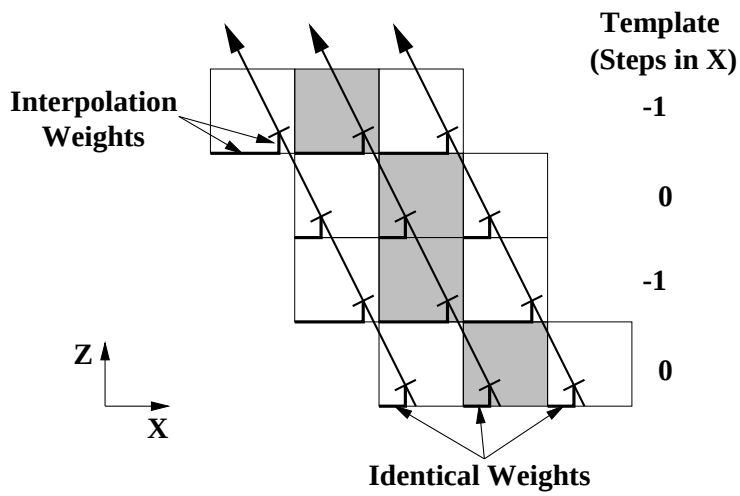

Figure 25: *A pre-computed template stores discrete ray steps and interpolation weights for sample points along the rays.*

crete rays step with unit distance in major viewing direction (the Z-axis in Figure 25), the template only needs to store the stepping in non-major directions (the X-axis in the figure). For parallel projections the template has n entries for an n^3 volume. The resampling weights for interpolation inside the cells are the same for all discrete rays and can be pre-computed and stored in the template as well.

For perspective projections we make the observation that all rays cast from the same scanline of the base-plane form a so-called *projection-ray plane* (PRP)

into the volume (see Figure 26). Inside each PRP the discrete rays follow the

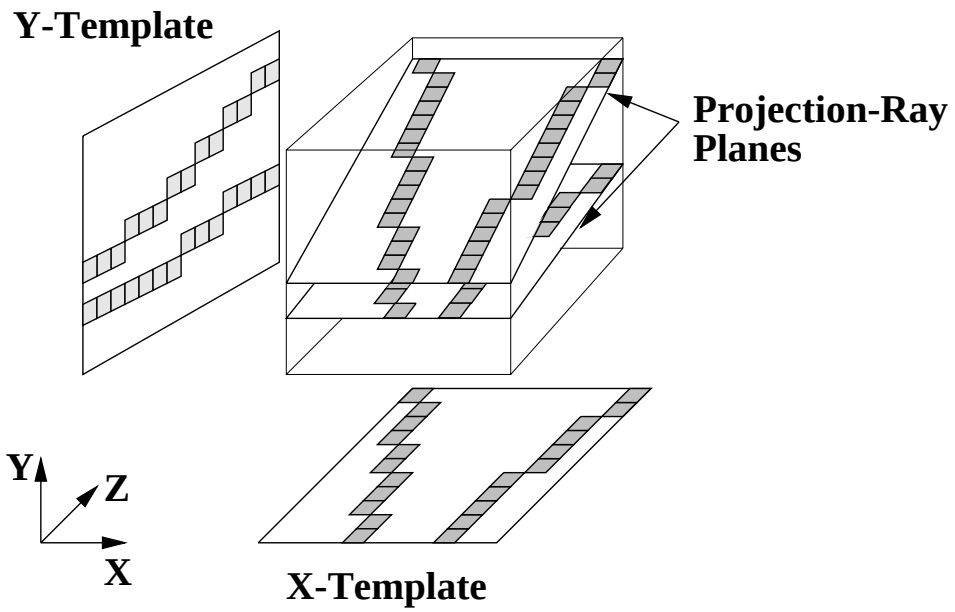

Figure 26: *Perspective ray-casting requires 2D templates of size n^2 each.*

same paths. The templates can be thought of as projections of the discrete rays along the two non-major directions (X and Y in Figure 26). Two *X- and Y-templates* of size n^2 each store the steps and weights of all perspective discrete rays.

Any ray exiting at the top face of the volume corresponds in a one-to-one fashion to a new ray that starts at a base-plane pixel n positions away (see for example the shaded cells in Figure 23). We call this a *wrap* of a discrete ray. Discrete rays can have at most one wrap in 2D and at most two wraps in 3D.

5.1.2 The Base-Plane Image

For orthogonal projections the dimensions of the base-plane is $n \times n$ pixels as illustrated using a 2D drawing in Figure 27. For non-orthogonal projections

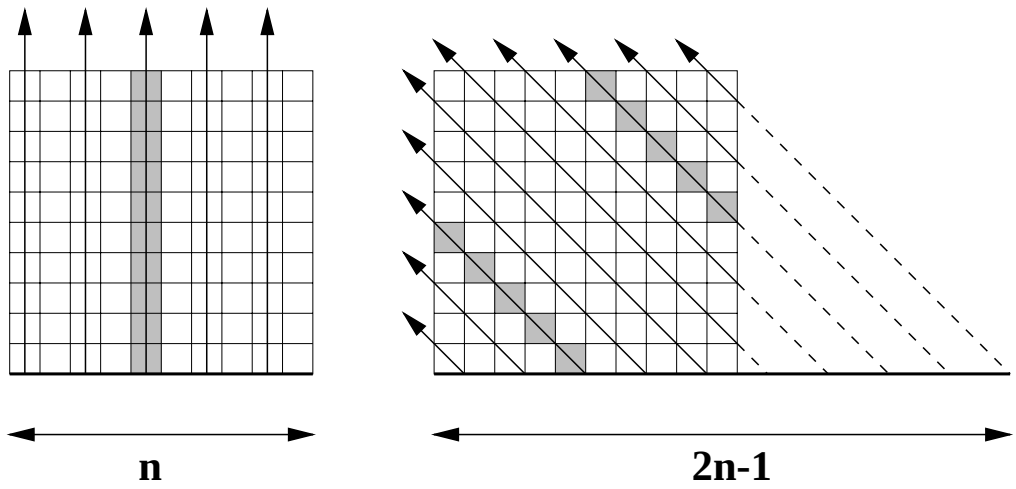

Figure 27: *Varying size of the base-plane depending on the viewing angle.*

the size of the base-plane becomes larger because not all discrete rays enter the volume at the front-most face. Figure 27 shows the worst case scenario of a projection under $\pm 45^0$ viewing angle where a total of $2n - 1$ rays are cast. In three dimensions the maximum base-plane size is $(2n-1) \times (2n-1)$ pixels.

The discrete ray generation algorithm is based on the fact the the viewing angle is constrained to $\pm 45^0$ around the major axis. Consequently, a *switch* of the base-plane occurs as soon as the major direction of the viewing vector changes (see Figure 28). Zero-order interpolation along voxels for projections at exactly 45^0 prohibits any visible artifacts due to the changing resampling pattern after a base-plane switch. However, the access pattern to the volume data changes dramatically. For two out of the three possible major directions the volume is accessed against storage order after a base-plane switch. We will come back to this issue in our discussion of high-bandwidth memory systems in Chapter 6.

The center of projection C and the field-of-view (FOV) in perspective projections influence the sampling rate of the base-plane image (see Figure 29).

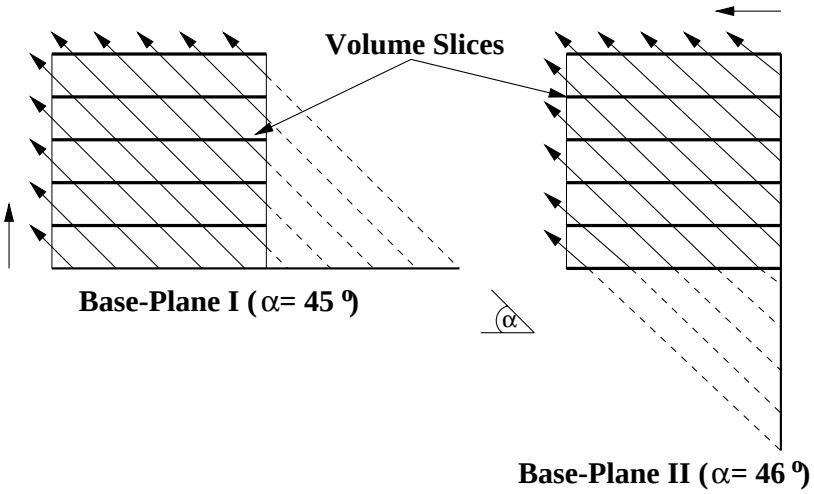

Figure 28: *Switch of the base-plane for viewing angle* $\alpha > 45^0$.

The discrete line algorithm casts exactly one discrete ray per pixel of the base-

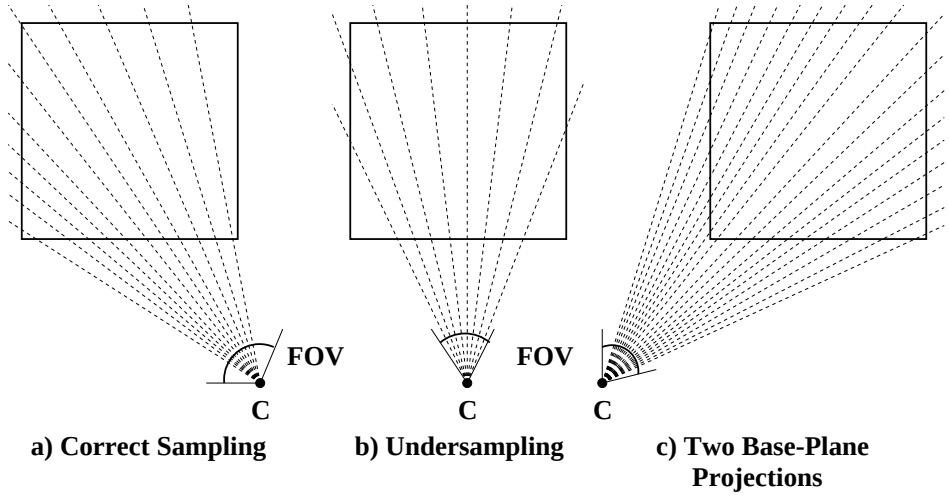

Figure 29: *Sampling of the base-plane image for perspective projections.*

plane, or a maximum of $2n - 1$ rays per scanline. In cases where the FOV extends across the dataset (Figure 29a) the template-based sampling approach guarantees better sampling than regular image order ray-casting, which would cast n rays spanning the FOV and send wasteful rays that miss the dataset.

However, for a small FOV the discrete line stepping yields undersampling in the active regions of the base-plane (Figure 29b). Figure 29c shows a case where two base-plane images contribute to the final view image. The worst case in 3D is the generation of three base-plane projections for a single perspective image.

5.1.3 Resampling the Volume

So far our discussion of template based ray-casting has focused on the discrete rays of cells without consideration to any specific resampling method. Figure 30 shows three different choices for resampling locations and interpolation methods. The simplest solution, shown in Figure 30a, is to use *nearest-neighbor*

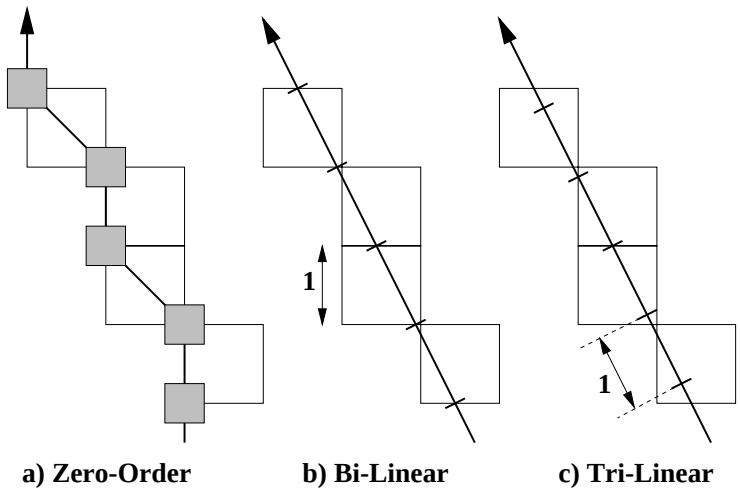

Figure 30: *Sampling distance and interpolation method along rays.*

sampling or *zero-order interpolation*. The sample values along the ray correspond to the voxels defined by the discrete line algorithm. Assuming that after n steps along the ray we are at voxel position $\vec{v}$, the voxel at position $\vec{v'}$ that

is sampled next is computed as:

$$\vec{v'} = \vec{v} + Ray_Template[n]. \tag{16}$$

Using this technique, Yagel reports a speedup of roughly 2 – 3x compared to a brute-force ray-caster [Yag91a]. A similar approach has been used by Schröder and Stoll [SS92] to achieve high frame rates on a SIMD multiprocessor.

However, zero-order interpolation leads to image artifacts for arbitrary viewing directions and is generally not acceptable in modern volume rendering applications. A notable exception are projections for $\pm 45^0$ viewing angle or orthogonal direction along a major coordinate axis. For these special cases the viewing rays start at the center of base-plane pixels and line up perfectly with the voxel positions. It is therefore better to use the original voxel values (i.e., nearest neighbor sampling) than to perform any form of interpolation.

To achieve higher accuracy than nearest neighbor sampling for arbitrary viewing directions, we employ linear or higher order interpolation. Figure 30b shows resampling at unit steps in major viewing direction and *bi-linear interpolation* on voxel slices pierced by the rays. This method leads to a very uniform resampling of the dataset due to the one-to-one mapping of sample locations and dataset cells. It has been successfully applied in shear-warp rendering algorithms [LL94]. However, the 2D interpolation may lead to aliasing artifacts if the voxel values or opacities contain high frequency components [Lac95a].

A consequence of the uniform stepping by one in major direction is that the sampling distance δ along the ray varies between 1 for orthogonal projections and $\sqrt{3}$ for projections along the main diagonals through the dataset. The under-sampling along the ray can be corrected by adjusting the opacity values

for each sample as follows:

$$\alpha_\delta = 1 - e^{-\alpha\,\delta}, \tag{17}$$

where α is the uncorrected opacity value, and α_δ is the corrected opacity value, depending on the sampling distance δ [Max95]. We found that the view dependent sampling rate is in practice not an important consideration due to the averaging nature of the compositing operator.

Figure 30c shows a more accurate resampling method using *tri-linear interpolation*. Instead of sampling with unit distance in major viewing direction, the resampling locations are generated at unit distance along the rays. This guarantees a view-independent sampling rate and true 3D interpolation without aliasing artifacts. As discussed in the previous section, all sample positions in the same slice across parallel rays have the same interpolation weights (see Figure 25). These weights can be pre-computed and stored in ray templates. Using this approach, Yagel and Ciula [YC94] report a speedup of 1.3 – 1.4x compared to brute-force ray-casting.

5.1.4 Warping the Base-Plane Image

The image produced by template based ray-casting is an *oblique projection* of the volume data [FvDFH90]. An affine image warp has to be applied to produce the final image. The homogeneous viewing transformation matrix M_v (see Equation 14) can be decomposed into a scaling component and a 2D warp

matrix (after [Lac95a]):

$$
\begin{aligned}
M_v &= \begin{bmatrix} m_1 1 & m_1 2 & m_1 3 & m_1 4 \\ m_2 1 & m_2 2 & m_2 3 & m_2 4 \\ m_3 1 & m_3 2 & m_3 3 & m_3 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} m_1 1 & m_1 2 & (m_1 3 - s_i m_1 1 - s_j m_1 2) & m_1 4 \\ m_2 1 & m_2 2 & (m_2 3 - s_i m_2 1 - s_j m_2 2) & m_2 4 \\ m_3 1 & m_3 2 & (m_3 3 - s_i m_3 1 - s_j m_3 2) & m_3 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & s_i & 0 \\ 0 & 1 & s_j & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= M_{\text{warp}}\, M_{\text{shear}}.
\end{aligned} \tag{18}
$$

The implementation of the 2D image warp is straight forward and can be accomplished using forward or backward projection [Wol90]. Real-time performance for the warping of large images can be achieved using special image processing instructions on commodity processors [PW96], texture mapping hardware on high-end graphics workstations [SKvW+92], or special-purpose image processing hardware [JJL96].

The additional 2D resampling of the image may cause blurring artifacts. However, in practice, we have never observed any significant image differences between template-based and traditional ray-casting.

5.2 ABC Gradient Estimation

We have seen in Chapter 4 that gradient estimation leads to unnecessary memory access and redundant computation due to overlapping cell neighborhoods. Furthermore, we want to achieve a fully pipelined algorithm, where all data transfers occur between consecutive stages of the pipeline. In this section we

present a novel gradient estimation technique that calculates the gray-level gradient not from voxel data, but from interpolated samples of neighboring rays. Figure 31 illustrates the idea in 3D. Consider the black sample at the

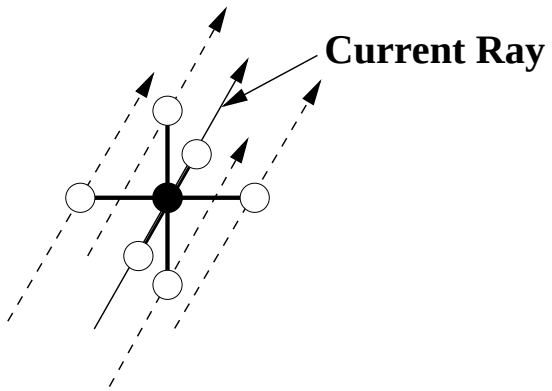

Figure 31: *ABC Gradient Estimation.*

current sample location. We call the associated ray the *current ray.* Central differences between samples of neighboring rays (the thick lines in the figure) allow us to estimate the gradient without further access to the voxel data. Because the computation involves samples along the current ray and from the rays above and below, we call this method *ABC gradient estimation.* The following sections present several ABC gradient methods. They differ by the number of samples used during the gradient computation.

5.2.1 6-Neighborhood Gradient

The simplest approach, shown in Figure 32 using a 2D drawing, is called the *6-neighborhood gradient.* We compute central differences using six samples values of neighboring rays. The resulting gradient vector has two components parallel to the base-plane, and one component in direction of the ray. Dashed lines

in the figure indicate the slices of interpolated samples parallel to the base-plane. The gradient components inside the slice of the current (black) sample

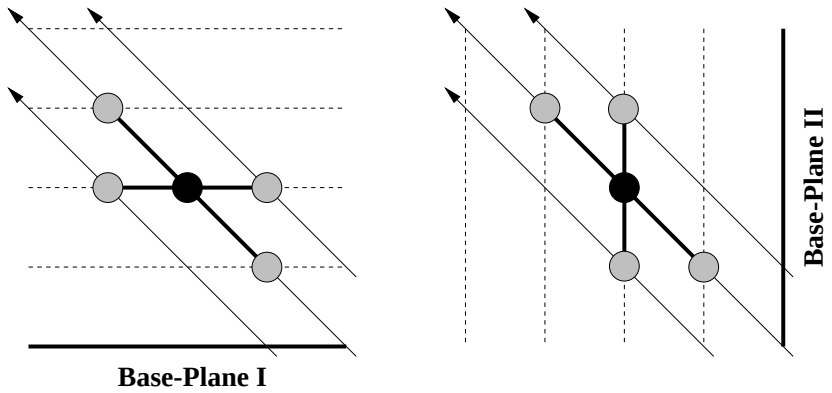

Figure 32: *6-Neighborhood Gradient.*

are orthogonal to each other. However, the gradient component in direction of the ray is slanted, depending on the viewing direction (see Figure 32). Furthermore, the central difference along the ray is computed over a view-dependent length. In the worst-case, for 45^0 viewing along a main diagonal of the dataset, the distance between samples along the ray is $\sqrt{3}$ units longer than the distance between the samples parallel to the base-plane.

A more critical problem occurs after a switch of the base-plane during rotations of more than $\pm 45^0$. The left drawing in Figure 32 shows the situation for 45^0 viewing direction, where an image is projected onto the horizontal base-plane. For an angle greater than $\pm 45^0$ the major viewing direction changes, as shown in the right drawing of Figure 32. Different samples are used to calculate the central differences parallel to the base-plane. The sudden change of the gradient leads to noticeable changes in object shading. The intensity variations of subsequent images during a base-plane switch result in intolerable temporal aliasing.

5.2.2 26-Neighborhood Gradient

One approach to alleviate the problems of the 6-neighborhood gradient is to use a higher order gradient filter on a $3 \times 3 \times 3$ sample neighborhood. Because 26 samples are involved in the computation, we call this the *26-neighborhood gradient* (see Figure 33). Samples along nine rays are used during convolution

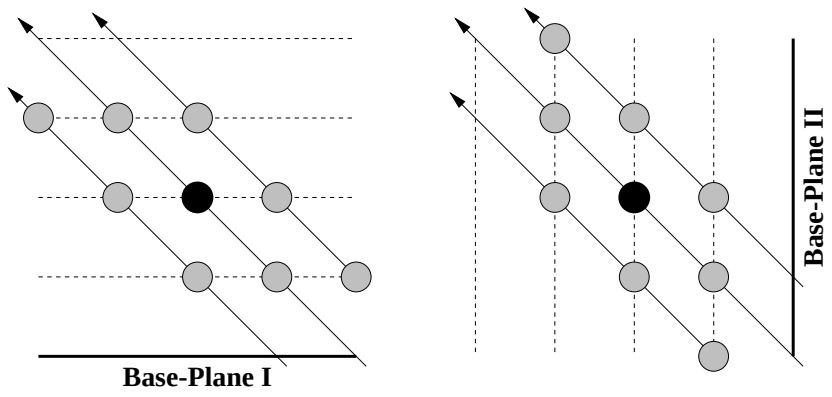

Figure 33: *26-Neighborhood Gradient.*

with a $3 \times 3 \times 3$ gradient filter, for example, the Sobel filter of Equation 2.

The 26-neighborhood gradient leads to better overall image quality when compared to the 6-neighborhood gradient. However, the convolution with a gradient filter is computationally expensive. The view dependent slanted direction of the rays leads to non-orthogonal $3 \times 3 \times 3$ sample neighborhoods. Furthermore, the switch of base-planes still leads to temporal aliasing, because the sample neighborhoods vary for different base-planes (see Figure 33).

5.2.3 10-Neighborhood Gradient

To circumvent the aliasing problems we need to ensure view-independent and orthogonal gradient components. A possible approach is to use the 6-neighborhood gradient, but with two gradient components orthogonal to the

current ray. Figure 34 shows how the samples on the left and right ray are used for a linear interpolation of two additional samples (in white). The gradient component between these samples is orthogonal to the ray direction. We

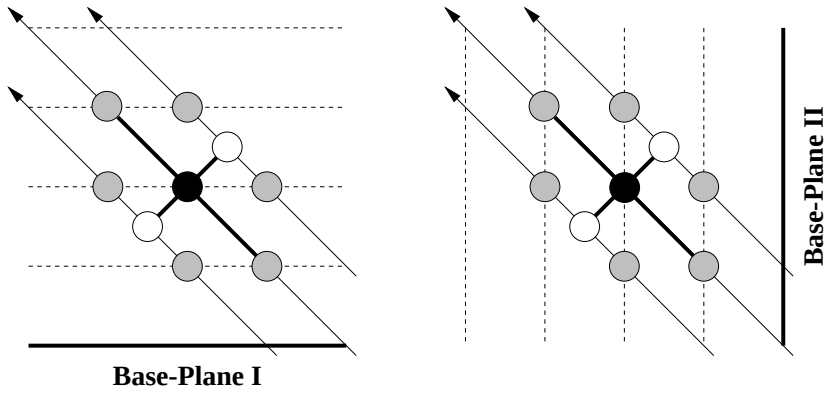

Figure 34: *10-Neighborhood Gradient.*

call this approach the *10-neighborhood gradient*, because 10 voxels participate in the computation in 3D. There is no temporal aliasing during base-plane switches, because the gradient components remain orthogonal to each other (see Figure 34).

However, some problems remain. The difference in lengths between the gradient components is aggravated when compared to the 6-neighborhood gradient. Furthermore, the local coordinate system defined by the gradient components is still view dependent. All light vectors used during the shading calculations must be transformed to this coordinate system for each frame. For perspective projections this amounts to transforming the light vectors to different local coordinate systems for every ray. Perspective projections also require different linear interpolation weights for every ray.

5.2.4 12-Neighborhood Gradient

Our final solution solves all of the above problems by calculating view-independent gradient components parallel to the main axes of the volume memory (see Figure 35). Two gradient components are computed inside the current slice. The third gradient component is computed perpendicular to the current slice using additionally interpolated samples (shown in white). In 3D, these additional samples require a bi-linear interpolation among ray samples. For each bi-linear interpolation we need four samples, plus two samples in

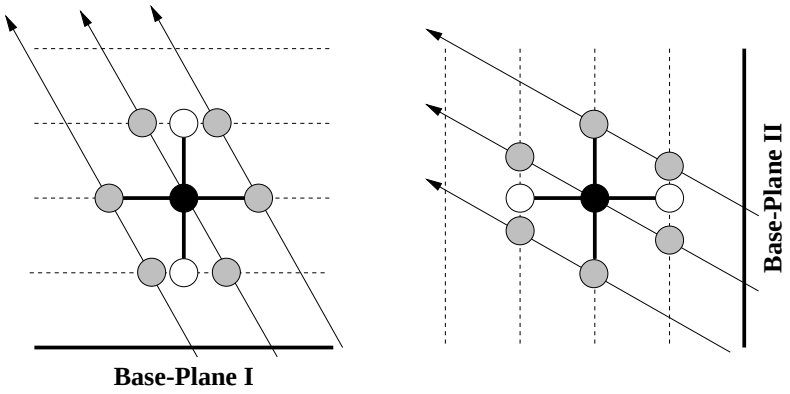

Figure 35: *12-Neighborhood Gradient.*

each of the other gradient component directions. Consequently, we call this technique the *12-neighborhood gradient*. An additional benefit of this method is that the central differences are taken over equal distances for all gradient components.

Chapter 7 contains a comparison of errors between the 6-, 10-, 12- and 26-neighborhood gradient and the analytic normals on a sphere. We also present results from a 90^0 rotation around the sphere with a base-plane switch that demonstrates the temporal aliasing problems for the 6-, 26-, and 10-neighborhood gradients.

5.2.5 Proof of Equivalence

We can easily proof that the ABC gradient is equivalent to the traditional gray-level gradient. Because the three gradient components are computed independently, it is sufficient to proof this equivalence for the X gradient component. To simplify the discussion, we will use V_i for the voxel value at integer location i, and S_t for the interpolated sample value at sample location t. Figure 36 shows the voxel and sample values we use for this proof. For simplicity, they are consecutively enumerated along the X axis.

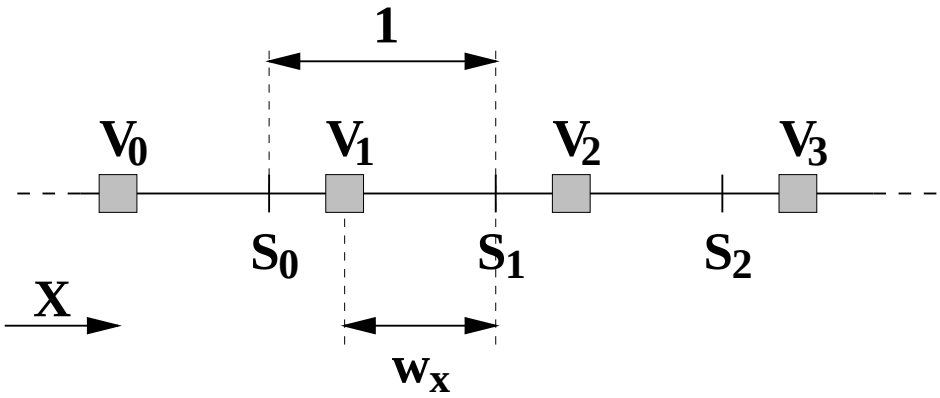

Figure 36: *Proof of Equivalence.*

Recall from Equation 3, that the gray-level gradient at voxel position x is computed as $\nabla f(x) = f(x+1) - f(x-1)$. Using G_x instead of $\nabla f(x)$ and V_x instead of $f(x)$, we compute the x-gradient components at the voxel locations V_1 and V_2 as:

$$\begin{aligned} G_x(V_1) &= V_2 - V_0 \\ G_x(V_2) &= V_3 - V_1. \end{aligned} \tag{19}$$

To compute the x-gradient at sample location S_1 requires a linear interpolation

between these gradients as follows:

$$\begin{aligned} G_x(S_1) &= (1-w_x)G_x(V_1) + w_x G_x(V_2) \\ &= (1-w_x)(V_2 - V_0) + w_x(V_3 - V_1) \\ &= (1-w_x)V_2 - (1-w_x)V_0 + w_x V_3 - w_x V_1. \end{aligned} \tag{20}$$

Alternatively, we can compute the ABC gradient at sample position S_1 using the central difference between interpolated neighboring samples:

$$\bar{G}_x(S_1) = S_2 - S_0. \tag{21}$$

Comparing Equation 21 to Equations 19 and 20 shows that the arithmetic complexity of ABC gradient estimation is much smaller than for the traditional gray level gradient due to the caching of previously interpolated samples. Expanding the linear interpolation terms of the samples leads to the same result as in Equation 20:

$$\begin{aligned} S_0 &= (1-w_x)V_0 + w_x V_1 \\ S_2 &= (1-w_x)V_2 + w_x V_3 \\ \bar{G}_x(S_1) &= (1-w_x)V_2 + w_x V_3 - (1-w_x)V_0 - w_x V_1. \end{aligned} \tag{22}$$

This proof holds only for linear interpolation. Furthermore, all arithmetic operations are assumed have infinite precision. This assumption is not true for any hardware implementation, and special care has to be taken to avoid problems due to arithmetic error accumulation.

