RAY TRACING IMPLICIT SURFACES FOR INTERACTIVE VISUALIZATION

by

Aaron M. Knoll

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2009

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Aaron M. Knoll

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Charles D. Hansen

Steven G. Parker

Ingo Wald

Claudio Silva

Hans Hagen

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Aaron M. Knoll in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Charles D. Hansen
Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David Chapman
Dean of The Graduate School

This dissertation is dedicated to Dr. Isaac Knoll, M.D., 1907-2003.

My grandfather belonged largely to the era before the microchip. He might not have understood the merit of computer science as an academic field, nor of graphics and visualization as topics of research. However, his steadfast commitment to education made my undergraduate and master's studies possible, and without question inspired me to pursue my doctorate. As the first member of my family to earn a PhD, I dedicate this work to his memory, in honor of the sacrifices he made in pursuit of my scholarship.

ABSTRACT

Implicit surfaces are useful representations of geometry in visualization. Since geometric data are nearly always discrete, it is typically desirable to pair them with a continuous reconstruction filter. An isosurface is a manifestation of that filter in implicit form. Traditionally, the preferred methods for rendering isosurfaces have been to extract and rasterize a triangle mesh, or to resample the implicit as simpler proxy geometry such as splats. Ray casting allows for direct and pixel-exact rendering of an implicit surface by root-solving for the intersection. Full ray tracing methods pair this intersection process with a spatial acceleration structure, such as a bounding volume hierarchy or octree. While traditionally slow compared to rasterization, this approach has several advantages. It enables multiple-bounce effects such as shadows and reflections, and allows for dynamic changes in the implicit without offline processing. More significantly for visualization, acceleration structures are traversed in logarithmic time, allowing for graceful scaling to large data. This dissertation presents several advancements in ray tracing implicit surfaces and its applications, namely single-ray and coherent multiresolution methods for isosurface ray tracing of octree-compressed large structured data, a coherent method for isosurface rendering of tetrahedral meshes, and algorithms for rendering arbitrary implicit forms robustly using interval and affine arithmetic. While computationally costly, techniques such as these map well to modern multicore CPUs and thread-parallel GPUs, and will continue to improve in efficiency and applicability as parallel hardware trends evolve.

CONTENTS

ABSTRACT . . . . . **iv**

LIST OF FIGURES . . . . . **xi**

LIST OF TABLES . . . . . **xiii**

ACKNOWLEDGMENTS . . . . . **xiv**

CHAPTERS

1. INTRODUCTION **1**

2. BACKGROUND AND RELEVANT WORK **6**

2.1 Implicit Surfaces 6
2.2 Mesh Extraction 9
2.3 Point-Based Methods 10
2.4 Volume Rendering 11
2.5 Ray Casting and Ray Tracing Implicits 12
2.5.1 Ray Tracing General-Form Implicits 13
2.5.2 Ray Tracing Isosurfaces 14

3. ISOSURFACE RAY TRACING OF OCTREE VOLUME DATA **16**

3.1 Motivation 16
3.1.1 Octree Traversal 17
3.1.2 Octree Hashing 18
3.2 Algorithm Overview 18
3.2.1 Constructing an Octree Volume 18
3.2.2 Ray Traversal and Voxel-Cell Duality 19
3.2.3 Computing the Min/Max Tree 21
3.3 Implementation 22
3.3.1 Data Format 23
3.3.2 Octree Data Lookup 24
3.3.2.1 Point location 25
3.3.2.2 Neighbor finding 25
3.3.3 Ray-Octree Traversal 25
3.3.3.1 Interior nodes 26
3.3.3.2 Cap nodes 27
3.3.3.3 Scalar leaves 27
3.3.4 Isosurface Intersection 27
3.3.5 Shading and Filling the Frame Buffer 28
3.4 Octree Construction Results 28
3.4.1 Data Compression 28

3.4.2 Further Compression . . . 29
3.4.3 Construction Performance . . . 30
3.4.4 Memory Footprint Comparison . . . 30
3.5 Rendering Results . . . 31
3.5.1 Comparison to Hierarchical Grid . . . 31
3.5.2 Richtmyer Meshkov Instability Results . . . 31
3.5.3 Scalability . . . 33
3.5.4 Time-Variant Volumes . . . 33

4. COHERENT MULTIRESOLUTION ISOSURFACE RAY TRACING OF OCTREE VOLUMES . . . 35

4.1 Motivation . . . 35
4.1.1 Coherence via Level of Detail . . . 36
4.1.2 System Overview . . . 36
4.2 Multiresolution Octree Volume Construction . . . 37
4.2.1 Extension to Multiresolution . . . 38
4.2.2 Modifications to Min/Max Tree Computation . . . 38
4.3 Coherent Octree Volume Ray Tracing . . . 39
4.3.1 SSE Packet Architecture . . . 40
4.3.2 Coherent Traversal Background . . . 40
4.3.3 Coherent Grid Traversal Algorithm . . . 40
4.3.4 Macrocell Hierarchical CGT . . . 41
4.3.5 Implicit Macrocell Grid Traversal of Octree Volumes . . . 42
4.3.6 Mapping Macrocells to Octree Nodes . . . 42
4.3.7 Default Slice-Based Traversal . . . 43
4.3.7.1 Culling empty cap-level macrocells . . . 44
4.3.7.2 Clipping the cell-level slice to active rays . . . 44
4.3.8 Cell Reconstruction from Cached Voxel Slices . . . 45
4.3.9 U,V Voxel Slice Filling . . . 45
4.3.10 Copying the Previous-Step $\vec{K}$-Slice . . . 47
4.3.11 Ray-Cell Intersection . . . 47
4.4 Multiresolution Level of Detail System . . . 47
4.4.1 Resolution Heuristic . . . 48
4.4.1.1 Stop depth . . . 48
4.4.1.2 Pixel-to-voxel width ratio . . . 48
4.4.1.3 Packet extents metric . . . 49
4.4.1.4 LOD mapping via $\vec{K}$ transition slices . . . 49
4.4.2 Multiresolution Traversal . . . 49
4.4.3 Smooth Transitions . . . 50
4.5 Shading . . . 51
4.5.1 Smooth Gradient Normals . . . 51
4.5.2 Shadows . . . 52
4.6 Results . . . 53
4.6.1 Coherent Traversal Analysis . . . 53
4.6.2 Packet Size . . . 54
4.6.3 Incoherent Behavior Without Multiresolution . . . 54
4.6.4 Multiresolution Results . . . 54
4.6.5 Overall Performance . . . 56
4.6.6 Comparison to Existing Systems . . . 58

4.6.7 Quality Comparison 60

5. COHERENT ISOSURFACE RAY TRACING OF TETRAHEDRAL MESHES 62

5.1 Motivation 62
5.2 System Overview 64
5.3 Isosurface Intersection 65
5.3.1 Extracting the Isopolygon 65
5.3.2 Ray-Isopolygon Intersection 66
5.3.3 SIMD Frustum Culling 67
5.3.4 Isopolygon Precomputation 67
5.4 The Implicit Bounding Volume Hierarchy 68
5.4.1 Building the BVH 68
5.4.2 Implicit BVH Traversal 69
5.4.2.1 Implicit culling 69
5.4.2.2 Speculative first-active descent 70
5.4.2.3 Frustum test 70
5.4.2.4 First-active ray tracking 70
5.4.2.5 Leaf traversal 72
5.5 Time-Varying Data 72
5.5.1 Schema I: Unique BVH Per Step 72
5.5.2 Schema II: Dynamic Refitting 72
5.6 Shading and Interaction Modalities 73
5.6.1 Shadows 73
5.6.2 Multiple Isosurfaces 73
5.6.3 Clipping Planes and Boxes 74
5.6.4 Transparent Depth Peeling 75
5.7 Results 75
5.7.1 Build Time and Performance 75
5.7.2 Rendering Performance 78
5.7.3 Scalability in Model Size 78
5.7.4 Traversal Efficiency 78
5.7.5 Multiple Isosurfaces, Shadows, and Transparency 80
5.7.6 Time-Varying Data Sets 80
5.7.7 Memory Overhead 81
5.7.8 Comparison to Existing CPU Based Approaches 82
5.7.9 Comparison to Existing GPU Based Approaches 82

6. RAY TRACING ARBITRARY-FORM IMPLICIT SURFACES 84

6.1 Motivation 84
6.2 Background 86
6.2.1 Ray Tracing Implicit Surfaces 86
6.2.2 Interval Arithmetic and Inclusion 87
6.2.3 Affine Arithmetic 89
6.2.4 Condensation and Reduced Affine Arithmetic 91
6.2.5 Inclusion-Preserving Reduced Affine Arithmetic 92
6.2.6 Ray Tracing Implicits with Inclusion Arithmetic 92
6.3 SIMD CPU Ray Tracing Algorithm 93
6.4 SIMD CPU Implementation 95
6.4.1 SSE Interval Arithmetic 96

6.4.2 Ray Packet Structure . . . 96
6.4.3 Traversal . . . 96
6.4.4 Division . . . 97
6.4.5 Precision Criterion . . . 97
6.4.6 Shadows . . . 99
6.4.7 Gradient Computation . . . 99
6.5 GPU Algorithm . . . 100
6.5.1 Application Pipeline . . . 101
6.5.2 Shader IA Library . . . 102
6.5.3 Shader RAA Library . . . 102
6.5.4 Numerical Considerations . . . 103
6.5.5 Traversal . . . 103
6.5.5.1 Initialization . . . 103
6.5.5.2 Rejection test . . . 104
6.5.5.3 Main loop . . . 104
6.5.5.4 Back-recursion loop . . . 104
6.5.6 Traversal Metaprogramming . . . 105
6.5.7 Shading . . . 105
6.6 Results . . . 105
6.6.1 Performance . . . 105
6.6.2 IA versus RAA . . . 105
6.6.3 Error and Quality . . . 106
6.6.4 Algorithm Coherence and Performance . . . 107
6.6.5 Feature Reproduction and Robustness . . . 109
6.6.6 Shading Modalities . . . 110
6.6.6.1 Shadows . . . 110
6.6.6.2 Transparency . . . 110
6.6.6.3 Multiple isosurfaces . . . 110
6.6.6.4 Reflections . . . 111
6.6.7 Applications . . . 112
6.6.7.1 Mathematical visualization . . . 112
6.6.7.2 Interpolation, morphing and blending . . . 112
6.6.7.3 Constructive solid geometry . . . 113
6.6.7.4 Procedural geometry . . . 113

7. CONCLUSION AND FUTURE WORK . . . 116

7.1 Isosurface Ray Tracing of Octree Volumes . . . 116
7.2 Coherent Multiresolution Isosurface Ray Tracing of Octree Volumes . . . 117
7.3 Isosurface Ray Tracing of Tetrahedral Volume Data . . . 119
7.4 Ray Tracing Arbitrary Implicit Surfaces . . . 120
7.5 Conclusion . . . 122

APPENDICES

A. OCTREE VOLUME HASHING AND TRAVERSAL . . . 124

B. ARBITRARY-FORM IMPLICITS **130**

REFERENCES **136**

LIST OF FIGURES

3.1 Voxel-cell duality and octree traversal 20
3.2 Retrieving a cell from a neighborhood of voxels. 20
3.3 Min/max tree construction from forward neighbors 21
3.4 Octree volume format illustrated 24
3.5 Ray traversing an octree node 26
3.6 Richtmyer Meshkov instability 32
3.7 Scalability with octree and grid ray tracing 33
3.8 Time-variant isosurface rendering. 34
4.1 Overview of coherent octree volume ray tracing 37
4.2 Voxel-cell mapping 38
4.3 Min/max tree construction for multiresolution 39
4.4 Coherent grid traversal 41
4.5 Coherent octree traversal via implicit macrocells 43
4.6 Culling empty macrocells from cap-node slices 44
4.7 Clipping cell slices to fit active rays 45
4.8 Slice-based cell reconstruction algorithm 46
4.9 Multiresolution transitions illustrated. 50
4.10 Shading with foward and central differences 52
4.11 Ray tracing with shadows 53
4.12 Qualitative impact of multiresolution 56
4.13 The visible female femur 57
4.14 Results for the coherent multiresolution system 59
4.15 Comparing multiresolution and single-resolution results on the RM data. 61
5.1 Ray-isopolygon intersection in an isotetrahedron 66
5.2 Implicit culling in the BVH 69
5.3 First-active descent, frustum test, and active ray tracking 71
5.4 Additional shading effects 74
5.5 Benchmark scenes for the BVH 76
5.6 Two examples of time-varying tet data sets 77

5.7 Large data and scalability 79
5.8 Teaser scenes from the tet mesh isosurface ray tracer 83
6.1 The inclusion property 88
6.2 Bounding forms resulting from the combination of two interval and affine quantities 90
6.3 Barth sextic surface on the CPU 94
6.4 Spatial interval bisection methods 95
6.5 Handling division with IA 98
6.6 The Klein bottle rendered using SIMD IA bisection 99
6.7 Gradient normal computation 100
6.8 The Mitchell and Barth Decic surfaces at various ε 108
6.9 Fine feature visualization in the Steiner surface 109
6.10 Shading Effects 111
6.11 The Barth sectic and decic surfaces 112
6.12 4D morphing example 113
6.13 CSG using inequalities on 3-manifold solids. 114
6.14 Sinusoid procedural geometry 114
6.15 An animated sinsuoid-kernel surface. 115

LIST OF TABLES

3.1 Compression achieved for various structured data 29
3.2 Octree-grid comparison 31
3.3 Frame rates of various time steps of the LLNL Richtmyer Meshkov data 32
3.4 Frame rates for the CSAFE heptane data 34
4.1 Results from clipping optimizations 55
4.2 Results with coherent packets 55
4.3 Richtmyer-Meshkov reference images for the coherent multiresolution system 58
5.1 BIH-style build vs. SAH for building the implicit BVH 76
5.2 Performance in frames per second for various data sets and platforms 78
5.3 Bucky ball replication and scalability 79
5.4 Traversal statistics of the frustum method. 80
5.5 Memory usage and BVH overhead 81
5.6 GPU performance comparison 83
6.1 Implicit surface performance on the CPU and GPU 106
6.2 Performance of various algorithms 107
B.1 Formulas of simple test surfaces used in Table 6.1. 134
B.2 Formulas of more complicated implicit surfaces used in Table 6.1. 135

ACKNOWLEDGMENTS

I first thank my wife Jeanmarie for the five years of our lives during which I pursued this doctorate. Two of these years we spent physically apart, while she pursued her own graduate studies at the University of Texas at Austin and Brown University. Not unexpectedly, our time together was often subject to the associated stresses of academic families with a two-body problem. Nonetheless, I wouldn't trade that time for any other in my life.

I cannot understate my thanks to my undergraduate colleagues Jamie Im and J Duncan, who were PhD students well before I was, and who provided invaluable advice and above all long-term friendship, which is so difficult to accomplish let alone maintain in our career-obsessed, relocation-prone existence.

My parents Mark and Marlene and my sister Laura also provided continual encouragement, particularly when I doubted my own abilities. Academia can be an introverted environment in which one places undue emphasis on the judgments of select peers. While starry-eyed optimism alone cannot sustain an academic career, a strong dose of it from supportive family members certainly doesn't hurt.

At the University of Utah, I thank (in no particular order) Mathias Schott, Andrew Kensler, Abe Stephens, Guo-Shi Li, Thiago Ize, Vincent Pegoraro, Roni Choudhoury, Kristi Potter, Miriah Meyer, Won-ki Jeong, and Louis Bavoil, indeed all of CS and SCI. Peter Djeu and Warren Hunt at the University of Texas, Austin; and Younis Hijazi and virtually everyone in the IRTG of the University of Kaiserslautern, were instrumental in helping me complete this dissertation. I would like to thank the professors on my committee, as well as Bill Mark, Chris Wyman, Alex Reshetov, John C. Hart, David Laidlaw, and numerous faculty mentors who helped me to develop my research interests.

My work has been supported primarily by the US Department of Energy, Los Alamos National Laboratory, the DOE VIEWS and VACET and CSAFE projects, and the German Science Foundation (Deutsche Forschungsgemeinschaft, or DFG). Specifically, funding has come from the U.S. Department of Energy through CSAFE grant W-7405-ENG-48, the National Science Foundation under CISE grants CRI-0513212, CCF-0541113, and SEII-

0513212, and NSF-CNS 0551724; and the US Deparment of Energy SciDAC VACET, Contract No. DE-FC02-06ER25781 (SciDAC VACET); and the DFG through the University of Kaiserslautern International Research Training Group (IRTG) 1131 for travel expenditures and collaborative opportunities. Additional thanks go to the National Science Foundation funding for the Center of Excellence for Interactive Ray Tracing and Photo Realistic Vision, for development support of the Manta software architecture; and to Mark Duchaineau of Lawrence Livermore National Laboratory for access to the Richtmyer-Meshkov Instability simulation data.

Finally, I would like to give special thanks to Chuck for his patience with an often headstrong student.

CHAPTER 1

INTRODUCTION

Interactive rendering of large structured and unstructured volume data is a persistent problem in visualization. Due to continual advancements in parallel computing, simulation data can be generated with more numerous time steps and higher fidelity. These simulation data model a wide range of real-world problems. Structured data computed on a regular grid are typically generated by finite difference or finite volume simulations for applications such as modeling climate, computational combustion for minimizing pollution, and reducing turbulence inside turbines for power generation. Unstructured data, such as tetrahedral or hexahedral meshes computed using finite elements methods, represent connectivity and adapt to irregular geometry, and can be applied in earthquake prediction, automotive crash worthiness, and aircraft design. In addition to simulation sources, volume data can be scanned with numerous devices such as x-ray computed tomography or magnetic resonance imaging machinery. These data are critical in a broad range of medical diagnoses, as well as security screening and nondestructive testing. Like computational simulations, modern scanning technologies increasingly employ higher resolution for greater fidelity.

To visualize large data sets, regardless of whether they are scanned or simulated, structured or unstructured, it is necessary to employ rendering methods that scale gracefully with higher geometric complexity. Conventional rasterization methods for fixed-function graphics hardware are of object-order time complexity, and encounter difficulty when the number of objects in a data set is large, particularly in the case of structured data with cubic object complexity. While unstructured data have better geometric adaptivity and compactness, scalable and efficient reconstruction and interpolation of tetrahedral and irregular kernels remain a challenge.

Implicit surfaces are useful geometric representations for volume visualization. In most analysis of spatial data, discrete samples are paired with a filter to generate a continuous field function. Isosurfacing consists of rendering an implicit surface defined by that filter and an

offset isovalue. While direct volume rendering is useful in visualizing global properties of these fields, isosurfaces are powerful in understanding local features, and the smoothing or interpolating behavior of the chosen filter. Moreover, in rendering large volume data, isosurfacing paired with efficient spatial culling [134] effectively reduces geometric complexity from N^3 to N^2, which in turn reduces the time complexity of rendering.

Accurate rendering of local implicit surface geometry is an equally important problem. Even in the absence of filtered discrete data, an implicit surface may be of arbitrary functional form which can be nontrivial to render correctly. Generally, this surface must be either ray-cast directly or rasterized via approximating geometry. The most popular method for rendering implicits has been to extract and rasterize a triangle mesh using marching cubes [139, 73] and more sophisticated variants. While many mesh extraction methods exist with varying quality and efficiency, their time complexity is on the order of the number of samples of the underlying implicit, which can become quite large particularly in the case of structured data in three dimensions. Nonpolygonal proxy geometries, such as disc splats or volumes, can encounter similar issues, when the sampling rate of the proxy is beneath the Nyquist limit of the filtered implicit.

Ray tracing algorithms allow for rendering implicit surfaces directly without any proxy representation, by sampling at each pixel in the output image. Ray casting isosurfaces involves solving for the root where the ray intersects the implicit surface. With an appropriate numerical method for the implicit form in question, this guarantees correct rendering of the surface, regardless of camera position, and enables on-the-fly changes in isovalue. Although the root-solving process is not as fast as rasterization of simpler proxy geometry, it can nonetheless be done efficiently and dynamically. Moreover, since the advent of programmable graphics processing unit (GPU) hardware, it has also proven effective to rasterize a bounding proxy, and employ a ray-casting intersection routine in a fragment shader. True ray tracing methods involve construction and traversal of a spatial acceleration structure, such as a grid, octree, kd-tree or bounding volume hierarchy. In the case of implicit primitives, the acceleration structure can be specialized to accommodate the data and filter in question. Ultimately, acceleration structures allow each pixel to be rendered at roughly $O(\log N)$ cost with respect to the number of geometric primitives. For sufficiently complex objects, and particularly volume data, this can make a parallel ray tracing approach more efficient than rasterization approaches, which in principle cost $O(N)$ for the entire frame.

The central premise of this dissertation is that ray tracing implicit surfaces is a powerful and efficient method for certain problems in visualization, and will continue to be so as graphics hardware evolves. In support of this statement, we specifically consider three problems: isosurface visualization of large structured data, isosurface visualization of unstructured data, and rendering of arbitrary implicit forms. The main technical components of our work involve efficient acceleration structure construction and traversal for volume visualization, and an efficient general-purpose intersection routine for general-form implicit surfaces. Specifically, we make the following contributions:

- **Isosurface ray tracing of large octree volume data.**
  In visualizing large volume data, we can employ the octree as both a data compression mechanism and acceleration structure for more efficient rendering, in conjunction with an implicit surface for voxel primitives as proposed by Parker et al. [90]. This permits rendering of multiple-gigabyte volume data that could not trivially be handled by a GPU direct volume renderer or dynamic isosurface extraction. Even with a conventional single-ray traversal approach, this system performs interactively on a multicore workstation, and outperforms ray tracing the same large data on a distributed cluster [19]. These techniques are presented in Chapter 3.

- **Multiresolution isosurface ray tracing using ray packets**
  In Chapter 4, we seek to optimize isosurface ray tracing of large volumes using coherent ray packets. We adapt the coherent grid traversal method of Wald et al. [130] to hierarchical grids and octrees, and find this technique can deliver greater performance, but at the cost of scalability. To remedy this, we turn to the multiresolution voxel representation inherent in the octree, and adapt our traversal algorithm to multiresolution isosurface reconstruction. While isosurfaces can indeed vary between resolutions, smoothing can actually be desirable in visualization, and the resulting system can achieve interactive performance on laptops and commodity desktops.

- **Isosurface ray tracing of tetrahedral volume data.**
  While grids and octrees are well-suited for structured volumes, isosurfacing unstructured polygonal or simplicial volume data benefits from acceleration structures that adapt to irregular geometry. We build and traverse an implicit bounding volume

hierarchy on top of a tetrahedral mesh. As the local implicit surfaces of first-order finite elements are inherently piecewise-linear, we can employ a specialized intersection algorithm inspired by the marching tetrahedra extraction method, in conjunction with a packet-polygon intersection test. Coherent BVH traversal and SSE enhancements allow for interactive ray tracing of large tetrahedral data sets on a desktop. This method is covered in Chapter 5.

- **Fast and robust ray intersection with arbitrary implicits.**
  Rendering general-form algebraic surfaces, first proposed by Hanrahan [39], is a classic problem in ray casting. Unlike point-sampling approaches, self-validated arithmetic methods allow for robust rendering of arbitrary implicit forms. Interval arithmetic is a well-known method of quantifying numerical error in computation [83]. Mitchell [82] first applied interval arithmetic to ray tracing implicit surfaces to guarantee robust intersection Improved inclusion methods such as affine arithmetic have also been employed in rendering [15, 17]. However, these methods have historically proven slow. We note that the numerical precision delivered by the root refinement algorithm of [82] is rarely required for visual accuracy. Moreover, by employing many of the same coherent SIMD optimizations as acceleration structure traversal algorithms, we can render closed-form algebraic and non-algebraic surfaces interactively on both the CPU. On the GPU, a stackless traversal algorithm proves effective for instruction parallel hardware. We illustrate these methods in detail in Chapter 6.

The practical efficiency of these systems is owed in no small measure to changes in CPU and GPU architecture. Partly because geometric complexity is seldom higher than pixel complexity, but also because due to the efficiency of the Z-buffer algorithm on fixed-function graphics hardware, ray tracing has until recently proven prohibitively slow compared to rasterization for most rendering tasks. However, over the course of the works in this dissertation, CPU hardware has changed dramatically, facilitating ray tracing techniques that were interactive only on small supercomputers in the late 1990s, such as the work of Parker et al. [90]. Moore's Law has ensured significant improvements in performance, but the most dramatic advances have come from thread-parallelism and SIMD vector instructions, coupled with efficient shared memory and cache. The proliferation of multicore CPUs in particular has enabled interactive ray tracing on mainstream hardware, particularly for appli-

cations involving rendering of large datasets. The first four contributions of this dissertation leverage the growing power of inexpensive shared-memory workstations, and more recently commodity multicore CPUs with shared cache.

At the same time, GPU hardware has evolved in ways that benefit interactive ray tracing. Over the past six years, programmable graphics hardware has progressed from simple processors for sequential vector instructions, to more general processors with support for branching, and recently massively data-parallel scalar processors with a light threading model. This has enabled vastly improved performance for programs employing branching, without sacrificing arithmetic computational power. These features make the GPU an ideal platform for solving ray tracing higher-degree implicits. The last component of this work is a full ray-tracing technique that is well-suited for current GPU's: it makes heavy use of floating point arithmetic and conditionals in tight loops, and significantly outperforms its counterpart on the CPU. Efficient acceleration structure traversal remains a challenge on the GPU, due to the irregular memory access patterns and code complexity of efficient CPU algorithms employing packets. Nonetheless, this obstacle is not insurmountable, and full accelerated ray tracing on the GPU has great potential.

With both GPU and CPU employing massively parallel SIMD hardware, computationally expensive ray tracing methods such as rendering implicits are increasingly practical. At the moment, both platforms have advantages and limitations: the applications in this dissertation are generally tailored to the strengths of their target hardware. However, common paradigms apply to both evolving GPU and CPU architectures. In particular, arithmetic performance will ultimately outpace memory access, suggesting that procedural methods for defining and rendering geometry will become increasingly prominent in graphics. We see implicit surfaces as part of this overall picture, and ray tracing as a viable method for interactive rendering of these geometries. We do not claim that ray tracing is the best solution for all rendering, nor indeed that it is ideal for all applications of implicit surface rendering. However, we do see it as playing a continued and important role in visualization, at the very least because implicit filters are ubiquitous in discrete data processing. Finally, isosurface ray tracing is an interesting case study in which algorithms are suited for the CPU and the GPU. It poses significant questions about appropriate tradeoffs between cache and arithmetic logic in hardware, the relationship between ray tracing and other means of sampling implicit surfaces, and the impact of compression and level-of-detail techniques.

CHAPTER 2

BACKGROUND AND RELEVANT WORK

Surfaces are ubiquitous in 3D computer graphics. Due to modeling simplicity and the efficiency of the z-buffer algorithm, the most common geometric surface representation is a piecewise-planar triangle mesh. However, in many cases it is useful to model the surface via a higher-order procedural form. The motivations for this approach include compactness of representation and control over local or global smoothness. Methods for modeling procedural geometry can be discrete, as in subdivision surfaces, or continuous, as in parametric and implicit surfaces. While parametrics are commonly used in modeling, implicits define surfaces based on a wide family of spatial filters.

An implicit form can be used to define a surface whenever discrete geometry is paired with a continuous reconstruction filter. The resulting isosurface is continuous with respect to the chosen support, and inherits the behavior of its parent filter. This functional definition of the surface enables a wide range of techniques built into the filter, including interpolation, morphing, and blending between levels of detail. In the case of visualization, a given data set often implies a specific filter. Even when the user employs an arbitrary filter, it is important to correctly render the filtered data.

This chapter provides a working definition of an implicit surface, and motivation for its use in graphics and visualization. It then covers relevant work in rendering implicits, employing ray tracing and other techniques.

2.1 Implicit Surfaces

In mathematics an explicit function $g : X \rightarrow Y$ is said to be *implicit* if for some $f : (X, Y) \rightarrow \mathbb{R}$ it satisfies $f(x, g(x)) = 0$. The *implicit function* has come to signify the function $f = 0$ itself; and more generally an implicit equation is understood to be anything of the form:

$$f : \Omega \rightarrow \mathbb{R}, f = 0 \tag{2.1}$$

For our purposes, given a function $f \in \mathbb{R}^n$ which we refer to simply as the *form*, or *filter*, we are interested in the zero set:

$$\{x \in \mathbb{R}^n \mid f(x) = 0\} \tag{2.2}$$

which more specifically in $\mathbb{R}^n$ is a *level set*. In 3-space, this set is a *level surface*. Thus, an *implicit surface* refers to a surface defined implicitly by function of three variables,

$$\{\vec{x} \in \mathbb{R}^3 \mid f : \mathbb{R}^3 \to \mathbb{R},\ f(\vec{x}) = 0\} \tag{2.3}$$

Typically, given an isovalue v, the form

$$f(\vec{x}) - v = 0 \tag{2.4}$$

denotes the *isosurface*, at v, of the implicit form f.

Thus, when we refer to implicit surfaces, or *implicits*, we are specifically referring to level surfaces of a chosen filter f. The choice of filter function is theoretically unlimited: it can be an algebraic or transcendental function; it can be defined globally over its domain or as piecewise kernels.

Since f is itself a function, confusion often arises from differences between the formal mathematical definition of an implicit function (as an explicit representable in implicit form) and the implicit form itself. This is further complicated by the implicit function theorem, which shows that any explicit function can be expressed as an implicit function, locally where continuously differentiable [102]. To this end the term *functional representation*, or "F-Rep" [92], has been proposed to describe objects defined by closed-form expressions $f = 0$, and inequalities such as $f \leq 0$ that model solids. However, implicits are at this point accepted terminology in mathematics as well as graphics. Nonetheless in our work, we will avoid using the ambiguous term *implicit function*, and instead refer to the implicit equation, as the *implicit form* of a filter f.

Ultimately in graphics, implicit surfaces are ways of functionally defining geometry on R^3, as opposed to defining geometry explicitly via a set of discrete points. In this sense,

a triangle mesh is simply a set of trimmed piecewise linear (planar) implicits; nonetheless meshes are often referred to as "explicit" geometry. Here, implicits are simply abstract reconstructions of some given geometry in typically simpler explicit form; the terms implicit and explicit have intuitive sense but have nothing to do with their formal definition.

Implicit surfaces were first introduced to computer graphics by Blinn [5] in visualizing molecular structures Blinn employed a quadratic polynomial to approximate a Gaussian distribution, and performed ray-casting combined with a hybrid iterative solver to find the intersection point. The resulting "blobby" shapes were popular in graphics and modeling from 1982 until the 1990s; however they proved difficult to render and model with, and ultimately unpopular outside of niche applications in molecular dynamics and biology.

Beginning with marching cubes [139, 73], generating an implicit surface from filtered volume data became a popular method of indirect volume visualization. Soon afterwards, implicit surface techniques were developed for extraction and rendering finite elements meshes, range scans, point clouds and other irregular data. At the same time, direct volume rendering methods evolved as a feasible and expressive method of rendering filtered data, without having to specify a discrete surface. Common sources of scientific data that benefit from isosurface rendering are:

- a regular (structured) volume, for example from a finite differences simulation, MRI or CT scan.
- a tetrahedral or hexahedral volume from a finite elements or similar method
- a raw point set generated from range scanning apparatus
- a scalar, vector or tensor field point set generated from simulation or measurement

The remainder of this chapter provides an overview of this work in implicit surface rendering and geometric data visualization. This field encompasses most of scientific visualization, and is too broad to adequately describe here. Interested readers are encouraged to consult [48], which provides an excellent overview of a wide variety of visualization techniques. In particular we focus on state-of-the-art ray casting and ray tracing methods for implicit surfaces, and proxy methods that are competitive in the arena of large-data visualization and rendering of implicit surfaces of arbitrary form or with general filters.

2.2 Mesh Extraction

Perhaps the simplest, and by far the most popular method for extracting a mesh from an implicit surface is via the marching cubes algorithm, developed for isosurfacing of volume data by Lorensen and Cline [73]. The procedure is to subdivide the lattice into cell primitives, and solve where the local implicit intersects the cell edges. In conventional marching cubes, cells are voxels of the volume, and the isosurface is approximated by solving where linear interpolation yields the desired isovalue along a voxel edge. By considering the values at vertices of the cell, the algorithm knows which edges will contain a surface. For a trilinear interpolant, 15 different mesh configurations exist for piecewise-planar iso-polygons within a given voxel. As these implicits are C^0 at cell edges, the resulting mesh is guaranteed to be continuous, although several cases yield ambiguous connectivity which must be resolved to prevent holes [85]. More generally, marching cubes allows a mesh to approximate any implicit, as solving for arbitrary $f(x,y,z)$ reduces to a 1D problem when any two coordinate axes are constant [6]. Ironically, a similar approach was first published by Wyvill et al. [139] for polygonizing blobby surfaces similar to those of Blinn. Nonetheless, isosurface extraction for volume rendering has proven a popular application of implicit surfaces.

For small volume data, marching cubes is well suited for dynamic extraction alongside rasterization. However for larger data, it becomes necessary to manage overall scene complexity, measured in the number of cells or tetrahedra visited by the extraction algorithm. As long as this is controlled, it is generally trivial to render the resulting mesh interactively on graphics hardware. To reduce the visited set of cells, Wilhelms and Van Gelder [134] proposed a min-max hierarchy embedded in a branch-on-need octree for trivially ignoring empty regions of a volume. Livnat et al. refined the interval tree concept in creating a span space tree [70]. Livnat and Hansen [69] extended the min-max octree technique with view-dependent frustum culling using a shearing transformation visibility test. Westermann et al. [132] implemented adaptive marching cubes on multiresolution octree data. Pesco et al. [94] proposed an occlusion test for the implicit itself, further reducing the cost of isosurface extraction at render-time. With few exceptions these techniques enable interactive rendering of data up to 512^3 resolution. For larger data, interactive visualization is difficult, and more recent extraction techniques have turned to topologically-guided generation of simplified meshes [64].

Other mesh extraction algorithms are appropriate for different applications. For unstruc-

tured volume data, marching tetrahedra (MT) was developed by Doi and Koide [23] to operate natively on tet meshes, though it can equally be applied to structured data due to the dual relationship between tetrahedra and voxels. MT and related algorithms have been employed in dynamic isosurface rendering of unstructured data [91, 53]. Though MT addresses some sampling issues of marching cubes with unstructured data, both algorithms generate poor-quality, irregular meshes, and fail to capture topological features such as singularities. Dual contouring [49] and dual marching cubes [86, 108] alleviate these problems, reducing overall triangle counts, and effectively capturing singularities. For generating high-quality triangle meshes, Scheidegger et al. [109] proposed an advancing front meshing algorithm using a curvature-based guidance field. Schreiner et al. [110] extended this method to structured and unstructured scalar volume fields, and modified the guidance field to minimize both geometric error and number of required samples.

Extraction methods have proven popular for rendering general-form implicits. Bloomenthal [6] presented a fast polygonizer for algebraic surfaces based on a generalization of marching cubes. Yu et al. [142] employed a coarse Bloomenthal polygonization in conjunction with a progressive subdivision-based refinement algorithm, achieving more regular tessellations in faster runtime. Stander and Hart [116] used critical points from Morse theory in feature-driven rendering of implicits. Paiva et al. [88] extended dual marching cubes with an interval arithmetic method for robust rendering. Varadhan et al. [123] employ dual contouring and IA to decompose the implicit into patches, and compute a homeomorphic triangulation for each patch.

2.3 Point-Based Methods

When the discrete geometry underlying the implicit surface is sufficiently dense, it becomes more efficient to approximate the surface with nonpolygonal proxy geometry. This geometry can then be rasterized using raw points, billboarded primitives, or a coarser volumetric representation.

Point-based rendering was conceived by Levoy and Whitted [68] and proven for large point-set data by Rusinkiewicz and Levoy [104] with their QSplat software. The process involves sampling a surface into points as an offline process and then rasterizing and shading these points as disc splats or similar proxies on a GPU. Semantically, point-based rendering is similar to ray casting without an explicit root-finding component; rather than picking samples

along a ray and incrementally solving for the root, it generates appropriate splats. Often, point-based methods employ ray casting techniques to blend between splats or to shade. Co et al. [14] extended the isosurface ray tracing technique of Parker et al. to point-based rendering on the GPU. Livnat and Tricoche [71] implemented a hybrid isosurface render that combined view-dependent marching cube extraction and point-based rendering, and handled reasonably large data interactively. Zhou et al. [143] develop a system for rendering higher-order finite element volumes efficiently. While rendering of point sets is efficient on the GPU, generating appropriate point set proxies can be expensive, and generally runs as an offline process. Adaptive particle-based sampling of implicits was proposed by Witkin and Heckbert [135]. Recent contributions have extended this approach with a curvature-guided energy redistribution method [79]. It is equally possible to use particle sampling of implicit surfaces for mesh generation [80].

2.4 Volume Rendering

Volume rendering is another approach to rendering implicit surfaces using the GPU rasterization pipeline. When the implicit is a structured scalar field, direct volume rendering (DVR), can approximate an isosurface by employing a sharp peak in the transfer function near to that isovalue [66, 67]. Clearly, this approach will produce artifacts when the convolution of the transfer function and underlying data set is undersampled. Nonetheless, DVR is an excellent approach to a wide variety of implicit surface rendering applications, and holds many advantages over discrete surface rendering, including the ability to trivially render multiple transparent surface approximations and provide better data classification [54]. Moreover, many of the space-skipping techniques employed in efficient ray casting and mesh extraction can also be utilized in DVR and volume ray-casting algorithms. Out-of-core methods [55] and distributed shared memory mechanisms paired with streaming [13] have proven capable of handling large volume data on the GPU, though these methods are not always scalable.

While structured volume rendering is ubiquitous and efficient due to fast GPU texture access and interpolation, GPU DVR algorithms for unstructured data are nontrivial. The best known methods rely on tet-by-tet ray casting [9], sorted rasterization of projected tetrahedra [11, 10], polygonal view-inedependent spherical shell proxies [31], and point-based rendering [143].

In rendering general-form implicit surfaces, one approach is to sample an analytical implicit into a regular grid by voxelization [119], followed by direct volume rendering. Rendering of recursively voxelized object space with interval arithmetic was first proposed by [136]. Discrete ray tracing [140, 118] and GPU-based volume ray casting are also valid approaches to rendering voxellized datasets.

Isosurface rendering and direct volume rendering have different strengths. The former is useful when subvoxel surface detail is critical, and the latter when a broader picture of the underlying data is desired. Transfer functions, particularly multidimensional transfer functions [54] can be highly expressive and flexible, and allow intuitive visualization of 3-manifold data segments as opposed to surfaces. However, the difference between object-order and logarithmic order rendering is significant when dealing with large datasets. Frequently, discrete isosurface rendering is appropriate in cases when conventional object-order volume rendering would be prohibitive, as in the case of large volume datasets; or where simplyfing LOD schemes for volume rendering are not desirable. In many cases, however, in visualizing boundaries such as the interface between two fluids, it remains desirable to render an isosurface.

2.5 Ray Casting and Ray Tracing Implicits

In ray casting, as first presented by Appel [2], viewing rays are generated from a camera and a frame buffer and intersected against objects in their trajectory, then finally shaded. Ray tracing, first proposed by Whitted [133], entails ray casting but with the additional tracing of secondary rays for shadows and refractions.

Over time, ray casting has come to signify any process of intersecting primary (viewing) rays with geometric primitives, including hybrid rasterization/ray-casting methods. In contrast, an algorithm is frequently termed a “ray tracing” algorithm if it can natively support casting of secondary rays, regardless of whether or not shadows or a full Whitted refraction model is actually used. While there is some disagreement over whether ray cast images without refractions can be termed ray tracing, in this dissertation we claim that a technique is ray tracing if it can support secondary rays, regardless of whether or not secondary rays are employed in a given image. Moreover, we prefer the term “ray tracing” to differentiate our algorithms, which involve ray traversal of an acceleration structure, from GPU ray casting algorithms which may rely on view-dependent rasterization at one or more stages of the

rendering pipeline. Similarly, while the terms "ray casting" and "ray intersection" are often used interchangably, we draw a distinction between the ray-primitive intersection test and the ray casting method of generating primary rays.

Much focus of ray tracing literature in graphics has been on using secondary rays for realistic lighting and shading models, such as global illumination [51]. More recently, the interactive ray tracing community has focused on making ray tracing faster through parallelization and optimization. Our work is primarily interested in ray tracing algorithms for their ability to intersect nonpolygonal geometry, specifically implicits.

In ray tracing an implicit surface $f : \mathbb{R}^3 \rightarrow \mathbb{R},\ f(x,y,z) = 0$, we seek the intersection of a ray

$$\vec{p}(t) = \vec{o} + t\vec{d} \tag{2.5}$$

with the implicit surface. By simple substitution of this parametric line equation into Equation 2.3, we have a unidimensional function

$$f_t(t) = f(o_x + td_x, o_y + td_y, o_z + td_z) \tag{2.6}$$

and we can then solve $f_t(t) = 0$ for the smallest $t > 0$.

Ray tracing implicits is a root-finding problem. Moreover, in ray tracing all surfaces are at some level formulated implicitly, in order to solve for the distance parameter t. In simple cases, finding the roots entails computing a closed-form solution or performing several iterations of Newton-Raphson or similar numerical methods. More complicated f with multiple roots and nonmonotonic behavior require robust techniques such as Lipschitz methods, bracketing or self-validated arithmetic.

2.5.1 Ray Tracing General-Form Implicits

Although it employed projection to solve ray-implicit intersection as a function of one variable, the blobby renderer of Blinn [5] was fundamentally a ray casting approach. Ray tracing of general algebraic surfaces was first explored by Hanrahan [39], using Descartes' rule of signs to determine if a given ray interval contained roots. While efficient, this "point sampling" approach is not robust when the implicit ray equation f_t contains multiple roots along an interval. Van Wijk [122] implemented a recursive root bracketing algorithm

using Sturm sequences, suitable for differentiable algebraics, and now employed in the popular POV-Ray software [96]. Kalra and Barr [52] devised a robust method for ray tracing algebraic and some non-algebraic surfaces given Lipschitz bounds of the first and second order. Hart [40] formulated the implicit as a signed distance function, and minimized this function along the ray. To robustly ray-trace arbitrary implicits directly from their expression, Mitchell [82] suggested the use of interval arithmetic for root isolation, followed by a separate root refinement algorithm such as regula falsi, bounded Newton-Raphson, or another globally convergent numerical method. De Cusatis Junior et al. [16] extended this technique to employ affine arithmetic.

More recent implementations have demonstrated ray-casting of implicit surfaces on the GPU. Loop and Blinn [72] implemented an extremely fast GPU ray caster approximating implicit forms with piecewise Bernstein polynomials over local tetrahedral domains. Romeiro et al. [100] proposed a hybrid GPU/CPU technique for casting rays through constructive solid geometry (CSG) trees of implicits. De Toledo et al. [18] demonstrated interactive ray casting of cubics and quartics using standard iterative numerical methods on the GPU. Fryazinov and Pasko [27] employ rule-of-signs interval methods in ray tracing generalized implicit (FRep) surfaces on the GPU.

2.5.2 Ray Tracing Isosurfaces

Direct ray tracing of implicits interpolating volume data was first demonstrated by [115], with rendering times on the order of seconds per frame. Parker et al. [90, 89] implemented a tile-based parallel ray tracer on a 128-CPU SGI workstation, achieving interactive rendering of isosurfaces from large (1 GB) structured volumes. They employed a hierarchical grid as a min-max acceleration structure on top of the raw raster data, and an analytical ray-voxel intersection technique based on the numerical solver of Schwarze [111] for cubic equations. DeMarle et al. [19] extended this system to a 32-node cluster, rendering an uncompressed 8 GB volume dataset at multiple frames per second on 64 CPUs. Wald et al. [129] employed progressive out-of-core rendering and SIMD ray tracing of a kd-tree to achieve significantly faster rendering speeds on a multicore workstation. Wyman et al. [138] extended the system of [90] with a global illumination approximation using a precomputed radiance cache and spherical harmonics.

Isosurface ray casting has not been restricted to the CPU. Hadwiger et al. [38] modified

volume ray casting techniques for the GPU to render discrete isosurfaces, by using the raster pipeline for high-level spatial culling and employing a hybrid point-sampling [39] and secant method for solving the intersection. Though not entirely robust, this method allows for rendering surfaces interpolated via a tricubic spline interpolant, at real-time rates. Stoll et al. [117] employed a hybrid rasterization/ray-casting approach to rendering piecewise quadric implicits, employing the SLIM surface model of Ohtake [87].

Ray casting algorithms have also been employed in rendering unstructured data. Garrity [30] proposed computing ray intersections with entry and exit faces of tetrahedra; this approach could trivially support isosurfacing albeit in linear time with respect to the number of objects traversed. Marmitt and Slusallek [76] present a Plücker-coordinate intersection approach for efficient volume ray casting of tetrahedral and hexahedral meshes, although this method is subinteractive on current CPUs.

We conclude this chapter by noting a common goal in all isosurface rendering applications. Regardless of whether the surface is extracted via a proxy or rendered directly via ray casting, adequate sampling is critical for accurate rendering of features. The main limitation of volume rendering, when employed in discrete isosurface rendering, is ensuring sufficiently high sampling frequency, whether samples are stored in planar proxy or computed via direct volume ray casting. Similarly, in point-based rendering and mesh extraction from large datasets, dynamically generating appropriate samples is more difficult than evaluating the implicit or rendering the proxy. With ray casting and ray tracing methods, sampling is automatically tailored to the camera. In this sense, ray tracing implicits is blind to sampling requirements; it is a "ground truth" method for visualization even when it is not the most efficient. A goal of the work in this dissertation is to show that ray tracing can be efficient, as well as correct, in rendering several classes of implicit surfaces with a variety of techniques.

CHAPTER 3

ISOSURFACE RAY TRACING OF OCTREE VOLUME DATA

In this chapter, we present a straightforward technique for ray tracing isosurfaces of large compressed structured volumes. The data set is first converted into a lossless-compression octree representation that occupies a fraction of the original memory footprint. An isosurface is then dynamically rendered by tracing rays through a min/max hierarchy inside interior octree nodes, using an efficient ray intersection routine for piecewise trilinear interpolants in nonempty cells. By embedding the acceleration tree and scalar data in a single structure and employing optimized octree hash schemes, we achieve competitive frame rates on common multicore architectures, and are able to handle large and time-variant data in a completely in-core CPU algorithm. The contents of this chapter were published as "Interactive Isosurface Ray Tracing of Large Octree Volumes" [61].

3.1 Motivation

This work was inspired by previous applications of isosurface ray tracing of large volume data [90, 89, 19, 129], and later by coherent grid traversal for polygonal scenes [130]. At the same time, we build on top of previous work involving min-max octree structures for online mesh extraction and rasterization [124, 132, 134]. The main motivation is that structured volumes, stored as scalars in a regular grid, are large but often inefficient. Particularly in the case of simulation data, or presegmented medical or biological scanned data, these volumes contain large amounts of empty space. Worse still, acceleration structures for isosurface ray tracing can occupy up to $3\times$ the memory footprint of the original data, in the case of a kd-tree of [129]. Although CPU cache architectures and the logarithmic complexity of ray tracing are ideal for handling large data, the overhead of uncompressed data and acceleration structures is high, and restricts the size of data that can be rendered while resident in memory. As seen in [19, 20], a cluster-based ray tracer with a distributed shared memory scheme can

perform interactively, but nonetheless incurs a high performance penalty relative to fully in-core rendering.

Data compression can be an effective alternative to out-of-core methods. From the standpoint of information theory, it is possible to represent a datum with fewer bits when the local gradient, or another measure of entropy, is zero or small [113]. This premise is the basis for all compression schemes. However, for the purposes of ray tracing we desire a compression mechanism that correlates to spatial location, so that information can be decoded at the same time an acceleration structure is traversed. Specifically, we desire that:

- compressed data can reside completely in local memory.
- decompression can be performed progressively at traversal time, with high spatial and cache coherence.

While achieving a maximal compression ratio is desirable, we are mostly concerned with representing the original data losslessly, and embedding that into an acceleration structure that can be traversed quickly. For this, we turn to the octree, a classic data structure often credited to Dijkstra [106] and first employed in volumetric image processing by Jackins and Tanimoto [47]. Functionally, the octree is the zero-order member of the wavelet family (first order being the Haar wavelet) [74]. While employing wavelets in volumetric compression is possible for this application, in our case the relative overhead of the acceleration structure is sufficiently high that the representation of each datum as a full byte is acceptable. This contrasts with wavelet compression methods designed to compress larger blocks, for example paired with out-of-core direct volume rendering [37]. Moreover, we are primarily interested in lossless compression of our voxel data, for which the octree is arguably better suited as opposed to higher-order wavelets.

3.1.1 Octree Traversal

Octrees are, in fact, theoretically optimal in terms of fewest traversal steps, assuming objects are contained uniformly within cells of the acceleration structure, with no overlap [8]. The combination of regular, hierarchical nature of the structure affords many different styles of traversal algorithm. The original Glassner implementation [32] proposed top-down point location testing along successive octree nodes hit by the ray. Samet [105] modified this marching procedure to incorporate a neighbor-finding algorithm, delivering

dramatic speedups. Sung [120] proposed a DDA traversal similar to a hierarchical grid. Finally, Gargantini and Atkinson [29] implemented a traversal similar to a kd-tree where the ray intersection with each octant midplane is ordered.

For regularly spaced geometry, octrees require fairly low memory access; however for meshes and unstructured volumes they conform poorly to irregular geometry. Because of this, and relatively expensive traversal strategies [41], octrees were overtaken by hierarchical grids as general-purpose ray tracing structures [45]. Nonetheless, in our application we are specifically concerned with regular voxel geometry. Moreover the ability to use a single structure for both ray traversal and scalar storage is tempting, and encourages our choice of the octree as an acceleration structure in this application.

3.1.2 Octree Hashing

It is worth noting previous work in efficient octree hashing. The general goal is *point location*: given (x,y,z) coordinates and the root node of the octree, retrieve a leaf node of the octree at that location. A related problem is *neighbor-finding*, in which we are given a leaf node and asked to find an adjacent neighboring leaf. While these two algorithms were pioneered by Glassner [32] and Samet [105] in ray tracing, their application extends to general use of any regular binary tree (quadtree, octree, etc.). Frisken and Perry [26] propose an efficient and concise hashing scheme using binary arithmetic on integer coordinates. We build upon their work to create our own fast, general-purpose hashing scheme.

3.2 Algorithm Overview

We compress an input volume from a 3D grid into an octree, similarly to the approach of Velasco and Torres [124] but encapsulating actual voxels as opposed to eight-voxel cells. Rather than extracting a mesh and sending geometry to the GPU, we ray trace the octree-encapsulated volume directly, using the octree as an acceleration structure to find a cell primitive. The implicit surface inside each cell is a trilinear interpolant patch; globally this defines a piecewise C^0 isosurface.

3.2.1 Constructing an Octree Volume

An octree volume is an adaptive-resolution, hierarchically compressed scalar field. Scalar values are stored at leaf nodes. At maximum octree depth, these correspond to the finest available data resolution. Scalars at less than maximum depth store coarser resolutions, by

factors of 8 per depth level. Interior nodes maintain pointers from parents to children. In our multiresolution LOD application, they also contain coarser-resolution representations of each of their children. We embed min/max pairs inside interior nodes, and compute these bottom-up during construction. The resulting octree is an acceleration structure for traversal.

Volume data could be natively computed and stored in this format; however for our purposes it is desirable to build an octree volume from a scalar field in a 3D array. The process of creating an octree volume is conceptually simple: given input data in the form of a 3D array, we group regions with low variance and output a hierarchically compressed octree volume. Specifically, we consider groups of 8 voxels nested within a parent node of the octree. If these voxels are identical (in lossless compression), or have a combined variance below a desired threshold (lossy compression), we compute their average and consolidate them into a single node at the previous depth level of the octree. By recursively consolidating nodes with low intervoxel variance, we can build an octree volume in bottom-up fashion.

3.2.2 Ray Traversal and Voxel-Cell Duality

Our choice to use the same structure for data and acceleration comes with a caveat: though our volume data consists of voxels, we ray trace an isosurface that is defined within *cells* of 8 voxels. Fortunately, there exists a dual relationship between voxels and cells. By logically shifting the position of all scalars backward by half a unit of voxel width, we remap our scalar field to cells (Figure 3.1).

Two options exist to accomplish this mapping in memory. One could expand each voxel to contain its forward neighbors, thus storing each cell completely. While this would require no additional searching through a structure to retrieve cell corner values, it requires 8 times the storage of the original volume. A far more sensible approach is to retrieve the forward-neighboring voxels and construct a cell from them. In effect, when a ray traverses an octree leaf node at (i, j, k) with a datum v, we instead intersect it with a box (cell) where i, j, k is mapped to the minimum vertex, and neighboring voxels in the octree are mapped to the remaining vertices. (Figure 3.2).

For a volume stored in a 3D array, querying the values of these neighbors is trivial: simply an array index into memory that is typically already in cache. For the octree, the process is more intensive. Here, we must employ point location to retrieve the voxel values of the forward neighbors. Full top-down point location from the root would result in an $O(log(N))$

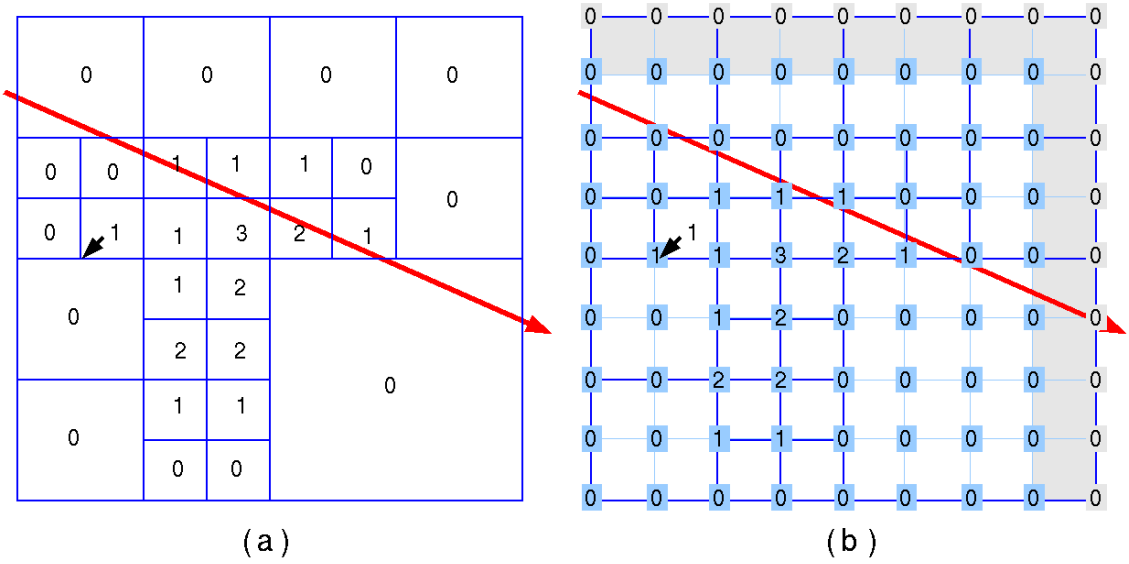


Figure 3.1: Voxel-cell duality and octree traversal. While the octree volume (a) is given with voxels at the center of each node, we actually seek to ray trace a field of cells with voxel values at the corners (b). To accomplish this, we observe a duality between voxels and cells, by mapping each voxel to the lower-left corner of a cell. Values outside the octree data (in gray) are defined to be zero. Thus, the ray traverses interior nodes of the octree, and intersects with a well-defined cell primitive composed of 8 voxels.

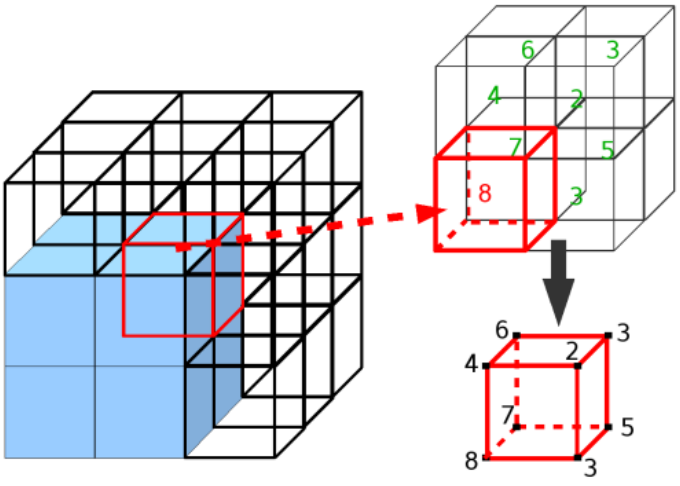


Figure 3.2: Retrieving a cell from a neighborhood of voxels. Given an octree interior node composed of eight voxels, we seek to intersect a leaf node consisting of a single scalar value. We perform neighbor-finding on the octree structure to retrieve the forward-neighboring voxels. This yields a cell of 8 voxels, which we then use as the intersection primitive from which we reconstruct the isosurface.

algorithm. However, with neighbor-finding techniques we can significantly reduce this lookup cost. The worst-case complexity of neighbor-finding is $O(log(N))$, but in practice the algorithm skews heavily toward the best-case of $O(1)$, when neighboring voxels lie within the same parent [106]. Even then, neighbor-finding on octree data must perform competitively with the $O(1)$ complexity of lookup on uncompressed 3D arrays. It is readily apparent that octree hashing, specifically neighbor-finding, is a fundamental algorithmic component of our work.

3.2.3 Computing the Min/Max Tree

Ray tracing cells defined by forward-neighbors (Figure 3.2) directly impacts the construction of our min/max tree. Specifically, a parent node in the octree must compute the minimum and maximum based not only its own children, but on voxels forward-adjacent to its children as well (Figure 3.3). Knowing this, one can compute a min/max pair for that leaf node based on the cell corner values. The min/max tree is then computed recursively, by finding for each parent node the minimum and maximum of all children min/max pairs. As we are only concerned with cells at the finest depth of the octree, it suffices to account for forward-neighbors once at the deepest leaf level, and thereafter compute each parent's

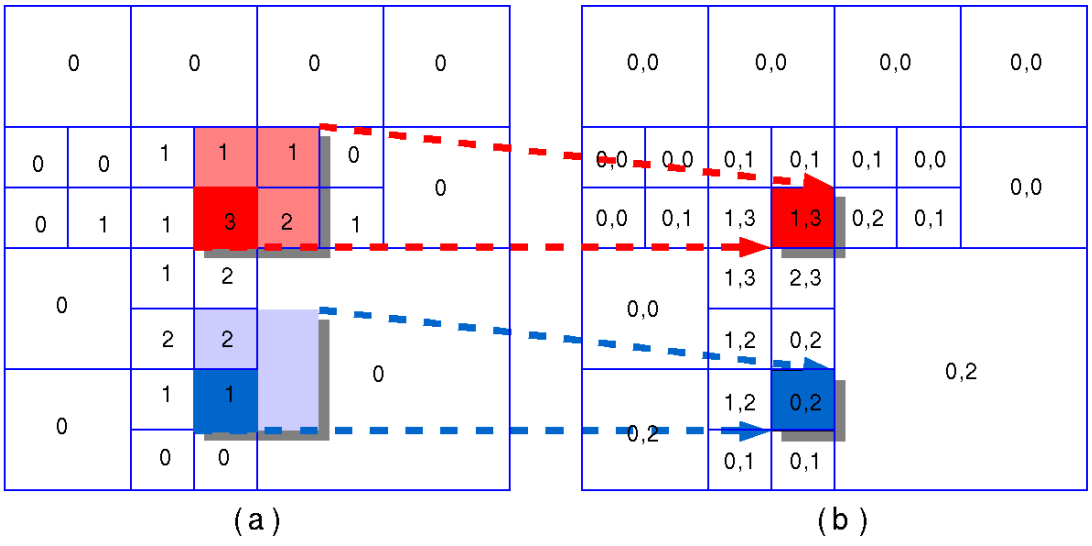


Figure 3.3: Min/max tree construction from forward neighbors. In the quadtree case, each leaf node must compute the minimum and maximum of its cell, hence account for the values of neighbors in the positive X and Y dimensions (a). This yields a min/max pair for the leaf node (b). Neighbors can potentially exist at different depths of the octree, as is the case for at the blue leaf node.

min/max pair based on the pairs of the 8 children.

Clearly, storing the min/max tree within the octree data structure entails some overhead. As compression is a major goal of our work, it would be unwise to store the min/max pairs of each scalar voxel, which would demand over three times the storage of the raw octree data. Instead, one could compute the extrema temporarily at leaf nodes, and begin storing the min/max tree at depth $d_{max} - 1$. Omitting the min/max pair at leaf nodes would seem to generate a looser tree and hurt performance, but in practice, it simply forces us to compute the minimum and maximum of forward voxels while we are looking them up via neighbor-finding. Logically, this approach entails an overhead of one min/max pair for 8 voxels, plus pairs for other interior nodes of the tree. This suggests approximately a 22% additional footprint on top of raw scalar octree-compressed data. While not insubstantial, that seems acceptable given the acceleration capabilities of the min/max structure.

While the octree volume does not yield great overall compression ratios (3:1 or 4:1), it is respectable as a lossless compression mechanism, and includes the min-max acceleration structure as well. For additional compression we can choose a set range of isovalues, and omit data outside that range. For rendering isovalues within that region of interest, this preserves lossless quality and allows less compressible data to reside in main memory. Overall, the octree can occupy an order of magnitude less memory footprint than a comparable kd-tree or hierarchical grid method.

3.3 Implementation

Our implementation builds on the theoretical foundation laid in the previous section, with details provided for the octree data format, point location and neighbor-finding, and the octree traversal itself. Pseudocode of these algorithms is provided in the appendices; however it is not necessary to understand our approach.

We chose not to employ SIMD or packets. Given our focus on large data, we would expect highly-variant scenes and at best modest speedups from coherent techniques. Wald et al. [129] reported little performance gain from coherent techniques on large data. Specifically, with comprehensive scenes of large volumes, a pixel can frequently cover multiple voxels. With agressively coherent techniques such as frustum-based traversals, this entails much unnecessary work and potentially a performance decrease over single-ray techniques. Moreover, we are first interested in how an optimized single-ray octree algorithm behaves

compared to known techniques, and the relative performance of octree volumes versus uncompressed structured data. Coherent octree traversal will likely be explored in future work, however.

3.3.1 Data Format

To avoid explicitly storing a full node for each leaf of the octree volume, we store nodes corresponding to the parent. In this scheme, at the maximum depth of the octree, all children are guaranteed to be leaves. Thus, at depth $d_{max} - 1$ of the octree, we employ a separate structure called a *cap*, consisting simply of 8 scalar values at d_{max}. All other interior nodes contain the scalars, min/max pairs, and pointers for 8 children. We denote any scalar value at noncap depth a *scalar leaf*, although admittedly scalars inside cap nodes are logically leaves of the tree as well. Scalar leaves are stored as a single value within within a parent interior node, and are indicated by a null child pointer. These three types of logical octree node are illustrated in Figure 3.4.

Rather than store full pointers, we store a 32-bit child_start and a single-byte offset per child. In early implementations, we used binary arithmetic masks and bit-counting to determine which nodes were leaves; in practice however this requires computation (specifically left-shifting by a nonconstant) that hampered performance. Ultimately, we use an array to indicate the offset of each child, or -1 if that child is a leaf. We use a second array, child_scalars, to contain the value of each child. In this application we only care about this value when the octant is a scalar leaf at submaximum depth; however future implementations could take advantage of this inherently multiresolution approach to provide a level-of-detail scheme. Details of the structure are provided in Appendix A.1.

To build our structure, we use a 3D array of rectilinear grid data as input. We determine N, the smallest power of 2 that encompasses the largest dimension of that volume, and choose the maximum depth $d_{max} = log_2(N)$. We then proceed from the bottom-up, assigning groups of 8 voxels from the original structured grid to the caps. Groups of 8 identical voxels are consolidated into a single scalar leaf of the parent. Pointers from interior nodes to children are subsequently filled in, until the root node completes the tree.

The min/max tree is computed simultaneously alongside bottom-up consolidation. As explained in the previous section and in Figure 3.3, we must consider not only the 8 child voxels of each parent, but their forward-neighbors as well. As a result, we compute the

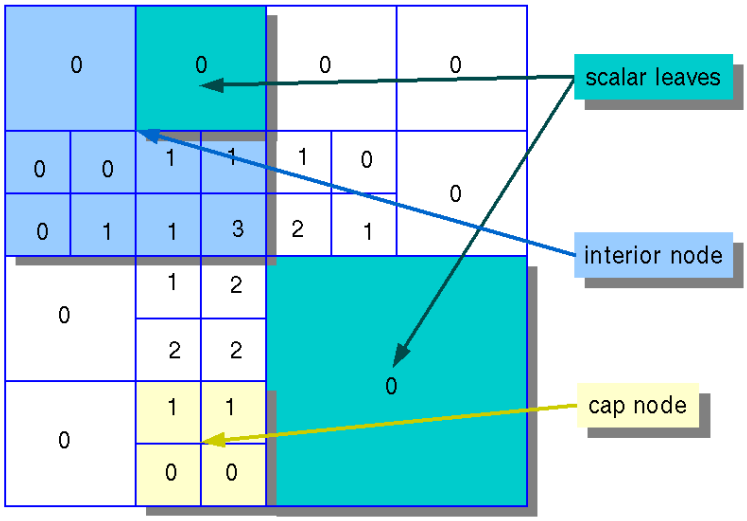


Figure 3.4: Octree volume format illustrated, showing examples of an interior node, a cap node, and scalar leaves. A scalar leaf is not a separate structure, but rather a single value embedded inside its parent interior node. Similarly, cap nodes are not leaves themselves but contain eight scalars at the maximal depth of the octree. Thus, nodes in this structure are the parents of nodes in the logical octree.

minimum and maximum of 27 voxel values, and store these in our min/max tree. Similarly to how we store a scalar leaf in child_scalars within the parent node, we store the minimum and maximum values of the eight children within their parent nodes. This allows us to reject children without actually traversing them, sparing us cache misses.

Scalar values are retrieved from the original data only for cap nodes, and used to compute the min/max tree. Afterwards, parent nodes are computed solely based on the values of their children. When child voxels consolidate into a parent, the corresponding child nodes are removed. This process continues recursively until the root node of the octree is completed.

3.3.2 Octree Data Lookup

Voxel-cell mapping manifests the need for a fast neighbor-finding routine, which brings us to octree hashing. As mentioned before, we adopt a scheme like that of Frisken and Perry [26], in which octree cells are defined on the interval $[0, 2^{d_{max}}]$. Then, given a vector in this coordinate space, we simply cast its components to integers and perform point-location from the root node of the octree.

3.3.2.1 Point location

Point location is simply top-down search through the octree; given an initial node, that node's current coordinates, and the coordinates of the desired destination. With full point location, the initial node would be the root, with all-zero coordinates. With neighbor-finding, one can begin point-location deep in the tree. Frisken and Perry [26] propose creating a single-bit mask corresponding to the current depth, with an offset shift to interleave the X,Y and Z components. Though theirs is an elegant algorithm, repeatedly left-shifting bits by arbitrary integers is expensive. Thus, we precompute $\mathsf{child\_bit\_depth[d]} = 1 << (\mathsf{max\_depth} - \mathsf{depth} - 1)$ and left-shift by 1 or 2 for X and Y components as necessary. We then proceed to compute the target child octant with binary & and integer inequality operations. We &-mask this value with the destination coordinates and bit-shift by constants corresponding to the X,Y and Z components. This yields the 0-7 octant offset of the child, and hence its index. We return the scalar value when we encounter a leaf; either a scalar leaf in an interior node, or a voxel within a cap node. Details can be found in Appendix A.2.1.

3.3.2.2 Neighbor finding

Given an origin node and a coordinate direction to a desired destination node, neighbor-finding entails recursion up the octree until we find a parent containing both nodes. Frisken and Perry [26] require foreknowledge of whether the neighbor is "left" or "right" on each X,Y,Z midplane. As our traversal only neighbor-finds when necessary, we omit this distinction, and begin at the depth of the origin node's parent. Our iteration consists of &-masking the origin and destination coordinates with the corresponding depth bit, and performing integer equality. When a common parent is found, the neighbor-finding function then relies on point location to find the leaf at the given destination. To minimize the memory footprint of the octree, we chose to omit parent pointers from the nodes of our octree. Effectively, recursing up the octree requires knowledge of parent indices. We provide this in the ray traversal algorithm itself, which fills a $\mathsf{parent\_trace[]}$ array containing the indices of all parents nodes for a given cap. Pseudocode for neighbor-finding is given in Appendix A.2.2.

3.3.3 Ray-Octree Traversal

Finally, we approach the problem of adapting a ray traversal scheme to our octree structure and its given hashing scheme. After experimenting with the methods of Sung [120] and Samet [105], the fastest traversal that emerged most resembled the technique of Gargantini

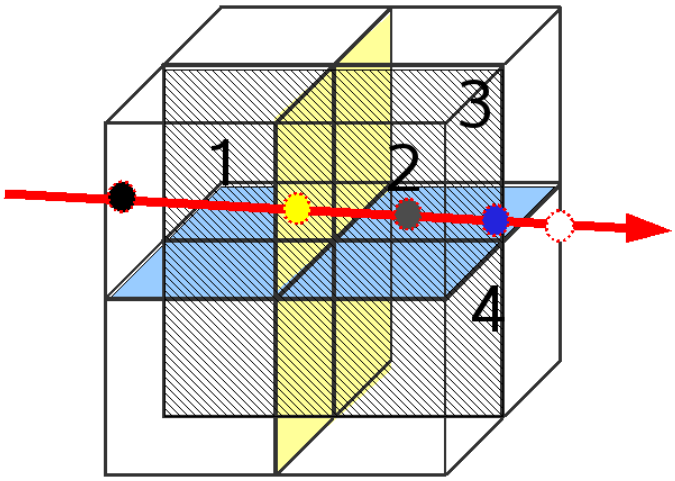


Figure 3.5: Ray traversing an octree node. The traversal algorithm sorts the intersection with the X, Y, and Z midplanes. As we are given the entry and exit intersections, we have the exact order of traversal of child octants.

and Atkinson [29]. The traversal is similar to that of a kd-tree, with splits along the X,Y and Z midplanes of each node. Gargantini and Atkinson proposed fully sorting child octants by the order of their traversal; this is the approach we take (Figure 3.5). We optimize it to exploit binary arithmetic on integer octree-space coordinates, similar to our neighbor-finding and point-location implementations.

Rays are generated in canonical octree space on $[0, 2^{d_{max}}]$, so no additional transform is required. We first perform a standard ray-bounding box test to discard rays that never intersect the volume. This test yields the entry and exit parameters (tenter, texit) for the root node of the octree, which we pass to our recursive traversal algorithm (Appendix A.3).

3.3.3.1 Interior nodes

The single-ray traversal first retrieves the octree node given by depth and node_index. Then, it computes the octree-space coordinates of the midplanes (Figure 3.5) that divide the child octants of this node. The computation-heavy section of the traversal involves evaluating penter and sorting the tcenter intersection distances in a separate array axis_isects[]. We use that array to sequentially march across the child octants in the correct order of their traversal. The algorithm has moderate initial cost associated with computing and sorting the midplane intersections; afterwards traversing the child octants is trivial. The first child octant is computed using the same constant shifting and binary-or as point location; afterwards

moving from one octant to the next merely requires inversion of the bitmask (axisbit) along the corresponding midplane axes traversed. Pseudocode is provided in Appendix A.3.

Our structure requires special traversal routines for scalar leaves and cap nodes. Exact details are left as an exercise for the reader; however, both are similar to interior node traversal in Appendix A.3.

3.3.3.2 Cap nodes

Cap intersection is identical to that of interior nodes, except for the block of code checking the isovalue against the min/max range and recursively calling the child traversal routine (Appendix A.3). In its place, we determine the values of the 8 voxels composing the cell (Figure 3.2). Before resorting to neighbor finding, we observe that given a voxel of interest intersected by a ray octree structure, anywhere from 1 to 8 voxels in this neighborhood will lie within the same cap node. Specifically, given the 0-7 child octant child, and a 0-7 direction "dir" to a desired neighbor, we simply check if (child & dir). If this evaluates false, the neighbor is simply cap.scalars[dir]. If it is true, we proceed with neighbor-finding to retrieve the value.

3.3.3.3 Scalar leaves

A scalar leaf is traversed recursively to the same depth as caps, even though it has no children and homogeneous value. When the traversal reaches cap depth, if the traversal encounters a neighborhood of identical voxels within the scalar leaf, we know that no isosurface is encountered. Otherwise, at the borders of the scalar leaf node, we perform neighbor-finding as we do for cap nodes.

Once we have the eight voxel values, we check that our isovalue lies within their minimum and maximum. If it does, we perform the isosurface intersection with the 8 voxels as corners of the cell.

3.3.4 Isosurface Intersection

The implicit in question is the trilinear Lagrangian interpolant:

$$f(x,y,z) = \left(\sum_{i,j,k=\{0,1\}} x_i y_j z_k \; v_{ijk}\right) - v = 0 \tag{3.1}$$

where (i, j, k) is the minimum coordinate of the cell, v is the isovalue, v_{ijk} is the value of that cell vertex, and $x_0 = i - x$ and $x_1 = x - i$ (similarly for y and z with respect to j, k).

To compute the ray-isosurface intersection, we seek a surface inside a three-dimensional cell with given corner values (Figure 3.2), such that trilinear interpolation of the corners yields our desired isovalue. We can find where a ray instersects this surface by simplifying Equation 3.1 a cubic polynomial equation in terms of t, and solving for the hit position by evaluating the ray at the first positive root of that polynomial. While the same recipe is generally used to generate the four coefficients of the polynomial, various techniques exist for finding the root.

Our implementation uses the same approach as the Neubauer iterative root finder employed by Marmitt et al. [75]. Here, a ray is iteratively reparameterized into subintervals within the cell in question, until a sign change is detected within the subinterval and a root is found. Compared to the analytical root finder based on Schwarze's cubic solver [111] used by Parker et al. [90], it is slightly faster and yields single-precision, numerically stable results.

3.3.5 Shading and Filling the Frame Buffer

While ray tracing delivers great flexibility in per-pixel shading methods, we are mostly interested in fast ray casting of the isosurface. Thus, our results show Lambertian shading with no shadows.

The traversal itself does not employ packets; however we use a packet architecture for ray generation and shading [4]. We do not defer normal computation due to the prohibitive cost of storing each cell per ray, or repeating the neighbor-finding process. However, the packet architecture allows diffuse shading to be performed in batch, which likely delivers some speedup over a more conventional single-ray tracer.

3.4 Octree Construction Results

3.4.1 Data Compression

Lossless octree compression by consolidating voxels with zero variance commonly yields a compression factor of 3 to 5, depending on the spread of isovalues within the data. In general, sparser data yield higher compression benefit (Table 3.1). Additional compression can be achieved by segmenting the data into iso-ranges of interest. For example, if we are

Table 3.1: Compression achieved for various structured data when converted to octree volumes, from Knoll et al. [60]. The second column shows clamped iso-ranges. Clamping all values outside a given range delivers additional octree compression, and preserves lossless compression for values within that range. "Full" indicates the full 0-255 range for 8-bit quantized scalars. Data sizes are in bytes, and include all features of the octree, including overhead of the embedded min/max tree.

DATA	ISO-RANGE	TIME STEP	SIZE		%
			original	octree	
heptane	full	70	27.5M	3.96M	14
	full	152	27.5M	9.5M	33
	full	0-152	4.11G	678M	16
RM	full	50	8.0G	687M	8.5
	full	150	8.0G	1.89G	25
	full	270	8.0G	2.48G	30
	64-127	270	8.0G	1.81G	22
CThead	full		14.8M	12.4M	84
femur	full		162M	163M	101
	100-163		162M	9.0M	5.5

mostly interested in isovalues from 64 to 127 in the Richtmyer Meshkov data, we can clamp scalars outside that range to those limiting values. As we see in Table 3.1, this allows us to compress a complex timestep of the LLNL data into under 2 GB, with full original quality within a sizeable isovalue range. Furthermore, if lossy compression is acceptable, one can more aggressively consolidate intervoxel variance. This could be desirable for large data that vary gradually in space. The effect would be to further quantize isovalues, and deliver extra compression.

3.4.2 Further Compression

Generally, our goal is simply to compress a single data timestep into a manageable footprint for limited main memory. Sometimes losslessly compressed data will be slightly too large to meet this constraint. One option is lossy compression via a nonzero variance threshold, which behaves similarly to quantization. A more attractive method, for our purposes, is segmenting data into interesting ranges of isovalues, and clamping scalars outside those values to the minimum and maximum of the range. This allows for lossless-quality rendering of isovalues within that range. For example, compressing only the 64-127 value range of timestep 270 of the Richtmyer-Meshkov data allows us to render that range on a

machine with 2 GB RAM (Table 3.1). This method is even better suited for medical data such as the visible female femur, when the user is specifically interested in bone or skin ranges. The full original CT scan has highly-variant, homogeneous data for soft tissue isovalues from 0-100, causing the octree volume to actually exceed the original data in footprint. However, considering only the bone isovalues 100-163, we achieve nearly 20:1 compression (Table 3.1). Not coincidentally, such "solid" data segments are best suited for visualization via isosurfacing.

3.4.3 Construction Performance

The bottom-up octree build algorithm is $O(N)$ with regard to the total number of voxels; nonetheless N can be quite large. On a single core of a 16-core 2.4 GHz Opteron workstation, building a single timestep of the 302^3 heptane volume requires a mere 8 seconds and negligible memory footprint, whereas a timestep of the Richtmyer-Meshkov data requires 45 minutes and a footprint of nearly 40 GB. The build itself creates an expanded full octree structure that occupies a footprint of four times the raw volume size. Thus, building octree volumes from large data requires a 64-bit workstation. Although an offline process, parallelizing and optimizing the build would be both desirable and feasible as future work. In addition, the current construction algorithm effectively samples coarser resolutions via recursive clustered averaging. Superior LOD quality could be achieved with bilinear or higher-order filtering.

3.4.4 Memory Footprint Comparison

Octrees generally occupy 20%-30% the memory footprint of the uncompressed grid data, including both the multiresolution LOD structure and min/max acceleration tree. Overall, our method has significantly smaller memory requirement than competing techniques. Employing a dense octree as opposed to a sparse (branch-on-need) octree would require twice the total footprint of the original volume. In systems rendering uncompressed volumes [90, 129], 3D array data are often padded to fit cache lines and bricked to preserve spatial locality, with a footprint penalty of around 15%. Our recursive octree construction inherently guarantees that nearby data will be relatively close in memory. In addition, overhead is required for the separate acceleration structure. The Parker et al. [90] macrocells entail a modest 4.5% for a macrocell depth of 5: around 400 MB for the LLNL data. The Wald et al. [129] kd-tree is more demanding, requiring up to 3 times the original data

footprint. Moreover, our octree volume may occupy less space than simply the acceleration structure of another method.

3.5 Rendering Results

3.5.1 Comparison to Hierarchical Grid

To gauge the performance of our octree traversal algorithm, we compare it to the performance of the Parker et al. [90] hierarchical grid on the same data (Table 3.2). We first consider the performance of each as an acceleration structure only, with both methods retrieving their data directly from the uncompressed original 3D array. The octree performs fairly well, albeit not as fast as the grid. Next, we compare grid and octree performance when looking up octree data via neighbor-finding. The octree surprisingly performs better than it did on array data, likely due to improved cache behavior. The grid performs top-down point location for the first lookup, and subsequently uses neighbor-finding; its results on octree data are noticeably slower. Thus, traversing a unified min/max structure encapsulating octree data yields a distinct advantage.

3.5.2 Richtmyer Meshkov Instability Results

In Figure 3.6 and Table 3.3, we consider the frame-rate performance across several timesteps of the LLNL Richtmyer-Meshkov instability field, a 2048x2048x1920 fluid dynamics dataset. Using octree compression we are able to render this volume at multiple frames per second on a 32-bit laptop; however for an indicator of performance on future multicore CPUs we benchmark fully interactive rates on a 16-node non-uniform memory access (NUMA) workstation of 8 dual-core 2.4 GHz AMD Opterons. For volumes as complex as the LLNL data, it is perhaps preferable to render a 1024^2 frame.

Table 3.2: Octree-grid comparison. Frame-rates for the same scene, traversed by our octree or a 5-deep hierarchical macrocell grid; using either uncompressed 3D array data or compressed octree data. Tests performed on the LLNL data at timestep 270, on a 16-core NUMA 2.4 GHz Opteron workstation. Octree traversal of octree data performs nearly as fast as hierarchical grid with uncompressed array data.