5.3 Summary

Figure 37 shows the new ray-casting pipeline with the two modifications introduced in this chapter: template-based address generation and ABC gradient

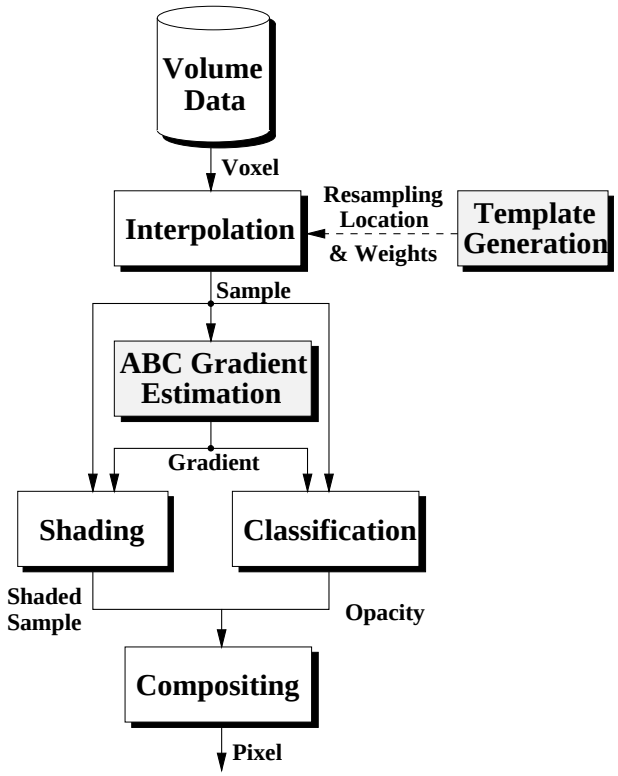

Figure 37: *Fully Pipelined Ray-Casting Algorithm.*

estimation. The main difference compared to Figure 22 is the fully pipelined dataflow. The voxel data is only accessed at the beginning of the pipeline by the interpolation stage. The template-based data traversal guarantees that each voxel is accessed exactly once per projection. The interpolated samples are cached and re-used during ABC gradient estimation, resulting in reduced arithmetic complexity. ABC gradient estimation also prevents overlapping neighborhoods between voxels used for interpolation and gradient estimation. Once all stages are filled with data, the fully pipelined ray-casting algorithm is capable of delivering one result per computation period. The algorithm

dramatically increases the data throughput when compared to traditional ray-casting.

In the next chapter we discuss several important architectural issues of parallel computer systems, such as parallel memory systems, data and task distribution, arithmetic unit replication, and communication between processing nodes. We look at ways of exploiting the inherent concurrency of ray-casting, and present three different ways of parallelizing the fully pipelined ray-casting algorithm.

Chapter 6

Parallel Architectures

For a volume rendering system that tries to achieve real-time performance it is imperative to use parallelism. The last chapter introduced pipelining, one form of parallelism, into the ray-casting algorithm. In this chapter we look at ways to explore parallelism in the architectural domain.

We start with a discussion of memory systems for volume rendering. In Section 6.1, we discuss the basic design considerations for memory systems. Section 6.2 introduces interleaving, the primary method to introduce parallelism into the memory. Section 6.3 presents block memory and vector memory, two methods of interleaving that allow high-bandwidth access to rectilinear volumetric datasets. We argue that vector memory provides a scalable solution to the high bandwidth requirements of volume rendering.

In Section 6.4, we introduce a vector memory organization based on a linear skewing of the address space that allows for conflict-free access to voxel scanlines from all three principal viewing directions. This memory system is used for the architectures presented in this dissertation. In Section 6.5 we discuss an important extension to the skewing scheme that allows for modular

hardware implementations with scalable performance

In Section 6.6 we apply unit replication, another form of parallelism, to the fully pipelined ray-casting algorithm outlined in the previous chapter. Finally, in Section 6.7, we present three approaches to the parallelization of pipelined ray-casting. Depending on how the the algorithms proceed we call these parallelization *ray-*, *beam-*, or *slice-parallel*. Each of these algorithms leads to a different architecture presented in the following chapters.

6.1 Memory System Basics

During the last decade, very large scale integrated circuit technology (VLSI) has increased the speed of logic faster than the speed of memory [HP90]. This trend has made the memory system the principal design challenge for every high-performance computer system. The three parameters that determine memory system performance are *access time*, *cycle time*, and *memory bandwidth*. Access time is the time it takes from data request to data delivery. Cycle time is the minimum amount of time between data accesses. Memory bandwidth determines the amount of requests that can be handled per unit time. Access and cycle times are determined by physical factors of the memory chip, such as technology and internal delays. The maximum achievable memory bandwidth is determined by the way the memory system is organized.

Memory modules are the smallest building blocks of the memory system. The two main classes of memory modules are *static random access memory* (SRAM) and *dynamic random access memory* (DRAM). SRAMs use multiple transistors to hold each bit of information, whereas DRAMs use a single transistor and a capacitor per bit. Consequently, DRAMs are about one hundred times more dense than SRAMs, but SRAMs are much faster and have only

about one-tenth of the access and cycle time of DRAMs [Fly95]. To meet the large storage requirements of volume rendering, our memory system needs to be composed of high-density DRAMs.

A *memory bank* consists of multiple memory modules that share the same input and output buses. Although only one memory module of the memory bank is capable to respond to a memory request at any given point in time, memory banks allow to hide the long access and cycle time of standard DRAMs using a technique called *interleaving*.

6.2 Memory Interleaving

The idea of memory interleaving is to partition the address space such that all data references are distributed fairly uniformly across memory modules. The simplest and most common form of interleaving assigns successive memory addresses to distinct memory modules. For m memory modules, enumerated from 0 to $(m-1)$, memory address a is assigned memory module number $k = a \bmod m$. Figure 38 shows the resulting partitioning of the address space across memory modules.

The *index* i into the memory module is calculated as $i = \lfloor \frac{a}{m} \rfloor$. The number of memory modules in an interleaved memory bank is called the *degree of interleaving*. Alternatively, a memory bank is said to be *m-way interleaved*. If m is a power of two, the calculation of $a \bmod m$ degenerates to a masking operation of the low-order bits of the address a. The index i into the memory module is provided by the remaining higher order bits of a. In this case the memory is said to be *low-order interleaved* [Fly95].

Using low-order interleaving is a standard technique in almost all high-performance computer systems. However, low-order interleaving only provides

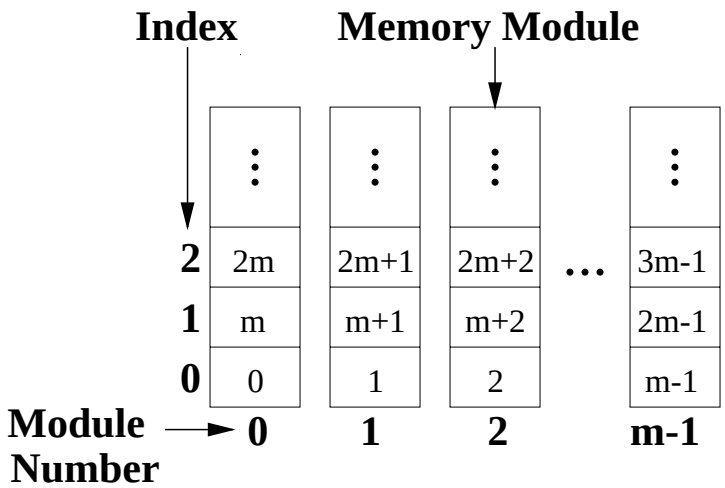

Figure 38: *Low-order interleaved memory system with m memory modules. Memory address a is assigned to memory module k according to $k = a \bmod m$.*

maximum bandwidth if the data access is sequential and creates no interference. Notice in Figure 38 that the addresses are stored successively in distinct memory modules. We call such an organization *rows across the memory*, because it is similar to the row-wise assignment of successive addresses in a 2D array. If the data access pattern is sequential and by row, the memory is able to exhibit maximum bandwidth. However, accesses to columns can have severe performance degradations, because only a subset of the memory modules (in the worst case only one) is being referenced.

Given two consecutive addresses a_i and a_{i+1}, we call their distance $a_{i+1} - a_i$ the *stride* of the access pattern. Low-order interleaved memory performs best for consecutive stride one (row) access, but performance breaks down for stride m (column) access. The problem occurs for any stride t such that t and m have a common factor, that is if $\gcd(t, m) > 1$, where gcd stands for the greatest common divisor [NM87]. For example, consider the access to a low-order interleaved memory system with stride m. For some starting address a,

the access pattern is $a, a+m, a+2m, a+3m, \cdots a+(m-1)m$. All of these references fall onto the same memory module at position $k = a \bmod m$. This leads to *memory access conflicts*. In multi-stage networks it may lead to a *memory hot spot* and network blockage [PN85].

Such stride accesses occur frequently volume rendering, which exhibits systematic references to matrix data. For low-order interleaved memory systems this may lead to worst case performance. For example, accessing the volume data against storage order significantly slows down performance of object-order volume renderers. One solution is to store the dataset three times, once for each main axis. This approach has been taken in current object-order shear-warp implementations [LL94]. However, data triplication increases the already high storage requirements of volume rendering.

The design of a dedicated architecture for volume rendering requires a memory organization that naturally supports the high-bandwidth needs of volume rendering without pre-processing and data replication. The difficulty is to design a memory system that sustains a continuous flow of data from the memory to multiple parallel processors.

6.3 Memory Systems for Volume Rendering

Consider a regular volumetric dataset with $n \times n \times n$ voxels. Figure 39 shows a $4 \times 4 \times 4$ dataset in its local coordinate system. Each voxel in the figure is represented by its address a, which is the $[zyx]$ tuple with the local coordinates of the voxel. For example, the voxel at location $x = 1$, $y = 3$, and $z = 0$ has address $a = [031]$. We refer to this standard arrangement of voxels as *volume space*.

For maximum performance we need to interleave this dataset across m

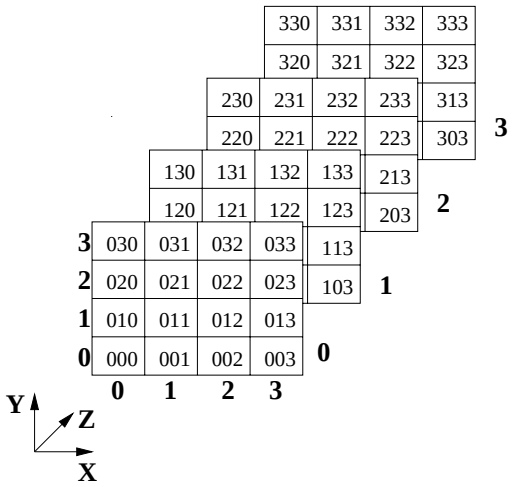

Figure 39: *Assignment of voxel addresses* $[zyx]$ *in volume space for a* $4 \times 4 \times 4$ *dataset. Numbers at grid points indicate the memory module where the voxel is stored.*

memory modules, each containing w words. Assuming that each voxel requires one word of storage (either 8 or 16 bits), we need a memory capacity of $wm \geq n^3$ in order to store the complete dataset. To distribute the volume space across memory modules, the voxel address $a = [zyx]$ must be decomposed into a memory module number k, and an index i into the module (the address of the word inside a module).

To interleave a regular, rectilinear volumetric datasets in memory means to subdivide volume space into smaller subsets that can be accessed during one memory cycle. The minimal building blocks of any rectilinear subdivision are voxels (0-dimensional), vectors (1-dimensional), rectangles (2-dimensional), and blocks (3-dimensional). In the following two sections we consider *block* or *vector* data subdivisions. We call the resulting memory systems *block memory* and *vector memory*.

6.3.1 Block Memory

The minimum voxel cell required for tri-linear interpolation consists of eight voxels, which can be stored in an eight-way interleaved memory. The easiest way to ensure conflict-free access to any cell of eight voxels is to use low-order interleaving in each of the three dimensions. A voxel with address $a = [zyx]$ is stored in memory module k at index i as follows:

$$\begin{aligned} k &= (x \mod 2) + 2\,(y \mod 2) + 4\,(z \mod 2), \\ i &= \lfloor \frac{x}{2} \rfloor + 2\,\lfloor \frac{y}{2} \rfloor + 4\,\lfloor \frac{z}{2} \rfloor. \end{aligned} \tag{23}$$

Equation 23 corresponds to using the least significant bits of the voxel address in each dimension (x_0, y_0 and z_0) as coding for the memory module number ($k = [z_0 y_0 x_0]$). The index i is given by the remaining bit pattern of the voxel address. Figure 40 shows the resulting assignment of voxels to memory modules in three dimensions. Notice that the lower left corner of the cell may fall on even grid points (dark outlined cells) or on odd grid points (dashed cells). This variation has to be considered during assignment of the interpolation weights. It can be corrected in software or in hardware by supplementing each memory module with a special weight modification unit [Kni93].

8-way interleaved memory has been used in special-purpose volume rendering architectures, for example in VOGUE [Kni93, KS94] and VIRIM [GPR+94] (see Chapter 3). Other block memory systems for volume rendering include the 64-way interleaved memory cube used in the PARCUM system, which allows conflict-free access to a macro volume element (i.e., a block) of 64 voxels [JS88] (see Section 3.3.1).

Block-based volume memory is a generalization of the techniques used to access rectangles of pixels in 2D raster display memory [CLR82, Gup81]. It offers a simple and effective memory organization that allows for compact

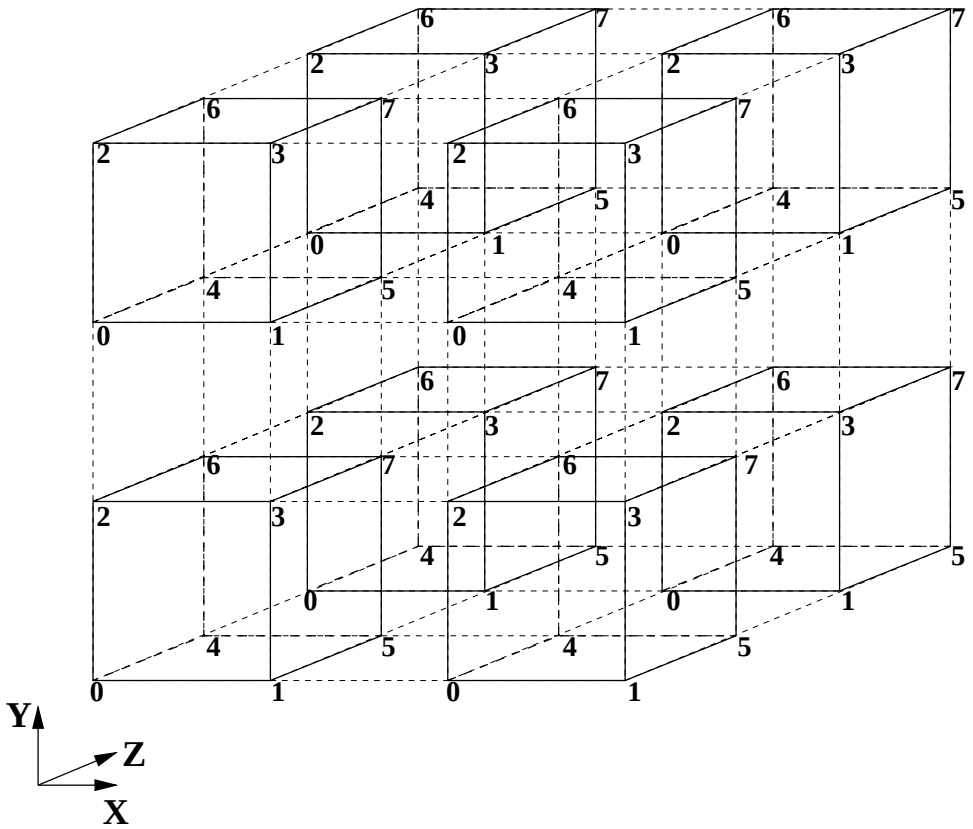

Figure 40: *Assignment of voxels to memory modules in an 8-way low-order interleaved volume memory.*

hardware implementations. Any cell of eight voxels can be fetched conflict-free in one memory access. Subsequent accesses occur at intervals corresponding to the cycle time of the memory. Performance can be improved by pre-fetching larger neighborhoods and using the fast DRAM page mode during accesses to pre-fetched voxels [KS94].

Block memory systems use a fixed degree of interleaving. The entire bandwidth of an 8-way interleaved memory bank is consumed by a single processor during interpolation or gradient estimation. To allow several processors to

operate in parallel requires to copy complete memory banks with m memory modules each. Each memory bank is statically connected to a different processor.

Although multiple block-based memory banks increase the data throughput, the static connection between memory and processors leads to problems for data and task distribution. Current implementations of block-based memory systems replicate the data in each memory bank, such as in the VIRIM architecture [GPR+94]. Other solutions require a high-bandwidth global network between processors, such as in the VOGUE architecture [Kni93]. The complexity of this interconnection becomes prohibitive for high resolution datasets.

6.3.2 Vector Memory

As we have seen in Section 6.2, low-order interleaving allows access to row vectors of data. However, any access to vectors with a stride greater than one leads to access conflicts and performance degradation. Vector memory systems allow accesses with a stride of the form 2^k, 10^k, or other even dimensions.

One way to design vector memory is to cause m and the stride t to be relatively prime. For example, for strides of the form 2^k it is possible to interleave the memory across $2^k \pm 1$ modules [Fly95]. Another approach is to apply a Boolean (or binary) matrix multiplication to the addresses [NM87]. Alternatively, addresses can be re-mapped using a hash table [Fly95]. Batcher [Bat77] and Frailong et. al [FL85] use XOR operations to map addresses to memory modules.

Vector memory offers an unlimited degree of interleaving and therefore an appropriate means to adjust the maximum bandwidth of the memory system to the expected maximum data throughput. The properties of vector memory, high data throughput and scalability, make it the solution of choice for our

real-time volume rendering architectures. The design challenge is to provide sufficient memory bandwidth while keeping the network between memory and processors at a manageable size.

Our vector memory uses a linear skewing of the address space that allows conflict-free access to any vector parallel to a principal axis [KB88]. This skewing scheme can be extended to allow for conflict-free access to voxel vectors along any of the 26 primary directions, i.e., along voxel scanlines and primary diagonals [CK95]. In the following section we explain the linear skewing that is used for the architectures described in this thesis. We start with the special case $m = n$, where the degree of interleaving is the same as the dataset dimension. In Section 6.5 we generalize our results to the case $m < n$, such that $\gcd(m, n) \neq 1$.

6.4 Skewed Memory Organization

For the mapping of a voxel with address $a = [zyx]$ onto a memory module k and index i we use the *skewing function* $\sigma : [z, y, x] \rightarrow [k, i]$. If $m \geq n$, the dataset can be mapped into memory by placing the voxel with address $[zyx]$ in memory module number k at index i as follows:

$$\begin{aligned} k &= (x + y + z) \bmod n \qquad & 0 \leq k, x, y, z < n, \\ i &= y + zn & 0 \leq i < n^2. \end{aligned} \tag{24}$$

The index mapping places adjacent voxels in X direction in the same relative locations of adjacent memory modules (i.e., rows across the memory). This choice of index mapping is arbitrary [KB88]. Figure 41a shows the resulting assignment of voxel addresses $[zyx]$ to memory modules, for $n = m = 4$. We define a *beam* to be a voxel scanline parallel to a coordinate axis in volume space. A beam parallel to the β-axis is called a $\beta-$beam. For example, rows

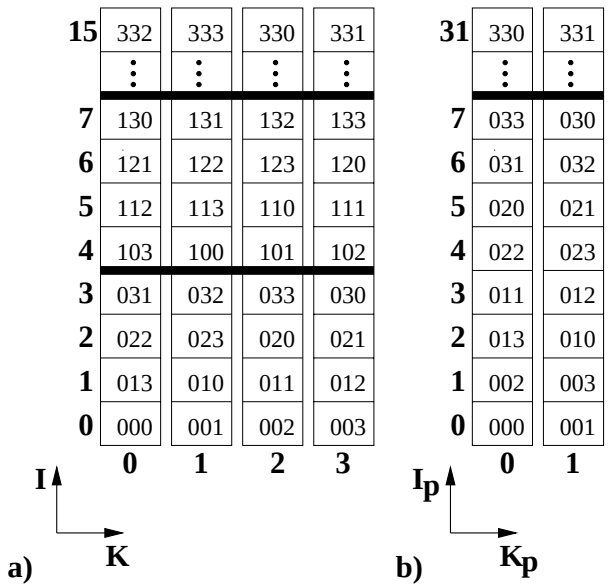

Figure 41: *3D skewed memory organization for a* $4 \times 4 \times 4$ *dataset. a) Dataset stored in* $m = 4$ *memory modules. b) Dataset stored in* $m = 2$ *memory modules. Thick lines indicate slice boundaries inside the memory.*

and columns are called X-beams and Y-beams, respectively, and Z-beams are called axles. If n is the dataset resolution, X-, Y-, and Z-beams have a stride of 1, n, and n^2, respectively. The skewed memory organization allows for conflict free access to any beam of voxels.

We define an *order* of voxels inside a beam based on their addresses in volume space as follows: For any two voxels v_1 with $a_1 = [ijk]$ and v_2 with $a_2 = [lmn]$, we say $v_1 > v_2$ if $[ijk] > [lmn]$. The distance by which two ordered beams are shifted is called their *skewing difference*. For example, beams that are stored in consecutive memory indices i in Figure 41a have a skewing difference of one.

6.5 Memory Partitioning for $m < n$

If $m < n$, the mapping is performed on partitions of beams in X direction, each of which is no longer than m. We require that n and m are not relatively prime. If $g = \frac{n}{m} \neq 1$ is their greatest common divisor, each memory module stores g voxels. This corresponds to a re-mapping of the skewed memory space by a *partitioning function* $\phi : [k, i] \rightarrow [k_p, i_p]$, where:

$$\begin{aligned} k_p &= k \bmod m \qquad & 0 \leq k_p < m, \\ i_p &= i\tfrac{n}{m} + \lfloor \tfrac{k}{m} \rfloor \qquad & 0 \leq i_p < \tfrac{n^3}{m}. \end{aligned} \tag{25}$$

Figure 41b shows the partitioned memory space for $n = 4$ and $m = 2$. It is important to notice that this skewing and partitioning of the memory space works for any n and m not relatively prime. In general, the computation of $(x + y + z) \bmod n$ or $k \bmod m$ involves a division operation. If n and m are powers of two, it degenerates to a masking operation with the low order bits of the operand.

Figure 42 shows the effect of address-partitioning in volume space. We refer to the partitioned beams as *partial beams*. Each partial beam contains m voxels, $m < n$. The figure shows the partial beams in storage order, i.e., the partial beams in X, but the skewed memory organization allows for conflict-free access to partial beams from any major direction.

The memory address i_p in each memory module can be computed incrementally for consecutive access to partial-beams. A γ-slice is a 2D array of voxels with constant γ coordinates. In the slice-parallel algorithm, we access voxels from X-, Y-, or Z-slices by consecutively fetching partial beams inside these slices. We denote the currently accessed partial beam number by N_p, the corresponding beam number by N_b, and the slice number by N_s. There are $\frac{n}{m}$ partial beams in one beam. In order to calculate i_p, we multiply i with

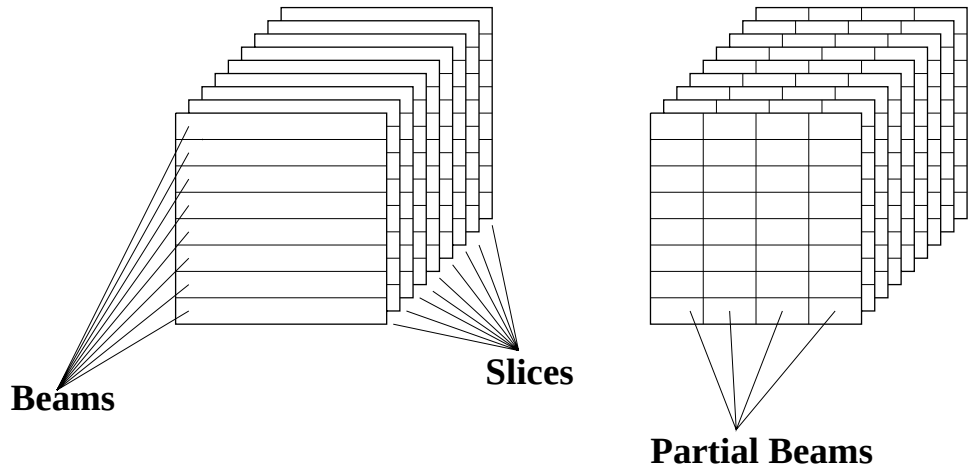

Figure 42: *Beams and the partitioning of beams into partial beams in volume space.*

$\frac{n}{m}$ and add the current partial beam number N_p:

$$\begin{aligned} i_p &= i\frac{n}{m} + N_p = (N_b + N_s n)\frac{n}{m} + N_p \\ &= N_p + N_b\frac{n}{m} + N_s\frac{n^2}{m}. \end{aligned} \tag{26}$$

For consecutive access to the slices, N_p, N_b, and N_s are incremented in order, for example, in positive coordinate direction.

Our skewed memory organization has several important advantages. Using m memory modules, it allows to store an n^3 dataset without data duplication. The skewing guarantees conflict-free access to partial beams of m voxels from all three major viewing axes. Finally, it allows for an unlimited degree of interleaving, which leads to scalable memory system performance.

6.6 Pipelining and Unit Replication

As mentioned in Chapter 4, pipelining leads to the processing of several data items in parallel, and it is considered one of the principal ways of introducing parallelism into an algorithm or architecture. The fully pipelined ray-casting algorithm presented in the previous chapter (Chapter 5) leads to the pipelined architecture shown in Figure 43. It consists of five processing stages: data

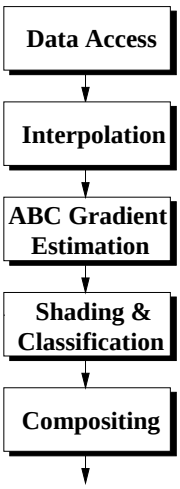

Figure 43: *Processor Pipeline for Ray-Casting.*

access, interpolation, ABC gradient estimation, shading and classification, and compositing. Notice that the voxel data is accessed only at the first stage of the pipeline. All subsequent data transfers happen between consecutive pipeline stages without further access to the volume memory.

However, pipelining itself can not deliver the required processing power for real-time volume rendering. At each stage of the pipeline there is only one arithmetic unit to perform computationally expensive operations, such as interpolation, shading, or compositing. In order to achieve higher performance we need to exploit another form of parallelism called unit replication [HJ88].