DATA	FPS	
	macrocell grid	octree
3D array	18.9	15.7
octree volume	8.0	**17.5**

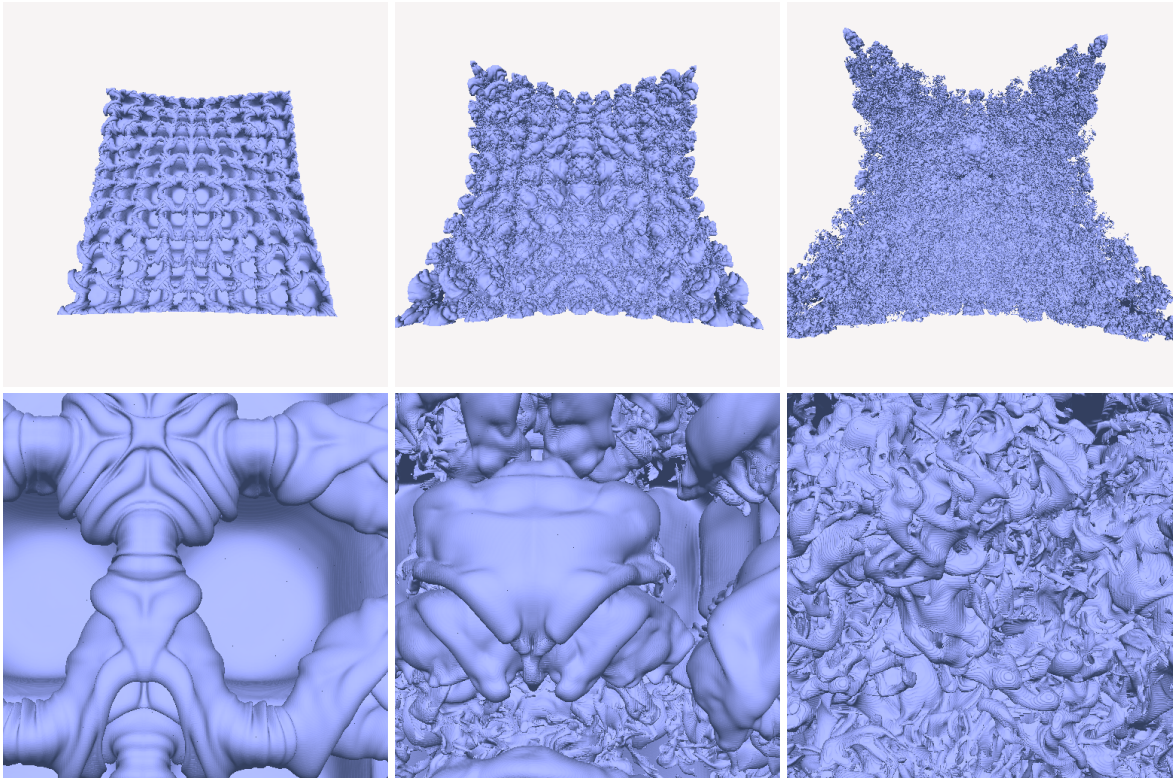


Figure 3.6: Richtmyer Meshkov instability. Top row, from left to right: timesteps 50, 150, and 270. Bottom: same timesteps, with a closer camera.

Performance with the LLNL data is competitive: on the Core Duo laptop, frame rates remain above 2 fps for most camera positions. Results on the Core Duo at timestep 270 actually exceed those achieved by DeMarle et al. [19] on a cluster of 32 PC's, albeit with a distributed shared memory system. They also perform on par with the Wald et al. [129] coherent kd-tree system, which reported around 1 fps on a dual 1.8 GHz Opteron at 640x480 for scenes similar to our far camera image.

Table 3.3: Frame rates of various time steps of the LLNL Richtmyer Meshkov data, on an Intel Core Duo 2.16 GHz laptop (2 GB RAM) and a 16-core NUMA 2.4 GHz Opteron workstation (64 GB RAM). Refer to Figure 3.6 for images.

SCENE	CORE DUO-512^2	NUMA-512^2	NUMA-1024^2
50, far	3.6	25.8	7.4
150, far	2.8	20.0	5.7
270, far	2.4	17.5	4.7
50, close	2.1	15.4	4.3
150, close	1.8	14.2	3.6
270, close	1.7	13.6	3.5

3.5.3 Scalability

While ray tracing is inherently parallel, complicated memory access could potentially compromise scalability on a shared memory or NUMA architecture. Thus, it is worth demonstrating that our technique scales well to multiple processors. We use the parallelization mechanism of the underlying Manta ray tracing architecture [4], which employs a dynamically load-balanced, tile-based thread parallelization scheme. Figure 3.7 demonstrates an efficiency of 91% with 16 processors, which behaves similarly to uncompressed 3D array volumes using the macrocell grid. Once again, the macrocell grid performs slightly faster, but without the benefit of compressed data.

3.5.4 Time-Variant Volumes

One limitation of GPU volume rendering is that, for time-variant volumes, GPU memory restricts the number of timesteps that can be stored and rendered in-core. Bus bandwidth prevents a GPU from streaming textures as effectively as geometry from the CPU. With octree volumes, we can compress full sequences of medium-sized time variant data to fit within main memory of a commodity laptop. The dataset in Figure 3.8 contains 153 timesteps, each of which would occupy 27.5 MB for a total of 4.11 GB. With octree compression, we compress the entire dataset in 678 MB, and render at multiple frames per second on a Core

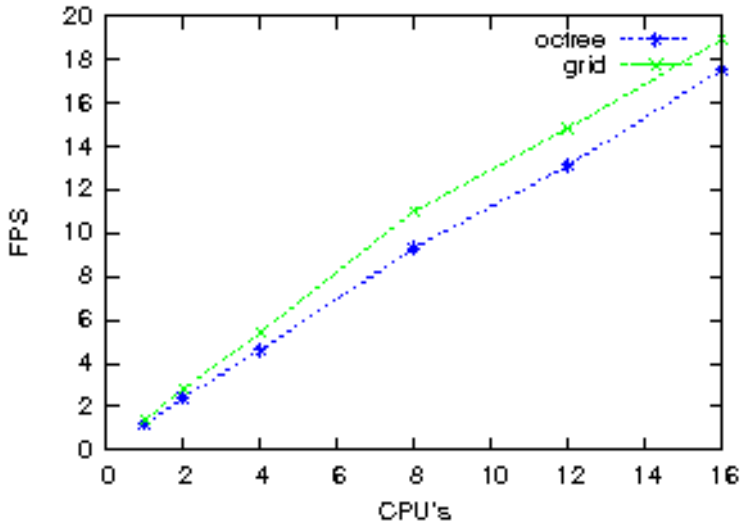


Figure 3.7: Scalability with octree and grid ray tracing. Scalability of our technique on 1,2,4,8,12 and 16 threads, on a 2.4 GHz Opteron NUMA workstation with the LLNL 270 far scene at 512^2. The slight change in slope at 8 threads corresponds to the use of local NUMA memory by two cores instead of one. This demonstrates that our octree technique scales as well as the hierarchical grid with uncompressed data.

Duo 2.16 GHz (Table 3.4).

Octree volumes are useful in that they allow data such as the LLNL to be visualized on machines with limited main memory. However, even in a workstation with 64 GB RAM, memory is a precious commodity. Compression would permit multiple timesteps of the LLNL data to be stored and rendered interactively in sequence.

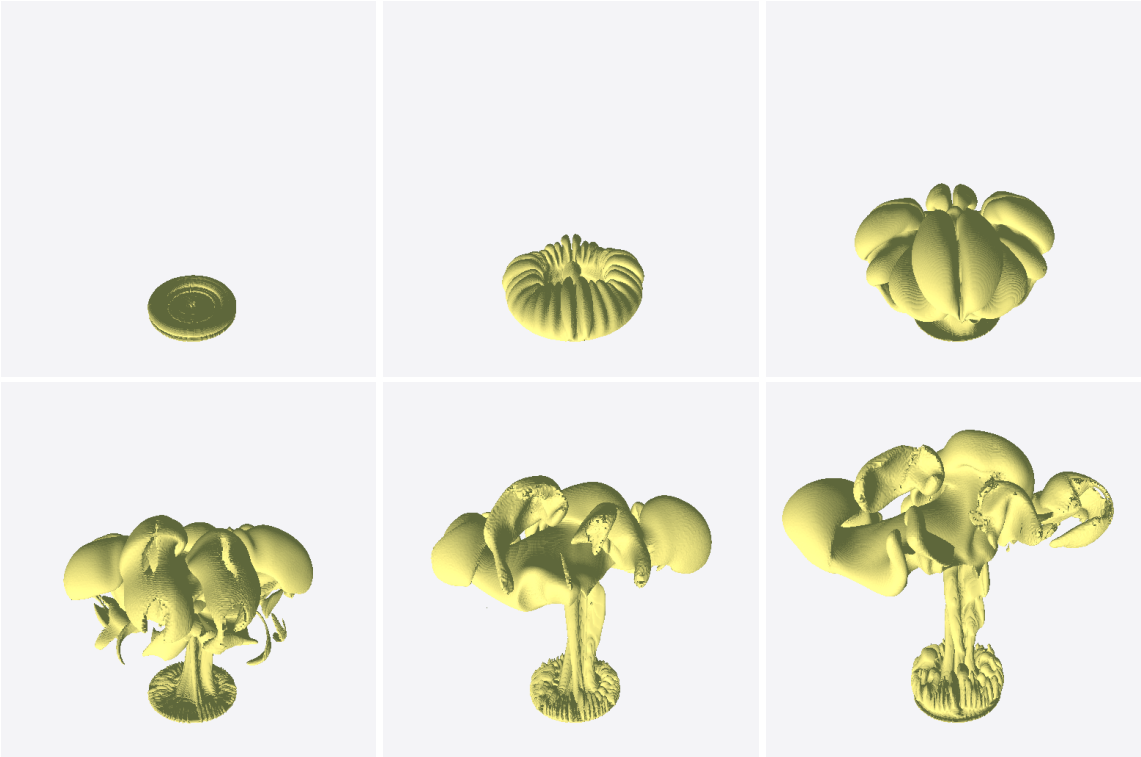


Figure 3.8: Time-variant isosurface rendering. Utah CSAFE heptane simulation, a 302^3 volume. The full sequence of 153 timesteps is stored in 678 MB as opposed to 4.1 GB uncompressed, permitting residency in main memory.

Table 3.4: Frame rates for the CSAFE heptane data, on an Intel Core Duo 2.16 GHz laptop (2 GB RAM) and a 16-core NUMA 2.4 GHz Opteron workstation (64 GB RAM). Refer to Figure 3.8 for images.

TIME STEP	CORE DUO-512^2	NUMA-512^2	NUMA-1024^2
25	17.0	87.1	29.2
50	11.1	60.3	18.0
75	5.7	36.7	9.6
100	4.1	26.6	6.6
125	3.5	28.0	7.1
150	3.2	23.1	6.3

CHAPTER 4

COHERENT MULTIRESOLUTION ISOSURFACE RAY TRACING OF OCTREE VOLUMES

This chapter describes a separate, optimized implementation of the system in Chapter 3. As in the previous chapter, a large data set is losslessly compressed into an octree, enabling residency in CPU main memory. We cast packets of coherent rays through a min/max acceleration structure within the octree, employing a slice-based technique to amortize the higher cost of compressed data access. By employing a multiresolution level of detail (LOD) scheme in conjunction with packets, coherent ray tracing can efficiently render inherently incoherent scenes of complex data. We achieve higher performance with lower footprint than previous isosurface ray tracers, and deliver large frame buffers, smooth gradient normals and shadows at relatively lower cost. In this context, we weigh the strengths of coherent ray tracing against those of the conventional single-ray approach, and present a system that visualizes large volumes at full data resolution on commodity computers. The contributions in this chapter appeared in "Coherent Multiresolution Isosurface Ray Tracing" [60].

4.1 Motivation

The primary goal of this work is to optimize ray tracing of octree volumes, and ideally to deliver interactivity on commodity CPUs. Our main vehicle for such performance gains is coherence. The general premise is to assemble neighboring rays into groups, or packets, with common characteristics. Then, rather than computing traversal and intersection per ray, we perform these computations per packet. High coherence occurs when rays in a packet behave similarly, intersecting common nodes in the efficiency structure or common cells in the volume. Thus, coherence depends on scene complexity as defined by the dataset and camera position.

In addition to the works that inspired the original single-ray octree volume ray tracing application (Section 3.1), we note existing literature that has employed multiresolution methods

for faster isosurface extraction [132] and volume rendering [63]. The point-based method of Livnat and Tricoche [71] is also worth mentioning for its adaptive generation of splats for isosurface rendering. Level of detail methods have already been employed in ray tracing; Igehy et al. [46] proposed ray differentials as a LOD metric for improved mipmap texture filtering. Yoon et al. [141] explored hierarchical splatting as a method of rendering massive mesh models. Djeu et al. [21] employed ray differentials in conjunction with subdivision surfaces for ray tracing LOD geometry. Certainly, the closest ray tracing approach to ours is the coherent kd-tree isosurface ray tracer of Wald et al. [129], which was recently extended by Friedrich et al. [25] with a multiresolution hierarchy for out-of-core progressive rendering.

4.1.1 Coherence via Level of Detail

Successful coherent systems have been optimized for relatively small dynamic polygonal data [130, 127] in which many rays intersect common primitives. In contrast, large volume data exhibit low spatial coherence, particularly from far-away camera positions. Isosurface ray tracing of large data using conservative 2x2 ray packets [129] has suggested performance generally on par with a single-ray system [61]. Coherent traversal may induce more intersection tests than a single-ray traversal, and without optimizations, actually perform worse than a single-ray tracer. To remedy this, we employ a multiresolution level of detail scheme: when the data set is sufficiently complex to hamper coherence, we render a coarser-resolution representation with higher coherence. The octree volume is inherently suited as a multiresolution LOD structure; coarser-resolution voxel data can be stored in interior nodes, allowing the original data, acceleration structure and all LODs to be stored for a fraction of the original uncompressed data footprint. To render a coarser LOD, one simply specifies a cut of the octree at a specified depth. The ray tracer then omits traversal and intersection of subtrees below that depth, and instead intersects coarser, larger cells at termination depth. As more rays intersect a common cell, coherence, and thus speedup, is achieved.

4.1.2 System Overview

As shown in Fig. 4.1, our system consists of offline construction of the multiresolution octree structure from the original data (A); followed by rendering of this octree using a thread-parallel SSE-optimized packet ray tracer (B) as in [129]. The latter distributes ray packets to worker threads (C), which then perform per-packet coherent traversal, SSE isosurface intersection, and shading in that order. Our main contributions involve extending the

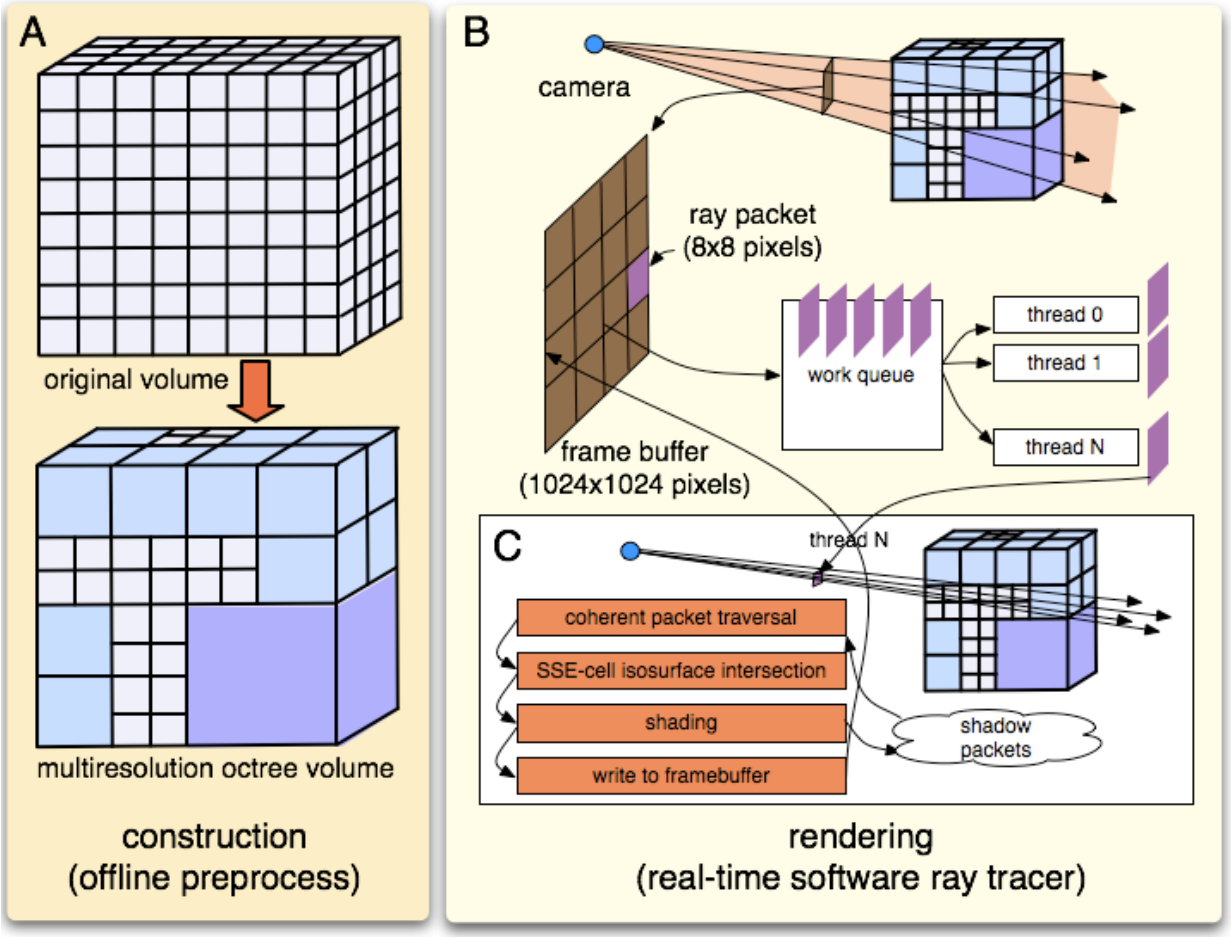


Figure 4.1: Overview of coherent octree volume ray tracing.

static-resolution octree volume to multiresolution (Section 4.2), devising a coherent traversal technique for the octree (Section 4.3) and leveraging the traversal technique to reduce the cost of compressed data access (Section 4.4). Ultimately, our system delivers interactive ray tracing on a desktop CPU while preserving image quality, and enables shading techniques that would be expensive in a conventional noncoherent octree volume ray tracer. Moreover, it allows for scalable rendering of large data that would be difficult for object-order volume rendering on single-GPU systems.

4.2 Multiresolution Octree Volume Construction

Constructing a multiresolution octree volume is fundamentally identical to building a single-resolution octree volume detailed in section 3.2.1 and 3.3.1 of Chapter 3. Our build still proceeds bottom-up, merging voxels with a variance beneath a given threshold (zero, for lossless compression). The only differences are that coarser-resolution voxels are actually

stored, and we require a slightly looser min-max tree.

4.2.1 Extension to Multiresolution

In multiresolution octree volume construction, coarser-resolution consolidated voxels are *always* computed and stored in interior nodes, regardless of whether or not they are leaves. Theoretically, a static-resolution octree volume could use a single array to contain *either* a pointer to a child subtree or a coarser-resolution scalar leaf. In practice, however, the memory savings of this approach were too small to justify the added computation. Multiresolution octree volumes are thus constructed exactly as in the static-resolution implementation [61]: nodes store eight-value arrays for child pointers and scalar leaves. The only difference is that multiresolution rendering actually uses nonleaf scalar data. Interested readers may refer to the pseudocode in Section A.1 of Appendix A for details.

4.2.2 Modifications to Min/Max Tree Computation

The only significant difference between multiresolution and static-resolution construction lies in computing the min/max tree. Static-resolution data, as discussed in Chapter 3, requires the min/max pair of a given voxel to reflect the minimum and maximum of 8 scalar vertices constituting the cell that maps to this voxel (Fig. 4.2). We do not store a min/max pair for each finest-level voxel due to the prohibitive 3x footprint. Instead, we compute them for the immediate parents of the finest voxels (cap nodes in Fig. 3.4 in Chapter 3). As shown in Fig. 4.3 (top), each leaf node must compute the minimum and maximum of its cell, hence accounting for the values of neighbors in the positive X and Y dimensions (left). This yields a min/max pair for the leaf node (right). Neighbors can potentially exist at different depths

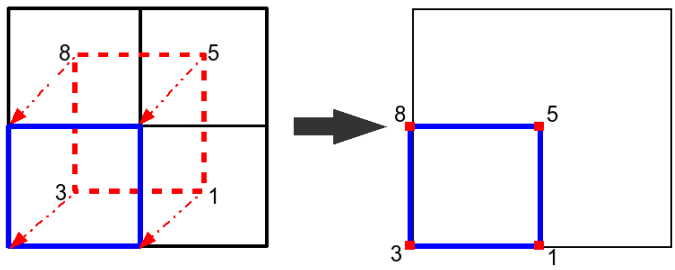


Figure 4.2: Voxel-cell mapping. Given a scalar-centered voxel, we construct its dual *cell* by mapping the scalar to the lower-most vertex, and assigning forward-neighboring scalars to the remaining vertices.

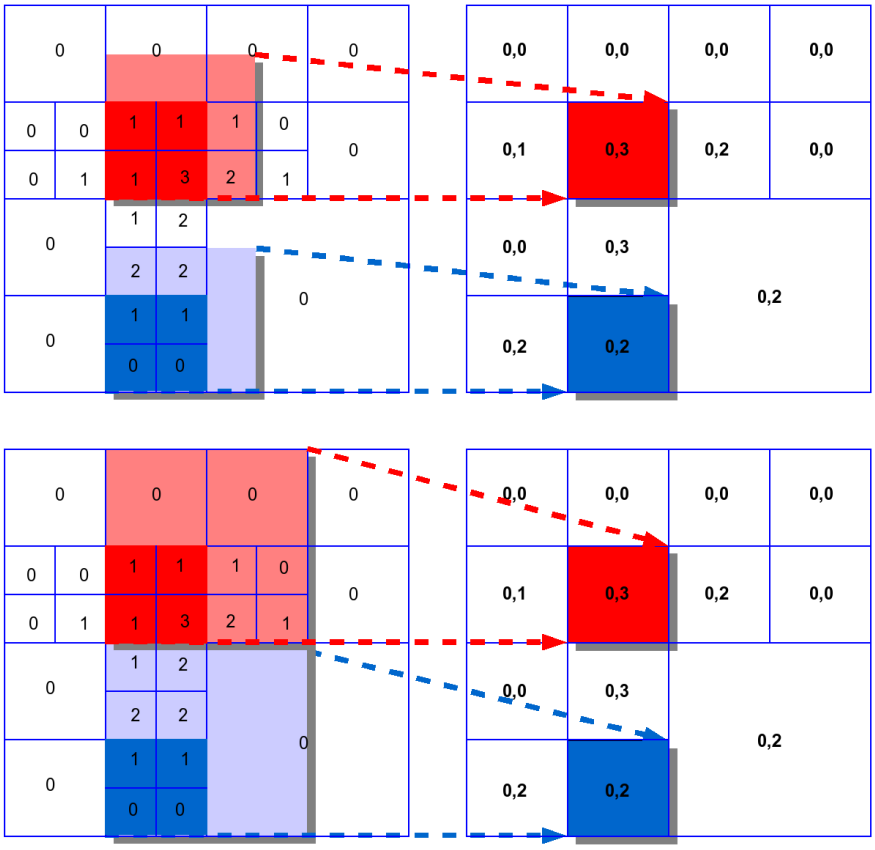


Figure 4.3: Min/max tree construction for multiresolution. Top: Each leaf node must compute the minimum and maximum of its cell, hence account for the values of neighbors in the positive X and Y dimensions (left). This yields a min/max pair for the leaf node (right). Neighbors can potentially exist at different depths of the octree, as is the case for at the blue leaf node. Bottom: For multiresolution data, we must include wider neighbors at coarser resolutions into the min/max computation.

of the octree, as is the case for at the blue leaf node.. For multiresolution data, cells may have any power-of-two width, and we accordingly consider forward-neighbors at each depth of the min/max tree (Fig. 4.3, bottom). As a result, the min/max tree for a multiresolution octree volume is looser than that of static-resolution data. In practice, the impact on performance is negligible for the data we test.

4.3 Coherent Octree Volume Ray Tracing

Having constructed a compact octree volume with an embedded min/max acceleration structure, we now turn to the task of building a coherent ray tracing system. In general, we seek to optimize for coherence as aggressively as possible, namely by implementing a

vertical SSE packet architecture and a frustum-based octree traversal similar to the coherent grid traversal of Wald et al. [130].

4.3.1 SSE Packet Architecture

A coherent ray tracer achieves its performance by operating on groups of neighboring or similar rays in packets. To exploit coherence during primitive intersection, we perform computations on SIMD groups of four rays (frequently referred to as *packlets*) and mask differing hit results as necessary. Performing these SIMD computations requires that we store ray information vertically within a packet. For example, ray directions are stored as separate arrays of X,Y,Z components, as opposed to a single horizontal array of 3-vectors. These vertical arrays are 16-byte-aligned, permitting us to access a packlet of four rays at a time in a single SSE register. Similarly, the packet structure stores aligned SSE arrays of hit results, such as hit position and normals.

4.3.2 Coherent Traversal Background

As an efficiency structure for ray tracing, the octree affords several different styles of traversal. With coherent ray tracing, we are given the choice between depth-first traversal similar to a kd-tree [131] or BVH [127], or a breadth-first coherent grid traversal (CGT) approach [130]. We choose the latter for several reasons. Our primitives are regular, non-overlapping cells, similar to large spherical particle data sets for which CGT has proven effective by Gribble et al. [35]. More significantly, the breadth-first nature of the CGT algorithm allows for a clever slice-based technique that amortizes voxel look-up from the octree when reconstructing the vertices of multiple cells.

4.3.3 Coherent Grid Traversal Algorithm

The original CGT algorithm departs from single-ray grid traversal in that it considers full slices of cells contained within a ray packet's bounding frustum, as opposed to marching across individual cells. The algorithm first determines the dominant X,Y,Z axis component of the first ray in each packet. This is denoted $\vec{K}$, and the remaining axes are denoted $\vec{U}$ and $\vec{V}$. Then, we consider the minimum and maximum u and v coordinates at the $k = 0$ slice, and note that the increment *du,dv* for a single unit along the march axis $\vec{K}$ is constant. We store this increment in a single SSE floating point vector, $d_{uv} = [du_{min}, dv_{min}, du_{max}, dv_{max}]$. Next, we determine the first and last k slice where the packet frustum intersects the volume.

We begin at the u,v extents, $e_{uv} = [u_{min}, v_{min}, u_{max}, v_{max}]$, the minimum and maximum of enter and exit points on that slice of cells. To intersect primitives, we truncate these values to integers and iterate over all cells in that given $\vec{U}$,$\vec{V}$ range. To march to the next slice, we add the constant increment. Thus, a nonhierarchical grid march is accomplished with a single SIMD addition and a SIMD float-to-integer truncation. Unlike a single-ray DDA grid algorithm [1], cells may be traversed in arbitrary $\vec{U}$,$\vec{V}$ order; however the $\vec{K}$ order is invariably front to back, permitting early termination. The 2D analog of this algorithm is illustrated in Fig. 4.4.

4.3.4 Macrocell Hierarchical CGT

The original CGT paper [130] implemented a two-level hierarchy, with a single layer of macrocells each corresponding to 6 grid cells. For small polygonal data, this was generally sufficient. As the smallest volume we test is 302^3, a more robust hierarchy could be desirable for our application. We extended the CGT algorithm to arbitrary number of macrocell layers similarly to Parker et al. [89], and found that a recursive 2^3 macrocell hierarchy – equivalent

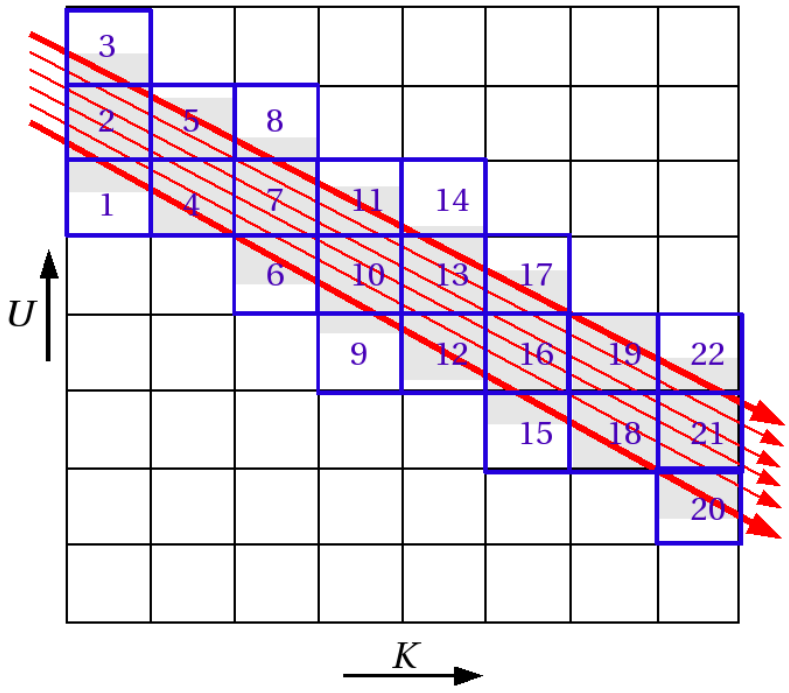


Figure 4.4: Coherent grid traversal. The CGT algorithm [130] traverses a packet of rays through a grid slice by slice along a major march axis $\vec{K}$, iteratively incrementing slice extents by the differential of the bounding frustum along the nonmajor axis $\vec{U}$ (and a third axis $\vec{V}$ in 3D).

to a full octree – consistently yielded the best performance for volumes larger than 256^3. The macrocell traversal employs an array stack structure to avoid recursive function calls: this stores the *u,v* slice and increment for all macrocell levels, the current slice within the current macrocell level, and the next slice at which to return to parent macrocell traversal. When all rays in a packet have intersected or the packet exits the root macrocell level, traversal terminates. The approach is that of a recursive grid sharing common coordinate space on the given volume dimensions, in which each macrocell block is a multiple M of its children. Thus, child coordinates are always an M-multiple of parent macrocell coordinates. Child macrocells, or the volume cells themselves, are traversed when *any* macrocell in a given slice is nonempty – specifically, when our desired isovalue is within that macrocell's min, max range. Then, the packet frustum traverses full slices of that macrocell level's children. As shown in Fig. 4.5, our hierarchical grid employs recursively superimposed macrocell blocks, with each parent containing 2^3 children, for alignment with the octree volume. We depict a 3-deep hierarchy, with blue, yellow and green extents corresponding to macrocell layers from coarsest to finest. Macrocells are only traversed when they contain our desired isovalue, as illustrated by the "surface" at the dotted line. With an octree, macrocells are implicit; min/max pairs are retrieved from the octree nodes via hashing.

4.3.5 Implicit Macrocell Grid Traversal of Octree Volumes

Our octree volume traversal is effectively coherent grid traversal of an implicit macrocell hierarchy, in which min/max pairs are retrieved from octree interior nodes instead of macrocells. Rather than repeatedly multiplying grid coordinates by the macrocell width M, octree nodes at all depths share a common coordinate space $[0, 2^{d_{max}}]$, where d_{max} is the maximum depth of the tree. Some macrocell traversal computation can be optimized for the binary subdivision of the octree. When recursing from a parent to traversing children, the macrocell grid multiplies the k-slice by the macrocell width M; in the octree $M = 2$, a bitwise left-shift. Computing the next macrocell slice requires a simple $+2$ addition.

4.3.6 Mapping Macrocells to Octree Nodes

Traversing implicit macrocells over an octree requires particular attention, as a single coarse scalar leaf node in the octree may cover multiple finer-level implicit macrocells. Given an implicit macrocell coordinate, we seek the deepest octree child that maps to it. We then use the min/max pair in the parent node, corresponding to that child, to perform the isovalue

culling test. As lookup is costly, we store the path from the octree root to the current node along the *u,v*-minimal ray of the frustum. We then use neighbor-finding as detailed in [61] to inexpensively traverse from one node to the next. Hierarchically recursing from a parent node to a child requires a single lookup step in the octree.

4.3.7 Default Slice-Based Traversal

At shallow levels of the octree, the packet frustum typically traverses a single common macrocell. At deeper levels, the *u,v* extents encompass multiple macrocells, so we must neighbor-find numerous octree nodes. By default, macrocell CGT stops iterating over a slice when *any* node is nonempty, and proceeds to traverse slices of children nodes. This ensures that traversal is performed purely based on the packet frustum as opposed to individual rays, and preserves the breadth-first coherent nature of the algorithm. Unchecked, it also causes numerous unnecessary octree lookups and ray-cell intersection tests. To mitigate this, we implement the two following optimizations.

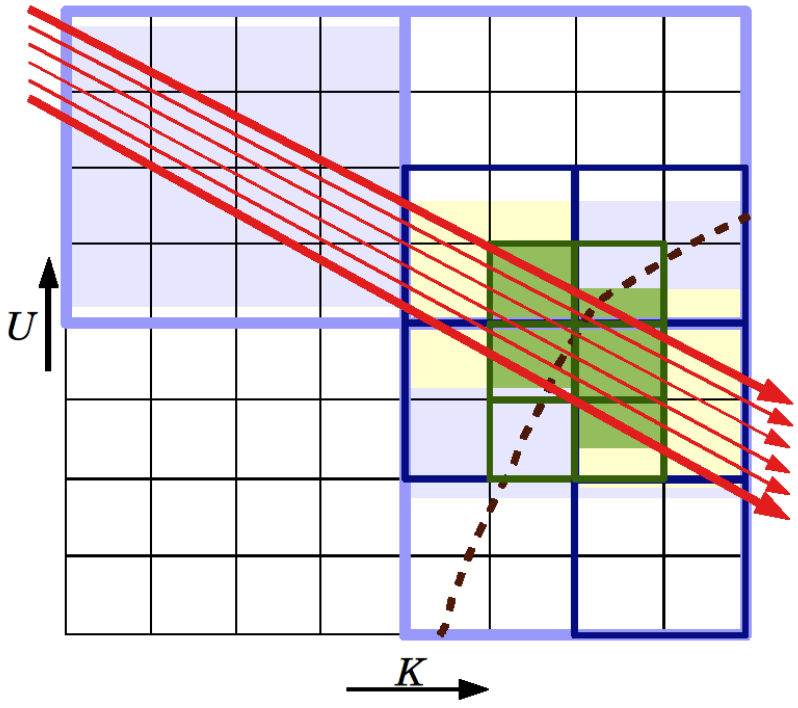


Figure 4.5: Coherent octree traversal via implicit macrocells.

4.3.7.1 Culling empty cap-level macrocells

To avoid unnecessary intersections and octree hashing, we clip the u, v slice corresponding to the deepest-level macrocells, one level above actual cell primitives. To do this, we iterate over the min/max pairs corresponding to the finest *available* octree depth. When traversing at maximum resolution, the deepest macrocells correspond to cap nodes (Fig. 3.4). Within this iteration, if a macrocell contains our isovalue, we compute new slice extents based on the minimum and maximum u,v coordinates. If the macrocell is empty, we omit it from extent computation. The effect is to clamp the u,v slice so that it more tightly encloses nodes with the desired isovalue. Fig. 4.6 illustrates this where we first clip slices of deepest macrocells, corresponding to cap nodes of the octree at depth $d_{max} - 1$. We narrow the u,v slice extents by omitting macrocells with ranges outside our value; only the shaded cells containing our isovalue are considered.

4.3.7.2 Clipping the cell-level slice to active rays

To further reduce the number of cell primitives in a slice, we intersect individual rays with the world-space bounding box formed by the current u,v slice. When rays have already successfully hit a cell, they are "inactive" and can be safely ignored even if they intersect

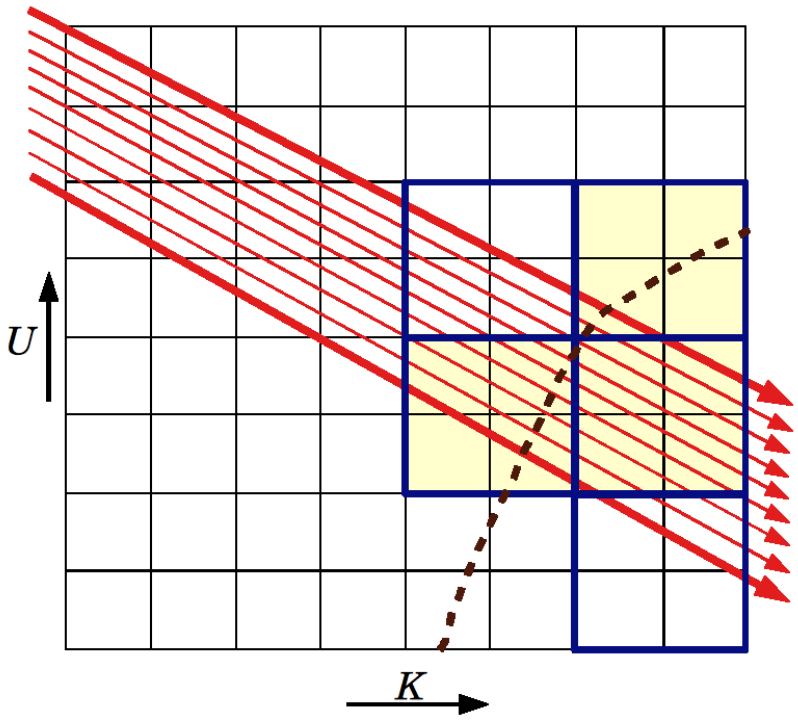


Figure 4.6: Culling empty macrocells from cap-node slices.

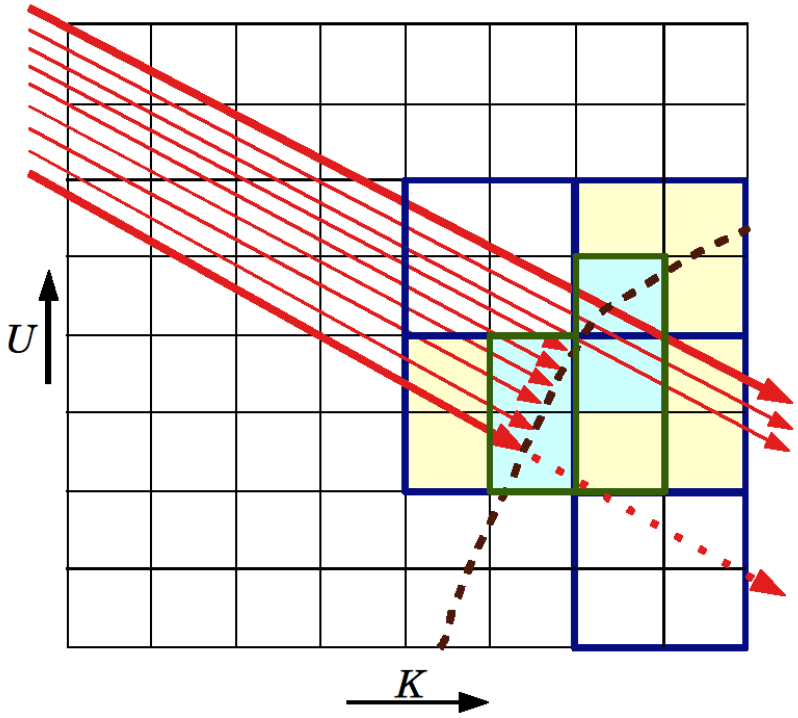


Figure 4.7: Clipping cell slices to fit active rays.

the slice bounding box. As shown in Fig. 4.7, this enables us to considerably shrink the *u,v* extents before intersecting a $\vec{K}$-slice of cells by simply computing the minimum and maximum of the enter and exit hit coordinates of active rays.