Unit replication at each stage of the ray-casting pipeline leads to the *parallel rendering pipelines* shown in Figure 44. By pipelining and unit replication we

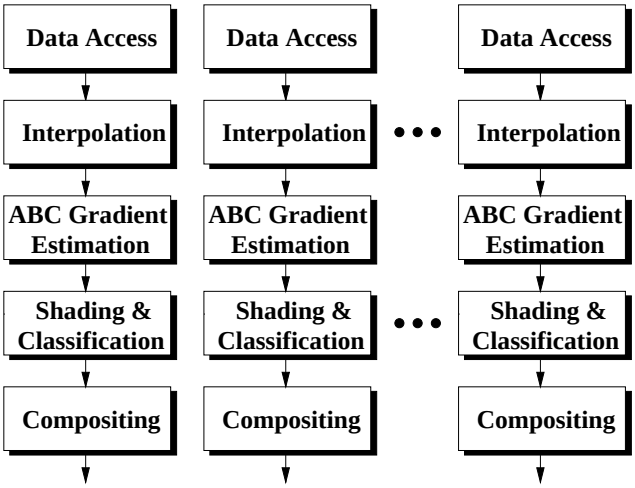

Figure 44: *Parallel Ray-Casting Pipelines.*

achieve the maximum level of parallelism and performance for the architecture. However, other important issues are the interconnections and data paths, both between processors and memory, and between processors of different rendering pipelines. The cost of communication in terms of area, input/output ports, packaging, and delay is high compared to the cost of logic and storage [HP90]. The types of communication networks and dataflow will be the distinguishing factors between the architectures we discuss in the reminder of this dissertation.

6.7 Parallel Pipelined Ray-Casting

Figure 45 shows three possible approaches to parallel ray-casting. According to the form of parallelism that is exploited we call these algorithms ray-, beam- or slice-parallel.

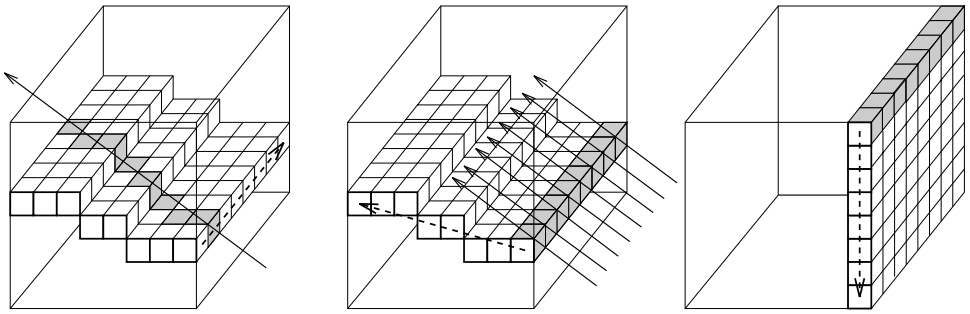

Figure 45: *Three different approaches to parallel ray-casting. Shaded voxels are processed simultaneously. The dashed arrows indicate in what direction the algorithm proceeds during subsequent timesteps.*

In the *ray-parallel* approach, all voxels along a ray are processed simultaneously (the shaded voxels in Figure 45a). The algorithm proceeds ray by ray in scanline order (the thick arrow in Figure 45a). The Cube-3 architecture, conceptually shown in Figure 46, implements ray-parallel ray-casting [PKC94, PWK94, HMK+95]. Ray-parallel processing allows for an efficient binary-tree implementation of compositing (indicated by VCUs in the figure). However, the simultaneous access to all voxels along a ray requires global bus between the volume memory and the rendering pipelines. The high bandwidth requirements for this processor-memory bus ultimately limit the performance and the scalability of the architecture.

An alternative to operating on all samples of a single ray is to simultaneously operate on samples of several neighboring rays [Chi93]. Depending on how the algorithm proceeds, we call these approaches *beam-parallel* (see Figure 45b) or *slice-parallel* (see Figure 45c). The beam-parallel ray-casting approach follows a group of rays by fetching consecutive beams in the major viewing direction [PWK95].

The slice-parallel approach, shown in Figure 45c, processes consecutive

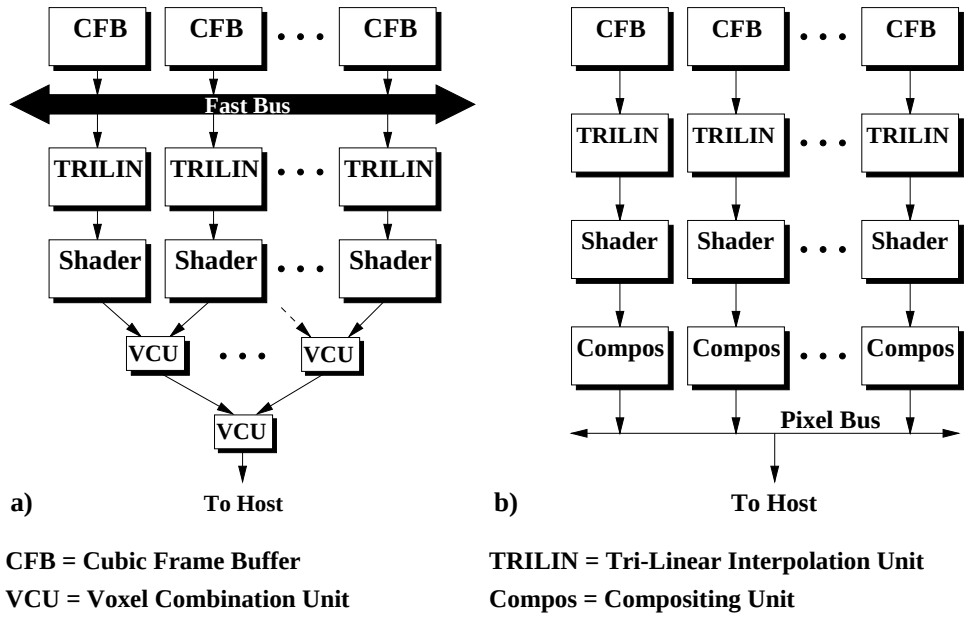

Figure 46: *The Cube-3 (a) and Cube-4 (b) Basic Architectures.*

data slices that are parallel to a face of the volume dataset. This processing order appears similar to multipass resampling [DCH88] or object order compositing algorithms [Wes90]. However, in addition to the object-order data traversal we incorporate all advantages of fully-pipelined ray-casting into the algorithm.

The primary result of beam- or slice-parallel processing is the direct connection of the memory to the rendering pipelines. Cube-4, described in Chapter 8 and Chapter 9, has only local, fixed-bandwidth connections between processing units. This is conceptually illustrated in Figure 46b. A pixel-bus collects and aligns the pixel output from the compositors. Because only pixels are being globally transferred, Cube-4 is scalable to high dataset resolutions.

6.8 Summary

In this chapter, we first discussed the basic design of a high-performance memory system using a set of parallel, interleaved memory modules. We have shown that low-order interleaved memory is inadequate for the requirements of real-time volume rendering because consecutive accesses with stride not equal to one lead to access conflicts and worst case performance.

We presented two memory organizations that interleave blocks or vectors of voxels across a set of parallel memory modules. We argued that vector memory systems provide scalability of performance with increased memory parallelism. We then presented a vector memory system using a linear skewing of the address space that allows for conflict-free access to any beam of voxels. This skewed memory system is the basis for all Cube architectures.

We transformed the fully pipelined ray-casting algorithm of the previous chapter into an architecture with several parallel rendering pipelines. Finally, we discussed three different approaches to parallel ray-casting. Depending on the order of data access, we call these approaches ray-, beam-, or slice-parallel ray-casting. The following chapters introduce three architectures based on each of these parallel ray-casting algorithms.

Chapter 7

Cube-3: The Ray-Parallel Architecture

This chapter gives an overview of the ray-parallel dataflow of Cube-3. Section 7.1.1 introduces a new method to use the coherence between neighboring discrete rays during interpolation. Because of the discrete steps along the rays, a naive interpolation scheme would use a sub-optimal neighborhood around the sample location. We discuss an alternative solution, called sheared tri-linear interpolation, that prevents this problem. Section 7.1.2 shows how ABC gradient estimation is integrated into the ray-parallel dataflow of Cube-3.

In Section 7.2 we present results of experiments using the proposed sheared tri-linear interpolation and ABC gradient estimation methods. Section 7.2.1 presents a comparison between image-order ray-casting, template-based ray-casting with traditional tri-linear interpolation, and template-based ray-casting with sheared tri-linear interpolation. Section 7.2.2 contains a comparison of image errors between 6-, 10-, 12- and 26-neighborhood ABC gradient estimation methods.

Section 7.3 presents the Cube-3 architecture. Its main architectural features are the de-skewing network between memory and rendering pipelines, the distributed 2D skewed buffer to facilitate access to voxels along rays, and the folded binary tree for compositing. Finally, in Section 7.4, we discuss the key problems that inhibit a cost effective implementation of Cube-3 in hardware.

7.1 Cube-3 Ray-Parallel Dataflow

Figure 47 schematically shows the Cube-3 dataflow. The volume data is stored in a *cubic frame buffer* (CFB). All the discrete rays belonging to the same scanline of the base-plane image reside on the same plane inside the volume, called the *projection ray plane* (PRP). By fetching all voxels on a PRP and transforming them accordingly into a 2D buffer, all discrete rays can be aligned parallel to an axis, e.g., along the vertical direction. For parallel projections this transformation is simply a shear of beams to the left or right. For perspective projections each voxel belonging to a discrete ray has to be shifted by a different amount. We refer to this process as *de-fanning*, since diverging rays are stored adjacent to each other in the 2D buffer.

As soon as two PRPs are stored in two 2D buffers (referred to as the *above* and *current* buffers in Figure 47), a tri-linear interpolation is performed to generate sample points on continuous rays using the voxels of the cells touched by the discrete rays as input data. The two 2D buffers generate one interpolated plane of continuous rays. Three such planes, *above*, *below* and *current*, are needed for local ABC gradient approximations using neighboring rays. The samples of the rays are shaded and opacities are assigned using a user controllable transfer function. The shaded rays are composited into a final pixel color using a binary-tree implementation of compositing according

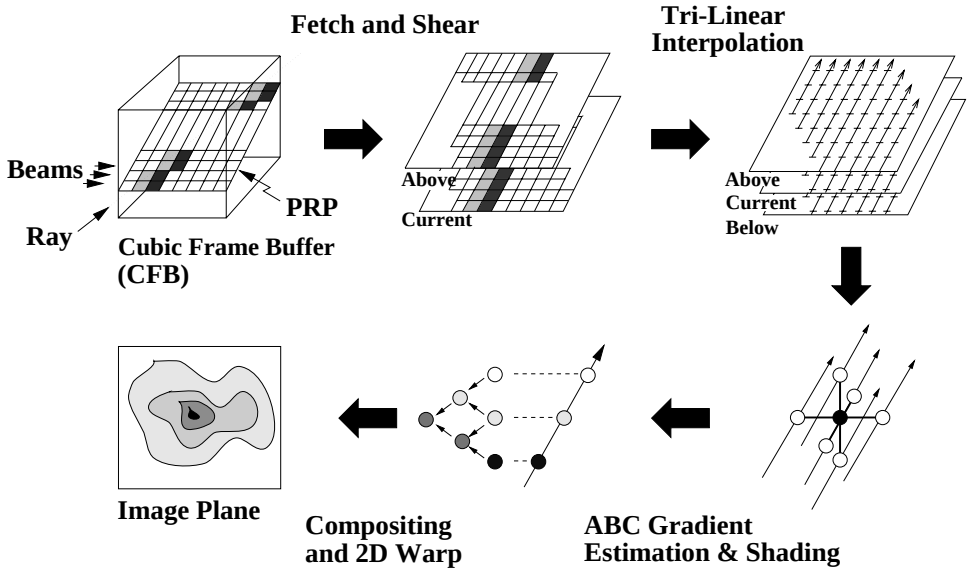

Figure 47: *Cube-3 Dataflow.*

to Equation 13 (see page 61 in Chapter 4). Other parallel projection schemes can also be employed, such as first or last opaque projection, maximum or minimum voxel value, and weighted summation.

7.1.1 Sheared Tri-Linear Interpolation

The key idea for interpolation in Cube-3 is to use neighboring discrete rays for tri-linear interpolation. Voxels of totally four discrete rays are used to interpolate one ray of continuous samples. Figure 48 illustrates this in 2D. The samples on one continuous ray have to be interpolated using tri-linear interpolation between samples of two discrete rays A (gray) and B (dark). Sample S1 can be correctly interpolated using voxels from A and B, since they form a rectangle, i.e., the rays do not make a discrete step to the left or right. Caching of voxels for the calculation of subsequent samples greatly reduces the data access and communication cost. However, there is a problem intrinsic to

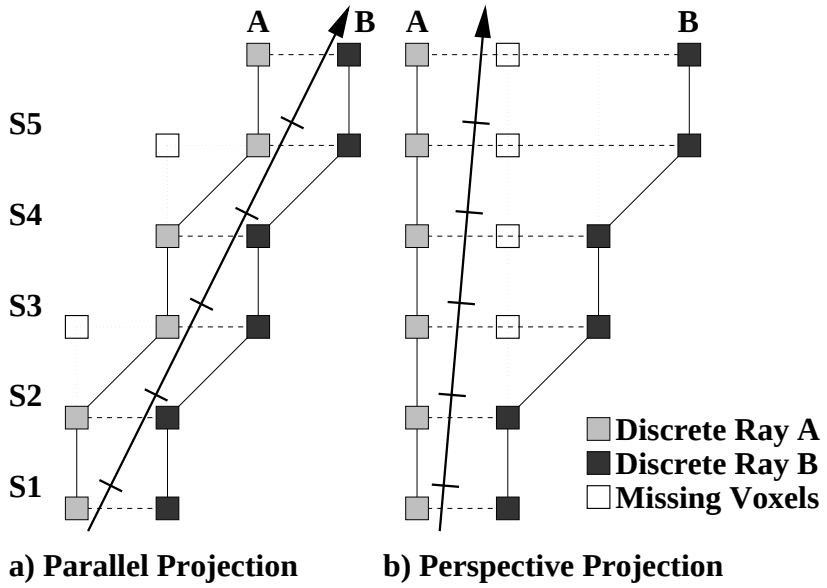

Figure 48: *Problems with Discrete Ray Interpolation.*

interpolation between discrete rays.

As soon as the discrete rays step to the left or right, as is the case for samples S2 and S4, the cell voxels form a parallelogram, and a straight forward interpolation produces the wrong sample values. The white square voxels in Figure 48a are needed to yield the correct result, but they reside on another discrete ray. This problem is exacerbated during perspective projections (Figure 48b). The discrete rays diverge, and the correct neighboring voxels are not being fetched from volume memory. For example, only two voxels of ray A contribute to the correct interpolation at sample point S3. In 3D, as many as six voxels may be missing in the immediate neighborhood of a sample point during perspective projections.

The solution is to perform a *sheared tri-linear interpolation.* Tri-linear interpolation is factored into four linear and one bi-linear interpolation. Instead

of specifying the sample location with respect to a corner voxel, each 3D coordinate along the ray consists of relative weights for linear interpolations along each axis in possibly sheared voxel neighborhoods. These weights can be precomputed and stored in the X/Y-templates discussed in Section 5.1. Figure 49 shows the necessary interpolation steps in 3D.

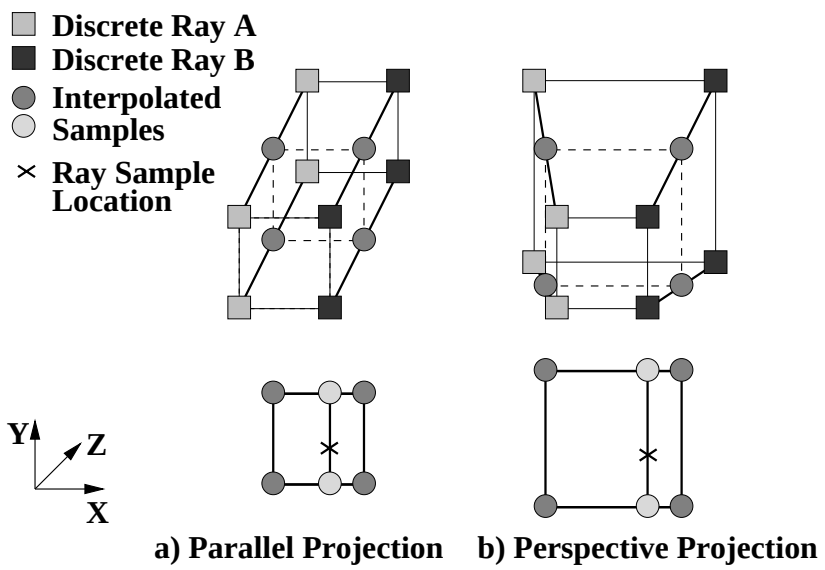

Figure 49: *Sheared Tri-Linear Interpolation.*

First, we perform four linear interpolations in direction of the major axis (the Z-axis in Figure 49) using eight voxels of four neighboring discrete rays inside the 2D buffers. These eight voxels are the vertices of an oblique parallelepiped for parallel projections (see Figure 49a) or of a frustum of a pyramid for perspective projections (see Figure 49b). Four voxels each reside on two separate planes one unit apart, which we call the front or the back plane. Therefore, only one weight factor has to be stored, corresponding to the distance between the front plane and the position of the ray sample point. The

resulting four interpolated values form a rectangle and can be bi-linearly interpolated to yield the final sample value. We split this bi-linear interpolation into two linear interpolations between the corner values and a final linear interpolation between the edge values. At the bottom of Figure 49 this is shown as two interpolations in X-direction followed by one interpolation in Y-direction.

For parallel projections, the eight voxels surrounding the sample point are arranged in an oblique parallelepiped, as shown in Figure 49a. For perspective projections, the surrounding voxels may form the frustum of a pyramid with parallel front and back planes as in Figure 49b. Due to the divergence of rays towards the back of the dataset, the volume spanned by this frustum increases, thereby reducing the precision of the tri-linear interpolation. However, we found that the distance between neighboring discrete rays at the end of the volume never exceeded two voxels for a 256^3 dataset with 60^0 field-of-view angles. Furthermore, in typical datasets the samples at the back of the volume have little influence on the final pixel color due to compositing along the ray.

As was shown in Figure 49, the sample points corresponding to the continuous rays have to be inside the polyhedron defined by the voxels on the four surrounding discrete rays. When constructing the discrete rays, all continuous rays start at integer positions of the base plane, i.e., they coincide with voxels of the first slice of the volume dataset. However, as Figure 50a shows, using these rays during ray-casting effectively reduces the tri-linear interpolation to a bi-linear interpolation, because all sample points along the ray fall onto the front planes of the parallelepipeds or pyramid frustum.

Using X and Y integer positions on the base-plane we can allow an offset from the base-plane in major direction as a degree of freedom and are able to perform sheared tri-linear interpolations (Figure 50b). For offsets in major direction that are too big, as shown in Figure 50c, some of the samples along

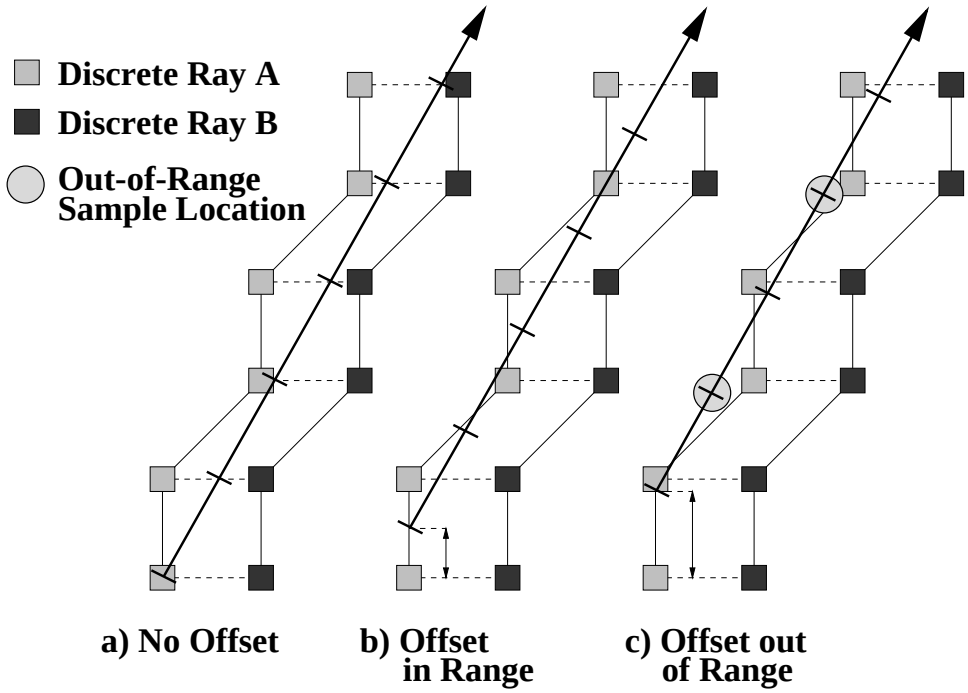

Figure 50: *Variable ray offsets in major direction and maximum offset calculation.*

the rays may fall outside the bounding box defined by the discrete rays.

In order to get an upper bound for admissible offsets, we have to understand how steps in non-major direction along discrete rays occur. Figure 51 shows the situation in 2D. The view vector is split into a dx component along the X-axis (dx and dy in 3D) and a unit vector in direction of the major axis (the Y-axis in Figure 51). Stepping in direction of the major axis, we add the viewing vector to the current sample position at S_n in order to get the new sample position at S_{n+1}.

Suppose that the addition of dx at point S_n leads to a step of the discrete rays in x direction. This step can only occur if S_n has a relative x offset with respect to the lower left corner voxel of more than $1 - dx$ for positive dx (or

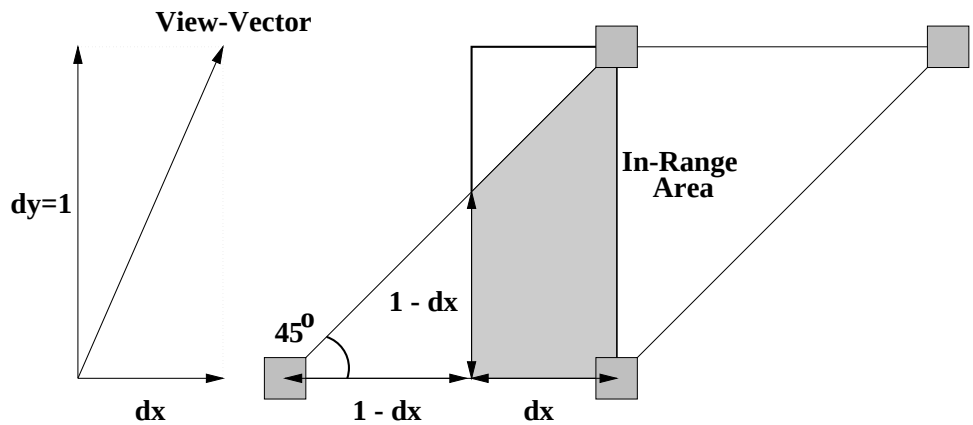

Figure 51: *Maximum Offset Calculation.*

less than $1 + dx$ for negative dx). In other words, sample S_n was inside the rectangle of size dx by 1 shown in Figure 51. However, only the shaded region of this rectangle contains sample positions inside the parallelepiped defined by the corner voxels. Taking the smallest side in major axis as the worst-case, this means that in-range samples have a maximal relative y offset of no more than $1 - dx$ for positive dx (no less than $1 + dx$ for negative dx).

Since we step with a unit vector in the direction of the major axis, all relative offsets along the ray are determined by the offsets of the first ray samples from the base-plane. The above argument easily extends to 3D, making the maximum allowed offset in direction of the major axis:

$$\begin{aligned} &min(1 - dx, 1 - dy), && dx, dy \geq 0 \\ &min(1 + dx, 1 - dy), && dx < 0, dy \geq 0 \\ &min(1 - dx, 1 + dy), && dx \geq 0, dy < 0 \\ &min(1 + dx, 1 + dy), && dx, dy < 0, \end{aligned} \tag{27}$$

where dx and dy are the components of the viewing vector in x and y direction, respectively. Notice that for 45^0 viewing angle dx and dy are 1, yielding an

offset of 0 and bi-linear interpolation as in Figure 50a. This fact will be of importance when discussing the results in Section 7.2.

In our implementation we cast a single ray from the origin of the image plane onto the base-plane using uniform distance between samples and choose the offset in major direction of the first sample after penetration of the base-plane. If necessary the offset is iteratively reduced until it satisfies the above condition. This leads to view dependent offsets in major direction and to varying resampling of the dataset.

Section 7.2 presents comparisons between image order ray-casting using a view independent sampling rate along the rays, tri-linear interpolation employing equation 1 using the correct voxels, and the proposed sheared tri-linear interpolation among discrete rays. The next section describes the method for correct orthogonal gradient estimation using 12 samples on neighboring rays.

7.1.2 ABC Gradient Estimation

In order to avoid temporal aliasing due to base-plane switching and to get gradient directions that are parallel to the main axes of the volume coordinate system, we use the 12-neighborhood gradient with two additional bi-linear interpolations to get samples that are perpendicular to the base-plane (see Section 5.2 in Chapter 5).

In the case of perspective projections, the front of each PRP is uniformly sampled with n rays one unit apart. As the rays diverge towards the back of the volume, the distance between rays increases, and the gradient estimation becomes less accurate. However, because of the usually small distance between rays and due to the averaging nature of shading, classification and compositing, these effects do not influence image quality for typical datasets.

With the gradient estimation and light vector directions, the sample intensity can be generated using a variety of shading methods (e.g., using lookup tables [LL94]). Opacity values for compositing are generated using a transfer function represented as a 2D lookup table indexed by sample density and gradient magnitude [Lev88].

7.2 Cube-3 Analysis

We implemented the different interpolation and gradient estimation methods in software and conducted several experiments. The first program, `VolRen` implements traditional image order volume rendering. Rays are cast from the image plane into the volume and sampled at uniform steps. The tri-linear interpolation is performed according to Equation 1 using the correct 8-neighborhood around sample points. The gradient is estimated using central differences of tri-linear interpolated values in a 6-neighborhood around each sample.

The second program, `True3D`, uses our real-time discrete ray-casting method, but instead of performing sheared tri-linear interpolation it fetches the exact 8-neighborhood around each sample point. The last program, `Sheared3D`, implements the same algorithm but with sheared tri-linear interpolation. Both `True3D` and `Sheared3D` can use any of the 6-, 26-, 10- or 12-neighborhood gradient methods for comparison purposes. For the implementation of these algorithms we used the VolVis volume visualization system, developed at the State University of New York at Stony Brook [ASK92, AHH+94].

7.2.1 Tri-Linear Interpolation Comparison

First we compare images resulting from `Sheared3D` to results obtained from `VolRen` and `True3D`. The normal approximation method used for `Sheared3D` and `True3D` was the 12-neighborhood gradient estimation.

The dataset, a CT study of a cadaver head of size $256 \times 256 \times 225$ voxels at 8-bit per voxel, was taken on a General Electric CT Scanner and provided courtesy of North Carolina Memorial Hospital. All programs use the same shading model. The opacity transfer function maps voxel values below 80 to $\alpha = 0$, has a linear ramp for α from 0 to 0.75 for values between 80 and 100, and assigns $\alpha = 0.75$ to values above 100. We chose this particular transfer function to classify bone in the dataset as opaque in order to try to maximize the display of aliasing effects on the forehead of the CT skull.

For the experiments we rotated the dataset by 70^0 around the horizontal axis with respect to the world coordinate system. During animations we rotated it around a vertical axis between 0^0 and 90^0 in steps of 5^0. As error measure between the resulting images we use the average Euclidean distance of RGB values between corresponding pixels. Figure 52 shows the dataset rotated by 45^0 around the vertical axis. The left image was generated using `Sheared3D` and the image on the right is the difference image, mapped to gray-scale, comparing the corresponding `Sheared3D` and `VolRen` images for this rotation angle.

Figure 53 shows the relative Euclidean error in percentage between images from `Sheared3D` and `VolRen` and between `Sheared3D` and `True3D`, respectively. The comparison with `VolRen` (top curve) shows how the error raises towards 45^0 rotation angle and reaches a minimum at 0^0 and 90^0. The peak at 45^0 is due to the different sampling distance along the ray, which is by $\sqrt{3}$ bigger for discrete line stepping (see Chapter 5). Furthermore, due to the offset

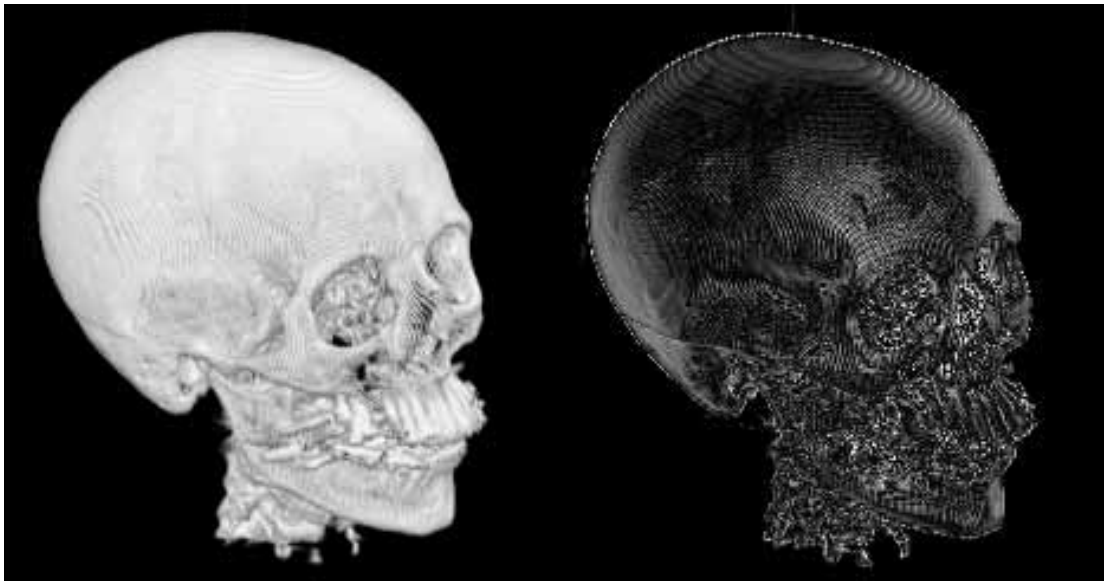

Figure 52: *Dataset rendered with sheared tri-linear interpolation (left) and the difference image to traditional volume rendering (right) for* 45° *rotation angle. This is the worst case for sheared tri-linear interpolation.*

considerations explained in Section 7.1.1, our algorithm performs only bi-linear interpolation as opposed to the the tri-linear interpolation in `VolRen`.

The comparison to `True3D` shows zero error for 45^0 because both algorithms perform bi-linear interpolation and use the same gradient estimation technique. The relative error in percent compared to `VolRen` stays below 1.3%, and compared to `True3D` it stays below 0.3%.

7.2.2 ABC Gradient Estimation Comparison

For the comparison of the different ABC gradient estimation techniques, we use a voxelized model of a sphere as dataset. The sphere is scan-converted into a $32 \times 32 \times 32$ dataset using the volume sampling method described in [WK93, WK94, Wan95]. The surface intersection points are obtained by

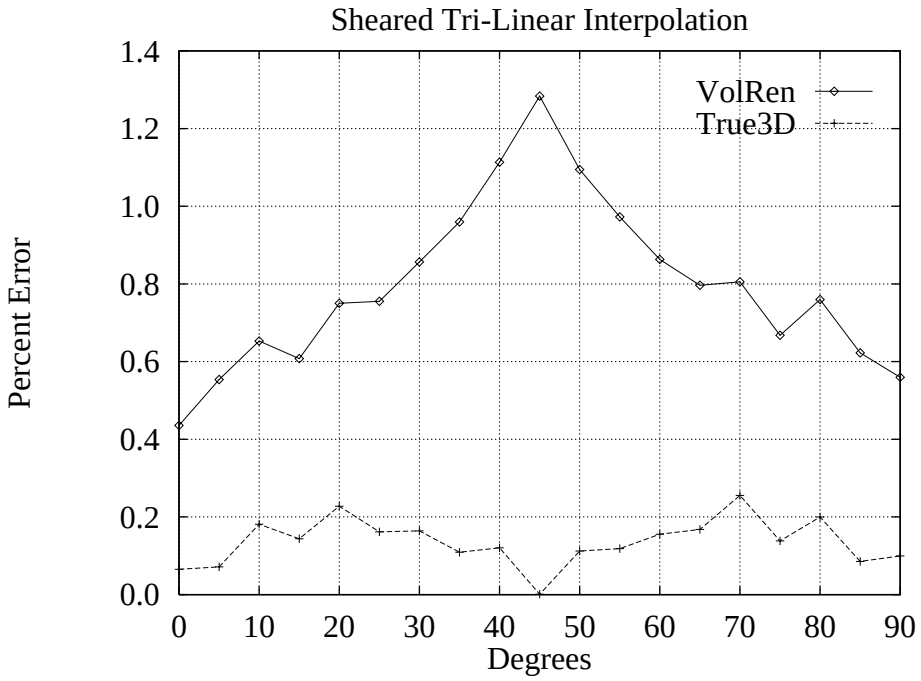

Figure 53: *Sheared Interpolation Error in Percent.*

thresholding, i.e., as soon as a certain sample value along each ray is exceeded we calculate the gradient at that point. Each gradient is compared to the true geometric surface normal. As error measure we use the magnitude of angular difference between two vectors. All differences are accumulated and averaged over all surface intersection points.

Figure 54 shows the results of rotating the sphere around a vertical axis between 0^0 and 90^0 in steps of 5^0. The top two curves compare the analytic normal with the 26- and the 6-neighborhood gradient, respectively. The error increases towards 45^0 rotation angle due to the non-orthogonality of the gradient directions which reaches a maximum at 45^0. Although the 26-gradient shows a little higher error magnitude, the difference between these two methods is not significant.

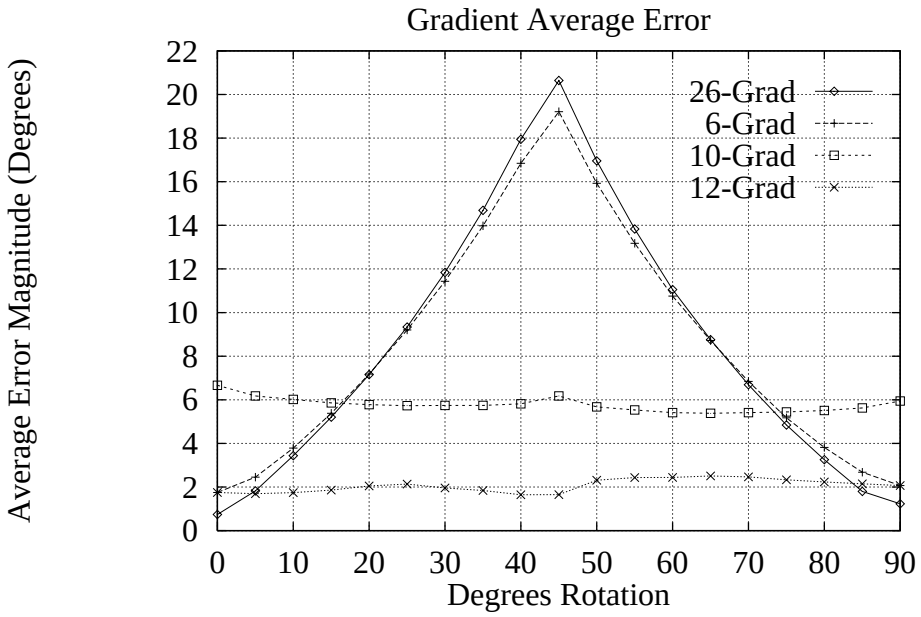

Figure 54: *Average Error Magnitude for ABC Gradient Estimations Compared to the Analytic Normal.*

The second curve from the bottom in Figure 54 shows the comparison of the analytic normal with the 10-neighborhood gradient estimation. The error magnitude is significantly smaller than for the other gradient methods. The error also increases towards 45^0 rotation angle. This is due to the different distances between samples that are used for the gradient calculations in the three orthogonal directions. The bottom curve shows the error of the 12-neighborhood gradient when compared to the analytic normal. The curve remains constant for all viewing directions, and the error magnitude remains around 3^0 which is substantially lower than for the other ABC gradient estimation techniques.

Figure 55 shows how the error propagates around the sphere for rotation

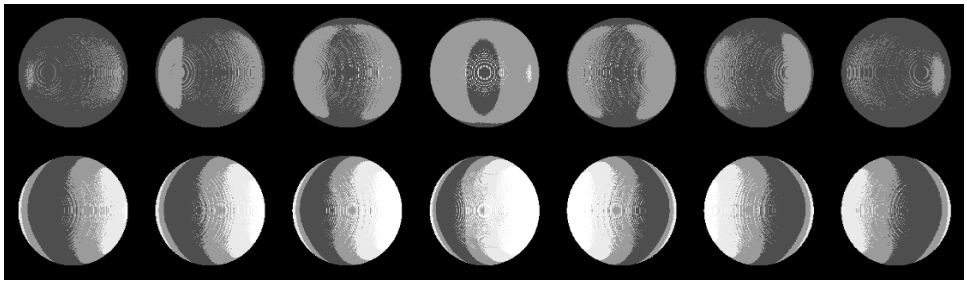

Figure 55: *Error magnitude of comparing surface normals of 10- (Top) and 26-neighborhood gradients (Bottom) to the true analytic normal of the voxelized sphere. Notice the jump of regions of high error for the 26-neighborhood gradient between* 45° *and* 50° *rotation angle. Dark:* $0^0 \leq |e| < 8.5^0$*, Medium:* $8.5^0 \leq |e| < 20^0$*, Light:* $20^0 \leq |e| < 31.5^0$*, White:* $|e| \geq 31.5^0$*. Rotation angles (left to right):* $30^0, 35^0, 40^0, 45^0, 50^0, 55^0, 60^0$*.*

angles from 30^0 to 60^0 in steps of 5^0. Dark shaded regions indicate regions of low error magnitude, light shaded regions indicate higher error magnitudes. The top row shows the 10-neighborhood gradient method with a fairly regular error transition from left to right during a switch of base-planes at 45^0 (center sphere). The bottom row, depicting the 26-neighborhood gradient method, shows a generally larger error magnitude. Additionally, the region of largest error jumps from the right side of the sphere to the left during the switch of base-planes. This jump leads to noticeable changes in image intensity during object rotation, an effect that we described as temporal aliasing in Section 5.2. Similar pictures for the 12-neighborhood gradient show no change in color due to the viewing angle independence and the low error magnitude.

7.3 Cube-3 Architecture

The Cube-3 architecture is highly-parallel and pipelined [PKC94]. Figure 56 shows a block diagram of the overall system. The CFB is a 3D memory

organized in n dual-access memory modules, each storing n^2 voxels using the special 3D skewed organization described in Section 6.4. PRPs are fetched as a sequence of voxel beams and stored in consecutive *2D skewed buffers* (2DSB). A high-bandwidth interconnection network, the *fast bus*, allows the alignment of the discrete rays on the PRP parallel to a main axis in the 2DSB modules.

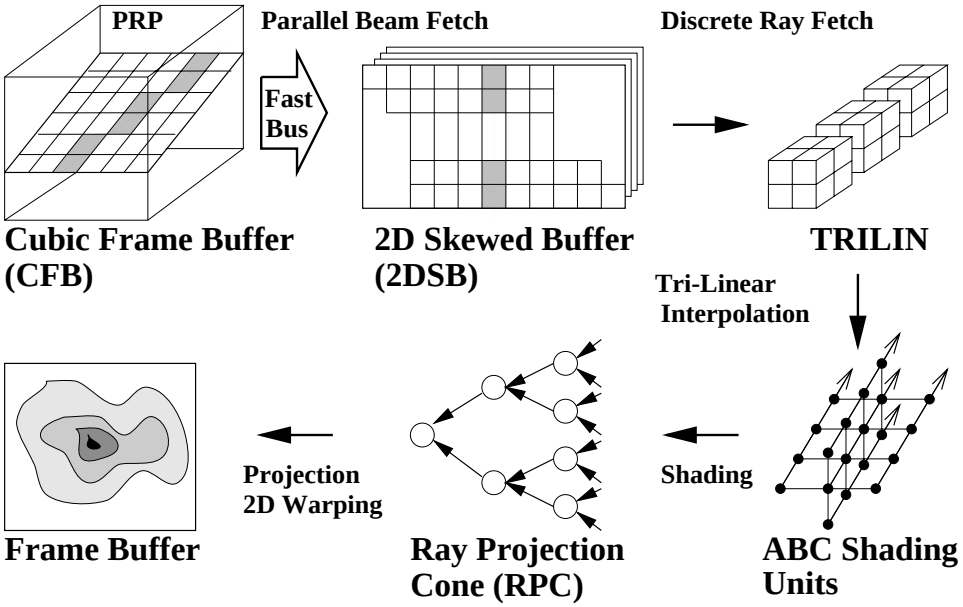

Figure 56: *Cube-3 System Overview.*

Three 2DSBs are used in a pipelined fashion to support sheared tri-linear interpolation. Aligned discrete rays from 2DSBs are fetched conflict-free and placed into special purpose Tri-Linear Interpolation (TRILIN) units. The resulting continuous projection rays are placed onto ABC Shading Units, where the gradients are estimated and each ray sample is converted into both an intensity and an associated opacity value according to lighting and data segmentation parameters. These intensity/opacity ray samples are fed into the leaves of a Ray Projection Cone (RPC). The RPC is a folded binary tree that generates in parallel and in a pipelined fashion the final pixel value using a

variety of projection schemes on the cone nodes. The resulting base-plane pixel is transmitted to the host where it is 2D transformed (warped) onto the viewing plane. The result is stored in the 2D frame-buffer.

The parallel conflict-free memory architecture of Cube-3 reduces the memory access bottleneck from $O(n^3)$ per projection to $O(n^2)$ and allows for very high data throughput. For a dataset size of 512^3 16-bit voxels we estimate a performance of up to 30 frames per second. Such a system would require 8 boards and a custom fabricated backplane.

7.4 Problems with Cube-3

There are several implementation issues of Cube-3 that make a practical realization difficult:

Fast Bus: The global Fast Bus must transfer all of the dataset voxels between memory and processing units for every frame. The bandwidth requirements for this bus scale with $O\ (n^3)$, n being the dataset resolution in one dimension. This is the biggest bottleneck of the Cube-3 architecture.

Size: A 512^3 Cube-3 implementation requires 512 DRAM chips for the CFB, $2n$ or 1024 SRAMs for the 2D skewed buffer (2DSB), 2×512 interface chips to the Fast Bus, 512 combined interpolation/shading units (LEAF), a total of 1023 Ray Projection Cone (RPC) nodes, and several addressing, control and multiplexor units. Using a VLSI implementation for the LEAF and RPC nodes this corresponds to 16 boards with about 350 chips per board.

Datapath Complexity: The datapaths of Cube-3 are very complex. The Fast Bus requires 512 wires running at 125 MHz and spanning 8 boards.

The RPC nodes are connected in a wrap-around binary tree fashion leading to problems at chip and board boundaries.

Control Complexity: A major problem is the control of the Fast Bus operation, namely how to distribute the control signals from the host to the numerous Fast Bus multiplexors. The switching operations inside the RPC change with every ray and it is unclear on how to centralize this control.

Scalability: Due to the binary tree nature of the RPC the scalability of Cube-3 is limited to powers of 2. A 1024^3 machine requires more than 32 boards due to the increased RPC complexity. Furthermore, with current technology it is inconceivable to build a Fast Bus with 1024 wires spanning those 32 boards.

7.5 Summary

This chapter presented the Cube-3 ray-parallel architecture. Using templates and shearing or de-fanning of beams, 2D projection ray planes of voxels are fetched from the volume dataset. Sheared tri-linear interpolation among voxels of neighboring rays produces interpolated samples which are used during ABC gradient estimation. A folded binary tree of compositors is used to generate the base-plane pixel.

Using software simulations we compared the proposed methods to traditional image order ray-casting. The error of using sheared tri-linear interpolation instead of performing image order ray-casting is below 1.3% relative difference in Euclidean distance of the resulting image pixels. We showed that use of 12-neighborhood gradients instead of 6-, 10-, or 26-neighborhood

gradients minimizes the average error compared to analytically computed normals. The 12-neighborhood gradient prevents temporal aliasing that arises from switching base-planes during object rotations.

We presented a list of problems related to the hardware complexity of Cube-3. It has been our goal to address these issues, mainly to simplify the datapath and control logic, decrease the machine size, and enhance the scalability. The Cube-4 architecture, presented in the remainder of this thesis, is the result of this process. Cube-4 adequately solves all of the problems of Cube-3 and satisfies all of our project goals.

Chapter 8

Cube-4: The Beam-Parallel Architecture

The main focus of this chapter is a description of the Cube-4 beam-parallel dataflow that leads to local, static, and fixed bandwidth memory-processor and inter-processor connections. The dataflow description in this chapter is not mathematically tight and rigorous, but rather descriptive and appealing to the intuition of the reader. Throughout the development of this project we established our techniques by practical example and experimentation. The explanation of the Cube-4 beam-parallel dataflow is no exception. However, we give a formal description of the Cube-4 slice-parallel dataflow in the next chapter.

To simplify the explanation, we first focus on a hypothetical example. We assume orthonormal projection and that we can fetch un-skewed beams from the memory without access conflicts. We call the resulting dataflow the un-skewed beam-parallel dataflow, and it enhances the general understanding of beam-parallel processing.

In Section 8.2 we present the actual beam-parallel dataflow using our skewed memory organization. The main complication arises from the skewing difference between successive beams that has to be corrected at every pipeline stage. However, in beam-parallel processing we do not attempt to deskew the data globally, but rather locally for small processing neighborhoods. The successive scanlines of base-plane pixels are still output in skewed order. Deskewing in 2D amounts to a simple address re-mapping inside the frame-buffer.

Section 8.3 presents the Cube-4 beam-parallel architecture. The beam-parallel dataflow results in localized, fixed connections between memory, tri-linear interpolation, shading and compositing units. Instead of global voxel communication over a fast bus, Cube-4 uses a simple, easy to implement pixel bus with only moderate bandwidth requirements.

In Section 8.4 we show that the performance of Cube-4 is mainly limited by the access speed of DRAM memories. The dataflow is simple and regular. It allows parallel processing and pipelining to be fully exploited at each stage.

8.1 Un-skewed Beam-Parallel Dataflow

Suppose we have a CFB organization that allows for conflict-free parallel access to any beam of voxels in X, Y and Z direction. Since we project each projection ray plane (PRP) in base-plane scanline order, we can fetch each PRP using this parallel beam access mechanism by following the steps stored in the Y template of the discrete rays. Figure 57 shows the dataflow for the simple case of a orthogonal parallel projection in +Z. Notice the coordinate system at the bottom of the figure. Because of the orthogonal projection, the PRPs are all horizontal planes and there is no step of the discrete rays in X or Y direction.

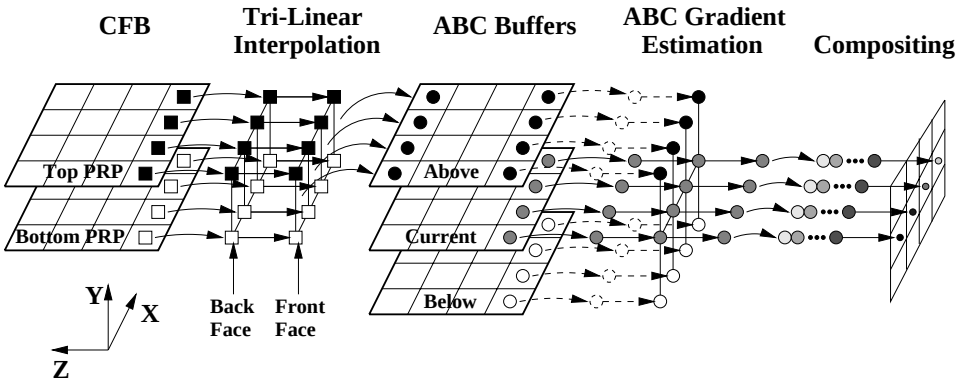

Figure 57: *Un-skewed Dataflow Example.*

In order to perform a tri-linear interpolation we need to fetch the eight voxels that form a cell around each resampling location. These voxels come from two adjacent PRPs which we call the top and the bottom PRP. In Figure 57 the top PRP contains the black voxels and the bottom PRP the white voxels. In order to be able to fetch beams from both PRPs during the same timestep we need to buffer one of them. This can trivially be accomplished by using a first-in-first-out (FIFO) buffer with the same depth as the PRP (i.e., depth n) connected to the memory modules of the currently fetched PRP. We discuss these and other hardware aspects of the design in Section 8.3.

Both top and bottom PRPs are fetched by accessing their beams from front to back, i.e., starting at the base-plane and proceeding in +Z direction into the CFB. In Figure 57 this corresponds to fetching the beams from the right side of the PRPs towards the left. The arrows in Figure 57 correspond to the data movement that takes place at each timestep. The arrows going from the right side of the PRPs correspond to fetching the beams from the CFB into the tri-linear interpolation units. We call the four voxels that are fetched directly from memory the back face of the interpolation cell.

The voxels from all the back faces are moved to the front faces inside

the tri-linear interpolation units at every cycle. Notice that neighboring cells share four voxels that have the same X coordinates. The data movement from memory and inside the tri-linear interpolation units can be implemented with fixed and local datapath connections. The tri-linear interpolation units are internally pipelined so that n of them produce a set of n interpolated samples per timestep. This can be easily achieved by factoring one tri-linear into seven linear interpolations.

In Figure 57 the currently interpolated samples are shown as black dots and are sent with straight datapath connections to the ABC buffers of the gradient estimation units. The middle plane of the three ABC buffers is the current buffer (gray dots), whereas the lowest plane is the below buffer (white dots). The dataflow between the buffers follows the same pattern as that of the top and bottom PRPs, i.e., the ABC buffers can easily be implemented using a similar FIFO buffering. Since the above samples come directly from the tri-linear interpolation units we do not need to store them (contrary to the drawing in Figure 57), but we need only two FIFO buffers for the current and below planes.

To estimate the gradient around a given sample location of the current buffer, we use two samples along the direction of the ray from the current buffer, two samples inside the current plane perpendicular to the direction of the ray from neighboring gradient estimation units, and two samples from the above and below buffers. Notice that the above and below samples around the current sample location have an additional delay of one timestep over the samples of the current plane. Taking simple differences between these samples yields a 6-neighborhood gray-level gradient estimation. Using a pipelined implementation of the required central difference calculations and n ABC gradient units in parallel we can estimate n gradients around the n sample locations

of the current buffer at every timestep.

After the gradient estimation follows the shading of the samples, not shown in Figure 57. With the gradient and the light vector description we produce n shaded samples of n continuous rays per timestep. This assumes that we are able to perfectly pipeline the shading calculations, which may be non-trivial for higher order, e.g., Phong, shading models. However, our simulations show very satisfactory results using a simple diffuse shading model. Other researchers have proposed fully pipelined Phong shading architectures [Kni93].

Once we are presented with n shaded samples at each timestep, we have to generate the final base-plane pixel values. In our simple example we get n samples contributing to n pixels of a base-plane scanline at every timestep. After n timesteps we have generated all pixels that fall on the current base-plane scanline. In order to generate these final pixel values we perform alpha blending or compositing, in this case in front-to-back order, using an opacity lookup table (transfer function) and the shaded sample value. In our example all samples flow straight into the compositing units. We can use simple accumulating adders to perform the compositing, yielding n final base-plane pixel values every n timesteps. Operations like first/last opaque, maximum or average projection can also be implemented in the compositors.

As a last step we have to transmit the base-plane pixels to the host where the transformation and resampling onto the view-plane is performed. Since in our example we get n base-plane pixels every n timesteps we can easily transmit one pixel per timestep to the host where they are buffered before the final 2D warp. In general this is true if the base-plane has extent $n \times n$. However, in the worst case the base-plane can have extent $2n \times 2n$ for a viewing angle parallel to a main diagonal of the CFB. That means we have to communicate $4n$ pixels during n timesteps to the host. This can either

be accomplished by operating it at four times the speed with which Cube-4 operates using a 16-bit bus or by making the datapaths four times as wide. Therefore, the pixel bus becomes 64-bit wide, a format supported by several standard bus protocols.

In this example we assumed that we can fetch beams of voxels in parallel and conflict-free out of the CFB. This can not be accomplished with regular memory addressing. It is easy to see that a straight forward memory scheme would lead to access conflicts for at least two of the three major directions of the CFB. Notice, however, that it may be conceivable to implement such a memory without resorting to the linear skewing that we use in Cube-4. The next section describes the actual Cube-4 algorithm, that uses a simple linear skewing mechanism in order to guarantee conflict-free access to any beam inside the CFB, independent of the viewing direction. Although the datapath connections do not follow the same fixed pattern, they remain local and short as in this un-skewed example.

8.2 Skewed Beam-Parallel Dataflow

The beam-parallel ray-casting algorithm discussed in Section 6.7 requires beams of voxels to be fetched from the CFB memory. Since we project PRPs in base-plane scanline order, we can fetch each PRP in n cycles using the parallel beam access mechanism provided by the skewed memory organization. Beams are fetched according to the steps stored in the Y template of the discrete rays.

The overall conceptional dataflow is illustrated in Figure 58. The data connections are symbolized by arrows and indicate the flow of voxels and samples. A very important aspect of the algorithm is that these data connections are fixed and remain unchanged during subsequent cycles.

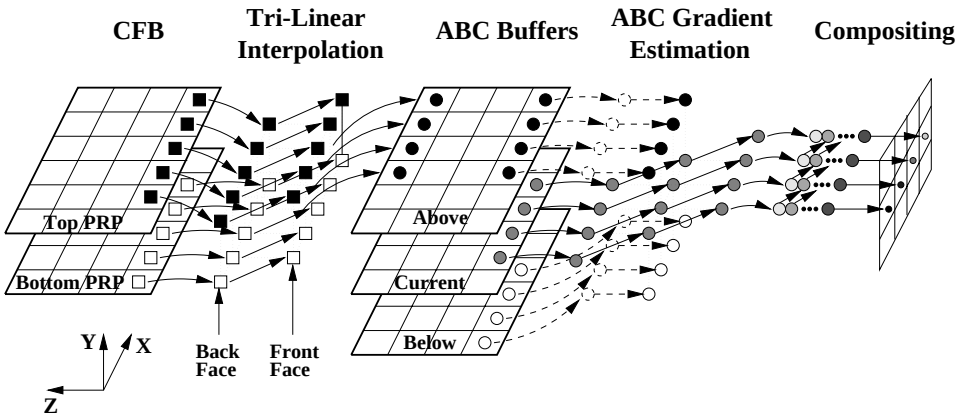

Figure 58: *Skewed Beam-Parallel Dataflow.*

Figure 59 shows two planes of voxels inside the skewed CFB memory. The numbers in the planes indicate the x coordinate of a voxel that are read from module k given certain y and z coordinates. Notice that voxels from neighboring beams have a different order of x coordinates (the so-called skewing difference) as can be seen for beams $B1$ and $B2$ or $B1$ and $T1$. This has a major impact on the data connections of the Cube-4 architecture as illustrated in Figure 58.

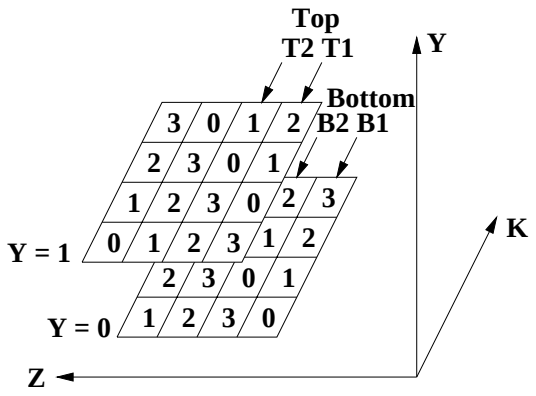

Figure 59: *3D skewing. Numbers indicate x coordinate of voxels stored in module number $k = (x + y + z)\text{mod}4$.*

Before going into detail, we give a brief overview of the dataflow. It starts at the left of Figure 58, where two beams are fetched from two consecutive PRPs (called the Bottom and Top PRPs) and sent to TRILIN units for tri-linear interpolation. The TRILIN units generate a beam of tri-linearly interpolated samples that is stored in the above buffer of the ABC shading units. Together with the current and below buffers, which contain previously interpolated samples, these buffers output three beams of samples that are fetched by the ABC gradient estimation units. These units estimate the gradient around each resample location, perform shading and classification on a beam of samples, and send the results to the compositing units for composition. The compositing units work on samples of n rays in parallel and perform one composition step for every ray per clock-cycle. The following sections look at each of these steps in more detail.