4.3.8 Cell Reconstruction from Cached Voxel Slices

Having clipped the primitive-level slice to as small a *u,v* extent as possible, we are ready to perform ray-cell intersection. Our ray-tracing primitive is a cell with 8 scalar values; one at each vertex. However, the data primitives in our octree volume are voxels. Using the same duality employed by min/max tree construction, we map octree voxels to the lower-most vertex of each cell (Fig. 4.2). Our task now is to reconstruct cells efficiently from the octree, exploiting coherence whenever possible.

4.3.9 U,V Voxel Slice Filling

In single-ray and depth-first traversals, cells are constructed independently, given a lower-most voxel from traversal, and using neighbor-finding to look up the remaining 7 voxels. However, adjacent cells share vertices – much neighbor-finding effort is duplicated. With our octree CGT, we can iterate over an entire slice of adjacent *u,v* cells, access each voxel

once, and store the results in a 2D array buffer. We add 1 to the maximal u,v slice extent to account for forward cell vertices in those directions. Then, we iterate over the u and v components of the slice, performing neighbor-finding from one coordinate to the next. By iterating in a scanline, the neighbor-finding algorithm need only find a common ancestor along one axis, and is slightly cheaper. We store the voxel results for this slice in a 2D array buffer, and look up values from this buffer to reconstruct four vertices of each cell in the slice. The remaining four vertices can be reconstructed in the same fashion by filling in a second buffer for the $k+1$ slice. Thus, to find the eight vertices of each cell, rather than neighbor-find seven forward-neighbors per voxel, we exploit our slice-based traversal to look up and cache $\vec{K}$-slices of voxels, amortizing and reducing the cost of data access. Fig. 4.8 illustrates filling of five successive slices, with like shades representing where cached voxels are used to avoid repeat neighbor-finding.

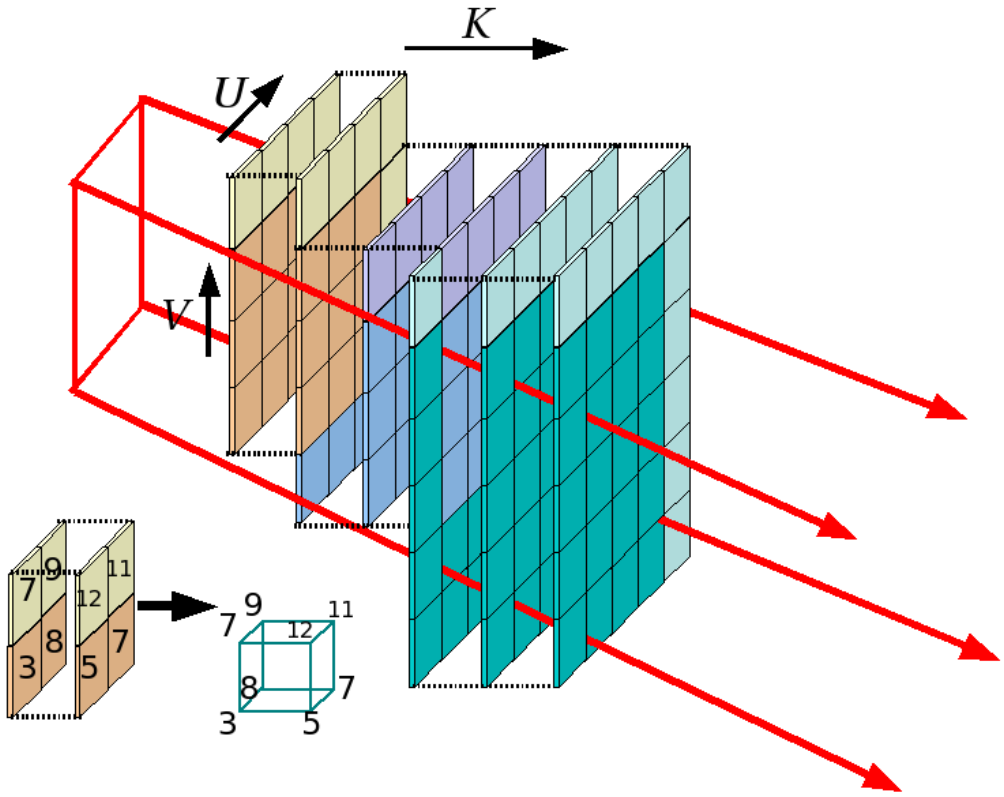


Figure 4.8: Slice-based cell reconstruction algorithm.

4.3.10 Copying the Previous-Step $\vec{K}$-Slice

In cell reconstruction, we also exploit voxel coherence along the $\vec{K}$ axis. For this, we note that vertices on either the front (k) or back ($k+1$) slice of each cell are shared from one traversal step to the next, depending on whether the $\vec{K}$ march direction is positive or negative. In either case, we can copy an advancing slice buffer from the previous traversal step into a posterior buffer of the current traversal step (Fig. 4.8). We must account for the traversal offset in the minimum u,v coordinates between the two buffers, and perform neighbor-finding for voxels not buffered from the previous step, due either to that offset or different maximal u,v extents.

Pseudocode for the full traversal algorithm is outlined in Algorithm 1 in Appendix A. Here, d_{uv} and e_{uv} are SSE vector variables, and k is an integer. Cap depth is $d_{cap} = d_{max} - 1$. For multiresolution, the algorithm is similar except we may intersect slices at lesser stop depth than d_{max}.

4.3.11 Ray-Cell Intersection

With our cached slice buffers, we can iterate over cell primitives and reconstruct cell vertices. To compute the ray-isosurface intersection, we iterate over all SIMD packlets, discarding packlets that are inactive (have already intersected) according to the per-packlet hit mask. For each packlet, we first check that each at least one actually intersects the bounding box of the cell in question, and then proceed to compute the ray intersection with the implicit isosurface.

For ray-cell intersection, we seek a surface inside a three-dimensional cell with given corner values (Fig. 4.2), such that trilinear interpolation of the corners yields our desired isovalue. This entails solving a cubic polynomial for each ray; the hit position is given at the first positive root. Our implementation uses the Neubauer iterative root finder proposed by Marmitt et al. [75]. Computation is performed per-packlet. If any ray in the packet intersects successfully, we compute the gradient normals for that packlet. We do not defer normal computation due to the prohibitive cost of reconstructing cell vertices twice.

4.4 Multiresolution Level of Detail System

Our optimized coherent traversal algorithm significantly outperforms single-ray traversal on simple scenes; and due to the lower data lookup cost even exhibits a factor-of-two speedup moderately incoherent scenes in which more than one ray per packlet seldom intersects the

same cell. However, coherence breaks down on highly complex scenes, where rays are separated by multiple cells that are never intersected. This pathological case is common with far views of large data sets. This behavior is detailed more fully in Section 6.6. The purpose of the multiresolution system is to manage pathological cases posed by large data, and preserve coherence with only minor sacrifice in quality.

4.4.1 Resolution Heuristic

4.4.1.1 Stop depth

The general vehicle for the multiresolution scheme is determining an effective depth at which to stop traversing children, and instead reconstructing cells to intersect. Coarser-resolution voxels are explicitly stored in the scalar leaf fields of interior nodes, regardless of whether a finer-resolution subtree exists. When the traversal algorithm stops, cell reconstruction proceeds exactly as it would at the finest resolution, except given a stop depth d_{stop} it increments the u,v coordinates by $2^{d_{stop}}$ instead of simply 1 at the finest resolution. Thus, the octree hash scheme operates on canonical octree space $[0, 2^{d_{max}}]$, regardless of LOD depth.

4.4.1.2 Pixel-to-voxel width ratio

A more difficult problem in formulating the multiresolution scheme is determining which parts of the scene should be rendered at which resolution. Generally, we note that when multiple voxels project to the same pixel, a coarser level of resolution is desirable. LOD techniques for volume rendering often use a view-dependent heuristic to perform some projection of voxels to screen-space pixels, and identify distinct regions of differing resolutions [63]. In the case of ray-casting with a pinhole camera, the number of voxels that project to one pixel varies quadratically with the distance from the camera. As aspect ratio is constant, we may simply consider the linear relation along one axis $\vec{U}$, namely the increment between each primary ray along $\vec{U}$, du. Then, we can render the coarser resolution at d_{stop} when $du = Q_{stop} * dV$, where dV is the $\vec{U}$-width of a voxel, and Q_{stop} is some constant threshold. As the $\vec{U}$-width of a single pixel, dP, is simply a multiple of du, we can simply reformulate our constant as a ratio of pixel width to voxel width dP/dV, where $Q_{stop} = (du/dP) * (dP/dV)$.

4.4.1.3 Packet extents metric

Ideally, our LOD metric should be evaluated per packet. An obvious choice would be the du width of the packet, given by the aforementioned u,v slice extents. One could render a coarser resolution whenever the number of cells in a slice at the current resolution surpassed some threshold. Unfortunately, at the same k-slice, the du_{packet} could vary between packets, causing neighboring rays to intersect different-resolution cells, hence resulting in seams. We desire a similar scheme that allows us to perform transitions consistently between packets.

4.4.1.4 LOD mapping via $\vec{K}$ transition slices

To ensure consistent transitions from one resolution to the next, we compute a view-dependent map from resolution levels to world-space regions along the major traversal axis $\vec{K}$. We note that the width of a pixel corresponds to the distance between primary rays along the $\vec{U}$ and $\vec{V}$ axes, which increases with greater t, as we move farther from the camera origin. If we consider a major march direction $\vec{K}$, we can find the exact k slice coordinate where any given number of voxels corresponds to exactly one pixel. This is similar to the per-ray metric approach, except it solves where $du = Q_{stop} * dV$ at a discrete $\vec{K}$-slice, k. As packets traverse the octree one $\vec{K}$-slice at a time, we have a constant world-space LOD function that can be computed on a per-packet basis.

We multiply the ratio of pixel width to voxel width, dP/dV, by the power-of-two unit width corresponding to each depth d of the octree. Then, we solve for the t parameter where this voxel width is equal to the distance between viewing rays, du_{camera}. Finally, we evaluate $\vec{K}$-component of the direction ray to compute the $\vec{K}$-slice where our fixed dP/dV ratio occurs, $k_{transition}[d]$. These mark the transition slices from each resolution to its coarser parent. The array is computed once per frame, using Algorithm 2 in Appendix A. The dP/dV constant is thus our base quality metric.

4.4.2 Multiresolution Traversal

Rather than determining the major march axis $\vec{K}$ per packet, we decide it once per frame based on the direction vector of the camera. While this causes some packets to perform CGT on a nondominant axis, in practice there is no appreciable loss in performance with a typical 60-degree field of view.

The traversal algorithm determines the initial transition slice when it computes the first k-slice of a packet, by finding the first $k_{transition}[d] < k$. Then, before recursively traversing a

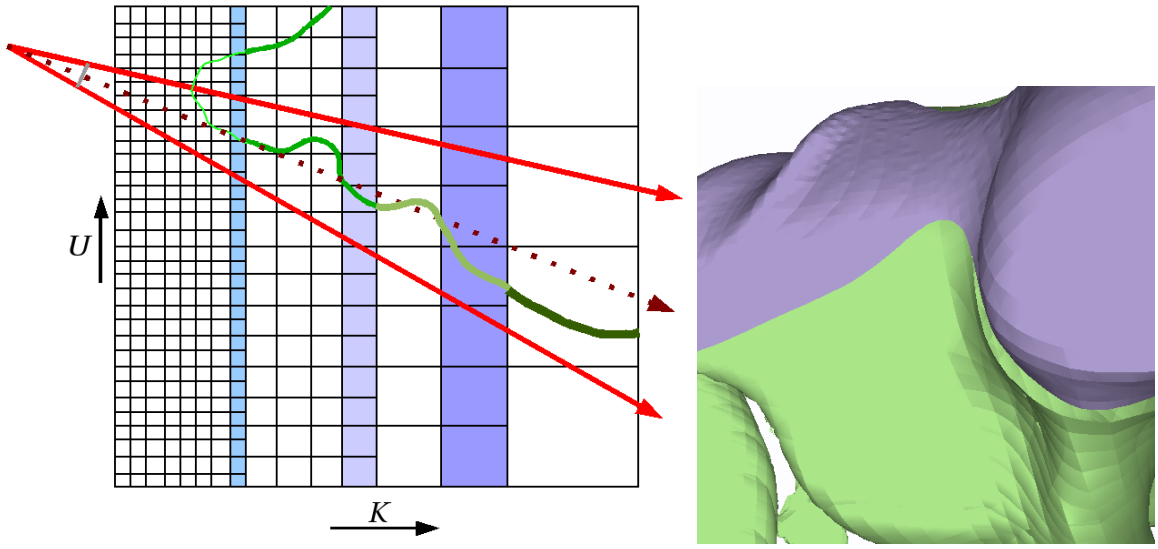


Figure 4.9: Multiresolution transitions illustrated. Left: multiresolution transition slices along the $\vec{K}$ axis. Right: transitions are smoothed by substituting coarse-LOD values at fine-level cell vertices at the transition slice.

child slice at the current resolution depth, we check if $k_{child} >= k_{d-1}$, the slice corresponding to transition to the *next* coarser resolution. When that occurs, we omit traversal of the child and perform cell reconstruction. The current resolution depth is then decremented, so the traverser seeks the subsequent coarser-resolution transition slice. This process is illustrated in Fig. 4.9 (left).

4.4.3 Smooth Transitions

Isosurfaces are piecewise patches over their respective cells, and can vary both topologically and locally from one resolution to the next. As such, discontinuities arise at transition slices between finer and coarser isosurfaces. While these discontinuous surfaces are technically "correct" with respect to each resolution, it is frequently desirable to mask the multiresolution transition and render a single smooth surface. To accomplish this, our slice-based reconstruction algorithm checks if each $\vec{K}$-slice is equal to the next k_d transition slice. If it is, we look up voxel data from the octree at coarser depth $d-1$ as opposed to the current default depth d. This guarantees identical voxel values on either side of the transition, and thus continuous surfaces (Fig. 4.9, right). Exceptions may occur in cases of gross disparity between each resolution of the scalar field, where topological differences cause a surface to *exist* at one resolution but not the other This is common in highly entropic regions

of the Richtmyer-Meshkov data. In these cases, it is desirable to omit smooth transitions and expose levels of detail via color-coding.

4.5 Shading

Our technique affords better flexibility in shading the isosurface. One limitation of the octree volume is that data access for cell reconstruction is expensive, discouraging techniques such as central-differences gradients that require additional neighbor-finding. With slice-based coherent traversal, we are able to amortize the cost of cell reconstruction as shown previously. Multiresolution allows us to simplify the casting of shadow rays and illustrate depth cues with less performance sacrifice.

4.5.1 Smooth Gradient Normals

By default, normals are computed using the forward-differences gradient at the intersection point within the given cell. The disadvantage of this method is that such gradients are continuous only within each cell. The isosurface itself is formed from piecewise trilinear patches with C_0 continuity at cell edges. For a more continuous normal vector field, and better visual quality, we can compute gradients on a central differences stencil to ensure C_1 continuity along cell edges.

To compute the central differences gradient, we use a stencil of three cells along each axis; thus 64 cell vertices (voxels) must be found during reconstruction. Reconstructing a 4^3 voxel neighborhood per-ray is costly in noncoherent octree volume isosurface ray tracing [61]. Coherent reconstruction with cached slices allows for smooth normals with far lesser penalty. In a noncoherent ray tracer this entails eight times the lookup cost of forward differences, causing worse than half the forward-differences performance. In our coherent system, we return to the slice-based cell reconstruction technique to amortize that cost of neighbor-finding. We simply retrieve two additional rows and columns of voxels, corresponding to $u_{min}-1, v_{min}-1$ and $u_{min}+2, v_{min}+2$ coordinates. In addition to our existing 2D array buffers for the k and $k+1$ slices, we store two additional buffers corresponding to the $k-1$ and $k+2$ slices. We then use this four-wide kernel with a central-differences stencil to compute the gradient: $\frac{1}{2}(V_{X-1,Y,Z)} - V_{(X+1,Y,Z)})$ along the X axis, and similarly for the Y and Z axes. Performance with central differences is typically 15%-30% slower than with forward differences. The visual improvement is arguably worth this cost (Fig. 4.10).

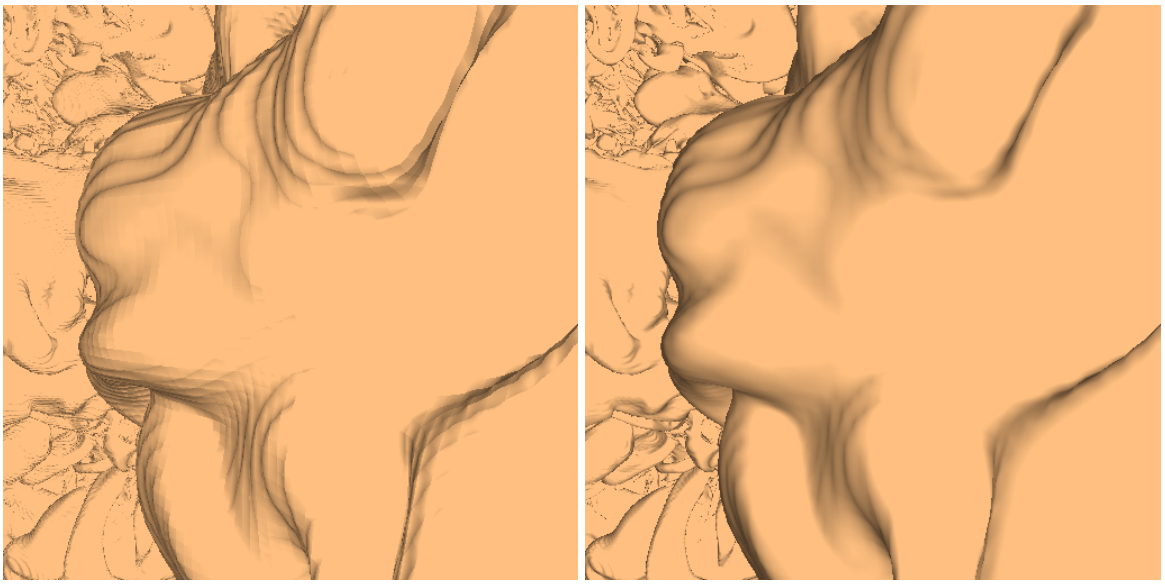


Figure 4.10: Shading with forward and central differences. Gradient normals computed on a forward differences stencil yielding 5.5 FPS (left), and a central differences stencil at 4.7 FPS (right) on an Intel Core Duo 2.16 GHz with a 512^2 frame buffer.

4.5.2 Shadows

An oft-cited advantage of ray tracing is that shadows can be computed trivially without adding geometric complexity or implementing sophisticated multipass texturing techniques. In practice, tracing shadows doubles the cost of casting each ray that successfully hits an object. Computing shadow rays in a coherent packet system is more complicated than for a single-ray tracer, as individual rays must be masked and shadow packets generated based on the hit results of the primary rays. Fortunately, point-light shadows may be cast from the light to the primary hit point; thus they share a common origin and benefit from coherent optimizations. Our primary goal being interactivity, we are interested in hard shadows that may not appear photorealistic, but adequately provide depth cues to the viewer. As such, we can exploit the LOD system to cast faster coherent shadow rays through a coarser-resolution representation of our volume – for example, using a shadow ray *dP/dV* of twice the viewing ray *dP/dV*. By coherently casting shadow rays through a coarser resolution, we can achieve higher performance and provide similar depth cues. This yields framerates only 20%-30% slower with shadows than without (Fig. 4.11).

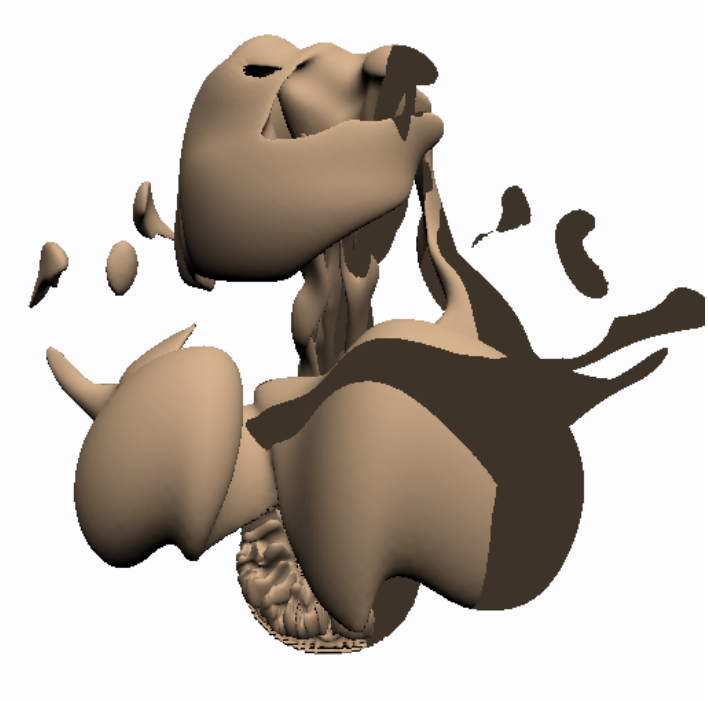


Figure 4.11: Ray tracing with shadows. With centrally-differenced gradient normals, the above shadowed scene renders at 3.9 FPS on an Intel Core Duo 2.16 GHz with a 512^2 framebuffer, as opposed to 5.1 FPS without shadows.

4.6 Results

For results of the octree construction and compression, refer to Section 3.4 in Chapter 3. In this section, we evaluate performance of our system by first considering coherent octree traversal alone, and then analyzing the performance of the multiresolution system.

4.6.1 Coherent Traversal Analysis

The main purpose of our slice-based algorithmic enhancements, and indeed of traversal itself, is to minimize the number of cells that must be intersected. By employing packets and the breadth-first CGT frustum algorithm, we are able to dramatically reduce both the computational and memory access costs of traversal. Finally, when multiple rays in a SSE packlet intersect the same object, we may effectively perform up to four intersections for the price of one. For these reasons, we are able to achieve significant speedups on highly coherent simple scenes. Even with moderately complex scenes where a pixel seldom contains more than one voxel, and SIMD intersection yields little speedup, slice-based reconstruction effectively doubles performance (Table 4.1). Moreover, rendering time is strongly correlated with the number of ray-cell intersections. Performance profiling reveals that only 5%-15%

of CPU time is spent in traversal, compared to over 70% in reconstruction and intersection.

4.6.2 Packet Size

Table 4.2 shows the impact of packet size on performance. Our implementation employs fixed packet size for traversal. This is appropriate for our application, as we seek to render isosurfaces with constant complexity. Later, we enforce this via the pixel to voxel width ratio in the LOD scheme. Empirically, we find that packets of 8x8 work best for scenes where one to 4 rays intersect a common cell. Packets of 16x16 rays yield little benefit even for simple data, and perform poorly on complex scenes of large data.

4.6.3 Incoherent Behavior Without Multiresolution

Complex scenes reveal the shortcoming of coherent traversal. Because traversal is not computed on a per-ray basis, but solely from the packet frustum corners, it frequently looks up cells that would have been correctly ignored by a more expensive single-ray traverser. Our clipping optimizations (Figs. 4.6, 4.7) noticeably alleviate this, as we can see in Table 4.1. However, for complex scenes such as far views of large data, rendering cost is totally bound by intersection (Table 4.2). Ultimately, frustum-based traversal causes large numbers of cells to be looked up, though no rays in the packet actually intersect them. This in turn causes many unnecessary intersection tests to be performed. Successful intersection tests are no less expensive, as packlet-cell intersection degenerates to single-ray performance without primitive-level coherence. These higher costs eventually overwhelm any gains made by more efficient traversal, and cause the coherent ray tracer, without multiresolution, to perform worse than a single-ray algorithm on sufficiently complex scenes.

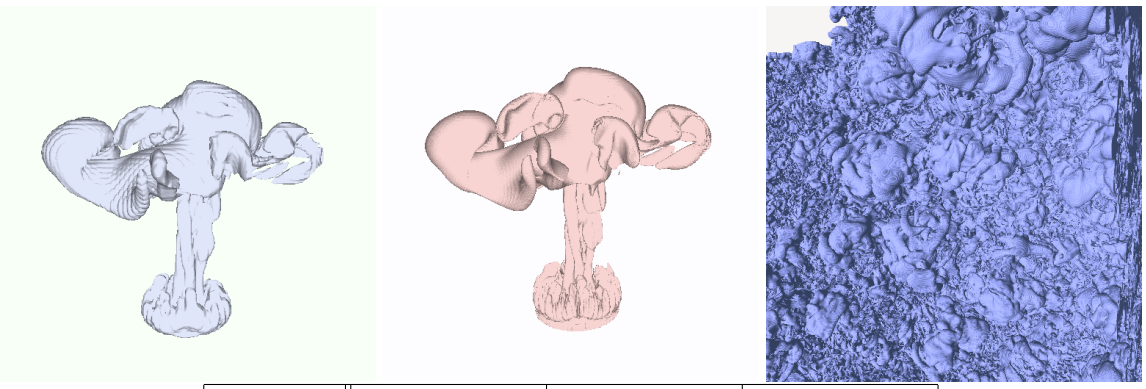
4.6.4 Multiresolution Results

The combination of multiresolution level of detail and coherence enables frame rates up to an order of magnitude faster for coherent scenes. With large volume data and small frame buffers, coherence is less common, but in general it is possible to decrease *dP/dV* to achieve interactive frame rates and interesting, albeit coarser-quality, representations of the data. For highly entropic large volume data, this behavior is frequently useful as coarser LODs inherently possess less variance, thus manifesting less aliasing. However, coarser LOD rendering are also less correct with respect to the original resolution, as shown in Fig. 4.12.

Table 4.1: Results from clipping optimizations, when ray-casting a moderately complex scene with low primitive-level coherence, from the heptane fire dataset (HEP302, Table 4.2). We compare single-ray traversal and 8x8 octree-CGT packet traversal with and without optimizations. *+slice:* use slice-based cell reconstruction. *+mcell:* clip the deepest macrocell slice extents to discard nodes not containing the isovalue. *+cell:* clip the cell slice extents to the set of active rays. *+mulres:* multiresolution scheme, with $dP/dV = 1$. Tests at 512^2 using one core of an Intel Core Duo 2.16 GHz.

TRAV.	LOOKUPS	ISECS	L/RAY	I/RAY	FPS
single	314707	166719	1.2	0.64	2.3
packet	1187798	469560	4.5	1.8	.78
+*slice*	1187798	469560	4.5	1.8	2.2
+*mcell*	561889	124221	2.14	0.47	3.9
+*cell*	270123	120514	1.0	0.47	4.6
+*mulres*	98055	44419	0.37	0.17	7.6

Table 4.2: Results with coherent packets, showing the net number of intersections per ray and frames per second with a single-ray tracer, and our coherent system with varying packet sizes. We examine three scenes of increasing complexity. Leftmost (HEP64) is the 64^3 downsampled heptane data, which has high intersection-level coherence. The full 302^3 heptane data (HEP302) has low intersection-level coherence, but benefits from coherent traversal. The 2048^3 RM data set yields even less coherence, and is a pathological case for packet traversal. Benchmarks on a single core of an Intel Core Duo 2.16 GHz, with a 512^2 frame buffer and multiresolution disabled.



SCENE	HEP64		HEP302		RM	
	I/RAY	FPS	I/RAY	FPS	I/RAY	FPS
single	0.70	1.9	0.64	2.3	3.58	0.57
coherent						
2x2	0.26	4.3	0.5	2.84	5.65	0.38
4x4	0.11	9.7	0.45	4.54	7.94	0.33
8x8	0.058	14.4	0.47	4.6	12.4	0.22
16x16	0.041	14.5	0.50	2.81	20.5	0.08

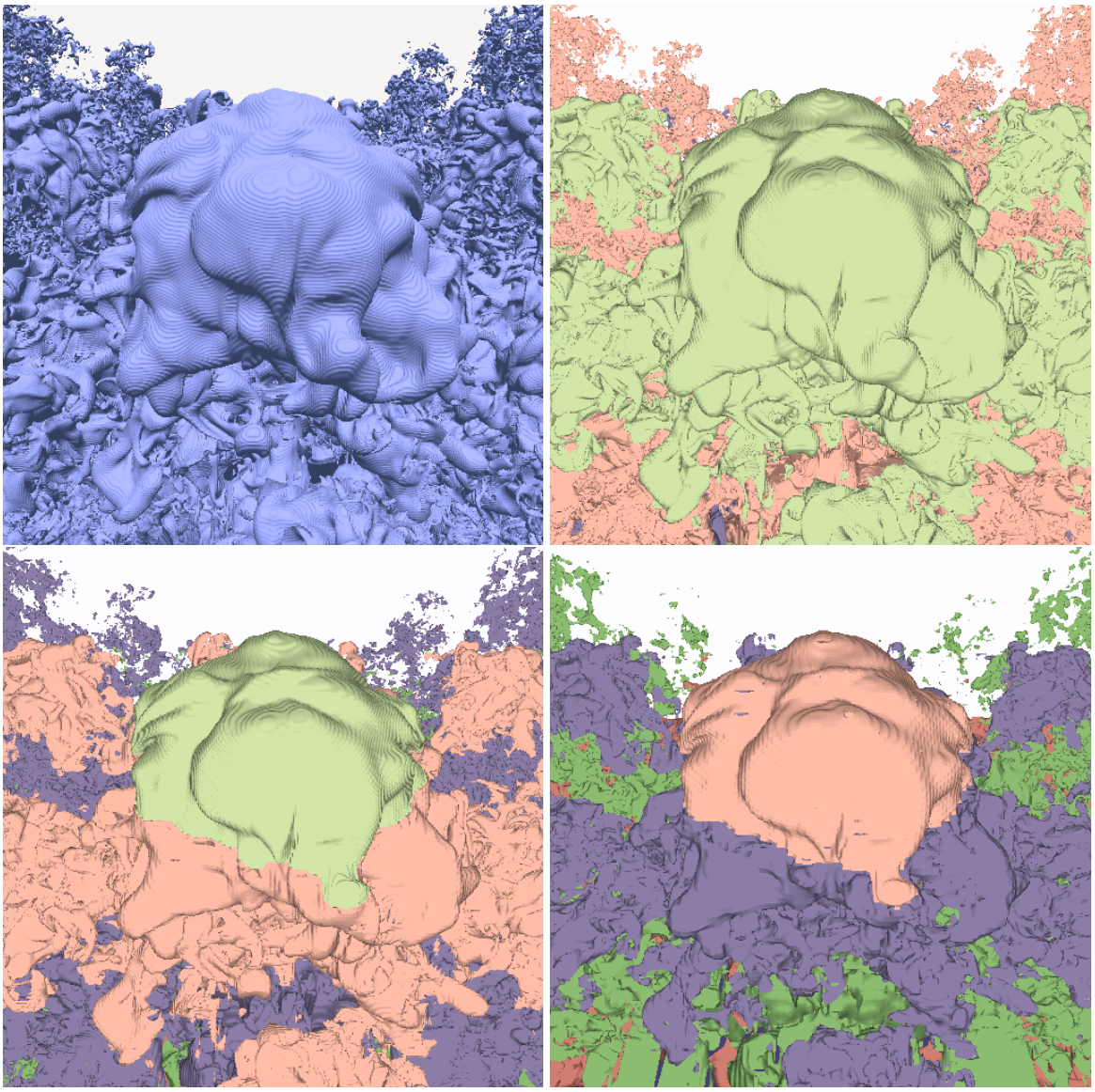


Figure 4.12: Qualitative impact of multiresolution on the Richtmyer-Meshkov data at t=270, isovalue 20. Top left to bottom right: single-ray, then coherent multiresolution with dP/dV of 1,2 and 4. On an Intel Core Duo 2.16 GHz with a 512^2 frame buffer, these render at 0.92, 1.0, 1.9, and 3.6 FPS, respectively. To illustrate LOD transitions, like shades indicate the same resolution.

4.6.5 Overall Performance

In best-case scenarios, our system significantly outperforms the single-ray tracer. With close camera views of the RM data and $dP/dV = 1$, we see order-of-magnitude improvement. The coherent technique usually yields modest improvements even for scenes with generally poor coherence. For sufficiently far camera angles viewing complex data, the single-ray system may actually outperform the coherent method, when using an LOD $dP/dV = 1$. For

these pathological cases, we recommend relaxing dP/dV for exploration, or resorting to single-ray traversal for quality.

Coherent traversal handles a difficult scenario for the single-ray system: a close-up scene deep within the volume, with an isovalue for which the min/max tree is particularly loose. Such is the case in the last example of Table 4.3. While single-ray suffers from data access demand, coherent traversal largely amortizes these costs and performs comparably to other scenes with similar complexity.

Another substantial advantage of coherence is that large frame buffers can be rendered relatively faster. Doubling the frame buffer dimensions generally causes a factor of four

Figure 4.13: The visible female femur. The original, full 617x512x512 volume occupies more space as an octree than uncompressed, due to the entropic nature of soft tissues. Bone, which is more appropriately visualized as an isosurface, can be represented by 100-163 isovalue segments, and compressed into an octree volume with a 20:1 ratio, including the multiresolution data and min/max acceleration structure. For this scene, coherent ray tracing scales well to large frame buffers. The image renders at 6.0 FPS at 512^2, versus 3.1 FPS at 1024^2 on an Intel Core Duo 2.16 GHz, with central differences and shadows.

Table 4.3: Richtmyer-Meshkov reference images for the coherent multiresolution system. Frame rates on an 2-core Intel Core Duo 2.16 GHz laptop (2 GB RAM) and an 8-core dual 3 GHz Intel Xeon (Clovertown) with 4 GB RAM; with our coherent multiresolution method with 8x8 packets and $dP/dV = 1$, and single-ray without multiresolution [61]. Refer to Fig. 4.14 for images.

SCENE	C.Duo,512^2		Xeon,512^2		Xeon,1024^2	
	single	8x8	single	8x8	single	8x8
50, far	2.5	3.5	14.5	19.5	4.5	6.5
150, far	1.9	2.5	11.1	15.0	3.4	4.9
270, far	1.1	1.1	7.2	12.4	2.2	2.2
50, close	2.0	6.9	12.1	39.8	3.6	14.2
150, close	1.7	8.1	12.0	44.2	3.5	14.6
270, close	0.2	4.7	1.4	38.6	0.4	9.2

slowdown in a single-ray tracer; by comparison the packet system frequently experiences a factor of two or better performance decrease, particularly when higher resolution leads to improved intersection-level coherence. For the segmented dataset in Fig. 4.13, coherent ray tracing scales especially well to large frame buffers.

4.6.6 Comparison to Existing Systems

Table 4.3 and Figure 4.14 show performance for the Richtmyer Meshkov dataset with our coherent multiresolution system with $dP/dV = 1$; and the single-ray implementation [61] with no multiresolution scheme. In the best-case scenario we achieve a factor of 23 faster than single-ray performance, and even in worst cases the coherent multiresolution implementation does not exhibit substantially inferior performance. These numbers compare favorably to other implementatations. For similar camera positions, we achieve the same 2 FPS RM data performance on an two-core Intel Core Duo as DeMarle et al. [19] report on a 64-processor cluster with a distributed shared memory layer. We are competitive with Wald et al. [129] for far views, and perhaps faster for close-up scenes, while generally requiring an order of magnitude lesser memory footprint. The performance of our system is also on par with that of Friedrich et al. [25]; however such comparison is not completely fair as that system employs LOD for progressive as opposed to dynamic rendering.

Comparison with state-of-the-art GPU methods is more difficult. Clearly, slice-based direct-volume rasterization on the GPU outperforms our method by well over an order of magnitude for small data (less than 512^3). For larger data, this gap is less pronounced, but

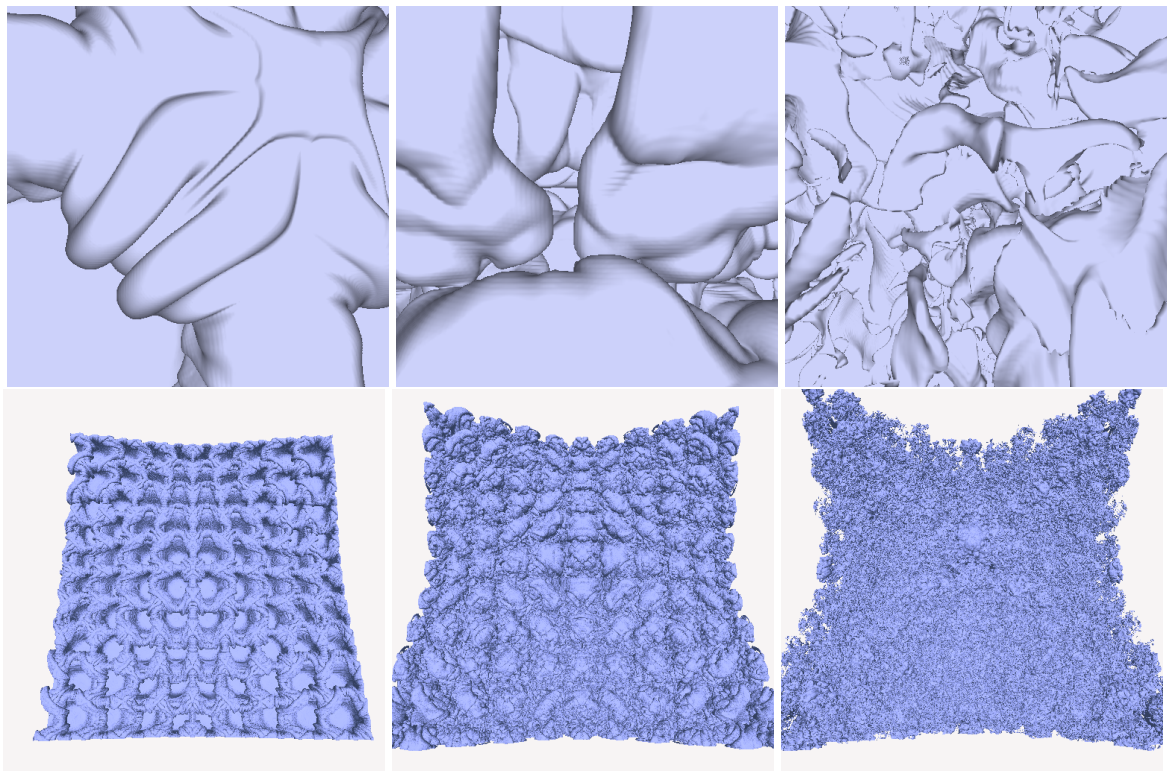


Figure 4.14: Results for the coherent multiresolution system. From left to right, timesteps 50, 150 (isovalue 20), and 270 (isovalue 160). Top: various close-up camera views, illustrating highly coherent scenes. Bottom: far views exhibiting generally poor coherence. We use $dP/dV = 1$.

GPU DVR methods can equally employ multiresolution compression schemes on blocks [37] and space-skipping and culling optimizations [103]. These techniques are still limited by bus latency, and to our knowledge data the size of the Richtmyer-Meshkov has yet to be visualized at original data resolution on a GPU. Out-of-core streaming and progressive rendering, as well as multi-GPU distributed systems, are clearly valid approaches to large-scale volume visualization [13]. Our system is still not as fast as some GPU isosurface raycasting approaches such as [38]. Particularly on a dual-core laptop; the CPU likely lacks the horsepower to compute isosurfaces using higher-order filters such as a tricubic spline kernel. However, it arguably scales better than out-of-core approaches for the GPU. Multicore workstations are increasingly inexpensive commodities, and share a more straightforward and scalable programming model. Ultimately in rendering large data, performance is bound more by memory access than by computation. To that end, multicore CPUs, with hierarchical

caches that directly access expandable mainboard memory, are increasingly attractive. Ray tracing algorithms are well-suited for both this application and platform.

4.6.7 Quality Comparison

We conclude this chapter by considering the impact of multiresolution isosurfacing on image quality. Fig. 4.15 compares results with and without multiresolution at $dP/dV = 1$, with forward and central differences gradients. The bottom images show per-pixel differences (computed using $1 - \text{abs}(reference - image)$ per color channel), comparing multiresolution and non-multiresolution results, both rendered as white surfaces on black background. In reproducing major features, renderings with and without multiresolution look essentially identical. However, the difference images reveal that though most features remain intact, actual isosurfaces are slightly offset when rendered at varying resolution – this accounts for the black pixels (shown in closeup) where the surface exists at one resolution but not another. Otherwise, most differences lie in the intensity of the gradient at different resolutions, as evidenced by grayscale pixels in the difference images.