8.2.1 Tri-Linear Interpolation

Every clock cycle, two beams of voxels are fetched by the TRILIN units from the Bottom and Top PRPs. Figure 59 shows that these two beams (e.g., B1 and T1) have a skewing difference of one. In other words, the x coordinates of voxels between any two PRPs differ by one in any memory module k. This skewing difference has to be corrected such that the TRILIN units at position k get the voxels with the same x coordinate. We achieve this by having straight data connections and connections with a shift of one in negative k direction between the Top and Bottom PRPs and the TRILIN units, respectively (see Figure 58). To guarantee that all TRILIN units have access to the correct 8-voxel neighborhood around a resampling location, the voxels of the beams make a shift of one position in positive k direction when they move from the so called back face to the front face inside each TRILIN unit. A consequence of

the fixed connections between the CFB and TRILIN units is that the resulting interpolated beams of sample values are still skewed with a similar skewing pattern as in Figure 59.

8.2.2 ABC Gradient Estimation

To estimate the gradient around a given sample location, three pairs of voxels in the directions of the CFB are required around the sample location. The actual differences in all directions can then be computed by taking central differences. For the gradient difference in major direction, it is necessary to have access to two beams from the current buffer that are two clock cycles apart. During the current cycle we can access three beams from the above, current, and below buffers for the other two gradient differences.

Figure 58 shows that three beams are output simultaneously by the ABC buffers. To be able to have fixed data connections between the ABC buffers and the ABC gradient estimation units, the beams from the current buffer have to be shifted one position in positive k direction during forwarding inside the ABC units. This shift can be explained using the same arguments as for the shifting inside the TRILIN units.

The beams from the above and below buffer have to be delayed by one cycle in order to be aligned with the middle beam from the current buffer. This extra delay takes care of the skewing difference between the beams that are output by the above and current buffers (compare beams $T1$ and $B2$ in Figure 59), whereas the skewing difference between the beams from the below and current buffers is increased to two (compare beams $T2$ and $B1$ in Figure 59). Consequently, the data connections from the below buffer to the ABC gradient estimation units make a shift of two positions in positive k direction to eliminate this skewing difference.

With the correct 12-neighborhood available in the ABC units, we are able to compute the gradient differences and produce a complete beam of shaded voxels every clock-cycle. This assumes that we can perfectly pipeline the shading calculations, which may be non-trivial for higher order, e.g., Phong, shading models. However, our simulations show very satisfactory results using a simple diffuse shading model. Other researchers have proposed fully pipelined Phong shading architectures [Kni93]. As part of our proposed research, we would like to investigate the use of deferred shading and texture mapping techniques in the context of beam- and slice-parallel architectures [DWS88, FPE+89, MEP92]. The sample and gradient values are available inside the ABC units and can be used for opacity lookup for subsequent compositing.

8.2.3 Compositing

The last stage of the Cube-4 architecture performs the composition operations of samples along the rays. Because the beams of shaded samples that are output by the ABC units are still skewed, the composition of a single orthogonal ray can not be done in a single compositing unit. Each sample point of the ray has to be composited in a different compositor. Looking at the skewing difference between two consecutive beams we see that the temporary compositing results of an orthogonal ray have to be moved to the next compositing unit in positive k direction for each consecutive composition step.

So far the dataflow example has focused on an orthogonal projection. In the following sections we explain the consequences of non-orthogonal parallel and perspective projections.

8.2.4 Steps in X and Y

Steps of the viewing rays in X and Y direction influence the skewing difference between subsequent beams. Discrete steps in X only affect the compositing units where the actual rays are composited, whereas steps in Y influence the interpolation and gradient estimation stages. For example, for positive steps in Y the skewing difference between consecutive beams becomes two (compare beams $B1$ and $T2$ in Figure 59). In order to keep the simple fixed data connections between the CFB and the TRILIN units, the voxels from the back face have to make a different shift as they move to the front face inside the TRILIN units. For negative steps in Y they are forwarded straight and for positive steps in Y they are shifted by two positions in positive k direction.

Apart from different internal forwarding of voxels inside the TRILIN units, a positive or negative step in Y also results in a sheared 8-voxel neighborhood for tri-linear interpolation. This is because the y coordinates of voxels in the back face of the TRILINs are either one unit in Y direction higher or lower than their counterparts in the front face. This can be corrected by using sheared tri-linear interpolation (see Section 7.1.1), but only in Y direction. The sheared tri-linear interpolation is a disadvantage, which will be corrected in the slice-parallel architecture presented in the next chapter.

The skewing pattern inside the ABC buffers is similar to that of the PRPs. It affects the forwarding of samples of the current plane inside the ABC units the same way as it influenced the forwarding inside the TRILIN units. Figure 60 shows that the component of the gradient in the direction of the ray is slanted with respect to the other two components because of the steps in Y of the interpolated samples. Following the discussion in Section 5.2 (Chapter 5), we need to perform two bi-linear interpolations in Y direction to estimate the 12-neighborhood gradient. The access to the additional samples

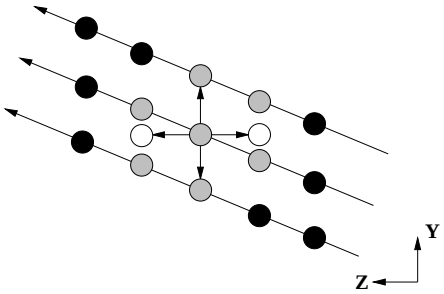

Figure 60: *12-neighborhood Gradient Correction.*

from neighboring rays requires additional datapath connections between the shading units.

8.2.5 Projection in Reverse Major Direction

Stepping in reverse major direction influences the forwarding of voxels inside the processing units. Instead of forwarding in positive k direction for none or positive steps in Y it becomes necessary to forward in negative k direction for none or negative steps in Y. This results in additional and bi-directional data connections between processing units. However, this can be avoided by performing back-to-front compositing inside the compositing units rather than front-to-back compositing. This complicates the hardware implementation of the compositing units, but the datapaths remain unaffected.

8.2.6 Perspective Projection

The divergence of perspective rays in X direction can be corrected during compositing of the rays. The way in which the compositing results are forwarded between the different units is different for each ray. Some compositing units may perform unnecessary work because not all of the samples between neighboring perspective rays are used. The divergence of rays in Y direction has to

be dealt with more carefully. Because PRPs may diverge by more than one unit in Y direction, the skewing difference between them becomes too big to be corrected by the fixed datapaths of Cube-4. To perform correct interpolation and gradient estimation during perspective projections it is necessary to always fetch one PRP that is one unit lower followed by one PRP that is one unit higher in Y direction. This is shown in Figure 61a.

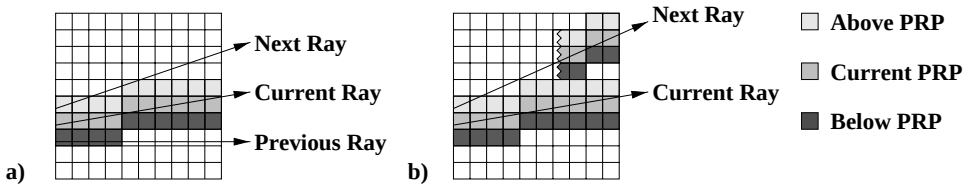

Figure 61: *a) Divergence of rays during perspective projection. b) Diverging perspective rays with unused beam inbetween.*

The shaded squares indicate the beams of three PRPs that need to be fetched in order to calculate the current ray. Only some beams can be shared between the current, next, or previous rays. This may lead to a performance drop by a worst-case factor of three per base-plane projection. However, this is seldom the case because it is not unusual for perspective PRPs to diverge by more than one beam towards the end of the dataset (see Figure 61b). Beams can be skipped between subsequent PRPs, which is not an uncommon situation for 512^3 or higher resolution datasets. An advantage over Cube-3 ray-parallel perspective projections is the fact none of the samples diverge for interpolation and gradient calculations.

8.3 The Cube-4 Beam-Parallel Architecture

Whereas the previous sections described the Cube-4 dataflow from a conceptual perspective, this section describes the Cube-4 architecture. Figure 62

shows part of a Cube-4 implementation with n CFB memory modules, TRILIN interpolation units, Shaders and Compositors, all connected through fixed-bandwidth datapaths. A global pixel bus connects the compositing units and transfers pixel data to an external host computer for post-processing and display. The shaded units in Figure 62 belong to a so called vertical module. Thicker lines in the figure indicate all datapath connections corresponding to module k.

Since the TRILIN units need access to two beams from consecutive PRPs, we include a FIFO buffer inside each CFB module. The beam from the Bottom PRP is output by the FIFO buffers whereas the beam from the Top PRP comes directly from CFB memory. The ABC Shading buffers are realized by using the output of the TRILIN units as the beam from the above plane and the output of two FIFO buffers as the beams of the current and below plane. The figure shows all datapaths that are needed for the beam-parallel dataflow described in Section 8.2, including perspective projections.

Information for the correct operation of Cube-4 can be forwarded in the datapath in form of additional control bits. They are summarized in Table 5. The wrap in X or Y bit fields indicate the exit and restart of a discrete ray

Control Signal	**Value**	**Bits**
Wrap in X	0,1	1
Wrap in Y	0,1	1
End of Ray	0,1	1
Step in X	-1,0,1	2
Step in Y	-1,0,1	2

Table 5: *Cube-4 Control Information.*

for non-orthogonal viewing angles. In such a situation, the compositing units

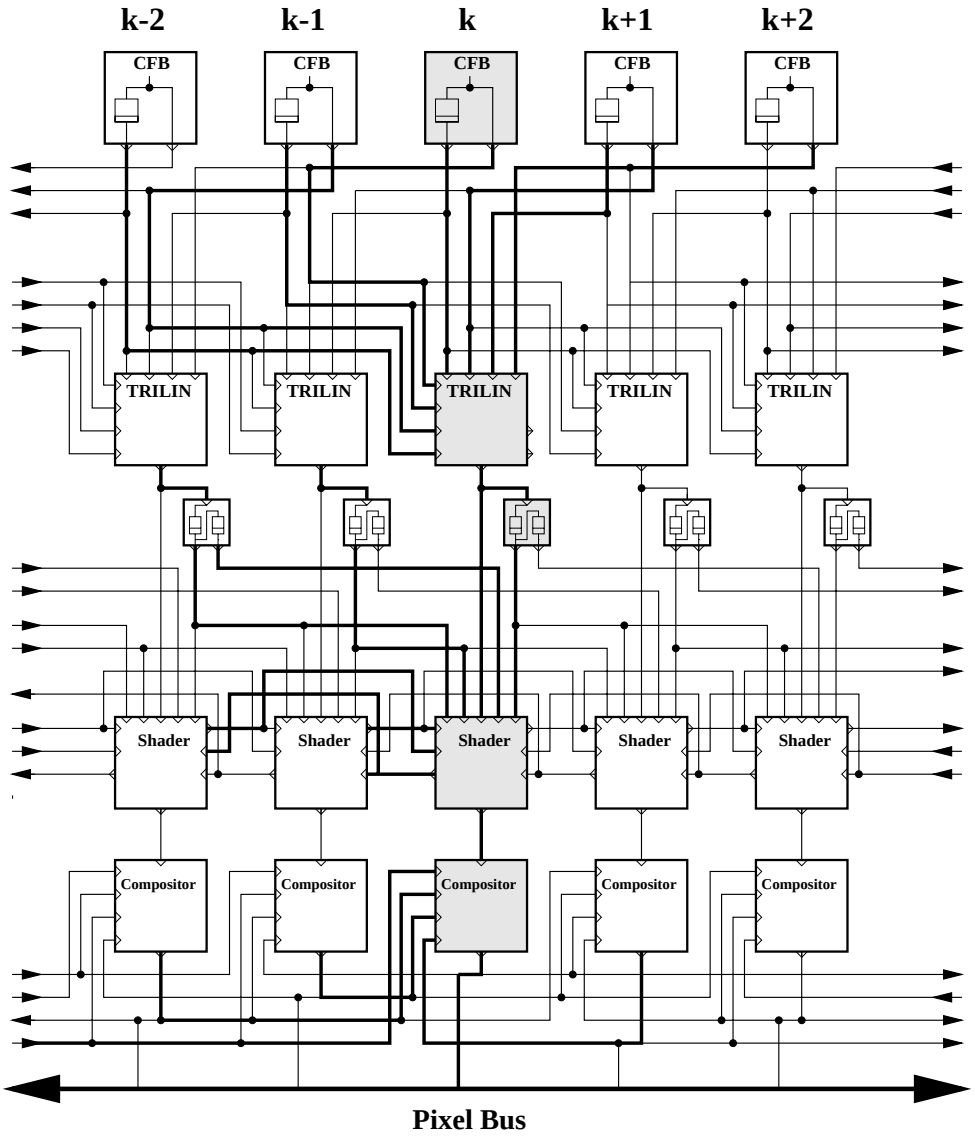

Figure 62: *Cube-4 Beam-Parallel Architecture.*

output the current pixel value on the pixel bus and start compositing a new ray. The end of ray flag indicates the end of a discrete ray and has the same effect for the compositing stage. The steps in X and Y bit fields carry the information concerning the forwarding of voxels and samples inside each of

the computational units. These control bits as well as the X and Y templates can be computed inside the TRILIN units [Wes95, KM96].

We implemented the Cube-4 algorithm inside the VolVis volume visualization system developed in our group [ASK92, AHH+94]. Parallel and perspective projections of several types of datasets achieved high image quality. The complete Cube-4 dataflow was simulated extensively in C, and we developed behavioral VHDL code [KM96].

8.4 Performance and Scalability

The fixed datapath connections and the simple control make it very easy to exploit parallelism and pipelining at every stage of Cube-4. The performance is only limited by the speed of the memory, assuming that each of the computational units can be perfectly pipelined. If n is the dimension of the dataset, p the projection rate in frames per second, the required *memory access frequency* f_m is:

$$f_m = n^2 p.$$

Table 6 shows several memory access frequencies for various dataset sizes assuming real-time performance with $p = 30$ frames per second.

Dataset Size n^3	**Memory Access Frequency** f_m
128^3	0.49 MHz
256^3	1.97 MHz
512^3	7.86 MHz
1024^3	31.46 MHz

Table 6: *Memory frequencies for* $p = 30$ *frames per second.*

Commercial DRAMs typically have a random access frequency of $f_d = 8.33$ MHz (assuming 120 ns cycle time). This allows for machine implementations with dataset dimensions $n \leq 512$ using n off-the-shelf standard DRAMs. By clocking the DRAMs at their maximum frequency f_d it is possible to reduce the number of physical memory chips to

$$n_d = \lceil \frac{n f_m}{f_d} \rceil,$$

each of them storing $\frac{f_d}{f_m}$ times the data of the fully parallel implementation. As an example, a 128^3 machine using standard DRAMs with $f_d = 8.33$ MHz requires only $n_d = 8$ physical memory chips instead of 128 for the fully parallel version. Each of the chips stores 16 times the data of the fully parallel version.

A high-resolution implementation requires higher memory access speeds than can be delivered by todays standard DRAMs. However, recent developments in DRAM technology have lead to faster devices [IEE92]. Synchronous DRAMs (SDRAMs) latch the data under control of the system clock, enabling pipelined read cycles. Enhanced DRAMs (EDRAMs) use on-chip static buffers to reduce the average access time. Currently these high-speed memory devices achieve an average access frequency of $f_d = 33$ MHz, allowing for 1024^3 implementations. Future improvements in memory technology allow faster DRAM access frequencies, thereby enabling more compact or higher-resolution Cube-4 implementations.

Table 6 assumes a fully parallel Cube-4 implementation with n memory units and n processing units in each stage of the projection pipeline. However, this also results in the same *processing frequency* f_p for the TRILIN, shader and compositor units, $f_p = f_m$. In order to significantly reduce the hardware complexity, we can increase the processing frequency and use fewer processing units that operate on multiple data items in a time-sliced fashion. We call

this technique *partitioning* or *horizontal integration* of the machine. Instead of n we use $\frac{n}{m}$ processing units, where n is the dataset dimension and m the partitioning factor. Given a projection rate of p frames per second, this results in an increased processing frequency of:

$$f_p = n^2 pm.$$

Figure 63 shows the tradeoff between the number of processing units per pipeline stage, $\frac{n}{m}$, and the processing frequency, f_p, for three dataset resolutions. Note that we use the number of units per pipeline stage and not the total number of processing units. Depending on available technology it is possible to combine two or more stages of the pipeline into one physical unit, a technique that we refer to as *vertical integration* of the machine.

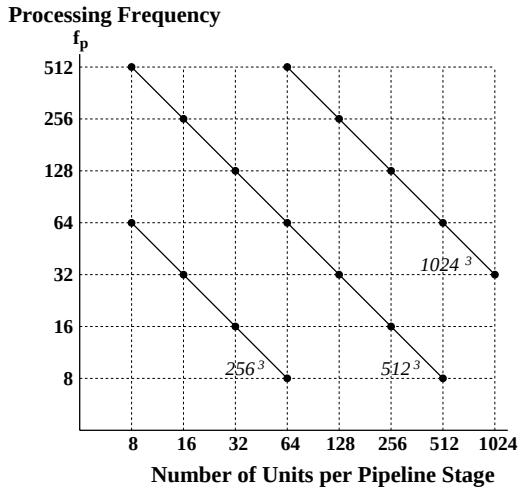

Figure 63: *Processing frequency, f_p, vs. number of units per pipeline stage, $\frac{n}{m}$, for three dataset resolutions. Rendering performance is $p = 30$ frames per second.*

It can be seen from Figure 63 that 64 units per pipeline stage running at 64 MHz suffice to implement a 512^3 machine. Using fast DRAMs such a design

could be implemented using 128 memory chips, bringing it into the realm of VME board sizes. Similarly, we can achieve EISA board implementations for 256^3 datasets using 8 units per pipeline stage running at 64 MHz and 16 fast DRAMs. Note that in the previous discussion we did not change the parameter p, assuming real-time projection rates of 30 frames per second for all tables, graphs and examples. Making p smaller further reduces the hardware requirements.

8.5 Summary

The Cube-4 beam-parallel architecture presented in this chapter is a first step towards a special-purpose scalable architecture that can deliver real-time high-quality ray casting of high-resolution datasets.

Beam-parallel ray-casting avoids any global communication of voxels and only requires a pixel-bus of moderate bandwidth. The resulting Cube-4 architecture is scalable, modular in design, and has the potential of rendering high-resolution datasets, such as 1024^3 16-bit voxels, at 30 frames per second. Using the fully pipelined ray-casting algorithm presented in Chapter 5 does not require any pre-computations and allows for 4D visualization of dynamically changing data.

However, beam-parallel processing leads to sheared tri-linear interpolation and some control complexity because of steps of the discrete rays. The Cube-4 slice-parallel architecture presented in the next chapter uses the correct 3D neighborhoods for interpolation. Furthermore, the constant skewing difference between beams along slices leads to a much simpler dataflow and an architecture with fewer datapaths. Consequently, the Cube-4 slice-parallel architecture has the best image quality and cost-performance ratio of the three

Cube architectures presented in this thesis.

Chapter 9

Cube-4: The Slice-Parallel Architecture

The beam-parallel dataflow model that we developed in the previous chapter results in localized, fixed connections between memory, tri-linear interpolation, shading and compositing units. However, beam-parallel processing requires to use sheared tri-linear interpolation and complicates the datapaths due to the view-dependent data traversal. In this chapter, we look at the slice-parallel dataflow. We will show that slice-parallel processing greatly simplifies the dataflow and the resulting Cube-4 architecture.

In Section 9.1 we present the Cube-4 slice-parallel dataflow, a main contribution of this research. Instead of the informal explanation style used in the previous chapter, we will use signal flow graphs (SFG) to formally specify the slice-parallel dataflow. We start by assuming parallel projection and $n = m$ processing pipelines and memory modules, where n is the resolution of the dataset.

In Section 9.2 we generalize our results for the case of $m < n$. This will allow us to implement the Cube-4 architecture with much fewer memory modules and processing pipelines than the dataset resolution. Section 9.3 shows how to implement perspective projections in the slice-parallel processing framework.

The slice-parallel dataflow leads to localized, near-neighbor datapaths for the Cube-4 slice-parallel architecture, described in Section 9.4. In Section 9.5 we show results from simulations and a prototype implementation of Cube-4 on the Teramac, a configurable custom hardware machine developed by Hewlett-Packard Laboratories. Our implementation proves the feasibility of the architecture. The measurements show linear increase of performance with increasing number of rendering pipelines.

Finally, in Section 9.6 we analyze the theoretical achievable performance which indicates real-time frame rates for up to 1024^3 datasets and linear scalability of performance with increased parallelism.

9.1 Cube-4 Slice-Parallel Dataflow

Recall from Section 6.4 that the skewing distance s is the distance by which two beams have been shifted (mod m) relative to each other. For example, Figure 42a shows that each beam of a slice (in volume space) has been shifted by $s = 1$ (in memory space) with respect to the beam below it. This means that, in general, beams can not be accessed from memory in the same order they have in volume space.

One solution to the problem is to permute fetched beams by an intermediate interconnection network between the memory and the processing units. This permutation of beams is called *unskewing*, because it reduces the skewing distance between consecutively fetched beams to zero. This approach has been

used in the Cube-3 architecture. However, the hardware complexity of such a global interconnection is high and becomes prohibitive for large m, limiting the scalability. In Cube-4 we take a very different approach that does not require any global communication except at the pixel level.

We now explain the datapaths and the resulting dataflow in detail using SFGs. A SFG is a directed graph with non-negative edge and node weights. A node stands for an arithmetic or logic function performed with zero delay and an edge stands for data transport. The order of operations is represented as directed edges emanating from the node that is to be executed first. The weight of the edge indicates by how many clock cycles the first operation must precede the second operation. We do not show edge weights of 0. An edge may also be viewed as a datapath from one operation to another and its weight as indicating the number of registers included in that datapath. The width of all datapaths is assumed to be constant. To simplify the discussion, we first restrict our attention to the case of $m = n$. Later, we discuss the generalization of these results to the case of $m < n$.

9.1.1 Tri-Linear Interpolation

Tri-linear interpolation requires 8 voxels arranged in a $2 \times 2 \times 2$ orthogonal voxel neighborhood. This is equivalent to two 2×2 voxel neighborhoods from consecutive data slices. Figure 64 shows one slice of a $4 \times 4 \times 4$ dataset in volume space and in skewed memory space. For simplicity we have indicated increasing voxel addresses along rows with consecutive letters. The neighborhoods required for the bi-linear interpolation inside the slice are surrounded by a box. Notice how the orthogonal neighborhoods are shifted and sheared in memory space due to the skewing difference between beams.

Assume that we fetch consecutive beams in positive Y direction from the

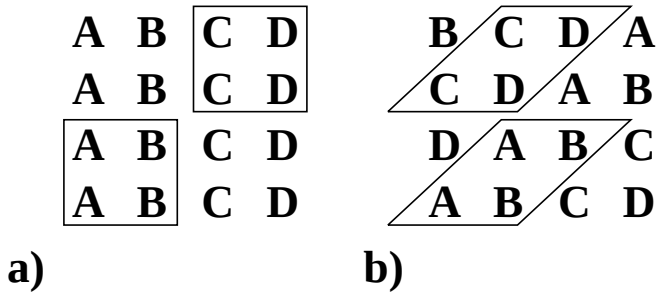

Figure 64: *Bi-linear neighborhoods a) in volume space and b) in skewed memory space.*

dataset. This corresponds to fetching consecutive rows in column direction in Figure 64b. The SFG in Figure 65 shows how the data is moved between pipeline stages. Dashed edges that leave on one side of the figure are connected

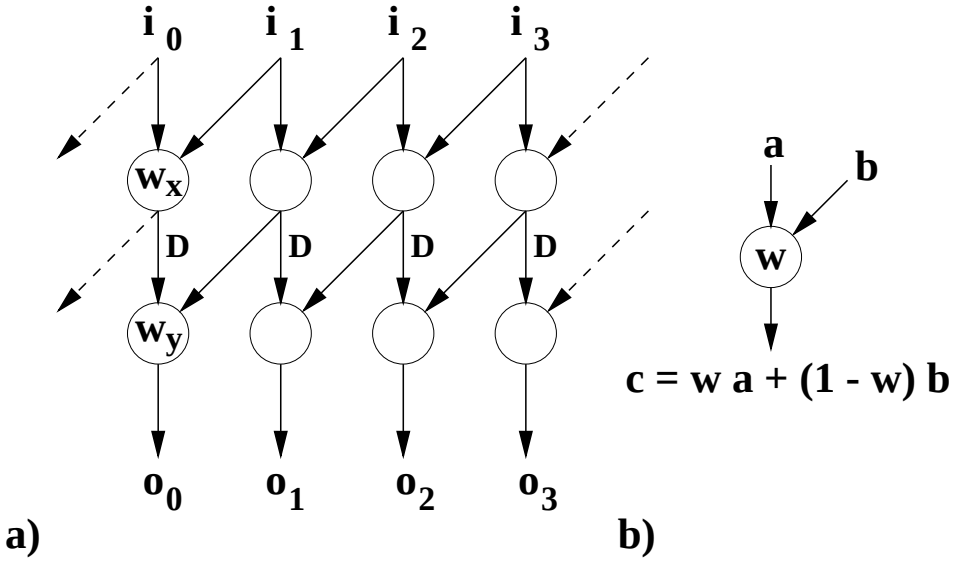

Figure 65: *a) SFG for bi-linear interpolation. Edges labeled with D indicate that the first operation must precede the second operation by D clock cycles. b) Each node performs a linear interpolation.*

to corresponding edges on the other side in a wrap around fashion. Each node in the graph performs a linear interpolation of its two inputs. The first stage

performs a linear interpolation between neighboring voxels of one beam, and the second stage performs a linear interpolation between the linearly interpolated samples of two consecutive beams.

Figures 66a through 67h show how the data flows in the corresponding hardware configuration for the bi-linear interpolation of a 4×4 voxel neighborhood.