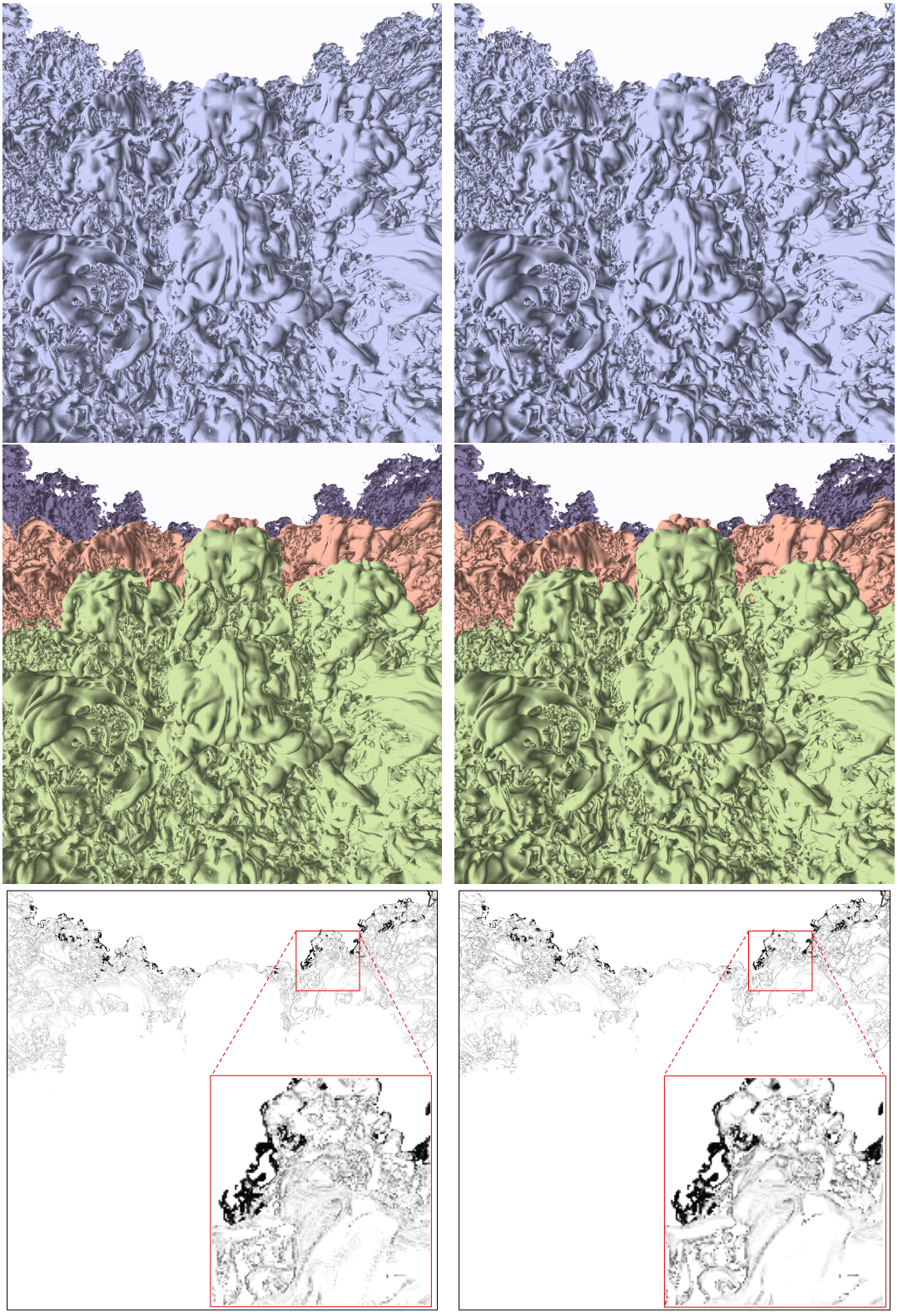


Figure 4.15: Comparing multiresolution and single-resolution results on the RM data. Top row: without multiresolution, with forward differences (left) and central differences (right), rendering at 1.8 and 1.3 fps, respectively. Middle row: multiresolution with $dP/dV = 1$, rendering at 4.2 and 3.1 fps for forward and central differences respectively. Bottom row: differences with and without multiresolution.

CHAPTER 5

COHERENT ISOSURFACE RAY TRACING OF TETRAHEDRAL MESHES

In this section, we discuss a system for interactively rendering isosurfaces of tetrahedral finite-element scalar fields using coherent ray tracing techniques on the CPU. By employing state-of-the art methods for polygonal ray tracing, namely aggressive packet/frustum traversal of a bounding volume hierarchy, we can accomodate large and time-varying unstructured volume data. In conjunction with this efficiency structure, we introduce a technique for intersecting ray packets with tetrahedral primitives. Ray tracing is flexible, allowing for dynamic changes in isovalue and time step, visualization of multiple isosurfaces, shadows, and depth-peeling transparency effects. The resulting system offers a scalable, dynamic and consistently interactive solution for visualizing unstructured volumes. This work was originally published as "Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes" [128].

5.1 Motivation

Large unstructured volumes pose a difficult problem in visualization. Due to its adaptive nature and simplicity, finite element (FE) analysis has experienced widespread adoption in simulations for numerous computational scientific and engineering disciplines such as CFD, meteorology, geology, and astronomy. While the size of most FE data is relatively small compared to large structured data from finite differences simulations, techniques for volume-rendering and isosurfacing unstructured data are less straightforward than those for rendering volumetric datasets in a regular grid.

Unlike marching cubes, which generates a piecewise linear approximation of a typically piecewise cubic isosurface, the marching tetrahedra algorithm of Doi and Koide [23] yields piecewise planar surfaces which represent the correct isosurface for first-order finite elements. However, as is the case with marching acubes, isosurface extraction remains bound

by data complexity and is often slow. Recent works have accelerated marching tet extraction on the GPU. Pascucci [91] showed that the vertex processor can be utilized to create appropriate quadrilaterals for the isosurface within a tetrahedron. Similarly, Klein et al.[53] exploit fragment programs for their quadrilateral computation. These GPU approaches yield overall rendering frame rates from 1 fps for million-tet data to 60 fps for smaller data sets. Object-order direct volume rendering methods for unstructured data [11, 31] can effectively approximate isosurfaces, and in some cases are modified to directly rasterize isosurfaces [137, 3]. However, GPU volume rendering is less efficient than for structured data, which benefits from built-in 3D texture fetching and interpolation hardware. Moreover, most methods exhibit poor scalability for even moderately complex tetrahedral meshes, even when these are sufficient small to reside in-core on a GPU. Currently, for all published rasterization-based GPU techniques, interactivity degrades significantly for larger datasets over 1 million tets. Lastly, point-based methods employing proxies [143] can render efficiently, but require copious preprocessing time and are ill-suited for dynamic data. Moreover, proxy methods resample the data before rendering them, which can result in loss of quality.

CPU-based ray tracing of tetrahedral data has been performed by Marmitt & Slusallek [77] using a new ray marching algorithm for directly traversing tet meshes using Plücker coordinates. In motivating the following work, we were inspired by the application of multicore CPU ray tracing to isosurfacing structured volume data [90, 129, 61]. Performance for such systems, as seen in the previous chapters, is largely limited by the ray-cell intersection for a trilinear interpolant implicit, which is computationally expensive. However, isosurfaces for first-order FE defined on tets are inherently polygonal, allowing for fast ray tracing via simpler geometric intersection tests. While it is difficult to adapt the current rasterization pipeline to tet rendering, ray tracing irregular data is quite efficient, and can interactively render dynamic objects using recent advances in coherent bounding volume hierarchy traversal [127, 65].

The main contributions of this work are extending polygonal bounding volume hierarchy construction to accomodate a tetrahedral mesh scalar field, implementing a fast coherent traversal algorithm for this BVH, and devising a fast packet-isourface intersection test for tetrahedra. The resulting system is surprisingly fast even on modest dual-core CPU hardware, and compares favorably to GPU systems in performance for rendering large unstructured data.

5.2 System Overview

Our core approach to ray tracing unstructured scalar fields is an implicit dynamic bounding volume hierarchy in the spirit of implicit kd-trees [129], combined with aggressive large-packet coherent ray traversal, and a specially designed packet-isopolygon intersection technique inspired by fast packet-triangle intersectors and the Marching Tetrahedra algorithm.

In unstructured grids, the scalar field is defined through linear interpolation over tetrahedral primitives; each such isotetrahedron can then contain one or more more isosurfaces given user-specified iso values. As with implicit kd-trees [129], we build a hierarchical data structure over these primitives such that each node in the hierarchy contains the minimum and maximum of the scalar field below that node's subtree; these isoranges can then be used during traversal to discard subtrees that cannot contain the isovalue. Instead of kd-trees, we opt for bounding volume hierarchies. In practice, they are at least as fast, equally efficient for time-varying data, and better suited to the irregular, overlapping geometry of unstructured volumes.

The implicit bounding volume hierarchy encourages a variation of the aggressive packet-frustum BVH traversal that was recently proposed for polygonal ray tracing [127]. This operates on much larger packets (typically 8x8 or 16x16 rays) than the 4-ray SIMD traversal proposed for implicit kd-trees, and uses frustum culling and speculative descent to minimize the number of ray-node traversal steps. Larger packets also imply better amortization of per-packet costs, and thus help in hiding the overhead induced through implicit culling. Since the implicit BVH is built over the space of all isovalues, the isovalue(s) of interest can be changed interactively any time, and even multiple isovalues can be trivially supported. A BVH also allows for easily updating the data structure once the scalar field or even vertex positions change, and thus allows for naturally supporting time-varying data.

When a packet reaches a leaf of the BVH, we intersect the isotetrahedra contained in that leaf using a new technique inspired both by marching tetrahedra [23] and fast packet-polygon tests. In both intersection and traversal, we will make heavy use of large-packet/frustum techniques recently developed in polygonal ray tracing. Unless otherwise specified, both intersection and traversal are assumed to operate on packets of 16×16 rays.

5.3 Isosurface Intersection

An isosurface is the implicit surface $f(\vec{x}) = v$ where a scalar field $f(\vec{x})$ takes on a given isovalue v. For conventional first-order finite elements, the scalar field is given as a tetrahedral mesh in which the scalar values are specified at the vertices A, B, C, and D; the scalar field inside each *isotetrahedron*, or *isotet*, is defined by linear interpolation

$$f(\vec{x}) = f(\alpha, \beta, \gamma, \delta) = \alpha A + \beta B + \gamma C + \delta D,$$

where $\alpha, \beta, \gamma, \delta$ are the barycentric coordinates of $\vec{x}$.

To intersect a ray $\vec{x}(t) = \vec{o} + t\vec{d}$ with any isosurface $f(\vec{x}) = v$ one can immediately substitute the ray equation into the linear interpolation, solve a linear system for t, and check that the solution lies within the isotet. However, we can also observe that for linear interpolation the isosurface must be planar. This plane is bounded by line segments along the edges of the isotet in which it exists, forming either a triangular or quadrilateral polygon as shown in the various cases of Marching Tetrahedra, and illustrated in Figure 5.1. We denote this polygon an *isopolygon* (or *isopoly*), as it represents the base geometric primitive we seek to ray-trace. Unlike solving the ray-parametrized implicit, this isopolygon must only be computed once per isotet traversed; that cost is amortized over all rays in the packet, and the full array of fast ray-polygon techniques can be applied.

5.3.1 Extracting the Isopolygon

To compute the plane equation and bounding edges of the isopolygon, we turn to the Marching Tetrahedra algorithm [23]. Vertices of the isopolygon lie on edges of the isotet, and isopolygon edges lie on the tet faces. Polygon vertices will lie only on those tet edges for which one vertex is greater and one is smaller than the isovalue. Having 4 vertices, there are only 16 cases for which a given vertex is either larger or smaller than the isovalue. For each of these cases, we can store how many vertices the resulting polygon will have, and the indices of the 2 tet vertices that span the edge on which that polygon vertex must lie. In SSE, this lookup is particularly simple: after loading the four vertices' isovalues into a SIMD register, an SSE comparison followed by a `movemask` operation will return the desired case. The result is conveniently returned in a 4-bit integer (one bit for each comparison) that can be directly used to index into the aforementioned table of 16 cases. Once we know

which tet edges contain isopolygon vertices, each isopoly vertex can be computed by linear interpolation along the two vertices of the corresponding tet edge.

5.3.2 Ray-Isopolygon Intersection

Once the vertices of our polygon are known, we can use an extension of Wald's triangle test [126] to intersect it. As shown in Figure 5.1 (left), ray-isopolygon intersection first computes the distance to the precomputed plane, then projects the ray hit point onto a suitable 2D coordinate plane. Here, each of the edges defines a (2D) half-space, which we orient to point towards the inside of the isopolygon. Since the isopolygon must be convex, we can then take the projected hit point and perform a 2D half-space test with each of the edges, rejecting the hit point as soon as any of these tests fails. This test can be performed efficiently for four rays in SSE for both triangle and quad cases.

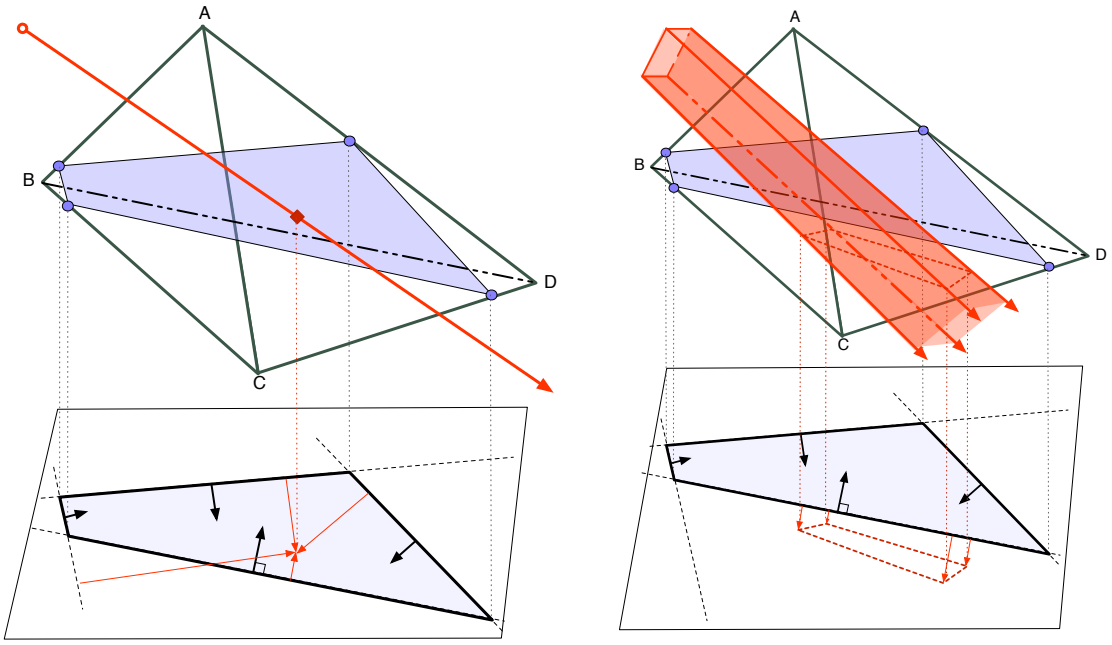


Figure 5.1: Ray-isopolygon intersection in an isotetrahedron. Knowing that the isosurface inside the tetrahedron is a plane, we first extract an isopolygon. We then compute the point where the ray pierces that polygon's supporting plane, and project both the polygon and that hit point to a 2D coordinate plane. In 2D, we then perform a point in (convex) polygon test by considering if the point is on each of the edges' positive half-spaces. The test can trivially be extended to support frustum culling: If all corner rays of the bounding frustum fail at the *same* edge, all rays inside the frustum must fail.

5.3.3 SIMD Frustum Culling

In addition to fast SIMD intersection, we also apply conservative "full miss" and "full hit" tests for the entire packet, using packet frustum culling [22, 7]. These tests require computation of the four corner rays bounding the packet frustum in SSE. For a given isopolygon, we can forgo individual ray intersections when all four bounding rays fail for the *same* 2D half-space test (Figure 5.1, right). Similarly, if all four rays pass all half-space tests, the entire packet passes through the triangle, and we must only perform a distance test for our component rays. Thus, intersection tests for individual rays are only required when the frustum neither fully misses nor fully hits.

The efficiency of frustum culling depends on the relative areas of the frustum and isopolygon within the plane. For complex scenes, tets are too small to have full hits, and frustum culling rarely succeeds. However, full misses are quite common due to the loose nature of the implicit BVH, making this test highly effective overall. Typically, frustum culling can reject 40–60% of the packet-isopolygon tests, though this ratio declines for larger models. Every time SIMD frustum culling rejects a packet test, all individual ray-isopolygon tests are avoided, for example, 256 tests for a 16×16 ray packet.

5.3.4 Isopolygon Precomputation

Isopolygon computation can be executed in three ways:

1. *Full precomputation.* Precompute all isopolys every time the user changes the isovalue(s) of interest.

2. *On-the-fly computation* from scratch on demand.

3. *On-the-fly computation with caching.* Compute isopolys only when needed, but keep a cache of already computed isotets; clear the cache every time the user changes the isovalue(s) or time step.

Full precomputation maximizes performance for navigation with static isovalues, but requires larger memory footprint and incurs delays when the user changes isovalue or time step. On-the-fly computation is slower during rendering, but offers greater flexibility with scene interaction. Caching in theory offers a compromise, but in practice is quite complicated in a multicore environment, as it requires the resolution of cache conflicts in a thread-safe manner, requiring significant synchronization overhead. We therefore opt for pure on-the-fly

computation by default. Due to the use of large packets – which allow for amortizing the on-the-fly computations over all rays in the packet – the overhead is in the range of 5–8%, which we believe is a tolerable price for the ability to arbitrarily change the time step or isovalue.

5.4 The Implicit Bounding Volume Hierarchy

The concept of the implicit BVH is similar to that of the implicit kd-tree [129] in that the acceleration structure is not built for a single isovalue, but rather as a tree of min-max isovalue ranges (as in Wilhelms & Van Gelder [134]). Each node stores the minimum and maximum of all scalar field values contained within that subtree. During traversal, we can consequently cull all BVH nodes that do not contain our desired isovalue. Once built, the implicit BVH structure is valid for all isovalues, and thus allows for simultaneously rendering multiple isosurfaces from the entire range of isovalues. As subtrees that do not contain the isovalue are never traversed, the only effective cost of supporting arbitrary isovalues is a slightly looser-fitting BVH.

5.4.1 Building the BVH

Building an implicit BVH for tets in fact is similar to building a BVH for triangle meshes. Most mesh-BVH builds rely on bounding boxes or centroids of their primitives as construction metrics [127, 125], and tets behave similarly to triangles in this regard.

Traditional bottom-up BVH builds [33] generally result in inefficient BVHs [42]. Recent BVH literature has favored top-down builds, which recursively partition primitives into two subgroups. Two partitioning strategies are of particular interest: the Wald et al. sweep surface area heuristic (SAH) build [127], and the Wächter et al. fast spatial median build as proposed in his bounding interval hierarchy (BIH) paper [125]. The SAH build employs a *surface area heuristic* [33, 42] to select a partition with lowest expected cost, but is costly to build. The BIH-style build is closer in spirit to spatial median builds and, as it requires no cost function evaluation, it builds significantly faster than SAH methods. In both constructions, nodes are partitioned until leaves contain 12 or fewer tet primitives. Empirically, we have found this fixed value to work best.

Our BVH node employs the same structure as [127], with a crucial modification: we interpret the isovalue v as a fourth dimension of the bounding volume, leading to 4D bounds $\{x, y, z, v\}$. This can then be stored and processed as SSE vectors. Integers for the child

node index and traversal bookkeeping follow, padded to ensure SSE-friendly 16-byte alignment. Storing isovalues alongside geometric extents allow all dimensions to be processed simultaneously in SSE.

5.4.2 Implicit BVH Traversal

Having constructed the implicit BVH, we now proceed to traversal. As previously mentioned, we employ the coherent traversal algorithm of Wald et al. [127], and extend it to implicit iso range culling. In general, this algorithm operates on large packets of rays, and tracks both a bounding frustum and the first "active" ray in the packet that intersects a current BVH node. Instead of intersecting each traversed node with *all* the rays in the packet, it employs optimizations such as speculative descent and frustum culling of nodes. Nodes not containing an isovalue in their min-max range are culled.

5.4.2.1 Implicit culling

As shown in Figure 5.2, at the heart of implicit BVH traversal lies the concept of culling subtrees that are known to be *inactive* – those whose isorange does not contain an isovalue.

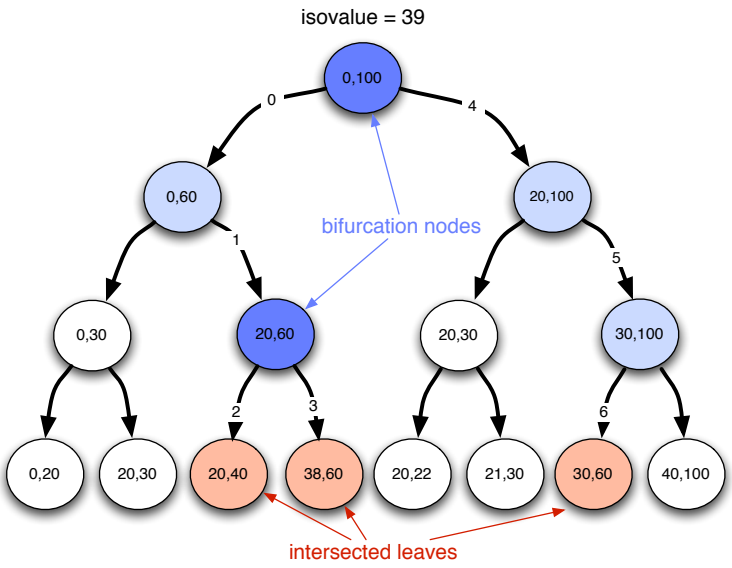


Figure 5.2: Implicit culling in the BVH. The implicit BVH is a min-max tree containing only a subset of BVH nodes containing our desired isovalue(s). We can speculatively descend the min-max tree until we reach a leaf, or an intersection test fails. Only at bifurcation nodes must we resort immediately to geometric packet-BVH traversal computation. Thus, geometric tests are performed as if the BVH had only been built over active nodes for a single isovalue.

As this test is very cheap, we naturally perform it first. In addition, we observe that each active node must have at least one active child, and if the first child is inactive, we can proceed to its active sibling. Only at *bifurcation nodes* - where both children are active - do we actually revert to the geometric tests outlined below. In the worst case, this behavior causes us to descend several times into a subtree that is not actually visible. Since these speculative descents are fast, however, this is still quicker than testing all the nodes for visibility, and even *if* the fast descent led to a subtree that is outside the packet's bounding frustum, this node would be immediately rejected by the frustum test outlined below.

5.4.2.2 Speculative first-active descent

For our first geometric traversal test, we examine the first *active* ray in the packet. If that hits the current node, we can immediately descend without performing any more ray-box tests, as illustrated in Figure 5.3, top. Since we never test whether any of the other rays actually hit the current node, this test is speculative. Though it may cause modest extra work when few rays in the packet are also active, this strategy allows many ray-box tests to be skipped when numerous consecutive rays are active.

5.4.2.3 Frustum test

If the first active test fails, we know that the packet at least partially misses the box, and can perform a frustum test to conservatively determine if the entire packet misses. Technically we employ an interval arithmetic [99, 7] test instead of a geometric frustum test, but the effect is similar in behavior. If the full packet missed, we reject the current node and go to the next node on the stack (Figure 5.3, middle).

5.4.2.4 First-active ray tracking

If both the speculative descent and frustum tests fail, we test all remaining rays until we find the first active one that hits the current node. Those rays that failed the test are marked inactive by tracking the index of the first active ray in the packet (all rays with a smaller index are known to be inactive). If no active ray could be found, we reject the node and pop the next subtree from the stack. Rays with indices higher than the first active one we found are not tested, and are speculatively descended into the subtree also (Figure 5.3, bottom).

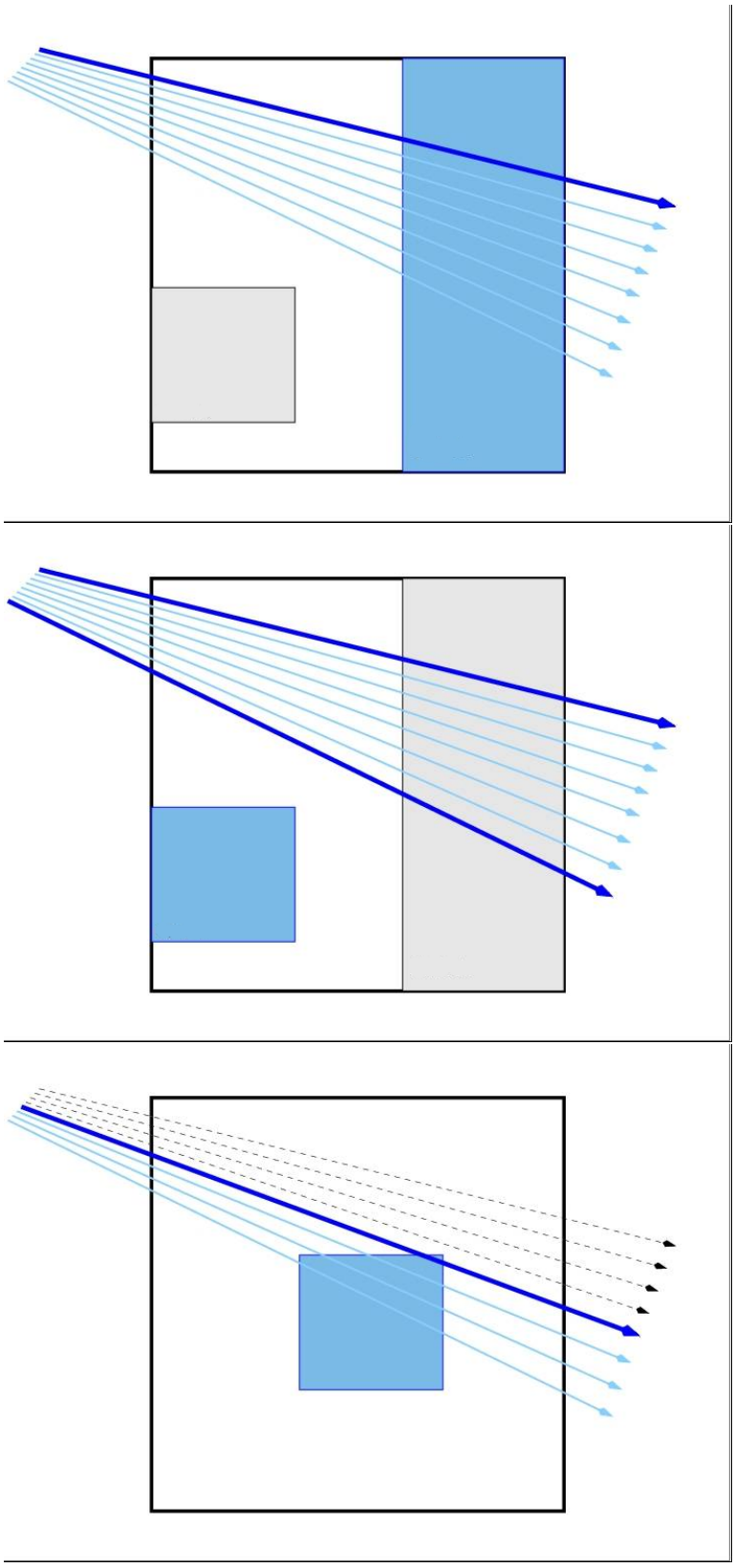


Figure 5.3: First-active descent, frustum test, and active ray tracking.

5.4.2.5 Leaf traversal

When encountering a leaf, we first perform a frustum test as for all other nodes. If that test passes, we iterate over all the tets referenced in that node, then determine that tet's isorange (which may be smaller than the node's isorange). We test that range, and finally either reject the tet or intersect it as described above.

5.5 Time-Varying Data

Time-varying data are extremely common in FE simulations. In the simplest time-varying tet meshes, geometry remains constant and only scalar values change. More complex scenarios include changing geometry and topology, and potentially dynamic addition and removal of elements from one time step to the next. To address these possibilities, we implement two schema for BVH construction, balancing performance and memory footprint. Results are analyzed in Sec. 5.7.6.

5.5.1 Schema I: Unique BVH Per Step

The naïve way of accommodating time-varying data is to compute a unique BVH for each time step. No render-time computation is necessary to progress from one time step to the next, regardless of changes in geometry or scalar element values. As we operate completely in host memory, this approach is in fact very efficient. However, for large data sets with many time steps such as the fusion data set, this approach may entail a considerable memory footprint.

5.5.2 Schema II: Dynamic Refitting

Fully computing a new BVH on-the-fly during rendering is too costly for large data, even using the fast BIH-style build. However, we observe that when tet mesh vertices change position but connectivity remains constant, the BVH structure will not change between time steps. Thus, simply refitting the nodes' bounding extents will yield a correct BVH. This technique has been successfully applied to ray tracing dynamic triangle meshes [127, 65]. The main drawback is that, particularly in cases of extreme geometric deformation, the refit BVH may perform worse than a BVH built from scratch for that particular time step. Fortunately, for tet meshes and our BVH, this method works extremely well due to the continuous nature of tet deformations in FE simulation, particularly for rigid bodies. Moreover, when

vertices remain constant but the scalar field changes, the BVH is identical for all time steps, as only the min-max isovalues must be updated.

As previously mentioned, minimum and maximum geometric bounds and isovalues are stored adjacently in 4D SSE vectors. Refitting the 4D extents can thus be accomplished with one SSE min and one SSE max per BVH node. Tet vertices and scalars are also stored as 4D points; thus computing the 4D bounds of a tet is also extremely efficient, requiring only 3 SSE min and max operations each per tet. It is straightforward to parallelize the update process. After the initial BVH has been built we find all the subtrees for a given level in the BVH hierarchy, and store their indices. During a refit, we can then update these subtrees in parallel. Once all subtrees are updated, a single thread refits the remaining few nodes close to the root node.

5.6 Shading and Interaction Modalities

Having leveraged these algorithms for efficient unstructured volume ray tracing, we describe several visualization modalities that can assist in understanding our data sets.

5.6.1 Shadows

Shadows add important visual cues in understanding shape (see Figure 5.4). In casting shadow packets, rays are generally coherent and share a common origin in the case of point lights. Unlike primary rays, shadow rays do not inherently form a regular beam, and thus have no concept of "corner rays" for SIMD frustum culling. Fortunately, shadow packets may still employ the Reshetov et al. [99] frustum-culling technique at traversal, as this requires no actual geometric frustum. The overall speed impact of shadow rays varies, but is typically lower than $2\times$.

5.6.2 Multiple Isosurfaces

Supporting multiple isosurfaces in an implicit BVH is straightforward, by simply testing whether a BVH subtree overlaps *any* of the isovalues before descending it. To follow the SIMD paradigm, we currently support up to four different isosurfaces, though it would be trivial to add more. Keeping the four isovalues in a SIMD vector, we can test when a BVH node's or isotetrahedron's iso range contains any of these four isovalues in parallel. These are in turn intersected with all the rays that actually hit the leaf node. Though rendering

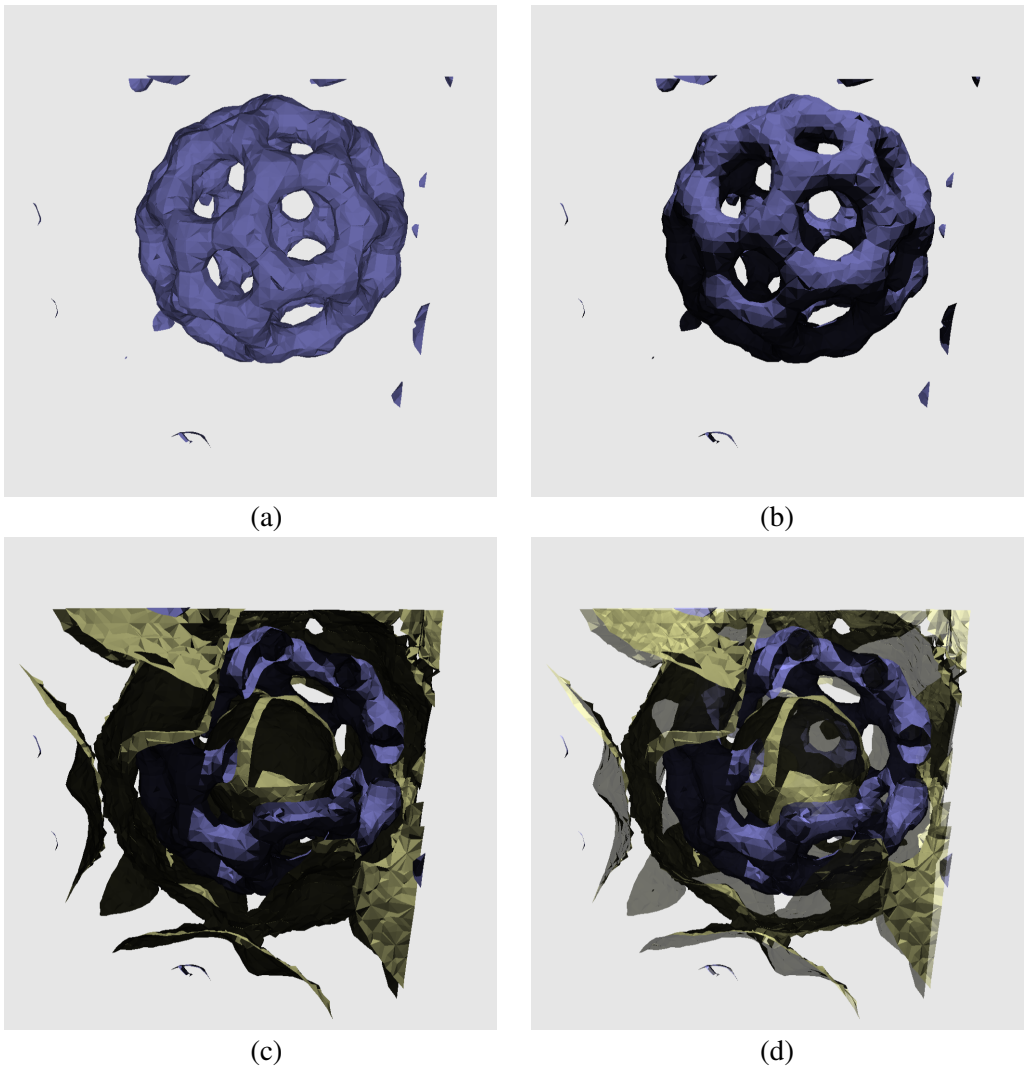


Figure 5.4: Additional shading effects. a) A bucky ball rendered with a single isosurface, and diffuse shading. b) After turning on diffuse shading with shadows. c) With a second isosurface and an interactive clip-box to expose the interior. d) Adding transparency as well. At 1024×1024 pixels on a Intel Core 1 duo laptop, these screenshots render at 15.6, 10.2, 5.4, and 2.6 frames per second, respectively. On our 16-core Opteron 3.0 GHz workstation, they render at 90, 70, 42, and 19 frames per second, respectively.

multiple surfaces can require tracing more rays per image, particularly when transparency is enabled, it causes no significant computation penalty in and of itself.

5.6.3 Clipping Planes and Boxes

While isosurfaces provide an intuitive way of visualizing a data set, one of their drawbacks is that the surface often occludes the data set's interior. For that reason, visualization systems often employ clipping planes (or boxes) that allow for cropping certain parts of the

model to expose its interior. We currently allow for a single box that may or may not extend to infinity (to simulate a plane), and use this to clip BVH subtrees. During traversal, if a node's subtree is completely enclosed in the crop box, we skip the subtree just as if it was out of the isorange. In SIMD, a box-in-box test is very cheap and can be amortized per packet, incurring negligible cost.

5.6.4 Transparent Depth Peeling

Rendering transparent isosurfaces also provides better understanding of the dataset. Though straightforward to implement, transparency multiplies the complexity of rendering an image by the number of transparent hits required. Though it is possible to implement by recording multiple hits per ray, in our packet architecture it is more elegant to implement as a shader via secondary rays. By simply specifying a minimum hit distance for each transparency ray, we can reuse the origin, corner rays and frustum of the original ray packet. Rays that do not require a transparency ray are disabled, sometimes leading to partially-filled packets, but incurring no additional traversal steps or isopolygon intersections. As shading is performed front-to-back, shadows and transparency are always computed accurately.

5.7 Results

In this section, we evaluate the system as a whole, and the overall success of coherent BVH ray tracing for tet-volume isosurfaces. For our benchmarks, we consider three representative machines: a laptop equipped with an Intel Core (1) Duo 2.33 GHz and 1 GB RAM; a 4-core dual Intel Xeon 2.33 GHz desktop with 4 GB RAM; and a 8-CPU dual-core (16 cores total) Opteron 3.0 GHz workstation with 64 GB RAM. Unless otherwise stated, all examples run at 1024×1024 pixels, and use packets of 16×16 rays. The data sets and scenes we used for our comparisons are depicted in Figures 5.5 and 5.6, and overall performance figures are given in Table 5.1.

5.7.1 Build Time and Performance

Because a tetrahedral mesh has far less geometric variation than a polygonal model (i.e., tets form a partition of space, and never overlap or self-intersect), the qualitative difference between a SAH and a BIH build is virtually nonexistent (Table 5.1). Because of the lower build times, we default to the BIH-style build. With the fast BIH-style build, most of the smaller data sets could in fact be rebuilt from scratch per frame.

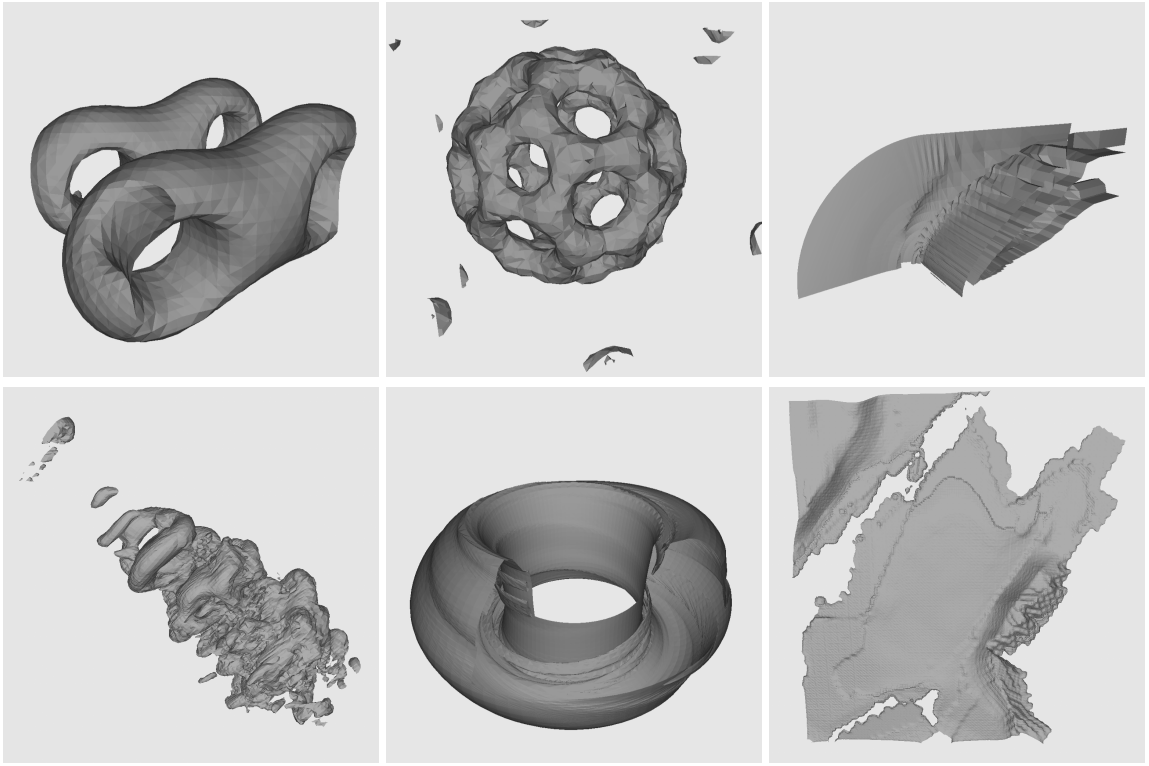


Figure 5.5: Benchmark scenes for the BVH. From left to right, top to bottom: ell32P (149k tets), bucky ball (177k tets), bluntfin (225k tets, two isosurfaces), tjet (1m tets), timestep 50 of the fusion data (3m tets), and the sf1 seismic data (14m tets). With simple shading, these examples run at 14.2, 13.3, 18.9, 10.1, 4.0 and 3.3 frames per second (1024×1024 pixels) on an Intel Core 1 Duo 2.33 GHz laptop with 1GB RAM, and at 116, 112, 95, 66, 57, and 32 frames per second on a 16-core 3.0 GHz Opteron workstation.