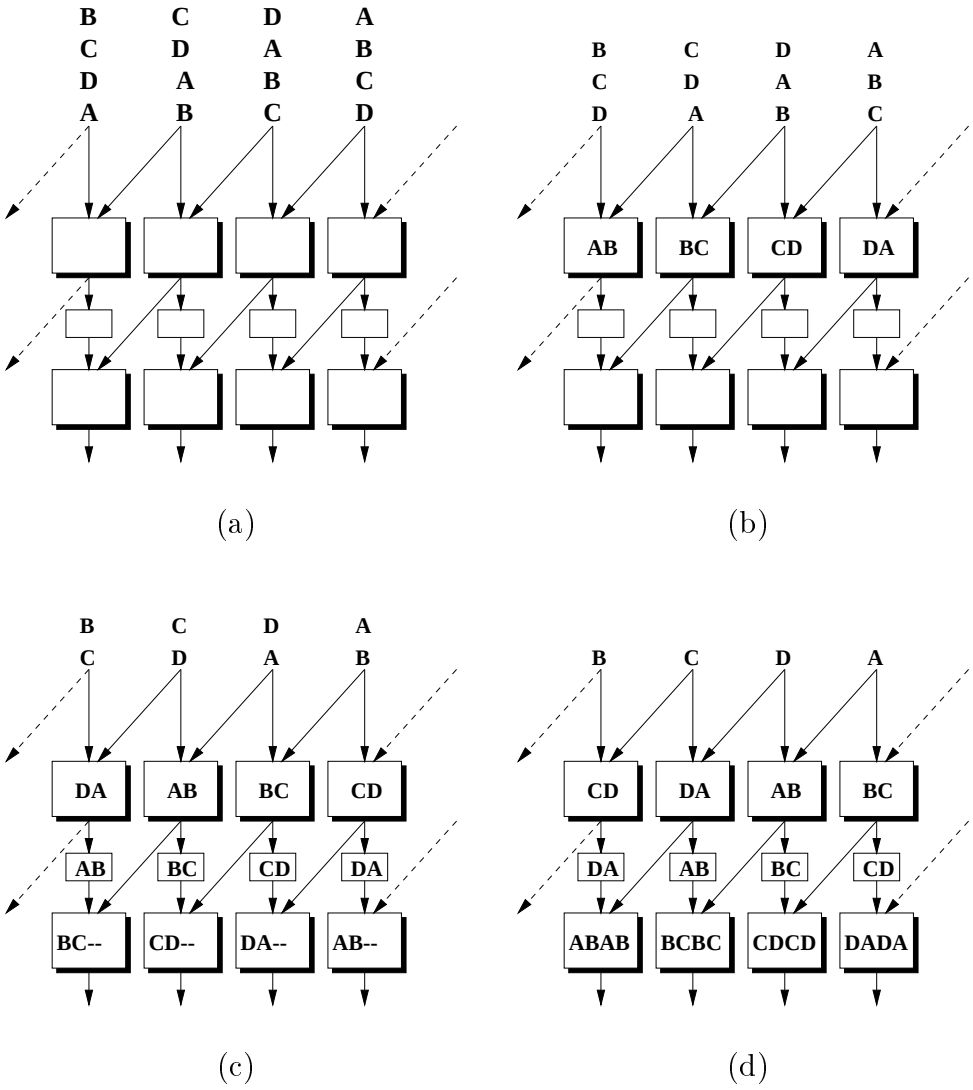

Figure 66: *Bilinear Interpolation Schedule (Part I)*

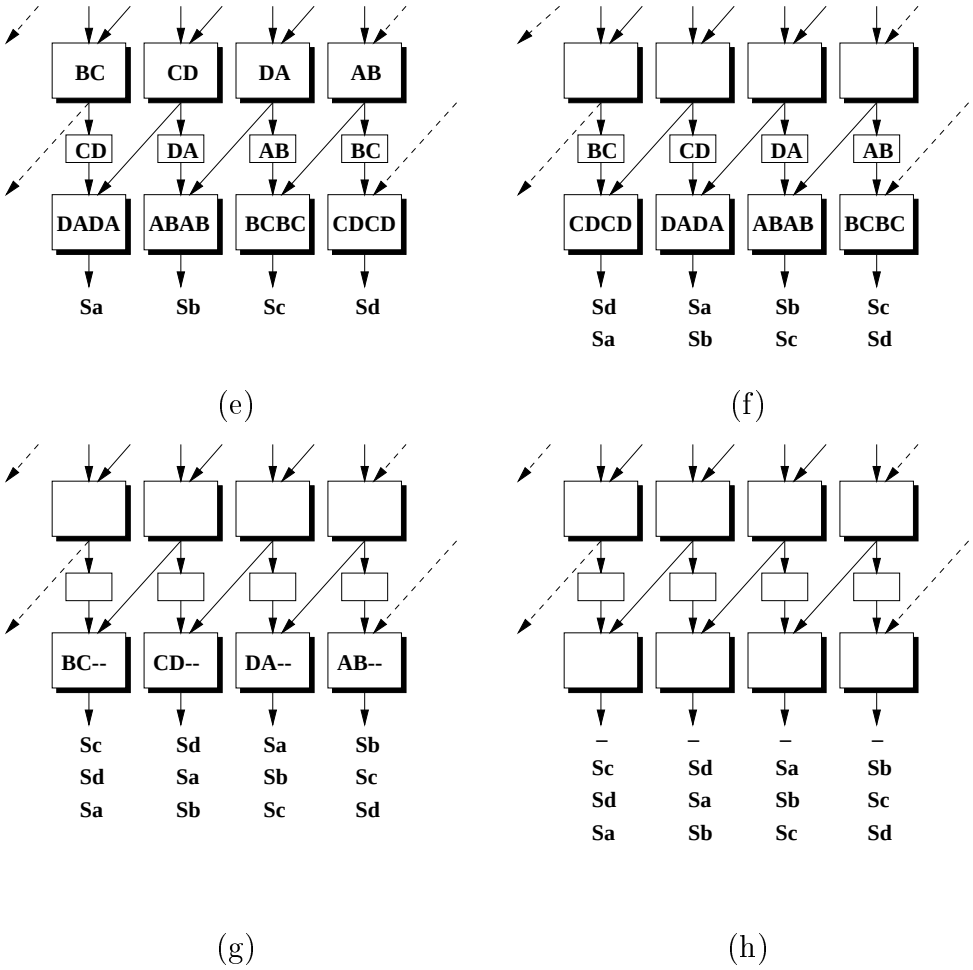

Figure 67: *Bilinear Interpolation Schedule (Part II)*

Each linear interpolation node is represented by a square cell that performs the necessary arithmetic operations. To simplify the explanation, each linear interpolation cell also contains a register that holds the two input values[1]. The delay along edges is represented by a register on the datapath between subsequent cells. The data that flows from the top to the bottom is represented using the same letters as in Figure 64. Each new drawing corresponds to another system clock period.

Looking at the SFG in Figure 65 and the example dataflow in Figure 66 and 67 we notice some important patterns. The datapath between memory and the first stage of linear interpolators is used to join two (spatially) adjacent voxels from a beam at a time. This is achieved by a *merger* of adjacent voxels inside the same beam. The datapath between the first and second stage of linear interpolators is used to join data of two (temporally) subsequent beams. Because the two beams are output in consecutive clock periods, this can be achieved by a *shift and delay*.

Another important observation is that the resulting bi-linear samples are output in skewed order. Although the skewing difference between input beams has been corrected, the dataflow does not attempt to de-skew the resulting values. As we will see in the following sections, this is true for all processing stages of the Cube-4 architecture. We effectively defer the de-skewing of the memory from 3D samples to 2D pixels, where it can be more efficiently implemented by a simple address permutation.

To perform a tri-linear interpolation, we need voxel data from two subsequent slices. Because voxels from the second slice are output n clock periods later, we need to delay data from the previous slice by n cycles. Furthermore,

[1] An actual hardware implementation is free to use zero or more internal registers inside each cell, depending on the required performance. A higher degree of pipelining leads to longer latency but shorter system clock period.

because of the skewing difference between beams of subsequent slices, we need to shift the non-delayed output from the memory by one position. Figure 68 shows the complete SFG for tri-linear interpolation for $n = m = 4$ using the SFG of Figure 65. The last stage of this SFG performs the linear interpolation

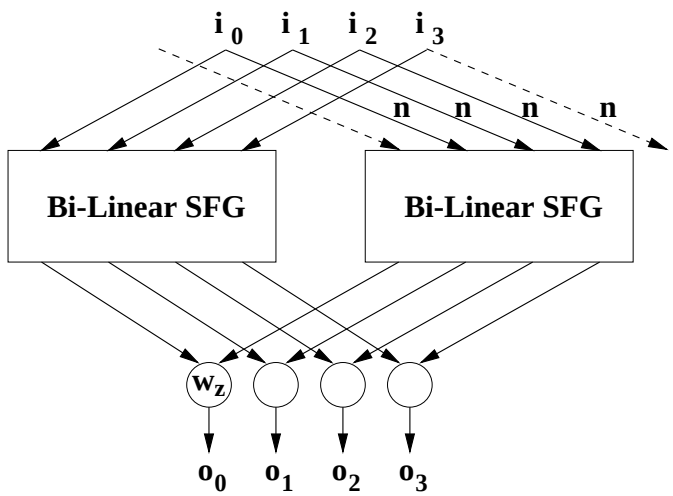

Figure 68: *SFG for tri-linear interpolation and $n = m = 4$. Edges labeled with n indicate that the first operation must precede the second operation by n clock cycles.*

between the bi-linearly interpolated samples of the two slices. Notice that this SFG easily extends to arbitrary values of n. The interpolation weights, indicated by w_x, w_y in Figure 65 and w_z in Figure 68, remain constant for all samples inside a newly interpolated slice. They can be easily pre-computed using the template-based address generation discussed in Chapter 4.

9.1.2 ABC Gradient Estimation

The dataflow for ABC gradient estimation is similar to tri-linear interpolation. We first explain the dataflow for the estimation of the X and Y gradient. It requires the collection of a 3×3 neighborhood inside each slice of interpolated

samples. Figure 69 shows a slice with 4×4 samples, both unskewed and skewed.

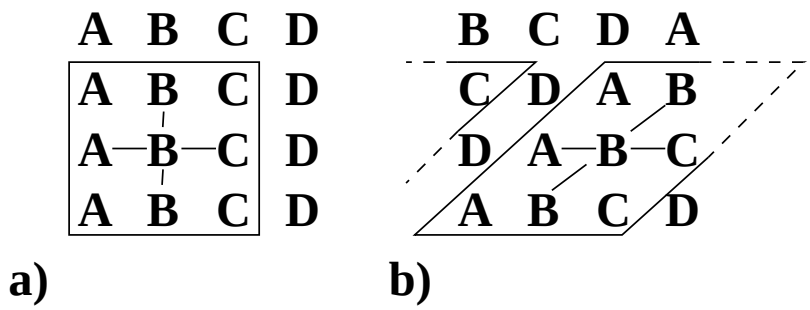

Figure 69: *Gradient neighborhood inside a slice of interpolated samples a) unskewed and b) skewed.*

The neighborhood required for one gradient estimation inside a slice of samples is surrounded by a box. Remember that samples are output each clock cycle by the tri-linear interpolation stage as skewed beams. To estimate the X gradients inside the slice (indicated by the horizontal line between samples) we need to merge and subtract sample values that are two positions apart inside each beam. Figure 70 shows the SFG that achieves this. The nodes in the first stage simply forward their inputs to the nodes of the following stage which compute the difference between their two inputs. Notice that this SFG merges the samples the same way as the bi-linear interpolation SFG in Figure 65, except that it needs two stages to merge samples with distance two in X direction.

The computation of the Y gradient (indicated by the vertical line in Figure 69) requires a shift and delay of samples from beams that are two positions apart in Y direction. Figure 71 shows the corresponding SFG. The node operations are the same as in the SFG for central differences in X shown in Figure 70. The shift and delay operation is equivalent to the shift and delay required for bi-linear interpolation, except that we need to shift and delay

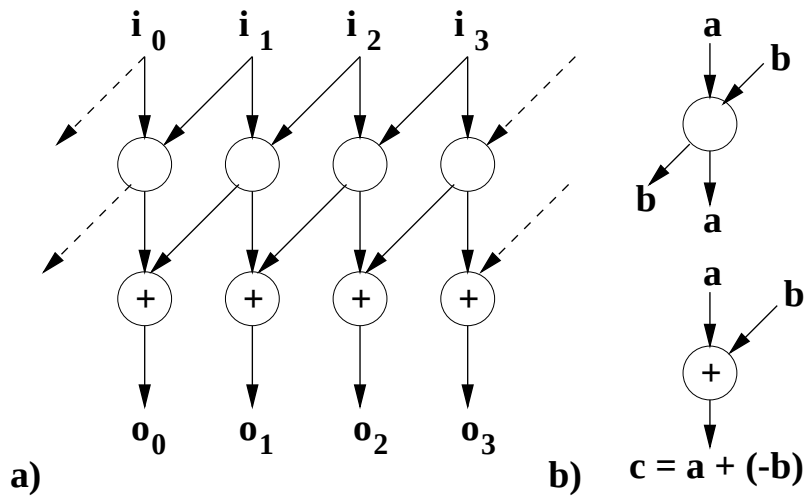

Figure 70: *a) SFG to compute central differences between samples for the X gradient. b) Nodes either forward the inputs or compute their difference.*

twice to merge samples from beams with distance two in Y direction.

As shown in Figure 60 on page 138, we need to compute two additional samples by bi-linear interpolation inside sample slices for corrected gradient estimation in Z. This additional interpolation is achieved by the bi-linear SFG of Figure 65 applied to the samples output from the tri-linear interpolation SFG. The weights for this interpolation are d_x and d_y, the X and Y component of the viewing vector, respectively. To compute the gradient component in Z direction, we need to choose among these corrected samples depending on the viewing direction as illustrated in Figure 72.

In 3D there are four cases to consider depending on the sign of d_x and d_y. Figure 73 illustrates this using a 2D drawing. The samples with a thick outline in the center of the figure represent the beam of the current slice for which the gradient components in Z are being computed. The dashed and shaded samples are bi-linearly corrected samples in the ahead or behind slice. For any given sample of the current slice, there are four possible choices for

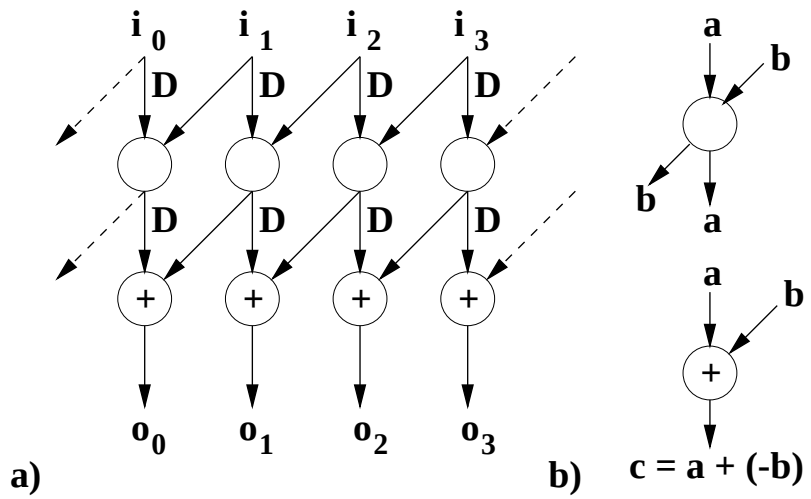

Figure 71: *a) SFG to compute central differences between samples for the Y gradient. Edges labeled with D indicate that the first operation must precede the second operation by D clock cycles. b) Nodes either forward the inputs or compute their difference.*

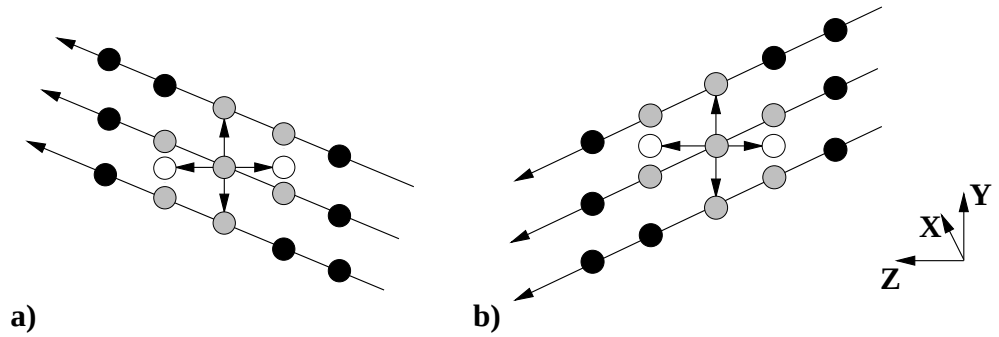

Figure 72: *Different choices for gradient correction in Z depending on the non-major component d_y of the viewing vector. a) $d_y > 0$ b) $d_y < 0$*

gradient estimation in Z in the ahead or behind slice, indicated by NE (North East), NW (North West), SE (South East), or SW (South West). Table 7 lists all valid choices for the ahead and behind slice depending on the non-major components d_x and d_y of the viewing vector.

Figure 74 shows the SFG to compute and forward the corrected samples

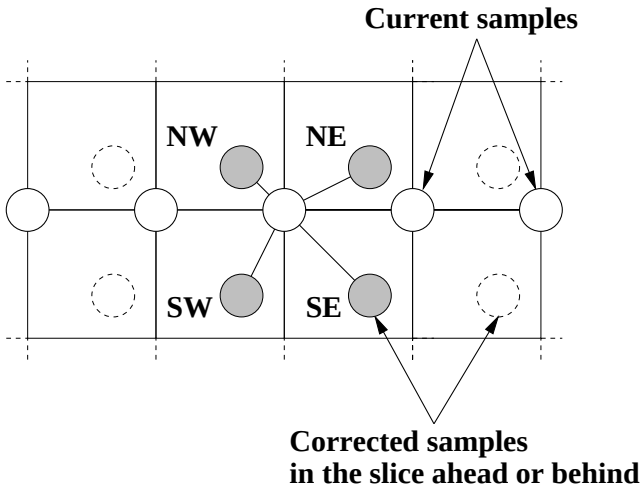

Figure 73: *Four possible choices among samples in the ahead or behind slice during gradient estimation in Z for the samples of the current slice.*

	$d_x < 0$		$d_x > 0$	
	Ahead	Behind	Ahead	Behind
$d_y > 0$	NW	SE	NE	SW
$d_y < 0$	SW	NE	SE	NW

Table 7: *Valid choices for gradient estimation in Z between the ahead and behind slice depending on the non-major components d_x and d_y of the viewing vector.*

inside each slice for gradient estimation in Z direction. The samples from the tri-linear units are first forwarded through the SFG for bi-linear interpolation to compute the additional samples for corrected gradient estimation. The nodes at the output of this SFG act as multiplexors to select the valid corrected samples for gradient estimation in Z according to Table 7. The first stage, activated by the signal *sel 0*, selects between East and West samples. The second stage, activated by the signal *sel 1*, selects between the North and

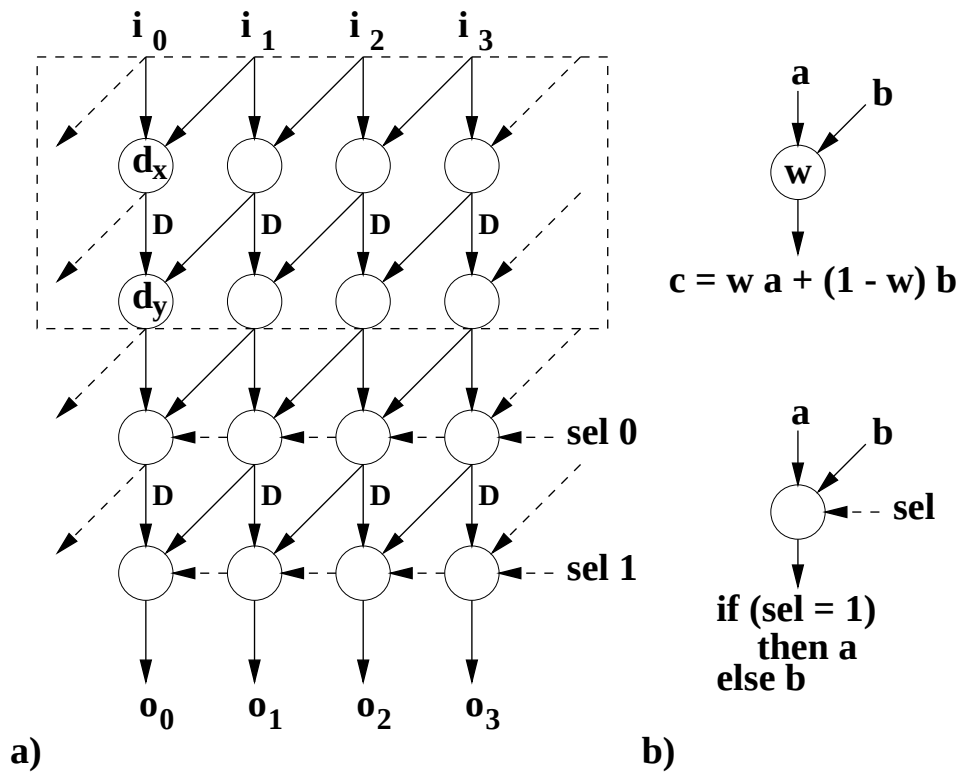

Figure 74: *a) SFG to compute and select corrected samples inside each slice for the Z gradient. Edges labeled with D indicate that the first operation must precede the second operation by D clock cycles. b) Nodes are either linear interpolators or multiplexors to select valid corrected samples depending on the viewing direction.*

South samples (see Figure 73). Notice that the structure of the selection stages is equivalent to the SFG for bi-linear interpolation. This is because we are computing inside 2×2 sample neighborhoods in both cases.

Figure 75 shows the complete SFG for ABC gradient computation. The inputs x_i are the skewed beams of samples output from the tri-linear interpolation units. The samples currently output by the tri-linear interpolation stage are input without delay as ahead samples. The ahead samples are delayed by n cycles to yield the current samples. A delay of the ahead samples by $2n$

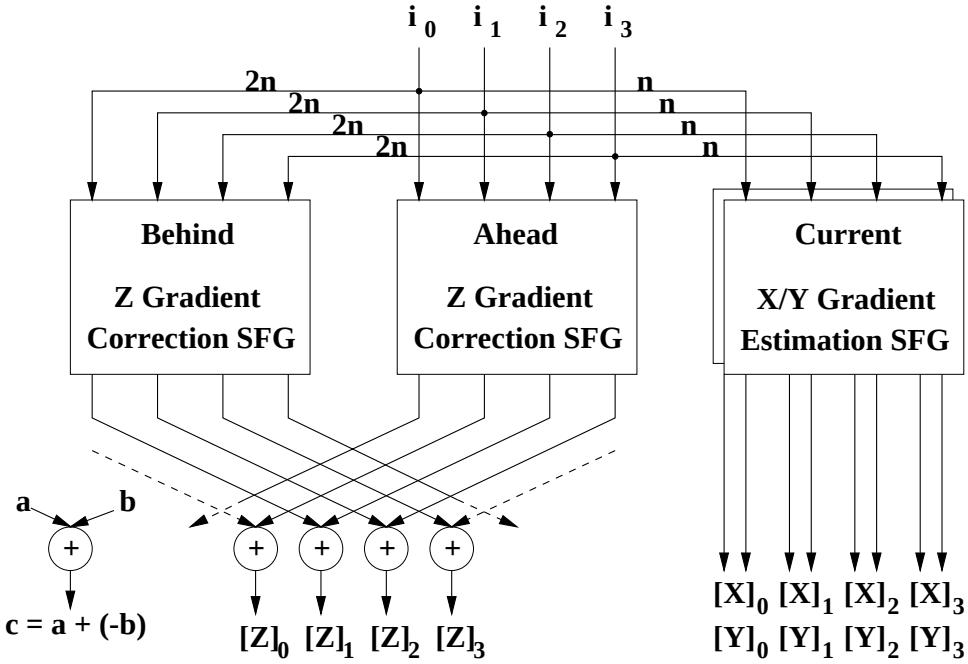

Figure 75: *SFG for ABC Gradient Estimation. Edges labeled with n or $2n$ indicate that the first operation must precede the second operation by n or $2n$ clock cycles, respectively.*

cycles produces the behind samples. All gradient computations are relative to the current slice. The gradient components in X and Y are computed on the current samples using the SFGs of Figure 70 and Figure 71, respectively. The ahead and behind samples are subjected to the gradient correction and selection using the SFG of Figure 74. The last stage in the SFG computes the central differences between the gradient corrected ahead and behind samples and outputs the gradient components in Z direction.

9.1.3 Shading and Classification

Using the gradient, each sample is shaded using any of the standard local illumination models. For maximum performance, we need to perfectly pipeline the

shading calculations. Other researchers have proposed fully pipelined Phong shading and vector normalization architectures [Kni93]. For our prototype implementation, we use a small, lookup-table based reflectance map shader [BvS95]. It allows to implement any higher-order shading model without expensive square root units. Classification is performed based on sample value and possibly gradient magnitude using a lookup-table opacity map.

9.1.4 Compositing

The shading stage produces consecutive beams of color intensity values within slices. In the slice-parallel dataflow, the compositing stage accumulates these intensity values to pixels stored in the base-plane. The total size of this base-plane buffer is $(2n)^2$, the maximum size of a base-plane (see Chapter 5). However, this accumulation buffer is distributed among m compositing units. The difficulty is how to forward the intensity values along a ray to the compositing unit that stores the intermediate base-plane pixel corresponding to that ray. Or, alternatively, how to forward the intermediate base-plane pixel value to the compositing unit that receives the next intensity value along the ray.

Consider a partially composited base-plane pixel that was produced after compositing slice S. We have to forward this pixel to the compositing unit which receives the next intensity value along the ray from the shader. Because all rays are 26-connected in discrete space, the next sample along the ray must come from a 3×3 neighborhood from the previous slice $S-1$. Using the discrete ray-templates of the template-based ray-casting algorithm [YK92], we can determine the position inside this neighborhood of the next intensity value along the ray. Using Figure 76, we can determine the forwarding pattern for all possible cases. The figure assumes that the major viewing direction is Z and that the dataset is stored along beams in X direction.

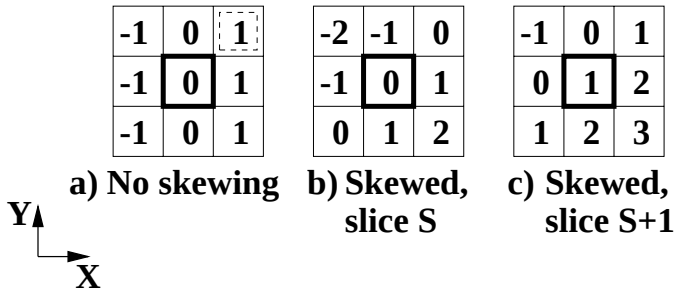

Figure 76: *Compositing Neighborhood.*

Figure 76a shows the 3×3 neighborhood for the case of no skewing. The center position, surrounded by a box, indicates the current position of the intermediate base-plane pixel. The numbers indicate the relative distance in X to the compositing unit that receives the next sample along the ray. For example, if the ray-templates indicate that the ray in discrete space makes a step in positive X and Y directions, the next sample is forwarded to the compositing unit one position in positive X direction (shown by a dashed box in the figure). Because of the skewing difference between beams inside slices, this forwarding distance is altered as shown in Figure 76b. Finally, Figure 76c shows the forwarding distances if we take the skewing between slices S and $S + 1$ into account. Because of the forwarding distances, each compositing unit has to be connected to three units in positive and one unit in negative X direction. Figure 77 shows the corresponding SFG for compositing. Notice that, due to the maximum skewing differences of -1 and $+3$ shown in Figure 76c, a minimum of five rendering pipelines is required.

The resulting pixels of the base-plane are still generated in a skewed order. However, pixel scanlines can easily be unskewed by a simple address-permutation inside or when stored into the frame-buffer.

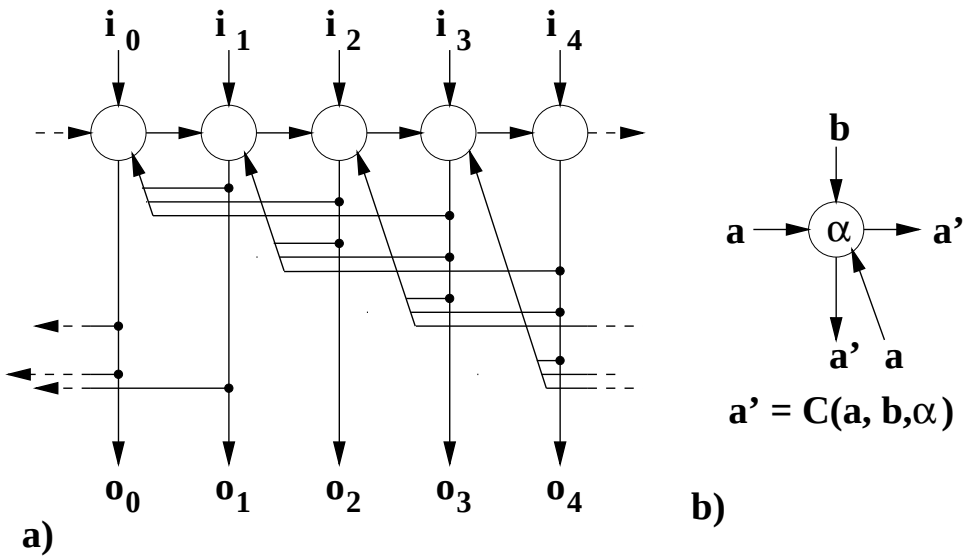

Figure 77: *a) SFG for compositing. b) Operation of a compositing node. C stands for front-to-back or back-to-front compositing. α is the current sample opacity, b is the current sample value, a is the partially accumulated result, and a' is the new result of the current compositing operation.*

9.2 Extensions for $m < n$

If $m < n$, we have to add two minor changes to the dataflow presented so far. Instead of complete beams we forward partial beams with $q = \frac{n}{m}$ samples each. The order of partial beam access is along beams. To fetch the data of a complete beam requires q cycles instead of one cycle. Consequently, all delay operations on edges in the SFGs, which are needed to gather data from consecutive beams, need to be changed from D to q.

The second change is required because of border cases between partial beams. For example, the tri-linear interpolation units at rightmost position m require voxels from the partial beam that will be fetched one cycle later. Figure 78 shows how to deal with these border cases using a technique we call *beam-extension*. The partial beam i, is delayed by one cycle, until the next

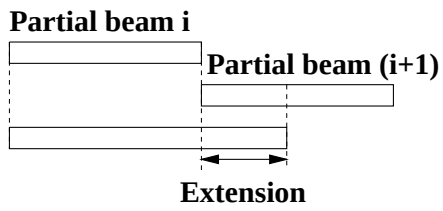

Figure 78: *Beam Extension.*

partial beam $(i+1)$ arrives. The overlap necessary for the border cases is available as an extension to beam i. Notice that we need to extend beams only in the direction of partial beam access. The amount of extension depends on the processing stage and varies between 3 and 4 data samples.

9.3 Perspective Projections

Perspective projection is nearly identical to parallel projection, except that the interpolation stage also needs to compute averages of larger neighborhoods for slices further away from the base-plane (cf. [LL94]). The first slice of data is uniformly sampled and scaled by a factor of one, which corresponds to shooting one ray per pixel of the base-plane. In all subsequent slices, the slices are scaled according to the viewing transformation, and a larger portion of the slice is sampled.

This averaging of larger neighborhoods can be implemented in hardware using additional interpolation stages that perform a simple box-filtering of slices. The maximum extent of this box filter, needed for the slice furthest away from the base-plane, is $1 + 2\tan\alpha$, where α is half of the field-of-view angle. For $\alpha < 45^0$, or any field-of-view less than 90^0, this corresponds to a maximum extent of 3 voxels. After samples from averaged slices have been computed, the subsequent algorithm remains the same as for parallel

projections.

In order to average the voxel slices, we need to compute the convolution of each slice with a 3×3 box filter:

$$y(i,j) = \sum_{k=0}^{2} \sum_{l=0}^{2} a(l,k)\, x(i-l, j-k). \tag{28}$$

We can further break down the 2D convolution into:

$$\begin{aligned} y(i,j) &= \sum_{l=0}^{2} a(l,0)\, x(i-l,j) + \sum_{l=0}^{2} a(l,1)\, x(i-l,j-1) \\ &\quad + \sum_{l=0}^{2} a(l,2)\, x(i-l,j-2). \end{aligned} \tag{29}$$

Each summation term in Equation 29 describes the 1D convolution of a beam of samples with a 1×3 filter. Figure 79 shows the SFG that implements this 1D convolution. For clarity of explanation we have labeled each input or output variable with only one index number. Dashed edges that leave on one side of the figure are connected to corresponding edges on the other side in a wrap around fashion. Each node in the graph performs a multiply-accumulate of one input and forwards the other input to the following node.

Table 8 shows the dataflow during 1D convolution of the 4×4 voxel slice shown in Figure 64b with a 1×3 filter kernel. The table only shows the data x' and y' of nodes in the first two rendering pipelines (0 and 1). At each timestep, the data moves through the three consecutive stages of the SFG in Figure 79. Notice how adjacent samples in each beam are merged during consecutive timesteps. The output of the 1D convolution SFG is still skewed.

To perform a 2D convolution of each sample slice with a 3×3 filter kernel we need to sum up to results of three consecutive 1D convolutions as specified by Equation 29. Figure 80 shows the resulting SFG. Because of the skewing difference between the output samples of the SFG in Figure 79, we need to delay and shift the outputs in order to add up corresponding summation terms.

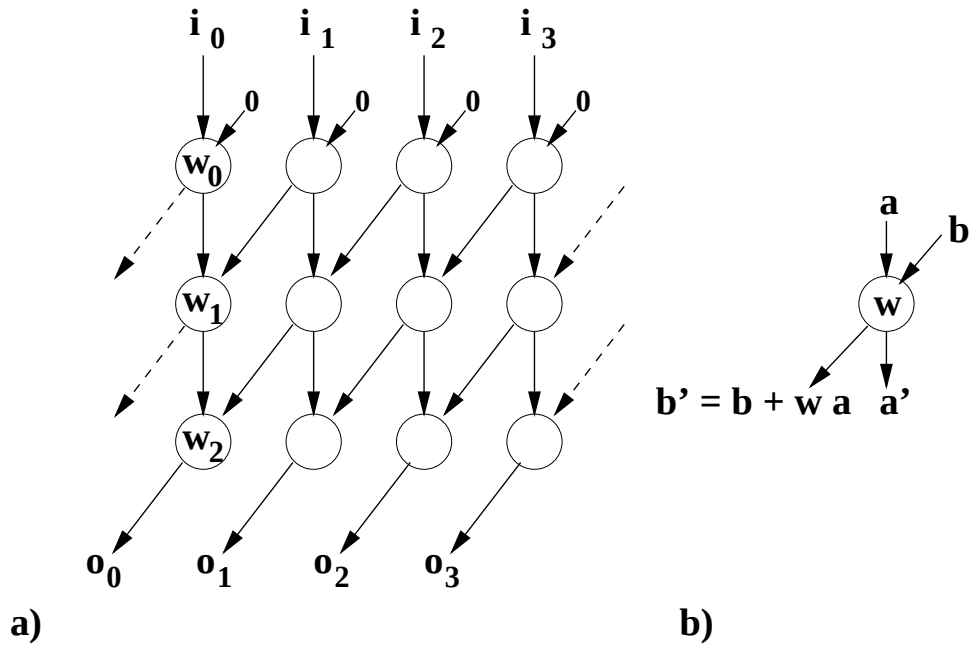

Figure 79: *a) SFG for 1D convolution with a* 1×3 *filter. b) Operation of each convolution node.*

Timestep	Stage	x'_0	y'_0	x'_1	y'_1
0	0	A	a_0A	B	a_0B
	1				
	2				
1	0	D	a_0D	A	a_0A
	1	A	$a_0B + a_1A$	B	$a_0C + a_1B$
	2				
2	0	C	a_0C	D	a_0D
	1	D	$a_0A + a_1D$	A	$a_0B + a_1A$
	2	A	$a_0C + a_1B + a_2A$	B	$a_0D + a_1C + a_2B$
3	0	B	a_0B	C	a_0C
	1	C	$a_0D + a_1C$	D	$a_0A + a_1D$
	2	D	$a_0B + a_1A + a_2D$	A	$a_0C + a_1B + a_2A$

Table 8: *Timing schedule for two pipelines (0 and 1) of the 1D convolution SFG.*

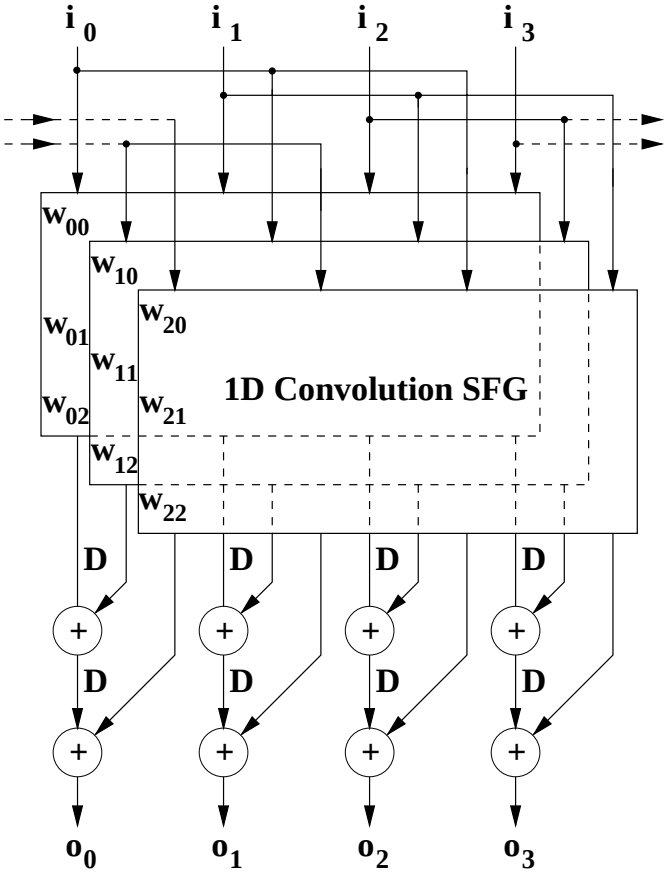

Figure 80: *a) SFG for 2D convolution with a* 3×3 *filter. Edges labeled with D indicate that the first operation must precede the second operation by D clock cycles. b) Operation of each convolution node.*

9.4 Cube-4 Slice-Parallel Architecture

Figure 81 shows the complete top-level diagram of the Cube-4 architecture with five rendering pipelines [PK96]. This is the minimal configuration due to the skewing difference for pixel exchange in the compositing stage The dataset is stored in the multiple cubic frame buffer (CFB) memory modules. Each rendering pipeline contains four types of processing units: CFB memory and

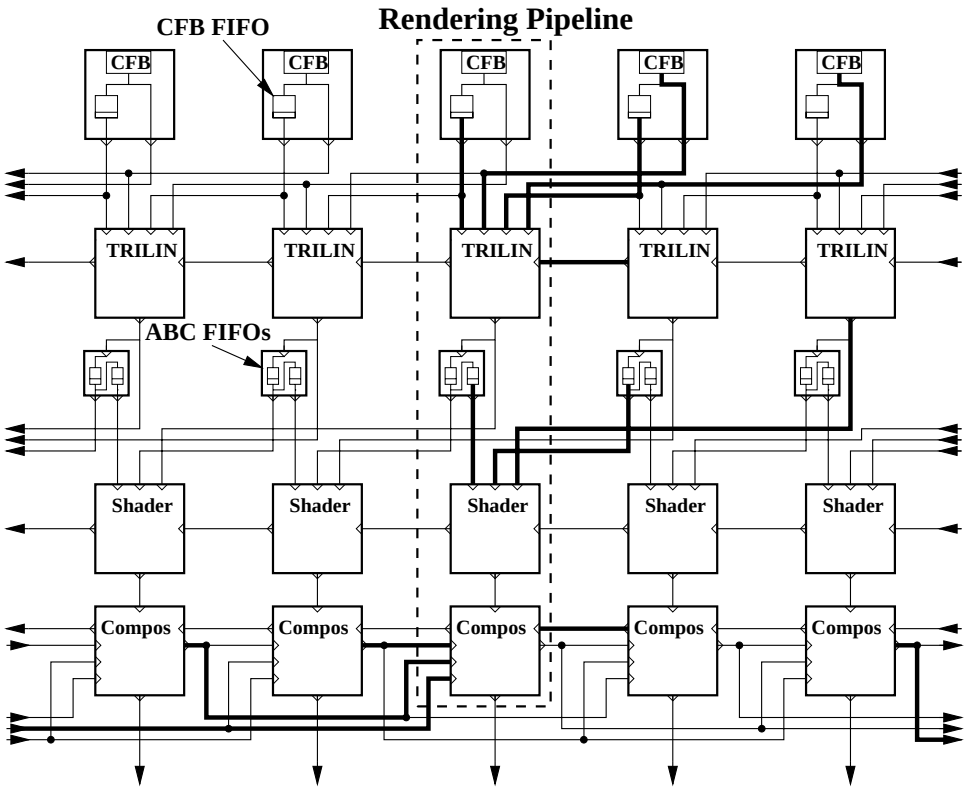

Figure 81: *Cube-4 slice-parallel architecture. Bold lines indicate all data connections of the rendering pipeline in the center. (CFB = Cubic Frame Buffer, TRILIN = Tri-Linear Interpolation Unit, Compos = Compositing Unit.)*

address generation, tri-linear interpolation (TRILIN), ABC gradient estimation and shading (Shader), and compositing (Compos). All datapaths have constant width, corresponding to the word-width of a voxel (e.g., 8 or 16 bits). The delay of data required for tri-linear interpolation and for the ABC gradient estimation is achieved by first-in first-out (FIFO) memories.

Control of Cube-4 is very simple and can be part of the dataflow. The host downloads the viewing vector into the CFB address-generation units. The ray templates are generated in hardware by adding the viewing vector to the current sample location and computing the resampling weights. From there

on, all necessary control signals travel with the data through the machine, making centralized control unnecessary.

9.5 Simulations and Prototyping

We have extensively simulated the algorithm and architecture in C and a high-level hardware description language (VHDL) [KM96]. Table 9 shows results from the VHDL simulation. The table shows rendering performance in frames per second versus the number of rendering pipelines for three different dataset resolutions. To translate the number of simulation cycles into frames per second, we assumed a relatively low processing frequency of 33 MHz.

Dataset	**# Pipelines**	**Cycles/frame**	**Frames/sec**
64^3	8	32,814	1,006
	16	16,422	2,009
	32	8,226	4,012
128^3	8	262,206	126
	16	131,118	252
	32	65,574	503
256^3	8	2,097,246	16
	16	1,048,638	31
	32	524,334	63
	64	262,182	126
	128	133,106	248

Table 9: *VHDL simulation results: Rendering performance as a function of the number of rendering pipelines.*

As a proof of concept we implemented a Cube-4 prototype on the Teramac, a configurable custom hardware machine developed at Hewlett-Packard Laboratories [KMS+96]. Figure 82a shows a picture of a 4-board Teramac system.

Teramac can execute synchronous logic designs of up to one million gates at rates up to 1 MHz [ACC+95]. The system has been built from custom field-programmable logic arrays (FPGAs) packaged in large multichip modules (MCMs). Figure 82 shows a picture of a single MCM, which carries 27 FPGAs. Each MCM measures 6.13×7.4 inches, weighs approximately 3 pounds, and has over 3000 pins. The Teramac system we used for our Cube-4 implementation includes 8 boards, 250 MB of RAM, 32 MCMs and 864 FPGAs.

Our prototype of Cube-4 on Teramac implements the design shown in Figure 81 with five rendering pipelines. The implementation is capable of producing parallel color projections of 128^3 8-bit per voxel datasets from arbitrary directions. Inside the shader units, we use a lookup-table based reflectance map shading. The total logic complexity for all five rendering pipelines is 330K gates. Compilation of the complete design onto Teramac takes less than one hour without user intervention.

The Cube-4 prototype generated an image of any of the 128^3 datasets in 1.5 seconds at 0.25 MHz, independent of dataset complexity, transfer function, or viewing parameters. The maximum processing frequency of Cube-4 on Teramac is 0.96 MHz without any performance optimizations. Figure 83 shows volume renderings of a CT lobster dataset and Figure 84 shows volume renderings of several other datasets. The use of different opacity and color transfer functions reveals different aspects of the data.

(a) (b)

Figure 82: *a) A four-board Teramac system. b) A Teramac multichip module (MCM) with 27 custom FPGAs routed through 39 signal layers.*

Figure 83: *Different volume renderings of a* 128^3 *CT lobster dataset generated by the Cube-4 prototype on the Teramac. Each image took 0.6 seconds at 0.96 MHz processing frequency.*

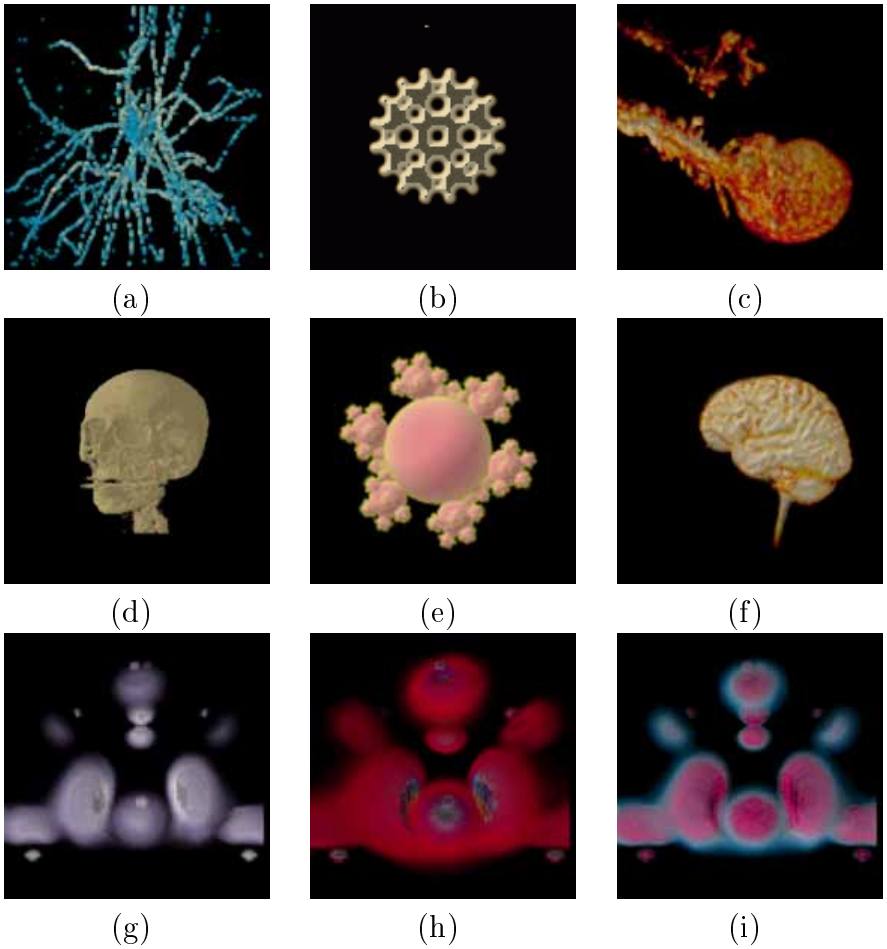

Figure 84: *Volume rendering images of* 128^3 *datasets produced by the Cube-4 prototype on Teramac. Each image took 1.5 seconds at 0.25 MHz processing frequency. a) Hippocampal pyramidal cell. b) Volume-sampled mechanical part. c) Bullfrog ganglion cell. d) CT head,* 45° *rotated. e) Volume sampled sphere-flake. f) MRI brain. Images g) through i) show the effect of different transfer functions on a simulated high-potential iron protein.*

9.6 Performance Analysis

The results we presented in the previous section indicate linear scalability of performance with increasing number of rendering pipelines. In this section, we look at the theoretical maximum performance of Cube-4. Assuming perfect pipelining of interpolation, shading, and compositing, we can continually enter data at the maximum possible rate, and the theoretical performance of Cube-4 is thus limited by the access speed of the memories.

If n is the dimension of the dataset, p the number of rendering pipelines, and f_p the processing frequency of the machine, the theoretical rendering rate f_r in frames per second is $\frac{p f_p}{n^3}$. Figure 85 shows the frame rate f_r as a function of the number of rendering pipelines p for three different dataset sizes. We show graphs for two different processing frequencies f_p. The solid lines shows graphs for $f_p = 33$ MHz, corresponding to the cycle time of SDRAM, the fastest currently available DRAM memory technology. The dashed lines show performance assuming 100 MHz processing frequency. Because current DRAM memory can not output data at this rate it has to be additionally interleaved per rendering pipeline. This additional interleaving is a standard memory bank arrangement as used in current general-purpose processors.

In order to allow for a compact implementation, we are currently developing an application-specific integrated circuit (ASIC) containing several of the Cube-4 rendering pipelines. We have a contract with a company that will fabricate such an ASIC. Preliminary estimates indicate that an ASIC containing 4 rendering pipelines requires less than 300 pins, including power and ground. Each ASIC requires only 400K gates, and internal memory for the ABC FIFO buffers of 40 K, assuming a total of 32 rendering pipelines.

We are designing a long PCI card system with 32 rendering pipelines or 8

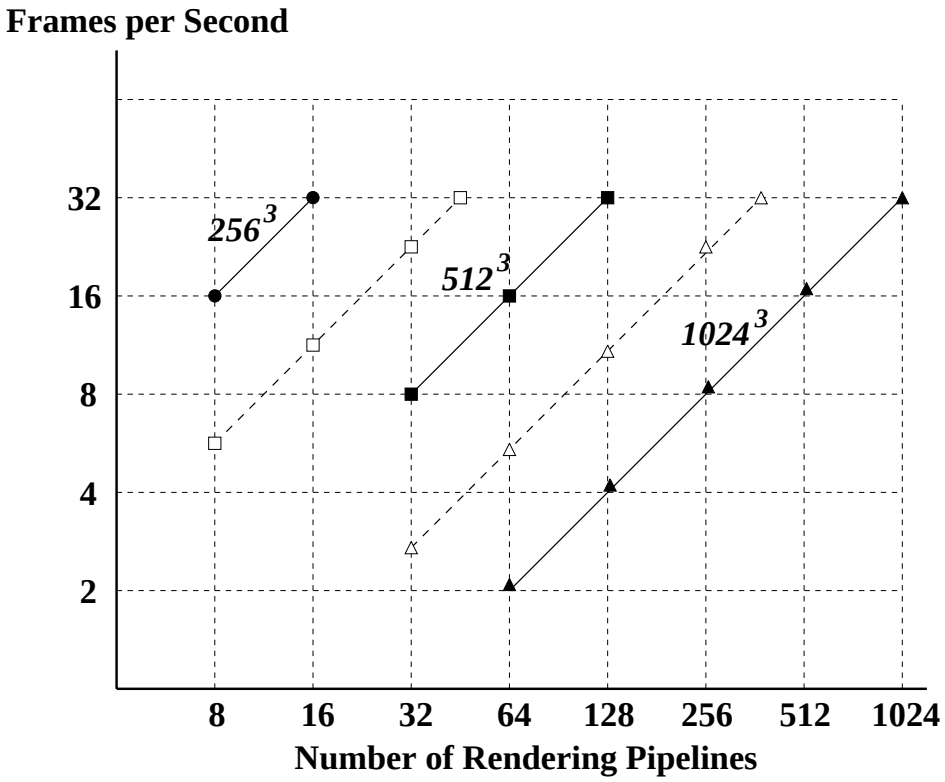

Figure 85: *Theoretical rendering performance of Cube-4 as a function of the number of rendering pipelines. We show graphs for different dataset sizes* ($\bigcirc = 256^3, \Lambda = 512^3, \triangle = 1024^3$)*. Solid lines indicate 33 MHz processing frequency, and dashed lines indicate 100 MHz processing frequency.*

Cube-4 ASICs, 32 SRAM chips, and a PCI host interface. Such a card would cost a few thousand dollars and provide 30 projections per second for 256^3 datasets. Larger systems for higher resolution datasets supporting 30 projections per second, 16-bit per voxel, can be built, such as a workstation board (e.g., VME size) for 512^3 datasets, and multiple boards for 1024^3 datasets.

9.7 Summary

In this chapter we have introduce the Cube-4 slice-parallel architecture. We have formally described the dataflow using signal flow graphs. Simple extensions to the dataflow allow parallel and perspective projections of high-resolution datasets with only a few rendering pipelines. The resulting Cube-4 slice-parallel architecture has local, fixed bandwidth datapaths. It is modular in design and scalable in performance.

Slice-parallel algorithms have been used in various forms by other researchers. Reynolds et al. [RGC87] and Lacroute and Levoy [LL94] use a shear-warp factorization of the viewing transformation to project the volume in a slice-parallel fashion onto the base-plane. Cameron and Underill [CU92] and Schröder and Stoll [SS92] have used slice-parallel approaches on massively-parallel SIMD machines.

Our hardware implementation of the slice-parallel ray-casting algorithm improves on these previous results in several ways. Shear-warp algorithms use linear 2D resampling filters [LL94], while the Cube-4 architecture implements accurate 3D resampling using tri-linear interpolation between data slices. Furthermore, Cube-4 does not use any pre-computations and stores only one copy of the dataset, allowing for real-time data input.

Chapter 10

Conclusions

Over the last decade, direct volume rendering has become an invaluable visualization technique for a wide variety of applications. Looking at the requirements of these applications, we make three key observations. First, real-time frame rates combined with interactive visual feedback during changes of the rendering parameters is essential if volume rendering is to become the primary visualization technique for 3D sampled or simulated datasets. Second, the trend in acquisition devices goes towards higher resolution 3D datasets that are acquired in a shorter period of time, possibly at real-time rates. And third, current volume rendering software or hardware is not capable of meeting these conflicting requirements.

This dissertation describes new special-purpose architectures for volume rendering that are capable of delivering real-time projection rates for high-resolution datasets. We employed an experimental research approach that goes beyond paper designs by implementing and testing hardware prototypes on the Teramac custom computing machine at Hewlett-Packard Laboratories. We have extensively simulated several design choices to validate many of the

key features of each design. The result of this design process and the main contribution of this dissertation is the Cube-4 architecture. Cube-4 meets all of our project goals and allows for a modular implementation in hardware with scalable performance.

In Chapter 2, we introduced the basic algorithms and acceleration techniques for volume rendering. We showed that pre-processing and data duplication are common features of most software optimization techniques. However, both of these techniques violate our project goals, which include interactive data classification, visualization of high-resolution datasets, and direct input of dynamically changing 3D data. We decided to implement ray-casting in our architectures because it provides very high image quality and leaves room for algorithmic improvements.

In Chapter 4, we looked at the basic ray-casting algorithm and analyzed its performance bottlenecks. We found that the same voxels have to be fetched several times during each projection, because of the non-uniform mapping of samples onto voxels, viewing dependent data traversal, and overlapping computational neighborhoods. This leads to excessive memory bandwidth requirements. Consequently, in Chapter 5, we introduced template-based data traversal and ABC gradient estimation using previously interpolated samples. Our improvements to the basic ray-casting algorithm lead to regular data traversal and a fully pipelined dataflow where each voxel is accessed exactly once per projection.

In Chapter 6, we lay the groundwork for the architectures presented in this dissertation. Our basic approach is to introduce a maximum amount of concurrency by using both pipelining and parallel processing. To access the data in parallel requires a distributed memory system. We presented a special

memory interleaving technique for high-bandwidth access to vectors of volumetric data. We showed that the pipelined ray-casting algorithm combined with arithmetic unit replication leads to a set of parallel rendering pipelines. We also looked at important issues in parallel processing such as data and task distribution and inter-processor communication. The chapter concludes with the three parallel ray-casting algorithms that lead to the three architectures presented in this dissertation.

Cube-3, presented in Chapter 7, implements ray-parallel processing where all rendering pipelines compute data along a single ray in parallel. We introduced sheared tri-linear interpolation that uses data coherency among neighboring rays during interpolation. We showed through various software experiments that the error introduced by this interpolation method is negligible, and that the 12-neighborhood ABC gradient estimation leads to accurate gradients without motion aliasing. We presented the Cube-3 architecture and discussed its shortcomings due to the global fast bus that interconnects memory and processing units.

In Chapter 8, we introduced the beam-parallel Cube-4 architecture that exploits the inherent parallelism among a group of rays that start on the same image scanline. We presented a novel beam-parallel dataflow that allows to locally connect memory and processing units and leads local communication with constant bandwidth between the processing units. Only the output of the compositing stage requires a global bus for the transport of pixels to the frame buffer. Together with a high-bandwidth memory system this novel approach offers an intriguing architectural advantage over existing techniques for ray-casting: The system performance can be scaled arbitrarily by increasing the number of memory and processing units. Performance of Cube-4 is only limited by current memory speeds.

In Chapter 9, we introduce the main result of this dissertation, the Cube-4 slice-parallel architecture. Slice-parallel processing operates on slices of the dataset which are retrieved by fetching consecutive voxel scanlines from the memory. The advantage of slice-parallel processing versus beam-parallel processing include correct tri-linear neighborhoods, simpler dataflow, and simpler implementation of perspective projections. We discussed the complete Cube-4 dataflow for parallel and perspective projections with the concise representation of signal flow graphs. And we presented results from a hardware implementation of Cube-4 on the Teramac custom computing machine at Hewlett-Packard Laboratories. Our results indicate linear scalability of performance with the number of rendering pipelines.

We believe that Cube-4 is an important contributions towards real-time visualization systems for volumetric data. It is a first step towards the design of add-on volume rendering accelerators for general-purpose machines. Additionally, the provision for direct data input without pre-processing may lead to special-purpose volume rendering hardware that can be embedded into real-time 3D acquisition devices.

10.1 Future Directions

Our future work has two main components. We intend to improve and enhance the Cube-4 functionality for volume rendering applications. A research contract with an industrial partner enables us to fabricate VLSI Cube-4 ASICs for further experimentation. A more remote goal is to generalize the architecture to include other applications such as texture mapping and image-based rendering.

First, we want to implement and simulate the proposed mechanism for

perspective ray-casting through slice scaling. In Chapter 9 we have outlined this approach and the resulting dataflow. However, this mechanism has not been implemented or simulated to test the resulting image quality. We intend to include perspective projections into our VHDL code and possibly into a future Cube-4 hardware prototype on the Teramac.