Table 5.1: BIH-style build vs. SAH for building the implicit BVH. Because the tetrahedra are distributed over space more evenly than triangles in a polygonal model, the render performance for between BIH-style build and SAH build is very similar, but executing the BIH-style build is much faster.

	ell32p	bucky	blunt	tjet	fusion (t=50)	sf1
#tets	148,995	176,856	224,874	1.0m	3m x 116	14m
render performance (frames per second)						
BIH	48.0	39.4	53.8	28.5	11.8	13.1
SAH	43.7	39.5	57.1	27.7	12.3	13.1
build time (ms, dual Intel Xeon 2.33 GHz)						
BIH	32	40	61	607	1402	4908
SAH	1647	1794	2710	20886	70119	311267

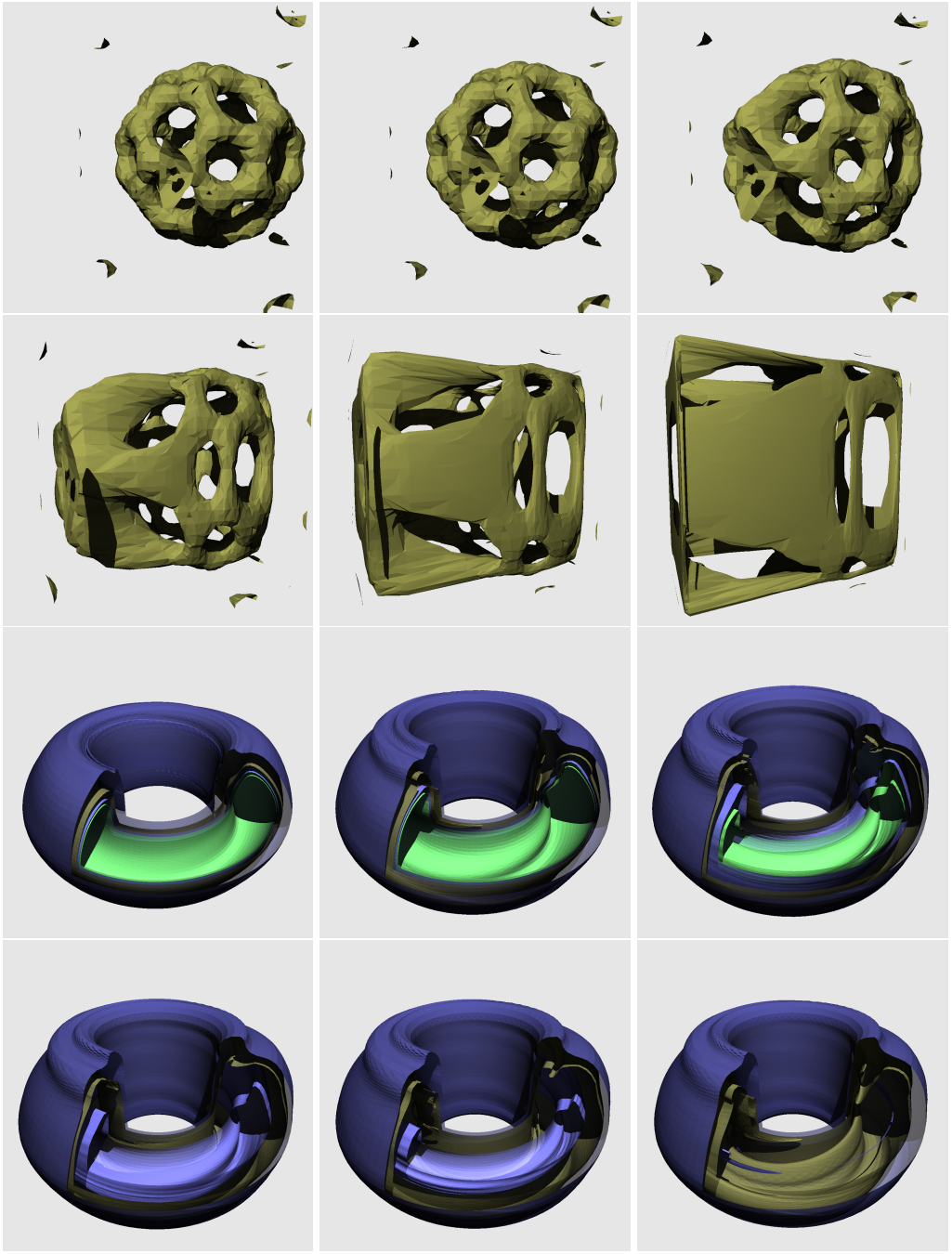


Figure 5.6: Two examples of time-varying tet data sets, rendered at 1024×1024 pixels, using a 16-core 3.0 GHz Opteron workstation. Top: An artificially created deforming bucky ball that shows severe deformation of its 226K tets, running at 50+ frames per second including shadows from a point light source. Bottom: The fusion data set with a time-varying scalar field (3m tets, 116 time steps), rendered with four layers of isosurfaces, a crop box, shadows, and transparency, running at 7 to 15 frames per second. Camera and light positions, time step, and number and parameters of the isosurfaces can be changed interactively.

5.7.2 Rendering Performance

As seen in Tables 5.1 and 5.2 and Figure 5.5, all of the static examples can be rendered at multiple frames per second even on the dual-core laptop. For static scenes, performance is typically linear in the number of CPU cores. Empirically, we found our application scales roughly linearly with respect to the number of pixels per frame. Thus, a frame buffer of 512×512 generally renders four times faster than at 1024×1024, enabling interactive rates for difficult scenes on the laptop.

5.7.3 Scalability in Model Size

Performance degrades gracefully when increasing model size, dropping only by 4x from from the smallest model (feok, 121k tets) to the most complex one (sf1, 14M tets). This is largely due to the logarithmic complexity of ray tracing efficiency structures, and the packet-amortized cost of memory access. To further evaluate scalability to large models, we have synthetically replicated a bucky ball $n \times n \times n$ times *without instancing*. As evident in Table 5.3, performance drops moderately even for hugely complex models of up to nearly a billion tets. Though they require workstation-class memory capacity, large unstructured data such as the STP bullet simulation (36m tets) render equally efficiently (Figure 5.7).

5.7.4 Traversal Efficiency

The key to this interactive performance lies in the aggressive large-packet traversal scheme. Speculative descent and frustum culling greatly reduce the number of individual ray-box tests during traversal by roughly a factor of 18–51 compared to tracing 2×2 packets (the smallest an SSE-based system can trace). Using packets allows for traversal and intersection code in SSE, which is crucial to realizing the performance potential of modern CPUs. Because we have transformed the ray-isotet intersection to a polygonal problem, the same frustum culling

Table 5.2: Performance in frames per second for various data sets and platforms. *Laptop* is an Intel Core Duo 2.33 GHz, 1 GB RAM. *Desktop* is a 4-core dual Intel Xeon 2.33 GHz, 4 GB RAM. *Workstation* is a 16-core cc-NUMA 3.0 GHz Opteron, with 64 GB RAM. Refer to Figure 5.5 for images.

	ell32p	bucky	blunt	tjet	fusion (t=50)	sf1
render performance (frames per second)						
laptop	14.2	13.3	18.9	10.1	4.0	3.3
desktop	48.0	39.4	53.8	28.5	11.8	13.1
workstation	116	112	95	66	57	32

Table 5.3: Bucky ball replication and scalability. Performance in frames per second on four Opteron 3.0GHz cores, for varying numbers of replication of the bucky ball scene (no instancing is used).

# replications	1	2^3	4^3	8^3	16^3
# tets total	177k	1.4m	11.3m	90.4m	724m
frames per second	43	16.7	6.2	2.0	0.80

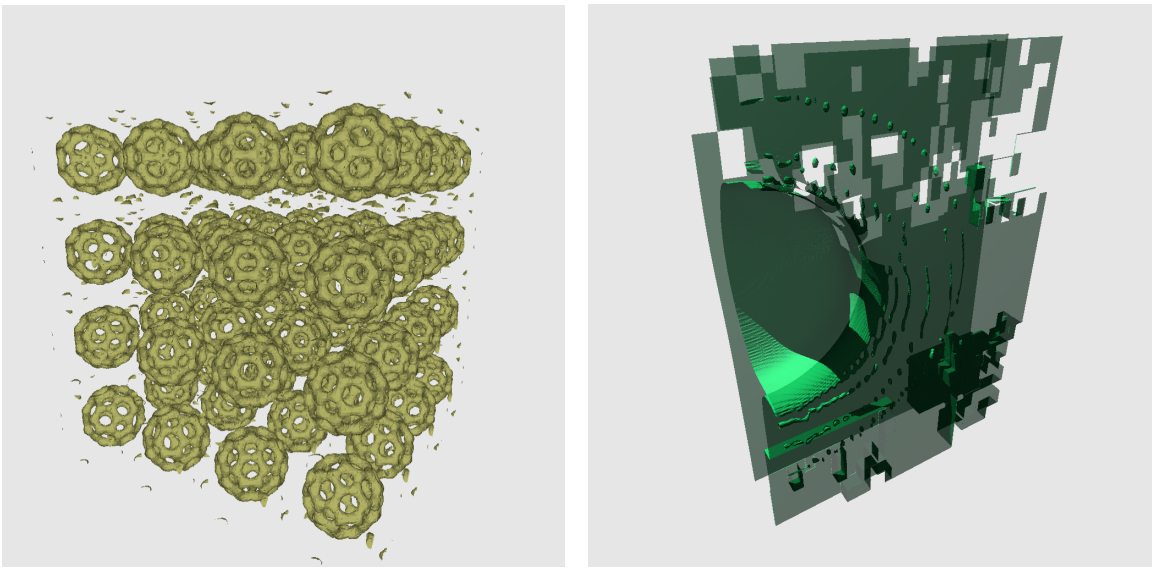


Figure 5.7: Large data and scalability. Left: 4^3 replicated buckyballs with 11.3m tets. Right: STP dataset with 36m tets. With simple shading, these datasets perform at 27.8 and and 26.9 fps, respectively, on a 16-core 3.0 GHz Opteron workstation with 64 GB RAM.

techniques can also be used to significantly reduce the number of individual ray-isopolygon tests, by about 2–3×, though for the most complex scene the number of ray-isopolygon tests actually increases (see Table 5.4). Finally, larger packets allow for amortizing per-packet operations like isorange culling and isotet extraction over the entire packet, thus reducing the total number of these operations per frame. As evident in Table 5.4, this reduces the number of isopolygon generations by about 6–40×, and the number of culling tests by 22–55×.

Regarding isopolygon caching versus on-the-fly recomputation, we note that because large packets reduce the number of isopolygon extractions, caching the isopolygons has a relatively low impact. Even when using only a single CPU and a large enough cache (so no conflicts occur, and all synchronization can be disabled), caching only increases total frame rate by 5–8% over on-the-fly recomputation. Thus, we opt for the on-the-fly recomputation by default.

Table 5.4: Traversal statistics of the frustum method (using 16×16 rays) vs. standard 2×2 packet traversal.

# scene	ell32P	bucky	bluntfin	tjet	fusion (t=50)	sf1
number of individual packet-box tests						
2x2	56.75	93.84	48.05	44.67	175.83	33.21
16x16	1.11	1.8	0.94	1.20	4.32	1.69
ratio	52×	52×	51×	37×	41×	20×
number of individual ray-isopolygon tests						
2x2	8.0	13.52	8.90	6.8	29.35	9.37
16x16	3.39	4.42	3.19	3.95	16.47	7.64
ratio	2.4×	3.0×	2.4×	1.7×	1.8×	1.22×
number of total packet isorange tests						
2x2	99.89	152.31	76.75	135.32	279.75	77.10
16x16	1.88	2.84	1.45	3.00	6.48	2.72
ratio	53×	54×	51×	45×	43×	28×
number of total isopolygon extractions (×1000)						
2x2	1908	354	2216	1154	7285	1943
16x16	64	10	69	110	296	373487
ratio	29×	34×	32×	10.3×	25×	5.2×

5.7.5 Multiple Isosurfaces, Shadows, and Transparency

Rendering multiple isosurfaces in itself does not significantly raise the cost of an image, due to the ray tracer's implicit occlusion culling – the $2\times$ drop in framerate in Figure 5.4 is due to the $2\times$ higher projected area of the model after adding the outer isosurface. However, as mentioned in Section 6.6.6, advanced shading bears a significant cost due to the higher number of rays traced. Shadows usually increase the render cost by about 2x if the rendered object covers the entire screen, and somewhat less, otherwise (also see Figure 5.4). Transparency similarly increases to the total number of rays traced per-frame, and thus increases the render cost. We typically limit the number of transparency rays to a user-specified maximum (2 by default), which can be changed interactively. All these effects can be supported simultaneously, even for large time-varying data sets (see Figures 5.6 and 5.4).

5.7.6 Time-Varying Data Sets

Precomputing a BVH and replicating vertex arrays for each timestep, as in Sec. 5.5.1, is only practical for small data or workstations with copious memory. For the fusion dataset this requires over 22 GB in memory footprint. Nevertheless, this scheme remains desirable, as moving across timesteps incurs no noticeable penalty in frame rate. Conversely, by employing a single BVH and refitting it per-frame (Sec. 5.5.2), the BVH and all 116 time steps of the fusion data occupy only 538 MB, allowing us to render that model on

the laptop. However, refitting requires updating the vertex array, all the BVH nodes, and some precomputed shading data (e.g., per-tet gradients) per frame. This update is fully parallelized, but scales poorly due to intensive and asymmetrical memory access on our workstation's cc-NUMA architecture. Effectively, refitting adds a significant per-frame cost that limits maximum performance to 3.5 fps on the workstation. Moreover, precomputation and refitting offer a classical trade-off between performance and memory consumption.

5.7.7 Memory Overhead

The bounding volume hierarchy structure occupies a significant footprint in main memory. In our implementation, the BVH requires two arrays: one for BVH nodes, at 32 bytes per node, and another for storing the lists of tet IDs that the leaf nodes refer to. The tetID list uses a constant amount of memory, requiring exactly 4 bytes per tet. The size of the node array depends on how many nodes are allocated, which in turn depends on the data and build strategy. In the worst case, a BVH would always split until each tet is contained in exactly one leaf, in which case a total of $2N-1$ nodes, (i.e., roughly $64 \times N$ bytes) would be allocated for the node array. In practice, the optimal BVH is much shallower, and uses only a fraction of that memory ($\frac{1}{4}^{th} - -\frac{1}{6}^{th}$).

For that worst-case assumption, however, Table 5.5 shows that for static scenes, memory overhead is around $4\times$ that of the raw input data. For the time-varying deformed bucky and fusion data sets, this overhead increases to a significant $18\times$ and $20\times$ if a separate BVH is stored per time step. If the BVH is shared over time, the overhead drops to 92% for the

Table 5.5: Memory usage and BVH overhead. Note that we report a worst-case upper bound on BVH memory ($2 \times N - 1$ nodes for N tets), as this is what our system actually pre-allocates memory for. In practice, only about one fourth to one sixth of that pre-allocated memory is actually used (i.e., memory overhead could be reduced rather easily should that ever become an issue).

scene	number of			raw	BVH per step		shared BVH	
	tets	verts	steps	mem	mem	ratio	mem	ratio
ell32p	149k	33k	1	2.8MB	9.6MB	4.1×	–	–
bluntfin	225k	41k	1	4.1MB	18MB	4.2×	–	–
SF1	13.9m	2.5m	1	251MB	906MB	3.6×	–	–
TJet	1m	163k	1	17.7MB	64.9MB	3.6×	–	–
bucky $\times 4^3$	11.3m	2.1m	1	205MB	734MB	4.2×	–	–
STP	36m	6.3m	1	1.7GB	7.2GB	4.2×	–	–
bucky. def.	176k	32k	20	12.7MB	234MB	18×	11.7MB	0.92×
fusion	3.0m	622k	116	1.1GB	22GB	20×	194MB	0.18×

deformed bucky while for the fusion data set the overhead is only 18%. In general, more time steps reduce the relative overhead, as they amortize input tet data footprint.

5.7.8 Comparison to Existing CPU Based Approaches

Our results compare favorably to the performance achieved by the Marmitt et al. Plücker-based tet marching algorithm [77], which reported 1.67 and 0.92 fps at 512×512 on a dual-Opteron for isosurfaces on the bluntfin and buckyball, respectively. On comparable hardware and frame buffer size, our system performs around 40 times faster. However, it is important to note that the Marmitt et al. method also supports semitransparent volume raycasting, which ours does not.

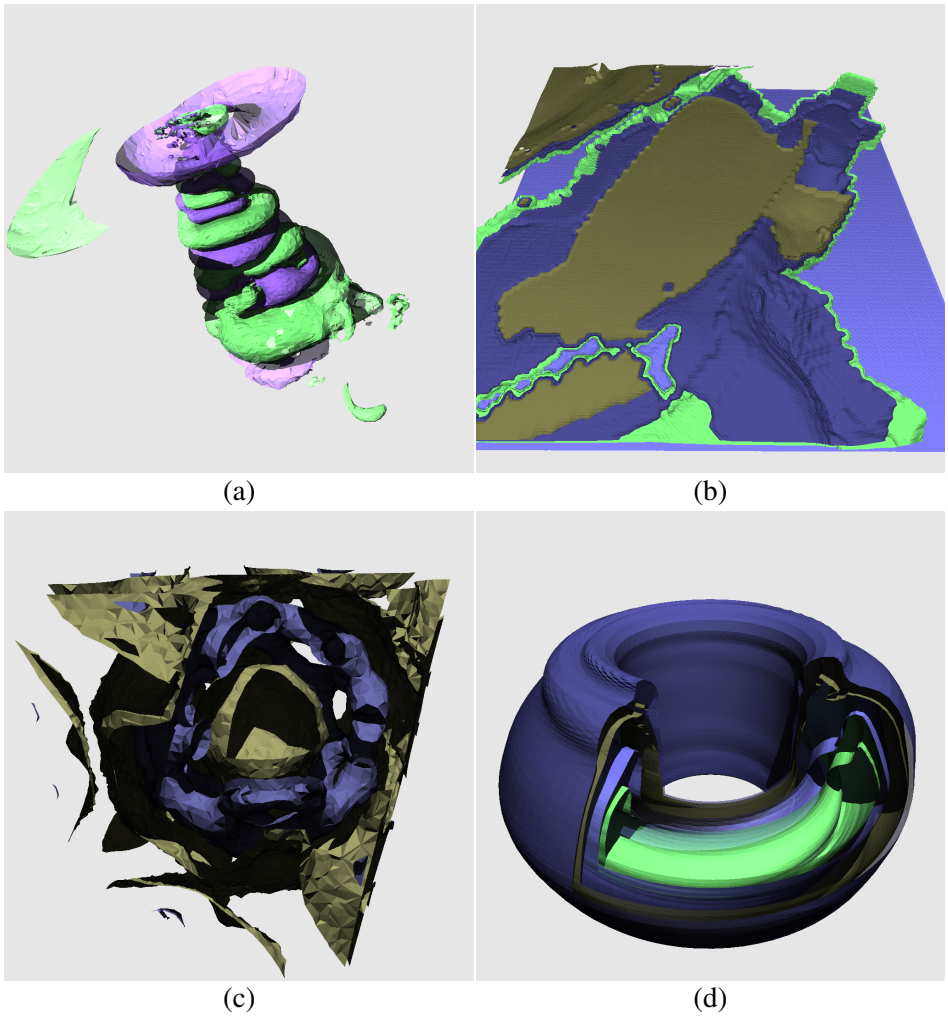
5.7.9 Comparison to Existing GPU Based Approaches

GPU hardware is continually changing, so comparing to previously published results would be an unfair comparison to already-outdated hardware. For that reason, we have decided to base our comparisons mainly on HAVS [11] and its isosurface extension [3], running on a state-of-the-art nVidia 8800 GTX. HAVS is well-known and freely available, thus an appropriate system for benchmarking GPU performance. As seen in Table 5.6, when isosurfacing small and moderate-sized datasets (less than 1M), ray tracing achieves roughly equivalent performance on a 4-core Xeon as rasterization on the nVidia 8800 GTX in the same desktop. For larger data sets, however, our method can outperform HAVS significantly, even for models that fit comfortably in GPU memory.

For small data such as the bluntfin, isosurfacing via the GPU ray-casting method of Georgii & Westermann [31] reports 175 fps at 512×512 on an nVidia 7900 GTX; our system achieves 160 fps on the 4-core Xeon desktop at the same resolution. However, their performance degrades significantly for larger datasets over 1M tets. We refrain from absolute comparison, but our system achieves similar performance for small data, and is substantially faster for large data. Again, it should be noted these GPU methods are designed for object-order volume rendering without acceleration structures, whereas our technique relies on logarithmic-order BVH traversal and is restricted to isosurface visualization. Nonetheless, CPU ray tracing is roughly competitive in performance with GPU methods for isosurface visualization of unstructured grids, and exhibits better overall scalability. In closing this chapter, Figure 5.8 provides an overview of effects supported by our system.

Table 5.6: GPU performance comparison, in frames per second, with HAVS [11, 3], running on an nVidia 8800 GTX, and our method on a 4-core Intel Xeon 2.33 GHz, at 1024×1024 resolution.

scene	ell32P	bucky	bluntfin	tjet	fusion	SF1
# Tetrahedra	149k	177k	225k	1m	3m	14m
BVH	48	39.4	53.8	28.5	11.8	13.1
HAVS	50	50	30	3.0	1.5	0.3



(a) (b) (c) (d)

Figure 5.8: Teaser scenes from the tet mesh isosurface ray tracer. a) tjet (1m tets) with shadows, transparent depth-peeling, and multiple isosurfaces b) SF1 (14m tets) with four isosurfaces. c) buckyball with two a clip-box, multiple isosurfaces and shadows, d) Time step 60 of the time-varying fusion data set (3m tets, 116 time steps), rendered with four isosurfaces, shadows, and transparency. With a 1024×1024 frame buffer, these examples render at 2.0, 3.1 5.4, and 0.8 fps, respectively, on an Intel Core Duo 2.33 GHz laptop with 1 GB RAM; and and 11, 18, 52, and 10 fps, on a 16-core 3.0 GHz Opteron workstation with 64 GB RAM.

CHAPTER 6

RAY TRACING ARBITRARY-FORM IMPLICIT SURFACES

In this section, we consider a more general problem in rendering a wide class of implicit surfaces. Existing techniques for rendering arbitrary-form implicit surfaces are limited, either in performance, correctness or flexibility. Ray tracing algorithms employing interval arithmetic (IA) or affine arithmetic (AA) for root-finding are robust and general in the class of surfaces they support, but traditionally slow. Nonetheless, implemented efficiently using a stack-driven iterative algorithm and SIMD vector instructions, these methods can achieve interactive performance for common algebraic surfaces on the CPU. A similar algorithm can also be implemented stacklessly, allowing for efficient ray tracing on the GPU. This paper presents these algorithms, as well as an inclusion-preserving reduced affine arithmetic (RAA) for faster ray-surface intersection. Shader metaprogramming allows for immediate and automatic generation of symbolic expressions and their interval or affine extensions. Moreover, we are able to render even complex forms robustly, in real-time at high resolution. The results of this chapter were published as "Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic" [59] for the CPU IA algorithm, and subsequently "Fast and Robust Ray Tracing of General Implicits" [58] for the GPU IA/AA method.

6.1 Motivation

Ray tracing methods for implicit surfaces have historically sacrificed either speed, correctness or flexibility. Piecewise algebraic implicits have been rendered in real-time on the GPU using Bézier decompositions [72], but approximating methods do not render arbitrary expressions directly, nor always robustly. Self-validated arithmetic methods, such as interval arithmetic (IA) or affine arithmetic (AA), are extremely general in that theoretically any composition of Lipschitz-boundable functions can be expressed as an inclusion extension and solved robustly. However, these approaches have historically been among the slowest.

This work discusses optimization of interval and affine arithmetic methods to allow for interactive ray tracing of arbitrary-form implicit surfaces on the CPU and GPU. We first create an optimized coherent intersection algorithm using SSE vector instructions, achieving interactive ray tracing for most simple surfaces on a dual-core CPU. We then derive a separate algorithm for the GPU, using a stackless interval bisection algorithm for general implicit intersection. We also implement an efficient implementation of a reduced affine arithmetic (RAA) that correctly preserves the inclusion property. Together, these allow real-time rendering of complex implicit functions. Shader metaprogramming allows users to design implicit forms and procedural hypertextures flexibly, with immediate results and full support for dynamic 4D surfaces. The ray tracing algorithm enables multibounce effects to be computed interactively without image-space approximations, enabling effects such as transparency and shadows. While the direct application is simply the computer graphics technique and a mathematical graphing tool, the ability to accurately and quickly render general-form implicits could ultimately facilitate isosurface rendering of geometric data with arbitrary user-specified filters.

Chapter 2 provides an overview of related work in rendering general implicit surfaces. Section 2.5.1 discusses ray tracing approaches for rendering algebraic and implicit surfaces. Approaches for specific implicits, such as [5] and [38], frequently combine point sampling and a superlinearly-convergent numerical technique, such as bracketed Newton and *regula falsi*, to converge to the intersection point of surfaces that contain multiple roots along a ray. These methods are fast and surprisingly sophisticated, but can fail near singularities when brackets are wide enough to contain multiple roots. Better numerical iterative techniques such as the Brent and Dekker methods exist [97], but require some tuning to the individual implicit. These secant-method derivatives are also employed in special-case implicit intersectors [75]. Analytical closed-form root-solving can also be effective [111], but frequently requires double precision for accuracy.

In ray-casting implicit surfaces, the problem lies in selecting appropriate brackets before employing a numerical root-finding method. Strategies for arbitrary-form implicits differ when bracketing assumptions cannot easily be made. Hanrahan [39] proposed a brute-force uniform sampling of points, using Descartes' rule of signs to isolate roots. Sturm sequences [122], bracketing based on Lipschitz conditions [52], and derivation and evaluation of signed distance functions [40] have been proposed; however these methods either

sacrifice robustness, generality, or the ability to directly render an implicit surface given its closed form. A common approach to rendering general implicit surfaces is to refactor them into a polynomial that can be easily solved. With low-degree polynomials this can be extremely efficient [72, 87] and sacrifice some robustness; or it can be more accurate but expensive [98].

The appeal of interval and affine arithmetic is that they allow for direct evaluation of the implicit function itself, and, assuming numerical stability, a robust method of determining whether a root is possibly contained within an interval bracket. Toth [121] first applied interval arithmetic to ray tracing parametric surfaces, in determining an initial convex bound before solving a nonlinear system. Mitchell [82] ray traced implicits using recursive IA bisection to isolate monotonic ray intervals, in conjunction with standard bisection as a root refinement method. Heidrich and Seidel [44] employed affine arithmetic in rendering parametric displacement surfaces. De Cusatis Junior et al. [16] used standard affine arithmetic in conjunction with recursive bisection. Sanjuan-Estrada et al. [107] compared performance of two hybrid interval methods with Interval Newton and Sturm solvers. Florez et al. [24] proposed a ray tracer that antialiases surfaces by adaptive sampling during interval subdivision. Gamito and Maddock [28] proposed reduced affine arithmetic for ray casting specific implicit displacement surfaces formulated with blended noise functions, but their AA implementation fails to preserve inclusion in the general case.

6.2 Background

6.2.1 Ray Tracing Implicit Surfaces

Recall that a surface S in implicit form in 3D is the set of solutions of an equation

$$f(x,y,z) = 0 \tag{6.1}$$

where $f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$. In ray tracing, we seek the intersection of a ray

$$\vec{p}(t) = \vec{o} + t\vec{d} \tag{6.2}$$

with this surface S. By simple substitution of these position coordinates, we derive a unidimensional expression

$$f_t(t) = f(o_x + td_x, o_y + td_y, o_z + td_z) \tag{6.3}$$

and solve where $f_t(t) = 0$ for the smallest $t > 0$.

In ray tracing, all geometric primitives are at some level defined implicitly, and the problem is essentially one of solving for roots. Simple implicits such as a plane or a sphere have closed-form solutions that can be solved trivially. More complicated surfaces without a closed-form solution require iterative numerical methods. However, easy methods such as Newton-Raphson, and even "globally-convergent" methods such as *regula falsi*, only work on ray intervals where f is monotonic. As shown in Fig. 6.1, point sampling using the rule of signs [39] fails as a robust rejection test on nonmonotonic intervals. While many methods exist for isolating monotonic regions or approximating the solution, inclusion methods using interval or affine arithmetic are among the most robust and general. Historically, they have also been among the slowest, due to inefficient implementation and impractical numerical assumptions.

6.2.2 Interval Arithmetic and Inclusion

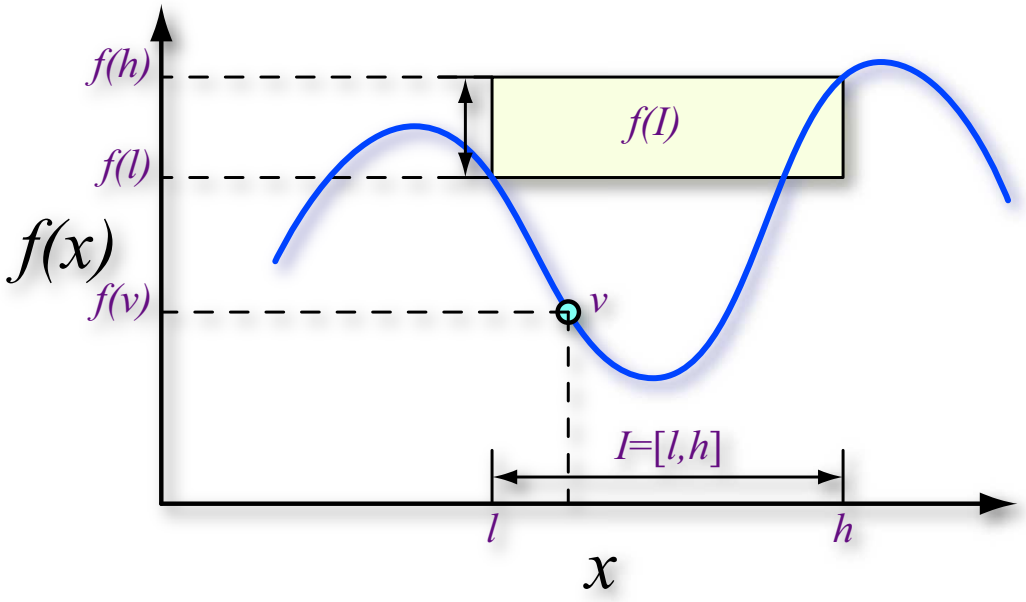
Interval arithmetic (IA) was introduced by Moore [83] as an approach to bounding numerical rounding errors in floating point computation. The same way classical arithmetic operates on real numbers, interval arithmetic defines a set of operations on intervals. We denote an interval as $\underline{\overline{x}} = [\underline{x}, \overline{x}]$, and the base arithmetic operations are as follows:

$$\underline{\overline{x}} + \underline{\overline{y}} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}],\ \underline{\overline{x}} - \underline{\overline{y}} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \tag{6.4}$$

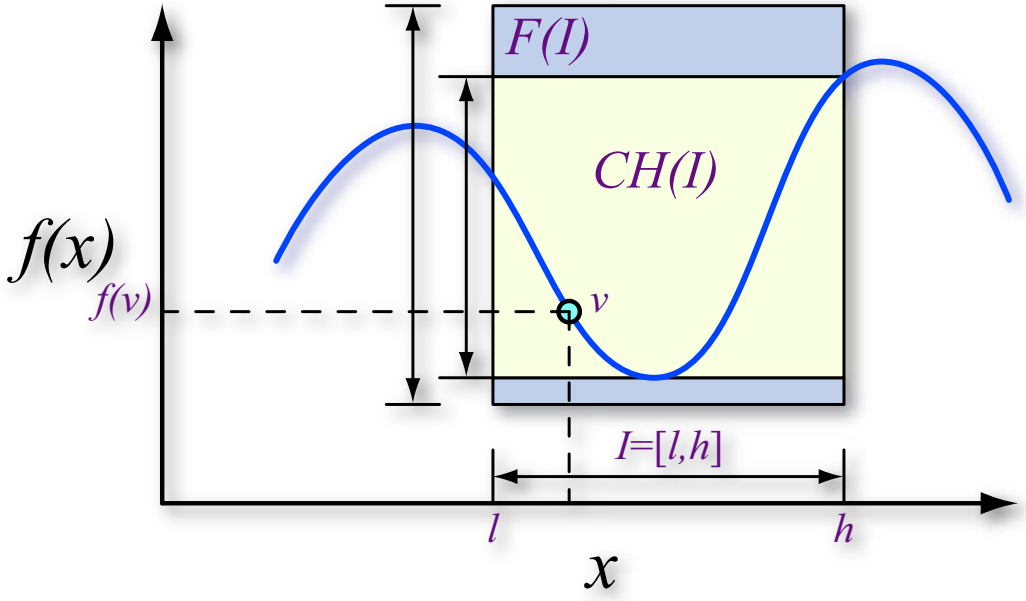
$$\underline{\overline{x}} \times \underline{\overline{y}} = [min(\underline{xy}, \underline{x\overline{y}}, \overline{x}\underline{y}, \overline{xy}), max(\underline{xy}, \underline{x\overline{y}}, \overline{x}\underline{y}, \overline{xy})] \tag{6.5}$$

Moore's fundamental theorem of interval arithmetic [83] states that for any function f defined by an arithmetical expression, the corresponding interval evaluation function F is an *inclusion function* of f:

$$F(\underline{\overline{x}}) \supseteq f(\underline{\overline{x}}) \ = \ \{f(x) \ | \ x \in \underline{\overline{x}}\} \tag{6.6}$$



(a)



(b)

Figure 6.1: The inclusion property. (a) When a function f is nonmonotonic on an interval I, evaluating the lower and upper components of a domain interval is insufficient to determine a convex hull over the range. This is not the case with an inclusion extension F (b), which encloses all minima and maxima of the function within that interval. Ideally, $F(I)$ closely envelopes the actual convex hull, $CH(I)$, enclosing the upper and lower Lipschitz bounds of f.

where F is the interval *extension* of f.

The inclusion property provides a robust rejection test that will definitely state whether an interval $\overline{\underline{x}}$ *possibly* contains a zero or other value. Inclusion operations are powerful in that they are composable: if each component operator preserves the inclusion property, then arbitrary compositions of these operators will as well. As a result, in practice *any* computable function may be expressed as inclusion arithmetic [82]. Some interval operations are ill-defined, yielding empty-set or infinite-width results. However, these are easily handled in a similar fashion to standard real-number arithmetic. A more difficult problem is converting existing efficient real-number implementations of transcendental functions to inclusion routines, as opposed to implementing an IA version from base operators. This requires ingenuity, but is usually possible and far faster than implementing an extension approximation from scratch.

The IA extension is often referred to as the *natural inclusion function*, but it is neither the only mechanism for defining an inclusion algebra, nor always the best. Particularly in the case of multiplication, it greatly overestimates the actual bounds of the range. To overcome this, it is necessary to represent intervals with higher-order approximations.

6.2.3 Affine Arithmetic

Affine arithmetic (AA) was developed by Comba & Stolfi [15] to address the bound overestimation problem of IA. Intuitively, if IA approximates the convex hull of f with a bounding box, AA employs a piecewise first-order bounding polygon, such as the parallelogram in Fig. 6.2.

An affine quantity $\hat{x}$ takes the form:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i \tag{6.7}$$

where the $x_i, \forall i \geq 1$ are the *partial deviations* of $\hat{x}$, and $e_i \in [-1,1]$ are the *error symbols*. An affine form is created from an interval as follows:

$$x_0 = (\overline{x} + \underline{x})/2,\ x_1 = (\overline{x} - \underline{x})/2,\ x_i = 0,\ \ i > 1 \tag{6.8}$$

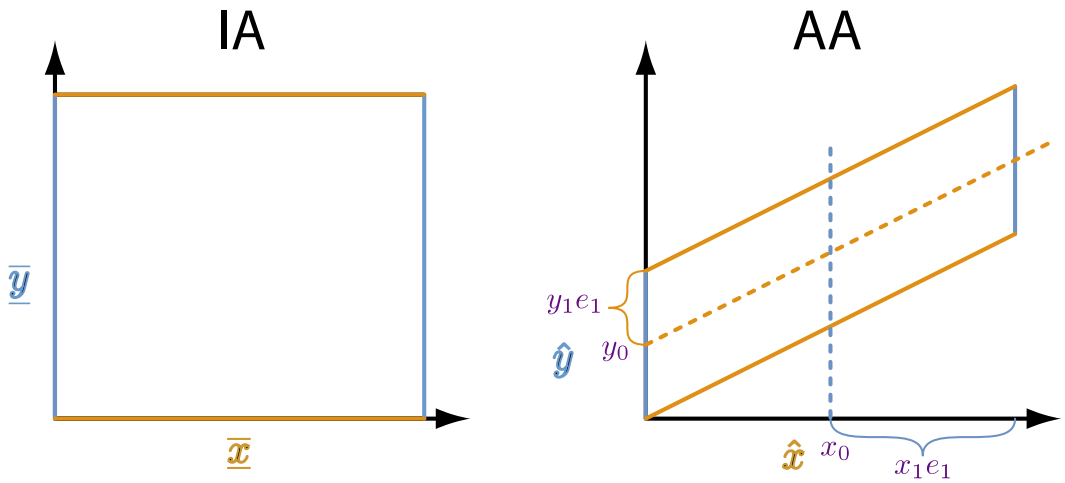


Figure 6.2: Bounding forms resulting from the combination of two interval (left) and affine (right) quantities.

and can equally be converted into an interval

$$\underline{\overline{x}} = [x_0 - \mathrm{rad}(\hat{x}), x_0 + \mathrm{rad}(\hat{x})] \tag{6.9}$$

where the *radius* of the affine form is given as:

$$\mathrm{rad}(\hat{x}) = \sum_{i=1}^{n} |x_i| \tag{6.10}$$

Affine operations in AA, where $\mathbf{c} \in \mathbb{R}$, are as follows:

$$\begin{aligned} \mathbf{c} \times \hat{x} &= \mathbf{c}x_0 + \mathbf{c}\sum_{i=1}^{n} x_i e_i \\ \mathbf{c} \pm \hat{x} &= (\mathbf{c} \pm x_0) + \sum_{i=1}^{n} x_i e_i \\ \hat{x} \pm \hat{y} &= (x_0 \pm y_0) \pm \sum_{i=1}^{n} (x_i \pm y_i) e_i \end{aligned} \tag{6.11}$$

However, *non affine* operations in AA cause an additional error symbol e_z to be introduced. This is the case in multiplication between two affine forms,

$$\hat{x} \times \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_i y_0 + y_i x_0) e_i + \text{rad}(\hat{x})\text{rad}(\hat{y}) e_z \tag{6.12}$$

Other operations in AA, such as square root and transcendentals, approximate the range of the IA operation using a regression curve – a slope bounding a minimum and maximum estimate of the range. These operations are also non affine, and require a new error symbol.

6.2.4 Condensation and Reduced Affine Arithmetic

The chief improvement in AA comes from maintaining correlated error symbols as orthogonal entities. This effectively allows error among correlated symbols to diminish, as opposed to always increasing monotonically in IA. Unfortunately, as the number of non affine operations increases, the number of noncorrelated error symbols increases as well. Despite computing tighter bounds, standard AA ultimately is inefficient in both computational and memory demands. To remedy this, AA implementations employ *condensation*. If $\hat{x}$ has n symbols, then it can be condensed into an affine entity $\hat{y}$ with $m < n$ symbols as follows [15]:

$$\begin{aligned} y_i &= x_i \quad \forall i = 0, \ldots, m-1 \\ y_m &= \sum_{i=m}^{n} |x_i| \end{aligned} \tag{6.13}$$

While $\hat{y}$ indeed bounds $\hat{x}$, condensation destroys all correlations pertaining to e_m. As a result, after condensation involving a symbol e_m, only positive-definite affine operations involving that symbol may be applied in order to preserve inclusion. Gamito and Maddock [28] employ a three-term reduced affine arithmetic that performs such condensation for every non affine operation. Though symbol correlation is destroyed, they construct their

specific extension evaluation to preserve inclusion. Nonetheless, condensation is ill-suited for arbitrary expressions, which may perform affine or non affine operations in any order.