We found that signal flow graphs provide us with a useful and intuitive graphical representation of the dataflow. Their main advantage is that they are not a representation for a specific architecture, but rather for the dataflow of the underlying parallel algorithm. Structured modifications on the graph allow to simplify and modify the dataflow. Signal flow graphs have been extensively used in the literature for VLSI array processor design [KL80, Pet93]. For future enhancements of the Cube-4 dataflow we intend to use formal methods developed for array processor design, such as scheduling, projection, and retiming.

There are several techniques to map signal or data flow graphs onto SIMD, systolic, wavefront, or even MIMD array architectures [Kun88]. The Cube-4 architecture shown in Figure 81 is only one of many mappings of the SFG onto hardware. We intend to develop different architectures for different price-performance requirements by trading off speed for hardware complexity. Furthermore, we anticipate that our SFGs lead to the further improvement of efficient parallel volume rendering algorithms that do not require global voxel communication. This may lead to near real-time implementations of volume rendering on commercial parallel computers and traditional graphics accelerators.

We have a research contract with Japan Radio Corporation (JRC) to design and implement a Cube-4 VLSI prototype ASIC. We anticipate to integrate five parallel rendering pipelines into one ASIC, thereby providing the ideal building

block for single-chip or multi-chip Cube-4 implementations. We intend to design and fabricate a PC board containing four Cube-4 ASICs, 32 MBytes of DRAM, and a PCI bus interface. This will allow us to perform extensive experimentation with 256^3 datasets to gain more experience and insight into all system and software aspects associated with a real-time volume rendering accelerator board.

We foresee that Cube-4 will become a commercially interesting solution for volume rendering acceleration in PCs, workstations, and in combined acquisition-visualization systems. However, we also believe that the potential of the Cube-4 architecture goes beyond volume rendering applications. The development in computer graphics goes towards increased scene realism using texture mapping or image-based rendering [CW93, MB95, SD96]. We can consider the set of texture images (mipmapped or not) and the set of images with depth required for these applications as 3D volumetric datasets. It is no surprise that todays texture mapping hardware is used to approximate volume rendering. Conversely, we believe that our approach to volume rendering has great merits for texture mapping and image-based rendering.

Texture mapping involves calculation of texture coordinates, resampling, filtering, and blending of texture data into the frame-buffer [SKvW+92]. Image-based rendering mainly consists of image-warping and image compositing, where image-warping can be further decomposed into image transformation, resampling, and filtering operations [Wol90]. The similarities between texture mapping, image-warping, and volume rendering are obvious. We intend to include coordinate transformations and higher-order filtering operations into the SFG framework presented in this dissertation. We want to investigate and simulate the resulting dataflow and design an appropriate new

architecture based on our experiences with Cube-4. We believe that the advantages of Cube-4, such as high memory bandwidth, modularity, and scalable performance, will be of great value for future texture and image-based rendering systems.

The combination of high-quality real-time volume rendering, texture mapping, and image-based rendering is a natural step in the evolution of computer graphics. Computer graphics drawing primitives have traditionally included vectors and surfaces. The success of texture mapping established 3D sampled data as an additional important component of the rendering pipeline. As a long term goal, we intend to enhance the graphics pipeline by 3D frame-buffer hardware that supports resampling, filtering, gradient estimation, and rendering of 3D volumetric or 2D image data. We believe that the results of this dissertation are a first and admittedly small step in this direction.

References

[ACC+95] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computing. In *Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines*, pages 32–38, Napa, CA, April 1995.

[AHH+94] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A diversified system for volume visualization research and development. In *Proceedings of Visualization 94*, pages 31–38, Washington, DC, October 1994.

[AK90] J. Arvo and D. Kirk. Particle transport and image synthesis. In *Computer Graphics*, Proceedings of SIGGRAPH 90, pages 63–66, August 1990.

[ASK92] R. Avila, L. Sobierajski, and A. Kaufman. Towards a comprehensive volume visualization system. In *Proceedings of Visualization 92*, pages 13–20, Boston, MA, October 1992.

[Bat77] K. E. Batcher. The multidimensional access memory in STARAN. *IEEE Transactions on Computers*, C-26(2):174–177, February 1977.

[Ben95] M. Bentum. *Interactive Visualization of Volume Data*. PhD thesis, University of Twente, Enschede, The Netherlands, 1995.

[BFGS86] L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. In *Computer Graphics*, pages 29–37. Proceedings of SIGGRAPH 86, November 1986.

[BKPP92] R. Bakalash, A. Kaufman, R. Pacheco, and H. Pfister. An extended volume visualization system for arbitrary parallel projection. In *Proceedings of the 1992 Eurographics Workshop on Graphics Hardware*, Cambridge, UK, September 1992.

[Bli77] J. F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics*, 11(2):182–198, July 1977.

[Bli82] J. F. Blinn. Light refelction functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(3):21–29, July 1982.

[BSVN89] G. J. Brakenhoff, E. A. Van Spronsen, H. T. M. VanD̃er Voort, and N. Nanninga. Three-dimensional confocal fluorescence microscopy. *Methods in Cell Biology*, 30:379–398, 1989.

[BvS95] M. Bosma and J. van Scheltinga. Efficient super resolution volume rendering. Master's thesis, University of Twente, Faculty of Electrical Engineering, Enschede, The Netherlands, August 1995. TR EL-BSC-079N95.

[CB92] D. Cohen and R. Bakalash. The Conveyor – An interconnection device for parallel volumetric transformations. In A. Kaufman, editor, *Advances in Graphics Hardware VI*. Springer Verlag, Berlin, 1992.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Workshop on Volume Visualization*, pages 91–98, Washington, DC, October 1994.

[Chi93] T. Chiueh. A novel memory access mechanism for arbitrary-view-projection volume rendering. In *Proceedings of Supercomputing '93*, pages 690–698, Los Alamitos, CA, November 1993.

[CHKP94] T. Chiueh, T. He, A. Kaufman, and H. Pfister. Compression domain volume rendering. Technical Report TR.94.01.04R, State University of New York at Stony Brook, Computer Science Department, Stony Brook, NY 11794-4400, April 1994.

[CK95] D. Cohen and A. Kaufman. A 3D skewing and de-skewing scheme for conflict-free access to rays in volume rendering. *IEEE Transactions on Computers*, 44(5):707–710, May 1995.

[CKBB90] D. Cohen, A. Kaufman, R. Bakalash, and S. Bergman. Real-time discrete shading. *The Visual Computer*, 6(1):16–27, February 1990.

[CLL+88] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, May 1988.

[CLR82] B. Chor, C. E. Leiserson, and R. L. Rivest. An application of number theory to the organization of raster graphics memory. *Conf. Rec. 23rd Annual IEEE Symp. Foundations of Computer Science*, pages 92–99, 1982.

[CM92] R. Crawfis and N. Max. Direct volume visualization of three-dimensional vector fields. *Workshop on Volume Visualization*, pages 55–50, October 1992.

[CM93] B. Corrie and P. Mackerras. Parallel volume rendering and data coherence. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 23–26, San Jose, 1993.

[CN93] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical Report TR93-027, Department of Computer Science at the University of North Carolina, Chapel Hill, 1993.

[CPB+88] D. Chin, J. Passe, F. Bernard, H. Taylor, and S. Knights. The princeton engine: A real-time video system simulator. *IEEE Transactions on Consumer Electronics*, 34(2):285–297, May 1988.

[CS80] E. Catmull and A. R. Smith. 3D transformations of images in scanline order. In *Computer Graphics*, Proceedings of SIGGRAPH 80, pages 279–285, August 1980.

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, January 1982.

[CU92] G. Cameron and P. E. Underill. Rendering volumetric medical image data on a SIMD architecture computer. In *Proceedings of Third Eurographics Workshop on Rendering*, May 1992.

[CW93] S. E. Chen and L. Williams. View interpolation for image synthesis. In *Computer Graphics*, Proceedings of SIGGRAPH 93, pages 279–288, August 1993.

[DCH88] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988.

[DH92] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In A. Kaufman and W. L. Lorensen, editors, *Workshop on Volume Visualization*, pages 91–98, Boston, MA, October 1992.

[DNR90] S. Dunne, S. Napel, and B. Rutt. Fast reprojection of volume data. In *Proceedings of the 1st Conference on Visualization in Biomedical Computing*, pages 11–18, 1990.

[Dür88] M. J. Dürst. Letters: Additional reference to "marching cubes". *Computer Graphics, ACM Siggraph Quarterly*, 22(2), 1988.

[DWS88] M. Deering, S. Winner, and B. Schediwy. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Computer Graphics*, Proceedings of SIGGRAPH 88, pages 21–30, 1988.

[Elv92] T. T. Elvins. A survey of algorithms for volume visualization. *Computer Graphics*, 26(3):194–201, August 1992.

[FGH+85] H. Fuchs, J. Goldfeather, J. P. Hultquist, S. Spatch, J. D. Austin, F. P. Brooks, Jr., J. G. Eyles, and J. Poulton. Fast spheres, shadows, textures, transparencies and image enhancements in Pixel-Planes. *Computer Graphics*, 19(3):111–120, July 1985.

[FGR85] G. Frieder, D. Gordon, and R. A. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics & Applications*, 5(1):52–60, January 1985.

[FL85] J. M. Frailong and J. Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *Proceedings of the International Conference on Parallel Processing*, pages 276–283, August 1985.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, pages 948–960, 1972.

[Fly95] M. J. Flynn. *Computer Architecture – Pipelined and Parallel Processor Design.* Jones and Bartlett Publishers, 1995.

[FP81] H. Fuchs and J. Poulton. Pixel-planes: A VLSI-oriented design for a graphics engine. *VLSI Design*, 2(3):20–28, 3rd Quarter 1981.

[FPE+89] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-Planes 5: A heterogeneous multiprocessor graphic system using processor-enhanced memories. *Computer Graphics*, 23(3):79–88, July 1989.

[FR93] P. A. Fletcher and P. K. Robertson. Interactive shading for surface and volume visualization on graphics workstations. In *Proceedings of Visualization 93*, pages 291–298, San José, CA, October 1993. IEEE Computer Society Press.

[FvDFH90] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, 2nd edition, 1990.

[GL82] R. Gemballa and R. Lindner. The multiple-write bus technique. *IEEE Computer Graphics & Applications*, 2(7):33–41, September 1982.

[Gla90] A. S. Glassner, editor. *Graphics Gems V*, chapter Normal Coding, pages 257–264. Academic Press, Inc., New York, 1990.

[Gla95] A. S. Glassner. *Principles of Digital Image Synthesis*, volume one and two of *The Morgan Kaufman Series in Computer Graphics and Geometric Modeling.* Morgan Kaufman Publishers, Inc., first edition, 1995.

[Gol84] S. M. Goldwasser. A generalized object display processor architecture. *IEEE Computer Graphics & Applications*, 4(10):43–55, October 1984.

[Gos94] M. E. Goss. An adjustable gradient filter for volume visualization image enhancement. In *Graphics Interface '94*, pages 67–74, 1994.

[GPR+94] T. Guenther, C. Poliwoda, C. Reinhard, J. Hesser, R. Maenner, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *Proceedings of the 9th Eurographics Hardware Workshop*, pages 103–108, Oslo, Norway, September 1994.

[GR83] S. M. Goldwasser and R. A. Reynolds. An architecture for the real-time display and manipulation of three-dimensional objects.

In *Proceedings International Conference on Parallel Processing*, pages 269–274, Bellaire, Michigan, August 1983.

[GR85] D. Gordon and R. A. Reynolds. Image space shading of 3-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 29:361–376, 1985.

[GRB+85] S. M. Goldwasser, R. A. Reynolds, T. Bapty, D. Baraff, J. Summers, D. A. Talton, and E. Walsh. Physician's workstation with real-time performance. *IEEE Computer Graphics & Applications*, 5(12):44–57, December 1985.

[Gup81] S. Gupta. Architectures and algorithms for parallel updates of raster-scan displays. Technical report, Computer Science Department Carnegie-Mellon University, Pittsburgh, PA, December 1981. CMU-CS-82-111.

[Han90] P. Hanrahan. Three-pass affine transforms for volume rendering. *Computer Graphics*, 24(5):71–78, November 1990.

[HB86] K. H. Höhne and R. Bernstein. Shading 3D-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging*, MI-5(1):45–47, March 1986.

[HCN94] B. M. Hemminger, T. J. Cullip, and M. J. North. Interactive visualization of 3D medical image data. Technical report, University of North Carolina at Chapel Hill, Department of Radiology and Radiation Oncology, 1994. TR 94-027.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2*. Adam Hilger, Bristol and Philadelphia, 1988.

[HMK+95] J. Hesser, R. Männer, G. Knittel, W. Straßer, H. Pfister, and A. Kaufman. Three architectures for volume rendering. In *Proceedings of Eurographics '95*, Maastricht, The Netherlands, September 1995. European Computer Graphics Association.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[Hsu93] W. M. Hsu. Segmented ray-casting for data parallel volume rendering. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 7–14, San Jose, CA, 1993.

[IEE92] Special report on high-speed DRAMs. *IEEE Spectrum*, 29(10):34–57, October 1992.

[Jac85] D. Jackel. The graphic PARCUM system : A 3D memory based computer architecture for processing and display of solid models. *Computer Graphics Forum*, 4(4):21–32, 1985.

[Jac88] D. Jackel. Eine Speicherarchitektur zur Darstellung und Verarbeitung räumlicher Datenfelder. *ITG - Fachbericht 102, Digitale Speicher*, 1988.

[JJL96] S. D. Jordan, P. E. Jensen, and B. A. Lichtenbelt. An architecture for high-performance 2D image display. In *Proceedings of the 11th Eurographics Hardware Workshop*, Poitiers, France, August 1996. (to appear).

[JS88] D. Jackel and W. Strasser. Reconstructing solids from tomographic scans - the PARCUM II system. In A.A.M. Kuijk and

W. Strasser, editors, *Advances in Computer Graphics Hardware II*, pages 209–227. Springer-Verlag, Berlin, 1988.

[Kae96] H. Kaeslin. Architecture of VLSI circuits. Lecture Notes, Integrated Systems Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland, 1996.

[Kau91] A. Kaufman. *Volume Visualization*. IEEE Computer Society Press, 1991.

[KB88] A. Kaufman and R. Bakalash. Memory and processing architecture for 3D voxel-based imagery. *IEEE Computer Graphics & Applications*, 8(6):10–23, November 1988.

[KBC91] A. Kaufman, R. Bakalash, and D. Cohen. Viewing and rendering processor for a volume visualization system. In R. L. Grimsdale and W. Strasser, editors, *Advances in Graphics Hardware IV*, pages 171–178. Springer-Verlag, Berlin, 1991.

[KCY93] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *Computer*, 26(7):51–64, July 1993.

[KH84] J. T. Kajiya and T. Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(3):165–173, July 1984.

[KL80] H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. *Introduction to VLSI Systems*, 1980.

[KM96] U. Kanus and M. Meissner. Cube-4 – A volume rendering architecture for real-time visualization of high-resolution datasets. Master's thesis, State University of New York at Stony Brook, 1996.

[KMS+92] J. Kaba, J. Matey, G. Stoll, H. Taylor, and P. Hanrahan. Interactive terrain rendering and volume visualization on the Princeton Engine. In A. Kaufman and G. Nielson, editors, *Visulization '92 Conference Proceedings*, pages 349–355, Los Alamitos, CA, October 1992. IEEE Computer Society Press.

[KMS+96] U. Kanus, M. Meißner, W. Straßer, H. Pfister, A. Kaufman, R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Cube-4 implementations on the Teramac custom computing machine. In *Proceedings of the 11th Eurographics Hardware Workshop*, pages 133–143, Poitiers, France, August 1996. To appear in Computers & Graphics.

[Kni93] G. Knittel. VERVE: Voxel engine for real-time visualization and examination. In *Computer Graphics Forum*, volume 12, No. 3, pages 37–48, September 1993.

[Kni94] G. Knittel. A scalable architecture for volume rendering. In *Proceedings of the 9th Eurographics Hardware Workshop*, pages 58–69, Oslo, Norway, September 1994.

[KS86] A. Kaufman and E. Shimony. 3D scan-conversion algorithms for voxel-based graphics. In *Proceedings ACM Workshop on Interactive 3D Graphics*, pages 45–76, Chapel Hill, NC, October 1986.

[KS94] G. Knittel and W. Strasser. A compact volume rendering accelerator. In *Volume Visualization Symposium Proceedings*, pages 67–74, Washington, DC, October 1994. ACM Press.

[Kun88] S. Y. Kung. *VLSI Array Processors*. Prentice Hall, 1988.

[Lac95a] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform.* PhD thesis, Stanford University, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford, CA 94305–4055, 1995. CSL-TR-95-678.

[Lac95b] P. Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. In *Proceedings of the Parallel Rendering Symposium 1995*, pages 15–22, Atlanta, GA, 1995.

[LC87] W. E. Lorensen and H. E. Cline. Marching–cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics*, Proceedings of SIGGRAPH 87, pages 163–169, 1987.

[Lev88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.

[Lev90] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[Lev92] M. Levoy. Volume rendering using the fourier projection-slice theorem. In *Proceedings of Graphics Interface '92*, pages 61–69. Canadian Information Processing Society, 1992.

[LG95] L. Lippert and M. Gross. Fast wavelet based volume rendering by accumulation of transparent texture maps. In *Computer Graphics Forum*, Proceedings of Eurographics 95, pages C–431 – C–443, September 1995.

[LH91] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume renderi ng. *Computer Graphics*, 25(4):285–288, July 1991.

[LL94] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 451–457, July 1994.

[LP84] A. Levinthal and T. Porter. Chap - A SIMD graphic processor. *Computer Graphics*, 18(3):77–82, July 1984.

[Mal93] T. Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[Max95] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[MB95] L. McMillan and G. Bishop. Plenoptic modeling. In *Computer Graphics*, Proceedings of SIGGRAPH 95, pages 39–46, 1995.

[Mea82] D. Meagher. Efficient synthetic image generation of arbitrary 3D objects. In *Proceedings of IEEE Computer Society Conference on Pattern Recognition and Image Processing*, June 1982.

[MEP92] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. *Computer Graphics*, 26(2):231–240, July 1992.

[MHC90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. *Computer Graphics*, 24(5):27–34, November 1990.

[MK93] T. Malzbender and F. Kitson. A fourier technique for volume rendering. In H. Hagen, H. Müller, and G. M. Nielson, editors, *Focus on Scientific Visualization*, pages 305–316. Springer Verlag, 1993.

[MMSE91] H.-P. Meinzer, K. Meetz, D. Scheppelmann, and U. Engelmann. The Heidelberg ray tracing model. *IEEE Computer Graphics & Applications*, page 34, November 1991.

[MO74] R. Mersereau and A. Oppenheim. Digital reconstruction of multidimensional signals from their projections. *Proceedings of the IEEE*, 62(10):1319–1338, October 1974.

[MPHK93] K. Ma, J. Painter, C. Hansen, and M. Krogh. A data distributed parallel algorithm for ray-traced volume rendering. In *1993 Parallel Rendering Symposium Proceedings*, pages 15–22. ACM Press, October 1993.

[MPHK94] K. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics & Applications*, 14(4):59–68, 1994.

[MPS92] C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. *Workshop on Volume Visualization*, pages 9–16, October 1992.

[Mur93] S. Muraki. Volume data and wavelet transform. *IEEE Computer Graphics & Applications*, 13(4):50–56, July 1993.

[Neu93a] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *1993 Parallel Rendering*

Symposium Proceedings, pages 97–104, San Jose, CA, October 1993.

[Neu93b] U. Neumann. *Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis*. PhD thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1993.

[NH92] P. Ning and L. Hesselink. Vector quantization for volume rendering. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 69–74, October 1992.

[NH93] P. Ning and L. Hesselink. Fast volume renderign of compressed data. In *Proceedings of Visualization '93*, pages 11–18, October 1993.

[NL92] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory MIMD architectures. *Workshop on Volume Visualization*, pages 17–24, October 1992.

[NM87] A. Norton and E. Melton. A class of boolean linear transformations for conflict-free power-of-two stride access. In *Proceedings of the International Conference on Parallel Processing*, pages 247–254, University Park, PA, August 1987.

[PD84] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(3), July 1984.

[Pet93] N. Petkov. *Systolic Parallel Processing*, volume 5 of *Advances in Parallel Computing*. North-Holland, 1993.

[Pho75] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 1(18):311–317, June 1975.

[PK96] H. Pfister and A. Kaufman. Cube-4 – A scalable architecture for real-time volume rendering. In *1996 ACM/IEEE Symposium on Volume Visualization*, pages 47–54, San Francisco, CA, October 1996.

[PKC94] H. Pfister, A. Kaufman, and T. Chiueh. Cube-3: A real-time architecture for high-resolution volume visualization. In *1994 ACM/IEEE Workshop on Volume Visualization*, pages 75–83, Washington, DC, October 1994.

[PN85] G. F. Pfister and V. A. Norton. 'Hot Spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[PW96] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[PWK94] H. Pfister, F. Wessels, and A. Kaufman. Sheared interpolation and gradient estimation for real-time volume rendering. In *Proceedings of the 9th Eurographics Hardware Workshop*, pages 70–79, Oslo, Norway, September 1994.

[PWK95] H. Pfister, F. Wessels, and A. Kaufman. Cube-4: A scalable architecture for real-time volume rendering. Technical Report TR.95.01.15, State University of New York at Stony Brook, Computer Science Department, Stony Brook, NY 11794-4400, January 1995.

[RGC87] R. A. Reynolds, D. Gordon, and L.-S. Chen. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38(3):275–298, 1987.

[Sab88] P. Sabella. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics*, 22(4):59–64, August 1988.

[SD96] S. M. Seitz and C. R. Dyer. View morphing. In *Computer Graphics*, Proceedings of SIGGRAPH 96, pages 21–30, August 1996.

[SF90a] M. R. Stytz and O. Frieder. Three-dimensional medical imaging modalities: An overview. *CRC Critical Reviews in Biomedical Engineering*, July 1990.

[SF90b] K. R. Subramanian and D. S. Fussell. Applying space sudivision techniques to volume rendering. In *Proceedings of Visualization '90*, pages 150–159, San Francisco, CA, 1990.

[SF91] M. R. Stytz and O. Frieder. Computer systems for three-dimensional diagnostic imaging: An examination of the state of the art. *Critical Reviews in Biomedical Engineering*, pages 1–46, August 1991.

[SFF91] M. R. Stytz, G. Frieder, and O. Frieder. Three-dimensional medical imaging: Algorithms and computer systems. *ACM Computing Surveys*, pages 421–499, December 1991.

[SK90] D. Speary and S. Kennon. Volume probes: Interactive data exploration on arbitrary grids. *Computer Graphics*, 24:5–12, 1990.

[SK94] L. Sobierajski and A. Kaufman. Volumetric ray tracing. In *Volume Visualization Symposium Proceedings*, pages 11–19, Washington, DC, October 1994.

[SKvW+92] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. In *Computer Graphics*, Proceedings of SIGGRAPH 92, pages 249–252, July 1992.

[SM93] R. L. Sollenberg and P. Milgram. Effects of stereoscopic and rotational displays in a three-dimensional path tracing task. In *Human Factors*, pages 483–499, 1993.

[Sob94] L. Sobierajski. *Global Illumination Models for Volume Rendering.* PhD thesis, State University of New York at Stony Brook, Computer Science Department, Stony Brook, NY 11794-4400, August 1994.

[Sob95] I. Sobel. An isotropic 3x3x3 volume gradient operator. Unpublished manuscript, May 1995.

[Sra94] M. Sramek. Fast surface rendering from raster data by voxel traversal using chessboard distance. In *Proceedings of Visualization '94*, pages 188–195, Washington, D.C., October 1994.

[SS92] P. Schröder and G. Stoll. Data parallel volume rendering as line drawing. In *1992 Workshop on Volume Visualization*, pages 25–31, Boston, MA, October 1992.

[TL93] T. Totsuka and M. Levoy. Frequency domain volume rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 93, pages 271–278, 1993.

[TT84] H. K. Tuy and L. T. Tuy. Direct 2-D display of 3D objects. *IEEE Computer Graphics & Applications*, 4(10):29–33, November 1984.

[UH84] J. K. Udupa and G. T. Herman. *3D Imaging in Medicine*. CRC Press, 1984.

[UK88] C. Upson and M. Keeler. V-BUFFER: Visible volume rendering. *Computer Graphics*, 22(4):59–64, August 1988.

[VFR92] G. Vézina, P. Fletcher, and P. Robertson. Volume rendering on the MasPar MP-1. In *1992 Workshop on Volume Visualization*, pages 3–8, Boston, MA, October 1992.

[Wan95] S. Wang. *Synthesis, Manipulation, and Rendering of Volume-Sampled Geometric Models*. PhD thesis, State University of New York at Stony Brook, Computer Science Department, Stony Brook, NY 11794-4400, May 1995.

[Wes89] L. Westover. Interactive volume rendering. In C. Upson, editor, *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 9–16, Chapel Hill, NC, May 1989. University of North Carolina at Chapel Hill.

[Wes90] L. Westover. Footprint evaluation for volume rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 90, pages 367–376, August 1990.

[Wes91] L. A. Westover. *Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm*. PhD thesis, The University of North Carolina at Chapel Hill, Department of Computer Science, July 1991. Technical Report, TR91-029.

[Wes95] F. Wessels. Hardware designs for volume visualization. Technical Report TR.95.01.13, State University of New York at Stony Brook, Computer Science Department, Stony Brook, NY 11794-4400, January 1995. M.S. Thesis.

[WG91] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 91, pages 275–284, 1991.

[WH94] C. Wittenbrink and M. Harrington. A scalable MIMD volume rendering algorithm. In *Eighth International Parallel Processing Symposium*, pages 916–920, Cancun, Mexico, April 1994.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 6(23):343–349, June 1980.

[Wit93] C. Wittenbrink. *Designing Optimal Parallel Volume Rendering Algorithms*. PhD thesis, University of Washington, 1993.

[WK93] S. Wang and A. Kaufman. Volume sampled voxelization of geometric primitives. In *Proceedings of Visualization '93*, pages 78–84, San José, CA, October 1993. IEEE Computer Society Press.

[WK94] S. Wang and A. Kaufman. Volume-sampled 3D modeling. *IEEE Computer Graphics & Applications*, 14(5):26–32, September 1994.

[Wol90] G. Wolberg. *Digital Image Warping.* IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Woo90] D. A. Wood. *The Design and Evaluation of In-Cache Address Translation.* PhD thesis, University of California, Berkeley, Computer Science Division, California 94720, 1990. Report No. UCB/CSD 90/565.

[Wor91] J. Worlton. Toward a taxonomy of performance metrics. *Parallel Computing*, 17, December 1991.

[WS93] C. Wittenbrink and A. Somani. Permutation warping for data parallel volume rendering. In *Parallel Rendering Symposium, Visualization '93*, pages 57–60, San Jose, CA, 1993.

[Yag91a] R. Yagel. *Efficient Methods for Volumetric Graphics.* PhD thesis, State Univeristy of New York at Stony Brook, December 1991.

[Yag91b] R. Yagel. The flipping CUBE: A device for rotating 3D rasters. *6th Eurographics Hardware Workshop*, September 1991.

[YC94] R. Yagel and K. Ciula. High quality template-based volume rendering. Technical report, Ohio State University, 1994. OSU–CISRC-3/94-TR17.

[YCK92] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics & Applications*, pages 19–28, September 1992.

[YK92] R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum, Proceedings Eurographics*, 11(3):153–167, September 1992.

[YL95] B.-L. Yeo and B. Liu. Volume rendering of dct-based compressed 3D scalar data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):29–43, March 1995.

[YNF+92] T. S. Yoo, U. Neumann, H. Fuchs, S. M. Pizer, T. Cullip, J. Rhoades, and R. Whitaker. Direct visualization of volume data. *IEEE Computer Graphics & Applications*, 12(4):63–71, July 1992.

[ZH81] S. W. Zucker and R. A. Hummel. A three-dimensional edge operator. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(3):324–331, May 1981.

[ZKV92] K. Z. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Acceleration of ray catsing using 3D distance transform. In R. A. Robb, editor, *Proceedings of Visualization in Biomedical Computing*, pages 324–335, Chapel Hill, NC, October 1992. SPIE, Vol. 1808.