6.2.5 Inclusion-Preserving Reduced Affine Arithmetic

In our own search for a correlation-preserving reduced affine arithmetic, we adopted a formulation equivalent to that proposed by Messine [78]. In his AF1 formulation, condensation of an entity with $n+1$ total symbols,

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i + x_{n+1} e_{n+1} \tag{6.14}$$

entails arithmetic operations as follows:

$$\begin{aligned}
\mathbf{c} \pm \hat{x} &= (\mathbf{c} \pm x_0) + \sum_{i=1}^{n} x_i e_i + |x_{n+1}| e_{n+1} \\
\hat{x} \pm \hat{y} &= (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i) e_i + (x_{n+1} + y_{n+1}) e_{n+1} \\
\mathbf{c} \times \hat{x} &= (\mathbf{c} x_0) + \sum_{i=1}^{n} \mathbf{c} x_i e_i + |c x_{n+1}| e_{n+1} \\
\hat{x} \times \hat{y} &= (x_0 y_0) + \sum_{i=1}^{n} (x_o y_i + y_0 x_i) e_i + (|x_0 y_{n+1}| + |y_0 x_{n+1}| + \mathrm{rad}(\hat{x})\mathrm{rad}(\hat{y})) e_{n+1}
\end{aligned} \tag{6.15}$$

Here, affine operations enforce positive-definite correlations between error symbols. While this does not compute bounds as tight as conventional AA, it is suitable for fixed-size vector implementation, and is in most cases a significant improvement over IA. We therefore use this as our formulation for reduced affine arithmetic (RAA).

6.2.6 Ray Tracing Implicits with Inclusion Arithmetic

The inclusion property extends to multivariate implicits as well, making it suitable for a spatial rejection test in ray tracing. Moreover, by substituting the inclusion extension of the ray equation (Equation 6.2) into the implicit extension $F(x,y,z)$, we have a univariate extension $F_t(X,Y,Z)$. To check whether any given ray interval $\bar{t} = [\underline{t}, \overline{t}]$ *possibly* contains our surface, we simply check if $0 \in F_t(\bar{t})$. As a result, once the inclusion library is implemented,

any function composed of its operators can be rendered robustly. To pick domain intervals on which to evaluate the extension, one has a wide choice of interval numerical methods. The simplest option is pure recursive bisection of intervals, examined in the order of the ray direction [82, 16, 28, 59]. Alternatives involve quasi-Newton methods and variants of the Interval Newton algorithm [12, 107] that rely on the inclusion extension of the function gradient.

6.3 SIMD CPU Ray Tracing Algorithm

We first present the SSE interval arithmetic method of [59], an example of which is shown in Figure 6.3. Our algorithm simplifies the interval bisection method first proposed by Mitchell [82], and employs a variant of coherent octree traversal [57] as opposed to direct bisection of t intervals along the ray. Together, these decisions allow us to perform bisection in a nonrecursive manner, evaluate intervals quickly using SIMD vector instructions, and avoid unnecessary per-step interval multiplication. The simplicity and efficiency of this algorithm allow it to interactively visualize most implicit functions.

The conventional Mitchell algorithm [82] employs interval bisection to reject empty (rootless) intervals. For each nonempty interval, it then computes the *gradient interval*, and determines whether $0 \notin F_t'(\underline{t})$, i.e. if the function is monotonic over an interval $\underline{t}$. When this occurrs, Mitchell resorts to a robust numerical "refinement" method, such as non IA bisection or *regula falsi*. Interval Newton methods [12, 107] also compute $F_t'(\underline{t})$ per-iteration. Gradient interval computation proves expensive. Although previous works suggest these techniques offer improved convergence and efficiency compared to pure bisection, that supposition has been weakly scrutinized. In the context of coherent traversal, we find that interval bisection yields unequivocally *better* performance, and achieves equivalent visual results efficiently at coarser sampling rates.

To leverage SIMD vector operations, we perform interval bisection on four rays at a time. Rather than bisecting t along the ray direction as in Figure 6.4(a), we bisect space along a major directional axis K, similar to the coherent octree volume traversal proposed in [60], and illustrated in Figure 6.4(b). Particularly when the space between rays exceeds the domain sampling width ε, this ensures more regular sampling of the function across neighboring rays, and preserves the spatial lockstep of coherent traversal (see Section 6.6.4).

The process of evaluating intervals is then simple. Given an interval box $\underline{b} = \underline{x} \times \underline{y} \times \underline{z}$,

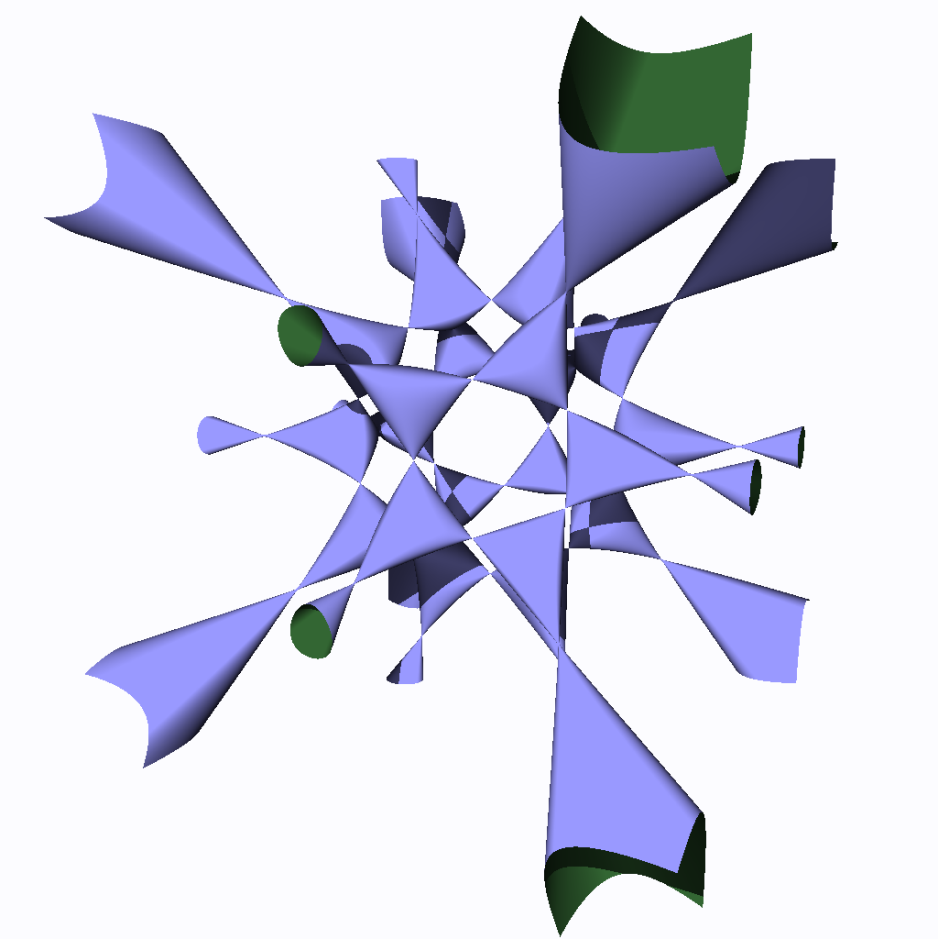


Figure 6.3: Barth sextic surface on the CPU. rendered roughly interactively at 9.0 fps (6.1 fps with shadows) with a 512^2 frame buffer on an Intel Core Duo 2.16 GHz.

our function f and its corresponding IA evaluation F, we evaluate whether $0 \in F(\overline{\underline{b}})$ for any ray in the packet. If so, we bisect that interval along the major march axis, or register a hit if a maximum depth threshold is reached. Rather than evaluating the IA extension of the implicit $F_t(\overline{\underline{t}})$ projected along the ray, as preferred by previous works, our K-bisection method evaluates the 3D implicit $F(\overline{\underline{x}}, \overline{\underline{y}}, \overline{\underline{z}})$ directly. This is convenient as both the IA extension and evaluation functions are natively given as $f(x,y,z)$ expressions. Moreover, our traversal algorithm computes domain intervals $\overline{\underline{b}}$ incrementally, requiring only three SSE additions per iteration. Conversely, evaluating $F_t(\overline{\underline{t}})$ requires IA evaluation of Equation 6.3: three IA multiplications and IA additions, or six SSE multiply, min, max and add operations in total.

The SIMD CPU algorithm shows that interval arithmetic can in fact be an effective method for rendering arbitrary-form implicits. While bound overestimation and computationally complex implicits can be costly to render using this technique, it is still orders of

magnitude better than the most recently published interval arithmetic or affine arithmetic method [24]. Interestingly, as we later show in Section 6.6.4, brute-force bisection outperforms more sophisticated quasi-Newton methods, particularly for the purpose of rendering implicits which requires relatively low numerical precision. Ultimately, in Section 6.6.4, we find that strategies for maximizing SIMD coherence and performance differ on CPU and GPU platforms.

6.4 SIMD CPU Implementation

Our application takes as inputs a domain $\Omega \subseteq \mathbb{R}^3$, and an implicit function expression. For simplicity, we chose to hard-code most functions as IA expressions; however the function can also be received from the user as a string and then parsed and compiled into IA code in a dynamic library on-the-fly.

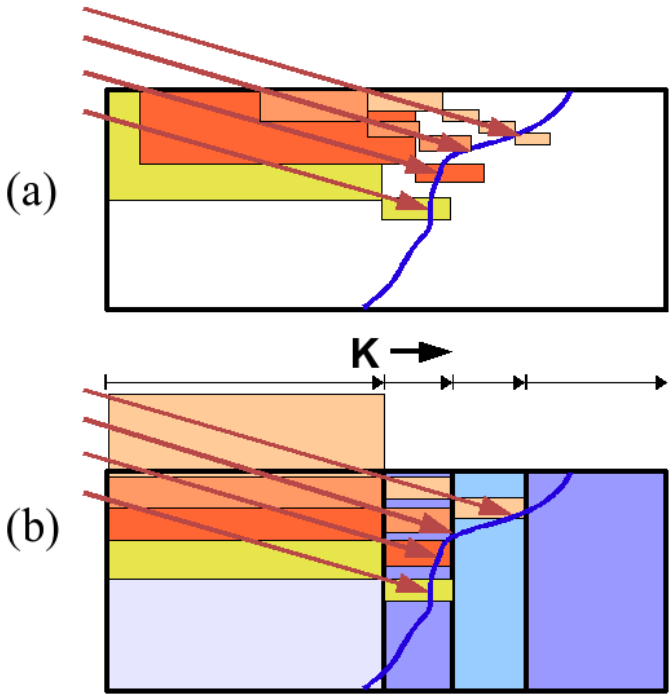


Figure 6.4: Spatial interval bisection methods. The conventional method (a) recursively bisects each ray along its parameter t until a surface is located to the satisfaction of a termination criterion. Our $K-marching$ technique (b) marches rays along a common axis in lockstep. Evaluating along 3D interval boxes B requires slightly less computation per iteration than evaluating the projected function $f_t(t)$. More importantly, traversing along a common spatial axis induces more coherent behavior between rays in a packet.

6.4.1 SSE Interval Arithmetic

The foundation of our implicit ray tracing system is our own SSE IA library, which allows us to quickly evaluate intervals in SIMD. Implementation is straightforward; interval multiplication is particularly efficient as SSE itself is relatively fast for both multiplication and minimum/maximum operation. The only nontrivial operators are periodic functions such as modulus and sine; and division which requires special-case handling during traversal (see Section 6.4.4). Examples of SSE IA pseudocode are given in Algorithm B.1 in Appendix B. We deliberately ignore IA rounding rules for numerical conditioning. Empirically, we find floating point round-off errors are insignificant compared to the termination tolerance of our bisection algorithm. One could likely devise numerically ill-conditioned functions that would require IA rounding, but in practice it is not a major issue.

6.4.2 Ray Packet Structure

We chose conservative 2x2 packets for our implementation. Above all, we wish to evaluate baseline performance with SIMD ray tracing using 4-wide SSE vectors; thus behavior of our system should be consistent on wider SIMD hardware, such as a GPU or FGPA. Though larger packets coupled with multilevel algorithms could be significantly faster [99], 2x2 packet traversal is better-suited for general-purpose ray tracing, and easily allows our implicits to be integrated into a ray tracer as geometric intersection primitives. The actual packet architecture should generalize to any coherent ray tracer; our packet implementation consists of origin and direction stored for each X, Y, Z axis in SSE packed floats. Packets also store the ray hit parameters t, and a mask indicating which rays have hit.

6.4.3 Traversal

Once the user has supplied a function, a domain box $\Omega \subseteq \mathbb{R}^3$, and a maximum depth d_{stop}, we are ready to perform traversal. As in coherent grid traversal [130], we first find K, the dominant axis of the first ray in the packet, and denote the remaining two axes U and V. We then perform a standard ray bounding-box test on our domain. We store the actual t_{enter} and t_{exit} parameters as well as the intersections with the K entry and exit planes, t_{Kenter} and t_{Kexit}. Now, we consider the total increment along K, $t_{Kexit} - t_{Kenter}$, and compute the total U and V increments over the entire domain. As our implementation is iterative, not recursive, we store an array containing a traversal "stack" for each depth $\{0..d_{stop} - 1\}$, containing the t, K, U and V increments bisected at each level.

The algorithm then simply marches from one K slice to the next, incrementing the t, K, U and V positions once per step and keeping track of current and next values, orthogonally for each ray using SSE. It constructs intervals from the K, U and V current and next values. This enables us to iteratively increment domain intervals simply with three SSE additions, as opposed to three SIMD IA multiplications and additions using the conventional t-marching method. Branching is only used to omit intervals when $t < t_{enter}$, and exit when all rays hit successfully or have $t > t_{exit}$. We store and check a flag for each depth, which indicates when both sides of a K-subtree have been traversed. When this happens, we decrement the depth, and exit traversal when $depth = -1$.

At each march iteration, we evaluate the IA function expression on this domain interval $B = X \times Y \times Z$. If $0 \in F(X,Y,Z)$, we "recurse" by incrementing d and using the bisected increments one level deeper. We register a hit on the surface when $d == d_{stop} - 1$ (or another hit criterion is met, such as $||F(B)|| < \delta$, as in Section 6.4.5). Finally, we mask rays that successfully hit or terminate traversal when all rays hit. Traversal is illustrated in Figure 6.4(b), and pseudocode is given in Algorithm 7.

6.4.4 Division

IA division requires a slight modification to the above algorithm. In theory, IA division by intervals containing zero is ill-defined, similar to division of real numbers by zero. Fortunately, we can easily detect and handle these cases. For two intervals $\overline{a}$ and $\overline{b}$, when $0 \in \overline{b}$, we define $\overline{a}/\overline{b} = [-\infty, \infty]$. When rays traverse these intervals, they will *always* find a surface within and recurse to maximum depth. Thus, without modification to the traversal, asymptotes will be rendered. To avoid rendering asymptotes, we simply neglect to register a hit when $\overline{f} - \underline{f} = \infty$. This principle is illustrated in Figure 6.5. With division correctly handled, our traverser will work for literally any function composed of IA operators.

6.4.5 Precision Criterion

In our implementation, d_{stop} determines the default precision for rendering the implicit. Roughly, this corresponds to a domain precision of $2^{-d_{stop}}$, though indeed this varies by ray. However, for a more view-independent domain-space metric, the user may optionally specify an ε, such that $||\overline{b}||_2 < \varepsilon$ serves as hit criterion, where $\overline{b}$ is an interval box $\overline{x} \times \overline{y} \times \overline{z}$. In this case, the stopping depth is determined adaptively per packet as

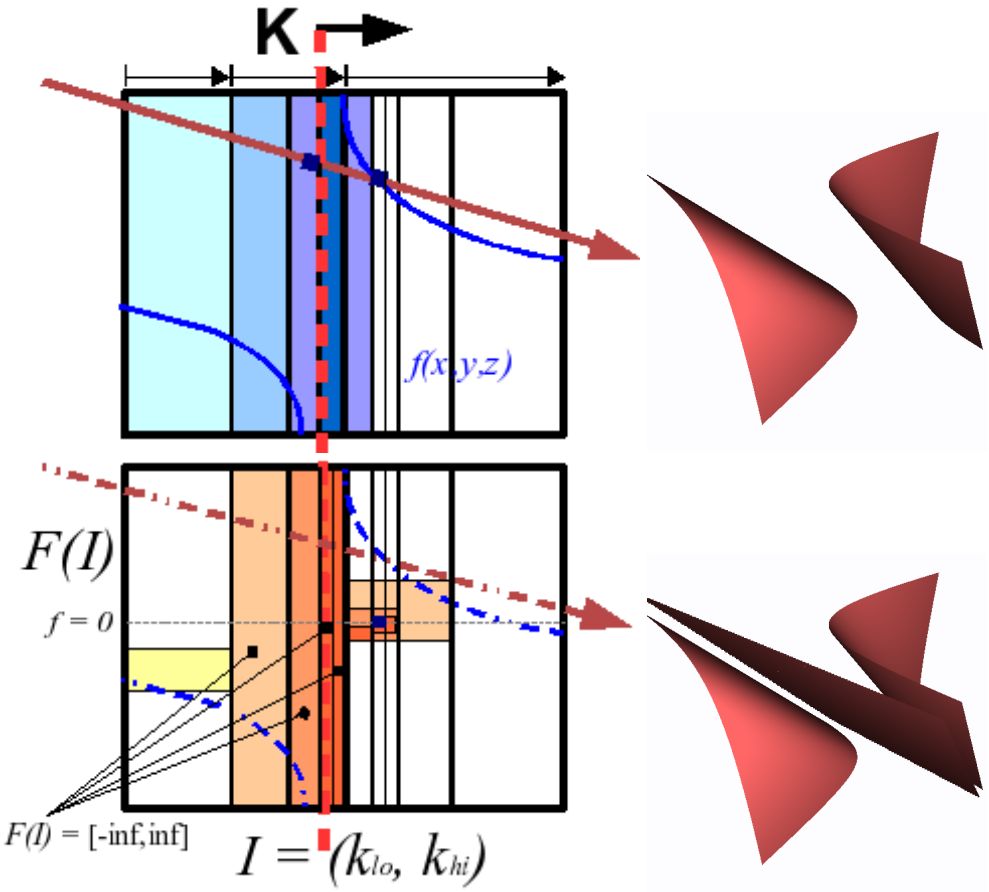


Figure 6.5: Handling division with IA. For functions with division, and intervals containing zero near an asymptote, our IA implementation returns "infinite" $F(I)$ intervals (bottom left). As a result, these regions are always subdivided until termination (top left). Fortunately, we may detect this infinite case within the traverser before registering a hit, and thus choose whether or not to visualize asymptotes.

$$d_{stop} = log_2(\Delta_{packet}/\varepsilon) \tag{6.16}$$

where for world-space ray entry and exits $\vec{P}_r$ with the domain box Ω, and their corresponding K-coordinates K_r,

$$\Delta_{packet} = max_{r \in packet} \frac{(||\vec{P}_r^{exit} - \vec{P}_r^{enter}||_2)^2}{|K_r^{exit} - K_r^{enter}|} \tag{6.17}$$

Alternately, the user may specify a range tolerance δ, in which case our algorithm registers a hit when $||F(B)|| < \delta$. Empirically, the performance differences between these metrics proved minor, and at low precision the d_{stop} method yields more continuous results for neigh-

boring rays. Thus, in practice we use d_{stop} as the default metric for evaluating performance at varying sampling quality.

6.4.6 Shadows

In ray tracing, hard shadows are fairly trivial, requiring a shadow ray cast for every primary camera ray that hits a surface. On the CPU this typically entails a 20% to 50% decrease in frame rate, depending on the coherent behavior of shadow rays. Fortunately, useful shadow rays require less accuracy than primary rays; it frequently suffices to cast shadows to a coarser termination depth, such as $d_{stop} - 2$, while employing a higher depth for primary rays. As shadows are primarily useful as depth cues, this is generally acceptable. The performance penalty is reduced, and loss of shadow detail is seldom perceptible (Figures 6.3 and 6.6).

6.4.7 Gradient Computation

For Lambertian shading, we require the surface normal at the ray hit position, given by the $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ partial derivatives at that point. While analytical gradients can be manually defined, they are not strictly necessary. If the user fails to define partials, we employ central differences by evaluating our function (using SSE, not SSE IA evaluation) six times to create a central differences stencil. The results look excellent in most cases, and have no

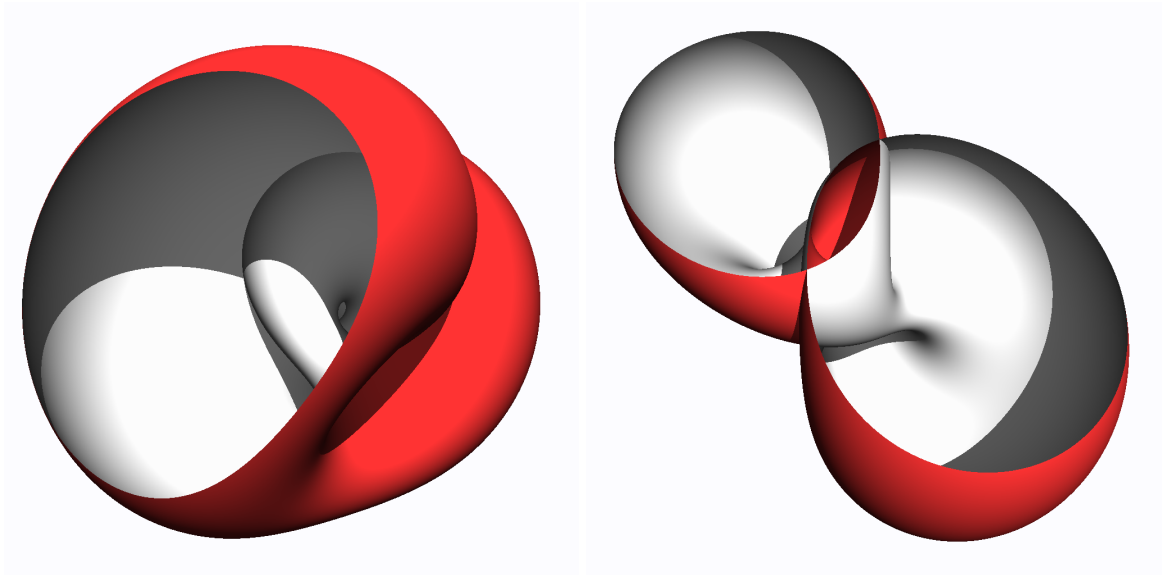


Figure 6.6: The Klein bottle rendered using SIMD IA bisection. Dynamic shadows aid the visualization and are trivial with ray tracing. Images rendered at 4.0 fps and 2.9 fps, respectively, at $d_{stop} = 12$.

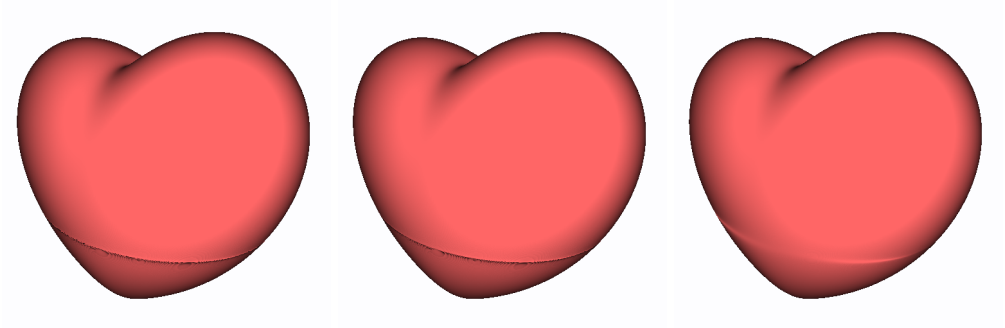


Figure 6.7: Gradient normal computation, on the Heart function $f(x,y,z) = (2x^2 + y^2 + z^2 - 1)^3 - (.1x^2 + y^2)z^3$. Left: using analytical partial derivatives as gradient, we see shading artifacts where the gradient magnitude approaches zero. Center: with a central differences stencil of width $\Delta_S = 0.001$, the results are visually indistinguishable. Right: smoother normals with $\Delta_S = 0.01$. All images render at 6.7 fps using the SIMD CPU method of [59].

appreciable impact on performance. We allow the user to specify stencil width; this is frequently beneficial for surface regions with near-zero gradient magnitude (Figure 6.7).

6.5 GPU Algorithm

The GPU technique involves a Cg implementation of the interval bisection algorithm (Section 6.5.5) and an implementation of reduced affine arithmetic (Section 6.5.3) suitable for the NVIDIA G80 architecture. Overall, shader languages such as Cg 2.0 allow for a more graceful implementation than the optimized SSE C++ counterpart on the CPU. Just-in-time shader compilation, in conjunction with metaprogramming, can easily and dynamically generate IA/AA extension routines from an input expression. Nonetheless, implementing a robust interval-bisection ray tracer on the GPU poses challenges. Principally, the CPU algorithm relies on an efficient iterative algorithm for bisection, employing a read/write array for the recursion stack. Storing such an array per-fragment occupies numerous infrequently-used registers, which slows processing on the GPU. Similar problems have clearly hampered performance of hierarchical acceleration structure traversal for mesh ray tracing [95]. Our most significant contribution is a traversal algorithm that overcomes this problem. By employing simple floating-point modulus arithmetic in conjunction with a DDA-like incremental algorithm operating on specially constructed intervals, we are able to perform traversal without any stack. Though this algorithm would be inefficient on a CPU, it is well-suited for the GPU architecture thanks to efficient floating-point division.

In implementing affine arithmetic to mitigate IA bound overestimation, it was immediately clear that a full array-based implementation of conventional AA would be impractical on the GPU. Though efficient, the reduced affine arithmetic method proposed by Gamito & Maddock [28] only preserves inclusion under specific circumstances. Fortunately, with modifications ensuring that the last error term is positive-definite, a formulation similar to that of Messine et al. [78] implements a correct inclusion for all compositions of AA operations. In adopting such an arithmetic, we implement a robust reduced AA suitable for ray tracing on the GPU. Particularly for complex forms requiring cross-multiplication between interval entities, this yields more correct results at lower required precision than standard IA, and superior frame rates for most functions.

6.5.1 Application Pipeline

As input, the user must simply specify a function in implicit form, a domain $\Omega \subset \mathbb{R}^3$, and a termination precision ε that effectively bounds relative error (see Section 6.6.3). User-specified variables are stored on the CPU and passed dynamically to Cg as uniform parameters. Some runtime options, such as the implicit function, choice of inclusion algebra, or shading modality, are compiled directly into the Cg shader through metaprogramming. In simple cases, the CPU merely searches for a stub substring within a base shader file, and replaces it with Cg code corresponding to the selected option. More advanced metaprogramming involves creating routines for function evaluation. Given an implicit function expression, we require two routines to be created within the shader: one evaluating the implicit f, and another evaluating the inclusion function, the interval or affine extension F. We use a simple recursive-descent parser to generate these routines in the output Cg shader. Alternately, we allow the user to directly provide inline Cg code. Because the shader compiler identifies common subexpressions, this is seldom necessary for improving performance. Our only examples employing inline code are special-case conditional evaluations in CSG objects.

Though our system is built on top of OpenGL, we use the fixed-function rasterization pipeline very little. Given a domain $\Omega \subset \mathbb{R}^3$ specified by the user, we simply rasterize that bounding box once per frame. We specify the world-space box vertex coordinates as texture coordinates as well. These are passed straight through a minimal vertex program, and the fragment program merely looks up the automatically interpolated world-space entry point

of the ray and the bounding box. By subtracting that point from the origin, we generate a primary camera ray for each fragment.

6.5.2 Shader IA Library

Implementing an interval arithmetic library (Section 6.2.2) is straightforward in Cg. Most scalar operations employed by IA (such as min and max) are highly efficient on the GPU, and swizzling allows for effective horizontal vector implementation (Algorithm 5 in Appendix B), unlike SSE SIMD on the CPU. Transcendental functions are particularly efficient for both their floating-point and interval computations.

On the GPU, we can use JIT compiling and metaprogramming to dynamically generate efficient IA routines for large integer powers. Russian peasant multiplication [81] allows us to recursively create routines for scalar evaluation of integer powers. Then, we can employ the IA rule:

$$\overline{\underline{x}}^{\,n} = [min(\underline{x}^{\,n}, \overline{x}^{\,n}), max(\underline{x}^{\,n}, \overline{x}^{\,n})] \tag{6.18}$$

The bound-efficiency of this IA rule, combined with the fast evaluation of integer powers via logarithmic decomposition of multiplications, frequently makes evaluation of high-degree polynomials more efficient with IA than with AA (see Section 6.6). This suggests that implicit surfaces defined by fully-expanded algebraic functions will in fact perform better with interval arithmetic.

6.5.3 Shader RAA Library

In implementing our RAA library on the GPU, we adopt a formulation similar to AF1 in Messine et al. [78], with changes to the absolute value bracketing that are mathematically equivalent but slightly faster to compute. We implemented AF1 with $n = 1$ using a float3 to represent the reduced affine form. We also experimented with $n = 2$ (float4), and $n = 6$ (a double-float4 structure). For all the functions in our collection, the float3 version delivered the fastest results by far. We also found that the computational overhead of the bound-improved AF2 formulation [78] was too high to be efficient. Examples of the float3 version of the forms in Equation 6.15 are given in Algorithm 6 in Appendix B.

The float3 implementation of AF1 makes for a versatile and fast reduced affine arithmetic. Particularly for functions with significant multiplication between noncorrelated affine variables, such as the Mitchell or the Barth surfaces involving cross-multiplication of Chebyshev polynomials, significant speedup can be achieved over standard IA.

6.5.4 Numerical Considerations

A technical difficulty arises in the expression of infinite intervals, which may occur in division; and empty intervals that are necessary in omitting nonreal results from a fractional power or logarithm. While these are natively expressed by nan on the CPU, GPU's are not always IEEE compliant. The G80 architecture correctly detects and propagates infinity and nan, but the values themselves (inf $= 1/0$ and nan $= 0/0$) must be generated on the CPU and passed into the fragment program and subsequent IA/AA calls.

Conventionally, IA and AA employ a rounding step after every operation, padding the result to the previous or next expressible floating point number. We deliberately omit rounding – in practice the typical precision ε is sufficiently large that rounding has negligable impact on the correct computation of the extension F. However, numerical issues can be problematic in certain affine operations: RAA implementations of square root, transcendentals and division itself all rely on accurate floating point division for computing the regression lines approximating affine forms. Though inclusion-preserving in theory, these methods are ill-suited for inaccurate GPU floating point arithmetic; and a robust strategy to overcome these issues has not yet been developed for RAA. We therefore resort to interval arithmetic for functions that require regression-approximation AA operators.

6.5.5 Traversal

With the IA/RAA extension and a primary ray generated on the fragment unit, we can perform ray traversal of the domain $\Omega \subset \mathbb{R}^3$. Though not as trivial as standard numerical bisection for root finding, the ray traversal algorithm is nonetheless elegantly simple, and can be found in its entirety in Algorithm 7 in Appendix B.

6.5.5.1 Initialization

We begin by computing the exit point p_{exit} of the generated ray and the bounding box Ω. We reparameterize the ray as $\vec{r}(t) := \vec{p}_{enter} + t(\vec{p}_{exit} - \vec{p}_{enter})$. The interval $\bar{t}$ along the ray intersecting Ω is now $[0,1]$. We now perform a first rejection test outside the main loop.

6.5.5.2 Rejection test

In the rejection test, we evaluate the IA/AA extensions of the ray equation to find X, Y and Z over $\bar{t}$, and use these (as well as scalars w, r_i for time and other animation variables) to evaluate the extension of our implicit function. The result gives us an interval or affine approximation of the range F. If $0 \in F$, then we must continue to bisect and search for roots. Otherwise, we may safely ignore this interval and proceed to the next, or terminate if it is the last.

6.5.5.3 Main loop

If the outer rejection test succeeds, we compute the effective bisection depth required for the user-specified ε. This is given by the integer ceiling:

$$d_{max} := \text{ceil}(\ \log_2(\frac{||\vec{p}_{exit} - \vec{p}_{enter}||}{\varepsilon})) \tag{6.19}$$

We initialize our depth $d = 0$, and distance increment, $t_{incr} = 0.5$. Now, recalling the bisection interval $\bar{t}$, we set $\underline{t} := \bar{t} + t_{incr}$. We then perform the rejection test on this new $\bar{t}$. If the test succeeds, we either hit the surface if we have reached $d = d_{max}$, or recurse to the next level by setting $t_{incr} := t_{incr}/2$, and incrementing d.

If the rejection test fails, we proceed to the next interval segment at the current depth level by setting $\bar{t} := \underline{t}$. Within the main loop, we now perform another loop to back-recurse to the appropriate depth level.

6.5.5.4 Back-recursion loop

In back-recursion, we decrement the depth d (and update t_{incr}) until we find an unvisited segment of the bisection tree. This allows us to perform ray bisection iteratively, not recursively, and without employing registers to mimic a recursion stack. Specifically, we perform floating-point modulus ($\underline{t}\ \%\ 2t_{incr} = 0$) to verify whether the current distance has visited one or both bisected segments in question. Currently on the G80, the fastest method proves to be performing division and examining the remainder. Back-recursion proceeds iteratively until it finds an unvisited second branch of the bisection tree, or $d = -1$ in which case traversal has completed.

6.5.6 Traversal Metaprogramming

The traversal algorithm largely remains static, but some functions and visualization modalities require special handling. To render functions containing division operations, we must check whether intervals are infinitely wide before successfully hitting, as detailed in Knoll et al. [59]. Multiple isovalues and transparency require modifications to the rejection test and hit registration, respectively, as discussed in Section 6.6.6. More generally, modifications to the traversal algorithm are simple to implement via "inline" implicit files (Section 6.5.1). We allow the user to directly program behavior of the rejection test, hit registration and shading. This is particularly useful in rendering special-case constructive solid geometry objects.

6.5.7 Shading

Phong shading requires a surface normal, specifically the gradient of the implicit at the found intersection position. We find central differencing to be more than adequate, as it requires no effort on the part of the user in specifying analytical derivatives, nor special metaprogramming in computing separable partials via automatic differentiation. By default we use a stencil width proportional to the traversal precision ε; variable width is often also desirable [59].

6.6 Results

All benchmarks are measured in frames per second at 1024x1024 frame buffer resolution.

6.6.1 Performance

Table 6.1 shows base frame rates of a variety of surfaces using single ray-casting and basic Phong shading. Performance on the NVIDIA 8800 GTX is up to $22\times$ faster than the SIMD SSE method on a 4-core Xeon 2.33 GHz CPU workstation. Frame rate is determined both by the bound tightness of the chosen inclusion extension, and the computational cost of evaluating it. In practice, the order of the implicit form has little impact on performance. Forms of these implicits can be found in Appendix B.3.

6.6.2 IA versus RAA

For typical functions with fairly low-order coefficients and moderate cross-multiplication of terms, reduced affine arithmetic is generally $1.5-2\times$ faster than interval arithmetic. For functions with high bound overestimation, such as those involving multiplication of large

Table 6.1: Implicit surface performance on the CPU and GPU, for various surfaces at $1024x1024$ resolution, with corresponding renderings indicated by the figure numbers in parentheses. The CPU SIMD algorithm is benchmarked on a four-core 2.33 GHz Intel Xeon desktop, using only IA. The GPU algorithm runs on an NVIDIA 8800GTX; results are shown with both IA and RAA. Results in these first three columns are evaluated with common $\varepsilon = 2^{-11}$; the last column labelled "converged ε" shows performance at the the highest ε yielding a correctly converged visual result, using either IA or RAA on the GPU.

	CPU	GPU		
ε	2^{-11}	2^{-11}	2^{-11}	converged ε
arithmetic	IA	IA	RAA	IA or RAA
sphere	15	75	147	165 /RAA /2^{-10}
steiner (6.9)	7.5	34	40	38 /RAA /2^{-12}
mitchell (6.8)	5.2	16	58	60 /RAA /2^{-10}
teardrop (6.10a)	5.5	102	115	121 /RAA /2^{-10}
4-bretzel (6.10c)	13	78	48	90 /IA /2^{-10}
klein b. (6.10b)	11	30	110	101 /RAA /2^{-12}
tangle (6.10d)	3.2	15	68	71 /RAA /2^{-10}
decocube (6.12)	5.5	28	27	28 /IA /2^{-11}
barth sex. (6.11l)	7.4	31	76	88 /RAA /2^{-10}
barth dec. (6.11r)	0.92	4.9	15.6	15.6 /RAA /2^{-11}
superquadric	18	119	8.3	108 /IA /2^{-12}
icos.csg (6.13l)	1.8	13.3	-	13.3 /IA /2^{-11}
sesc.csg (6.13r)	1.6	8.9	-	7.2 /IA /2^{-13}
sin.blob (6.15)	0.71	6.0	-	6.0 /IA /2^{-12}
cloth (6.14l)	2.2	38	-	44 /IA /2^{-9}
water (6.14r)	2.2	37	-	44 /IA /2^{-9}

polynomial terms (e.g., the Barth surfaces) or Horner forms, RAA is frequently 3 to 4 times faster. Conversely, thanks to an efficient inclusion rule for integer powers, IA remains far more efficient for superquadrics, as evident in Table 6.1. As explained in Section 6.5.4, IA is currently required for extensions of division, transcendentals, and fractional powers.

6.6.3 Error and Quality

As seen in Equation 6.19, a global user-specified ray-length precision ε is used to determine a per-ray maximum bisection depth d_{max}. If a candidate ray interval $\underline{\bar{t}}$ contains a zero, then the actual error is

$$\varepsilon_{actual} \leq ||\underline{\bar{t}}|| = 2^{-d_{max}} \leq \varepsilon \tag{6.20}$$

This effectively specifies an upper bound on the absolute error in ray space t; by scaling by the magnitude of the ray segment over Ω, $||p_{exit} - p_{enter}||$, we normalize to bound relative error in world space. Our application also allows the user to specify a tolerance δ, which halts bisection only when the width of the interval $||F|| < \delta$. This would seem a more adaptive way of guaranteeing convergence, as bisection proceeds until the interval width is sufficiently small to better guarantee existence (or nonexistence) of a root. However, range interval width varies widely by function, and is more difficult for the user to gauge than the domain-space ε.

Choice of appropriate ε depends greatly on the implicit in question. For most of our examples, $\varepsilon = 2^{-11}$ yields a topologically correct rendering, and thus is suitable as a default. Figure 6.8 on the next page shows the impact of precision ε, controlling relative error, on the Mitchell and Barth decic surfaces, both examples with particularly high bound overestimation and sensitivity to low precision. RAA generally converges far more quickly than IA, given lesser bound overestimation at low ε. In addition, refining ε has lesser impact on frame rate once RAA has effectively converged. Finally, we note that increasing ε generates progressively tighter convex hulls around the ideal surface at $\varepsilon = 0$.

6.6.4 Algorithm Coherence and Performance

Table 6.2 shows the relative performance of various algorithms on the Mitchell and Barth Decic functions shown in Figure 6.8, at $\varepsilon = 2^{-11}$ and $\varepsilon = 2^{-22}$. Our suggested implementations (also used in Table 6.1) are shown in boldface. The efficiency of both CPU

Table 6.2: Performance of various algorithms on the Mitchell and Barth decic functions, using interval arithmetic only.

function	Mitchell		Barth Decic	
ε	2^{-11}	2^{-22}	2^{-11}	2^{-22}
CPU SSE				
t-bisection	5.0	1.0	0.90	0.061
K-bisection	**5.1**	**1.2**	**0.92**	**0.18**
Mitchell	0.54	0.22	0.19	0.036
GPU (IA)				
t-bisection	**16**	**6.2**	**4.9**	**1.4**
K-bisection	11	5.6	4.4	1.1
Mitchell	3.9	1.0	1.1	0.29

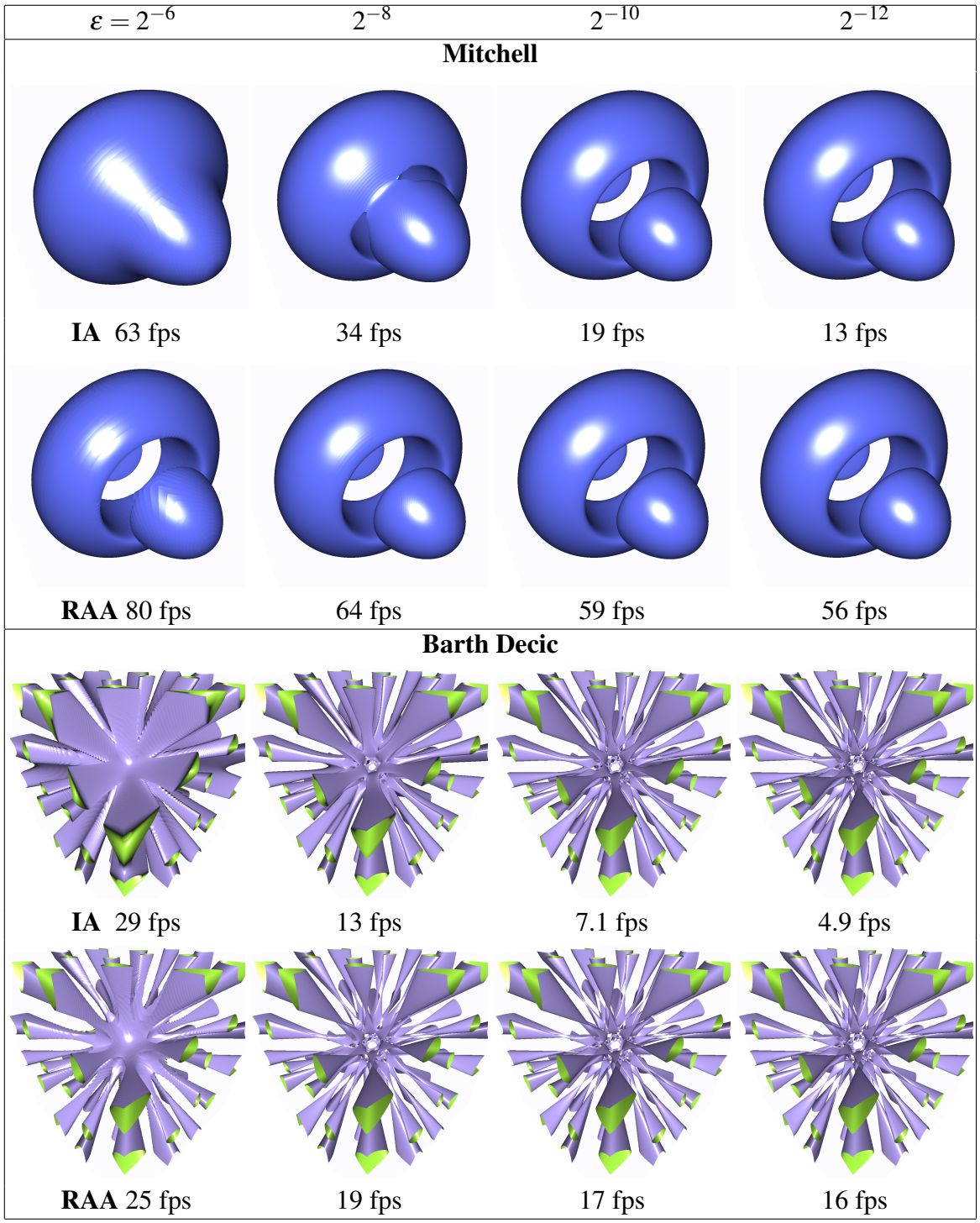


Figure 6.8: The Mitchell (top) and Barth Decic (bottom) surfaces, at various ε, with IA and RAA.

(Section 6.3) and GPU (Section 6.5) algorithms depends on exploitation of SIMD coherence. The CPU SSE algorithm benefits from explicit spatial coherence, as shown in Figure 6.4(b). With the t-marching method in SSE (Figure 6.4(a)), rays in the same packet can fall out of lockstep, destroying coherence. Conversely, the GPU algorithm requires more general instruction-level coherence, with a minimum of used registers. A modification of the GPU algorithm to march along the major **K**-axis yielded noticeable performance decrease. We also note that both the SSE CPU and GPU implementations of the Mitchell [82] algorithm (employing interval arithmetic followed by standard numerical root refinement) perform far worse than naïve bisection, particularly at higher ε. This can be attributed to the high cost of evaluating the gradient interval, and both worse instruction-level coherence on the GPU and spatial coherence in the SSE CPU algorithm. Though difficult to fairly evaluate on the GPU, our experimentation with SSE versions of other quasi-Newton methods such as Interval Newton method and [12] empirically suggested far worse results. However, these algorithms could prove desirable if efficiently mapped to SIMD architecture.

6.6.5 Feature Reproduction and Robustness

As it entails more floating-point computation than IA, RAA has worse numerical conditioning, particularly with smaller ε. Figure 6.9 illustrates the challenge in robustly ray tracing the Steiner surface with IA and AA. Both inclusion methods identify the infinitely thin surface regions at the axes, but a small $\varepsilon < 2^{-18}$ is required for correct close-up visualization of these features. Affine arithmetic yields a tighter contour of the true zero-set than IA, but

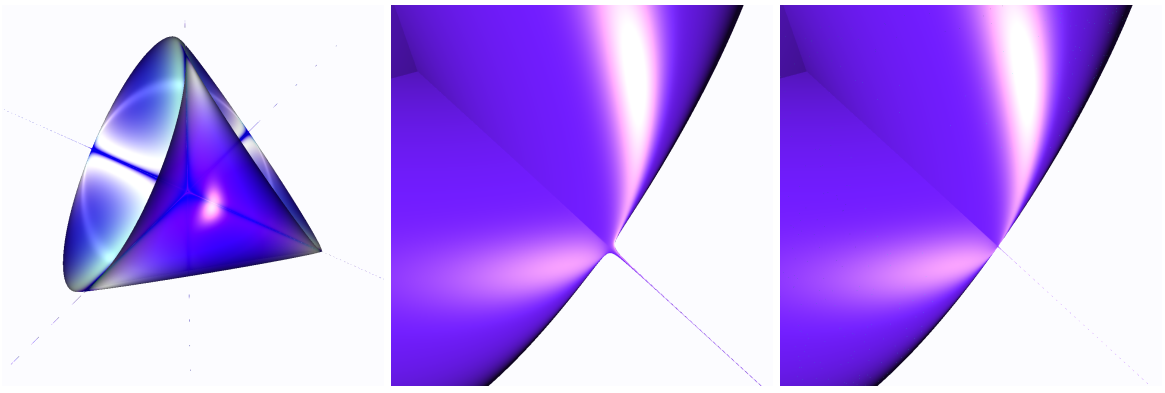


Figure 6.9: Fine feature visualization in the Steiner surface. Left to right: shading with depth peeling and gradient magnitude coloration; close-up on a singularity with IA at $\varepsilon = 2^{-18}$; and with RAA at the same depth.

with some speckling. While not crucial with IA, a comprehensive rounding strategy would greatly benefit the numerical stability of RAA. Nonetheless, both IA and RAA yield more robust results than noninclusion ray tracing methods [72] on the Steiner surface, or than inclusion-based extraction [88] on the teardrop (Figure 6.10(a)).

6.6.6 Shading Modalities

As our algorithm relies purely on ray-tracing, we can easily support per-pixel lighting models and multiple bounce effects, many of which would be difficult with rasterization (Figure 6.10). We briefly describe the implementation of these modalities, and their impact on performance.

6.6.6.1 Shadows

Nonrecursive secondary rays such as shadows are straightforward to implement. Within the main fragment program, after a successfully hit traversal, we check whether $\vec{N} \cdot \vec{L} > 0$, and if so, perform traversal with a shadow ray. To ensure we do not hit the same surface, we cast the shadow from the light to the hit position, and use their difference to reparameterize the ray so that $\bar{t} = [0,1]$, as for primary rays. Shadows often entail around $20-50\%$ performance penalty. One can equally use a coarser precision for casting shadow rays than primary rays. RAA is sufficiently accurate for secondary rays even at $\varepsilon > .01$; which can decrease the performance overhead to $10-30\%$.

6.6.6.2 Transparency

Transparency is also useful in visualizing surfaces, particularly functions with odd connectivity or disjoint features. With ray tracing, it is simple to implement front-to-back, order-independent transparency, in which rays are only counted as transparent if a surface behind them exists. Our implementation lets the user specify the blending opacity, and casts up to four transparent rays. This costs around $3\times$ as much as one primary ray per pixel.

6.6.6.3 Multiple isosurfaces

One may equally use multiple isovalues to render the surface. This is significantly less expensive than evaluating the CSG object of multiple surfaces, as the implicit extension need only be evaluated once for the surface. The rejection test then requires that *all* of those isovalues miss. At hit registration, we simply determine which of those isovalues hit, and

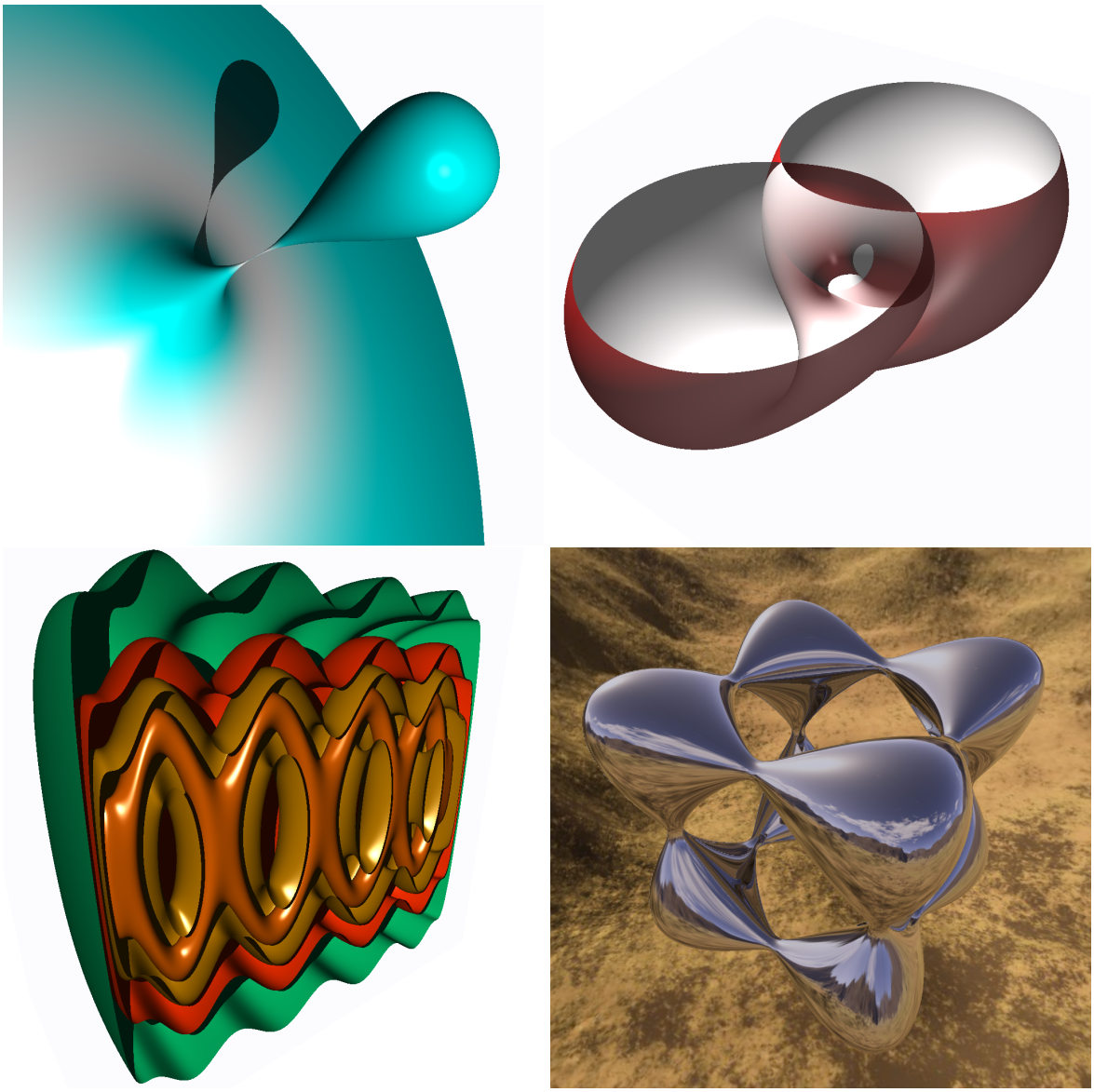


Figure 6.10: Shading Effects. Top left to bottom right: (a) shadows on the teardop (40 fps); (b) transparency on the Klein bottle (41 fps); (c) shadows and multiple isovalues of the 4-Bretzel (18 fps); and (d) the tangle with up to six reflection rays (44 fps).

flag the shader accordingly to use different surface colors. With no other effects, multiple isovalues typically entail a cost of anywhere from $10-40\%$.

6.6.6.4 Reflections

Reflections are a good example of how built-in features of rasterization hardware can be seamlessly combined with the implicit ray tracing system. Looking up a single reflected value from a cubic environment map invokes no performance penalty. Tracing multiple

reflection rays in an iterative loop is not significantly more expensive ($20-30\%$), and yields clearly superior results (Figure 6.10d).

6.6.7 Applications

6.6.7.1 Mathematical visualization

The immediate application of this system is a graphing tool for mathematically interesting surface forms in $3D$ and $4D$. Ray tracing ensures view-dependent visualization of infinitely thin features, as in the teardrop and Steiner surfaces. It is similarly useful in rendering singularities – Figure 6.11 shows the Barth sextic and decic surfaces, which contain the maximum number of ordinary double points for functions of their respective degrees in $\mathbb{R}^3$.

6.6.7.2 Interpolation, morphing and blending

Implicit forms inherently support blending operations between multiple basis functions. Such forms need only be expressed as an arbitrary 4D implicit $f(x,y,z,w)$, where w varies over time. As ray-tracing is performed purely on-the-fly with no precomputation, we have great flexibility in dynamically rendering these functions. The blending function itself can operate on multiple kernels, and be of arbitrary form. Figure 6.12 shows morphing between a decocube and a sphere by interpolating a sigmoid convolution of those kernels.

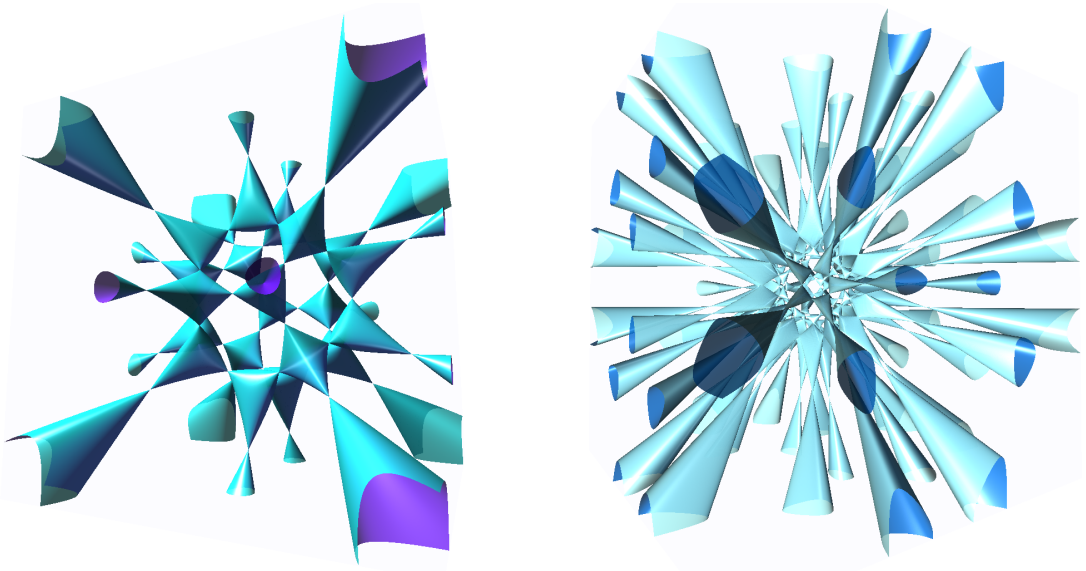


Figure 6.11: ***The Barth sextic and decic surfaces.***

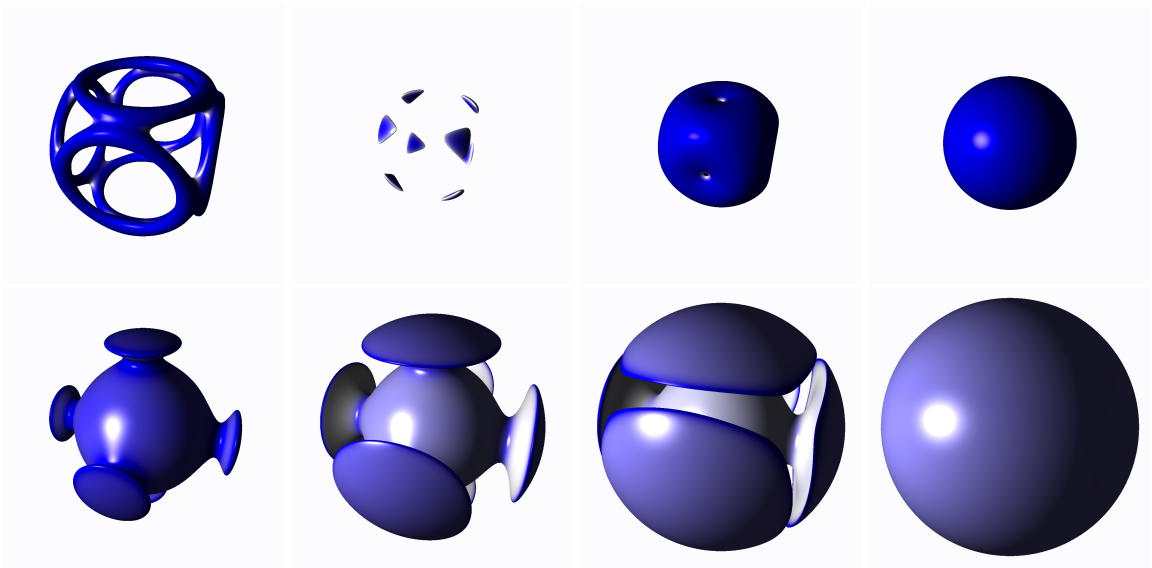


Figure 6.12: 4D morphing example. Morphing between a decocube and a sphere using a sigmoid interpolation function, running at $33-50$ fps.

6.6.7.3 Constructive solid geometry

Multiple-implicit CSG objects can accomplish similar effects to product surfaces and sigmoid blending, but with C^0 trimming. Unions and intersections of functions can be expressed natively using min and max operators, which are well-defined for both interval and affine forms. However, this inevitably requires evaluation of all sides of a compound expression. A more efficient approach employs 3-manifold level-sets, or inequality operations on CSG solids, as conditions over an implicit or set or implicits. This technique is frequently used in modeling F-Rep solid objects [92]. Given an implicit $f(\omega)$ and a condition $g(\omega)$, inclusion arithmetic allows us to verify $g_+ = \{\underline{g}(\omega) \geq 0\}$ or $g_- = \{\overline{g}(\omega) \leq 0\}$, given the interval form of the inclusion extension G over an interval domain $\omega \subseteq \Omega$. Then, one can render $f \cap g_+$ or $f \cap g_-$ for arbitrary level sets of g, as well as identify during traversal which surface is which. In the case of union, only the first condition need be evaluated if it contains a zero. Solid conditions are evaluated independently as boolean expressions; by determining which level sets are intersected inside the traversal, we can shade components differently as desired (Figure 6.13).

6.6.7.4 Procedural geometry

Implicits have historically been nonintuitive and unpopular for modeling large-scale objects. However, the ability to render dynamic surfaces and natural phenomena using

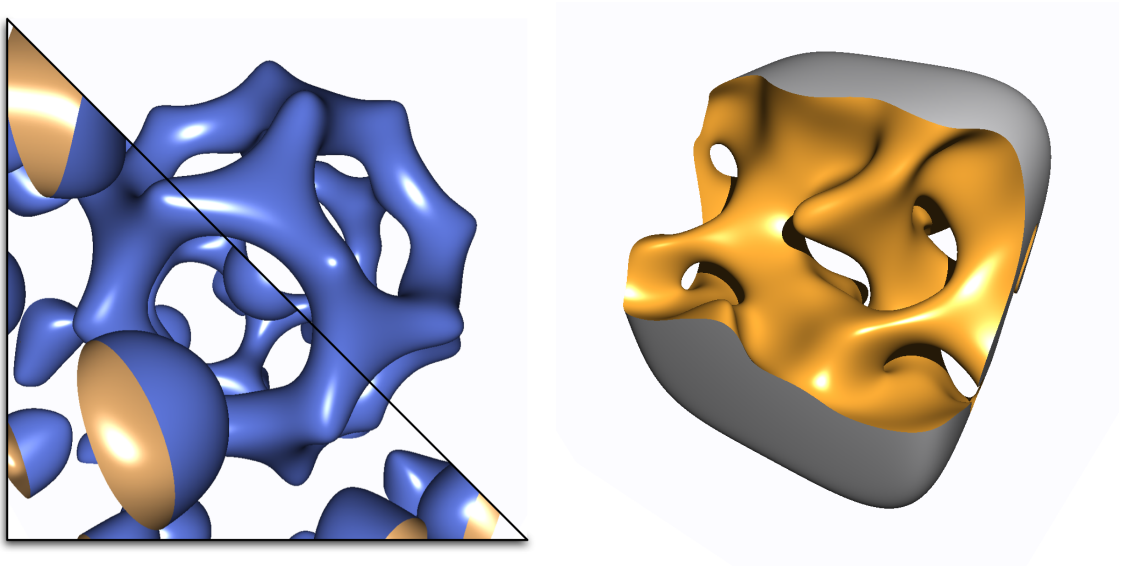


Figure 6.13: CSG using inequalities on 3-manifold solids.

combinations of known closed-form expressions could prove useful in modeling small-scale and dynamic features. Sinc expressions, for example, define closed-form solutions of simple wave equations for modeling water and cloth (Figure 6.14). Previous applications of implicit hypertextures focused on blended procedural noise functions [93, 28]. Recently, implicits based predominately on generalized sinusoid product forms similar to that in Figure 6.15 have been used within some modeling communities [50]. Arbitrary implicits are intriguing

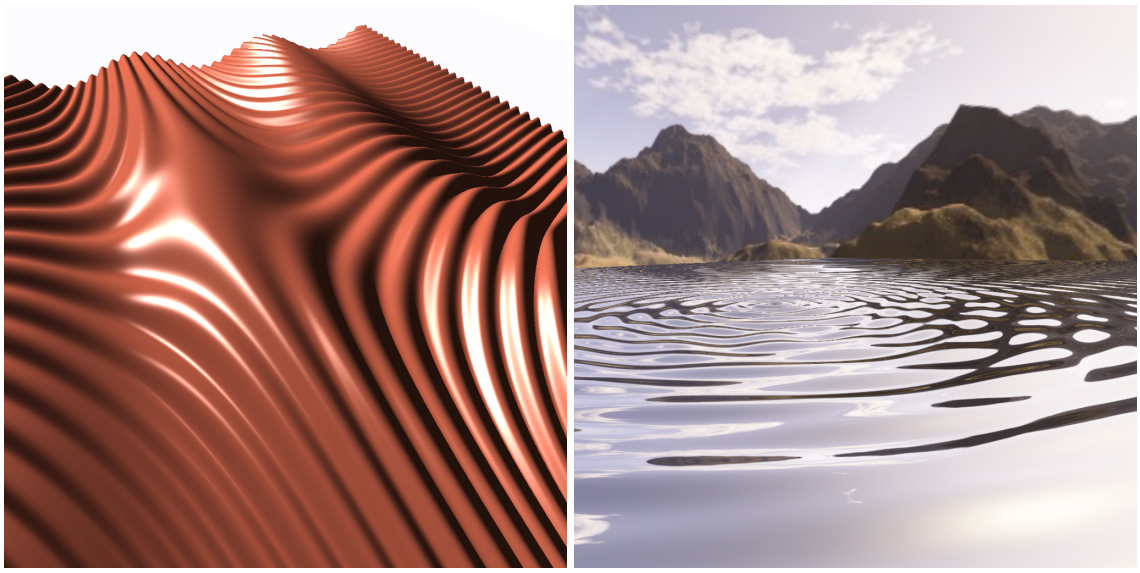


Figure 6.14: Sinusoid procedural geometry. With IA, these surfaces render at 38 and 37 fps, respectively.

in their flexibility, and ray tracing promises the ability to dynamically render entire new classes of procedural geometries, independently from any polygonal geometry budget.

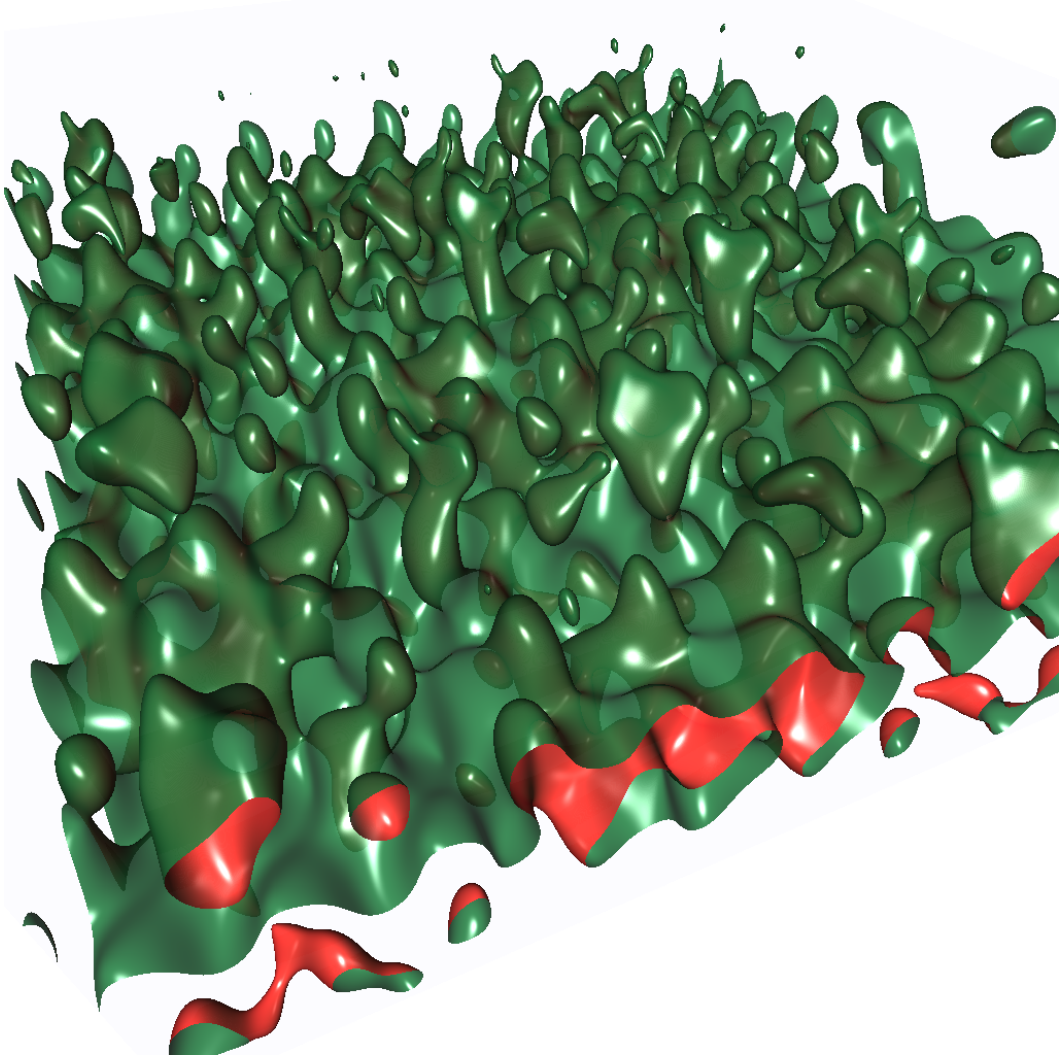


Figure 6.15: An animated sinusoid-kernel surface. Ray-traced directly on fragment units, no new geometry is introduced into the rasterization pipeline. IA/AA methods ensure robust rendering of any inclusion-computable implicit.

CHAPTER 7

CONCLUSION AND FUTURE WORK

We have contributed five implementations of implicit surface ray tracing with varying applications and hardware platforms. These systems show that implicit ray tracing, consisting of both fast spatial acceleration structure traversal and efficient ray-isosurface intersection, can be an effective algorithm for certain classes of problems in visualization. In particular, it is ideal for scalable rendering of large volume data, and pixel-exact intersection with implicit surfaces. In analyzing these systems, we consider the three broad problems addressed in this dissertation: isosurface rendering of large structured volume data, unstructured volume data, and general-form implicits.

7.1 Isosurface Ray Tracing of Octree Volumes

In Chapter 3, detailing work published from the RT06 Symposium [61], we showed that isosurface ray tracing can be employed in direct rendering of a compressed data structure for the purpose of large volume visualization. Our method allows for interactive exploration of large structured data on multicore computers using a fraction of the original memory footprint. Compressing volumes into octrees allows us to visualize data locally with the same quality as uncompressed arrays. While other spatial structures could deliver greater compression or faster traversal, the octree strikes a particularly good balance of these goals.

Our traversal is highly dependent on a fast octree hashing scheme. Our contributions in ray traversal and min/max tree construction are designed for this application alone; however, the point location and neighbor-finding implementations extend to general use of a binary hash tree. While benchmarking other applications of octree hashing falls outside the scope of this paper, our routines seem well-optimized for this application, and suggest general improvement over the code proposed by Frisken and Perry [26].

Octree ray tracing is not necessarily the ideal solution for general-purpose volume rendering. For smaller volume data with uniformly high isovalue variance, an octree can actually

occupy more space than a 3D array; moreover, the uniform grid and coherent kd-trees would likely outperform the octree for such scenes. However, in these cases a GPU volume renderer would generally be preferable to an interactive ray tracing solution. Thus, our method is primarily useful for large volumes, or medium volumes with numerous timesteps. Moreover, as large volumetric data sets are often the impetus for CPU ray tracing in the first place, this method is highly appropriate for its particular application.

Ultimately, this work illustrates that single-ray methods for large-volume isosurface rendering can perform roughly on par (albeit slightly slower) than packet methods, and that octree traversal is comparable in speed to grid and kd-tree traversal for this application. Moreover, techniques such as this continue to be attractive for rendering large structured data sets. Doubling each dimension of a 3D grid entails a factor of eight increase in memory footprint; this all but guarantees that main memory will continue to be a scarce resource in large volume rendering, and that logarithmic as opposed to object-order methods will prevail.

7.2 Coherent Multiresolution Isosurface Ray Tracing of Octree Volumes

In Chapter 4, drawing on work published in *The Visual Computer* [60], we employ coherent packet methods to accelerate the octree volume method of the previous chapter. Our approach was packet-based coherent ray tracing of large octree volume data using a multiresolution level of detail scheme to improve performance. Octree volume ray tracing allows for interactive exploration of large structured data on multicore computers using a fraction of the original memory footprint. While other spatial structures might deliver greater compression or faster traversal, the octree strikes a particularly good balance of these goals. With multiresolution and coherent traversal, we are able to trade quality for performance and render at interactive rates. Coherent traversal amortizes the cost of cell lookup, which allows for faster intersection and improved shading techniques.

One concern with the work in Chapter 4 compared to that in Chapter 3 is that LOD may not be an ideal solution for high-quality rendering, and ultimately performance gains from improved coherence may not justify the increase in code complexity and loss in visual quality. One of the major advantages of ray tracing, when compared to rasterization, is that performance depends logarithmically, not linearly, on geometric complexity. The single-ray tracer renders both simple and complex data at roughly equal, though slow, frame rates.

Coherent multiresolution essentially forfeits this advantage; it instead opts to improve best-case performance of simple scenes, while attempting to simplify complex scenes to mitigate worst-case performance. In a way, coherent ray tracing behaves similarly to rasterization in that its performance depends on LOD.

Nonetheless, for the purposes of large volume visualization, packet-based multiresolution isosurface ray tracing presents clear benefits. The main goal of our optimizations was to overcome limitations of single-ray octree volume ray tracing [61] and to ensure general interactivity. Overall, we accomplish that: our system is generally faster than single-ray noncoherent methods, allows for improved shading at reduced cost, and permits the user to trade visual quality for speed to better ensure interactivity. Moreover, as multicore CPUs increase in power and availability, techniques such as these become increasingly practical, and retain their scalability to large data and more cores in the long term.

Anecdotally, we have found this implementation to perform far better on recent multicore CPUs such as the Intel Clovertown and Penryn series, than on earlier architectures such as AMD Opteron and Intel Merom. We initially tested this system and its single-ray predecessor [61] on a NUMA Opteron workstation that cost nearly $50,000. The coherent technique performs almost equally well on an 8-core Clovertown that now costs less than $2,000; this is not the case for the single-ray technique. Performance is superior still on recent Intel Penryn chips, even at lower clock speed. This is likely due to improved SSE swizzling performance, but also the improved L2 cache model of recent CPUs.

As future work, the multiresolution octree could trivially be employed for out-of-core progressive rendering similar to Friedrich et al. [25], using the same compressed structure for LOD. Equally intriguing would be adapting the slice-caching reconstruction algorithm to perform direct volume rendering. Though computationally demanding, it could be implemented to take advantage of SIMD vector instructions [56], and would exhibit similar overall complexity to isosurfacing if the transfer function were sufficiently sparse. Also of interest would be employing generalized higher-order implicit surfaces [59] as intersection primitives, which could yield higher-quality reconstructions. Finally, generating coarser LODs with improved filtering, as well as smoothly blending between LOD levels as opposed to only interpolating at transitions, could improve visual quality.

7.3 Isosurface Ray Tracing of Tetrahedral Volume Data

In Chapter 5, previously published in TVCG [128], we have shown it is possible to ray trace isosurfaces of tetrahedral scalar fields at interactive to real-time frame rates, purely on the CPU. In doing so, we are able to correctly visualize large unstructured volumes, interactively manipulate isovalues and shader modalities, and handle time-varying data with hundreds of steps. The main algorithmic contributions of this method are the fast packet-isotetrahedron intersection test and extension of the coherent BVH to an implicit min-max tree over the tetrahedral volume. Our implementation naturally supports multiple isosurfaces, on-the-fly clipping, semitransparent depth peeling, and shadows. Accommodation of large data is limited only by host memory capacity, though the overhead of the BVH must be taken into consideration. Time-varying data can be handled by either precomputing an implicit BVH per time step, or by building a single BVH that is updated on the fly.

Compared to existing GPU methods, our system exhibits better scalability to large data, and is not limited by the GPU memory capacity. However, our current system is limited to isosurfacing, whereas existing GPU methods support direct volume rendering. Moreover, multicore CPUs are increasingly mainstream, and future GPUs could ultimately be capable of running a similar ray tracing system. Ultimately, the question is not one of GPU vs. CPU, but rather which rendering algorithm is used.

The BVH excels as a spatial subdivision structure for irregular data where objects overlap across axis-aligned boundaries. While it is difficult to directly compare geometric complexity of structured and unstructured datasets, the coherent BVH scales more gracefully to large data and scenes with poor primary-ray coherence than coherent grid traversal, or the packet octree traversal method of Chapter 4. While the grid and octree benefit greatly from multiresolution to reduce traversal-time complexity, the BVH seems less sensitive to pathological cases of incoherent viewing rays. This can likely be attributed to the unique first-active tracking approach of packet-BVH traversal [127]. Perhaps due to the regular nature of structured data and the relatively large number of primitives, coherent BVH traversal of an octree volume has not proven as effective. Nevertheless, the BVH could theoretically be used in constructing a min-max tree, alongside a compression/multiresolution scheme employing run-length encoding. Particularly for large volume data, this could prove competitive with kd-tree approaches [129]. In the case of unstructured data, it could also prove useful to employ compression techniques in reducing the footprint of the BVH and the tet mesh.

Since the publication of this work [128], Gross et al. [36] have demonstrated a highly efficient ray tracing system employing SSE traversal of 2x2 packets using skd-trees [43]. Although not as fast as our system at isosurfacing smaller datasets, it is nonetheless quite competitive, and capable of interactive volume rendering. Unlike the case of structured data, the lack of hardware support for fetching and interpolating from a tet mesh suggests that optimized CPU techniques and GPU ray casting methods will be competitive with GPU rasterization methods for rendering unstructured data.

The tet isosurfacing system opens several avenues for future work. We could extend BVH traversal to direct volume rendering methods, such as maximum intensity projection (MIP) or full transfer-function methods. Though the latter suffer from high traversal complexity, the BVH could still be useful for space-skipping when the transfer function is sufficiently sparse, as in [62]. Another intriguing extension would be support for higher-order finite elements in the spirit of Nelson & Kirby [84] or Rössl et al. [101]. This would require a completely different intersection routine, but the BVH traversal would remain unchanged. Also of interest would be more advanced lighting effects such as soft shadows, ambient occlusion, or global illumination, which can significantly improve understanding of data sets [34]. Finally, investigating scalable build algorithms could allow for rendering even complex data with arbitrary deformations without precomputation.

7.4 Ray Tracing Arbitrary Implicit Surfaces

In Chapter 6, covering work published in the RT07 Symposium [59] and *Computer Graphics Forum* [58], we have demonstrated a fast, robust and general algorithm for rendering implicits of arbitrary form, using interval and affine arithmetic. On both the CPU and GPU, the key to performance lies in optimization of the interval bisection algorithm. Coherent traversal using SIMD vector instructions; and a stackless fractional modulus traversal algorithm, aid the efficiency of CPU and GPU algorithms, respectively. These implementations perform up to four orders of magnitude faster than the previously published state-of-the-art interval arithmetic ray tracing method [24]. This performance is due to the SIMD-parallel efficiency of their respective platforms, and the reality that high rendering quality can be achieved with relatively coarse numerical precision, corresponding to fewer iterations of the interval bisection algorithm.

The GPU algorithm is arguably the state-of-the-art solution for guaranteed-robust ren-

dering of arbitrary implicit forms. The SSE CPU technique remains interesting in that the K-marching approach outperforms direct ray bisection. Similar algorithms, designed to leverage explicit coherence of SIMD vector instructions, could prove useful in future hardware such as Intel Larrabee [112]. The main advantage of bisection with interval and affine arithmetic methods, as employed in both methods, is the general guarantee of correctness in rendering a wide variety of implicit forms.

Following publication of the works in Chapter 6, two noteworthy methods for rendering arbitrary implicit surfaces on the GPU have been published. The blossoming approach of Seland and Reimers [98] is a hybrid GPU-CPU method based on refactoring arbitrary implicit functions into a Bernstein polynomials, and efficiently solving their roots using the De Casteljau algorithm. Given similar ε, its results appear comparably robust to our IA/AA method. However it is more complicated to implement and around 2x-4x slower. The other approach, by Singh and Narayanan [114], is an adaptive point-sampling method similar to Hanrahan [39] but using a two-level root-isolation and bracketing strategy, and implemented on an NVIDIA G80 GPU. While not robust for rendering singularities such as the teardrop, it is extremely fast, typically 2x–5x faster than our IA/AA method on the 8800GTX, and in practice it renders most implicit surfaces accurately. This latter method is intriguing, as it shows that brute-force linear methods can outperform even our bisection method on the GPU, which in turn outperforms most higher-order convergent methods. Nonetheless, this approach is only competitive on GPU architectures, where computational throughput is sufficient to overcome poor time complexity. Ultimately, we believe that the IA/AA method will prove a good balance between algorithmic simplicity, adaptivity and robustness, as CPU hardware becomes more parallel and as GPU hardware improves its handling of branching code.

Our interval and affine arithmetic algorithms for ray tracing arbitrary implicits would benefit from several extensions. Further development of approximating regression operations for RAA could allow for correct and fast affine extensions of transcendental functions and their compositions. Also, while robust per-ray, our system ignores aliasing issues on boundaries and subpixel features. To robustly reconstruct the surface between pixels would require supersampling and ideally beam tracing. More generally, the application front-end could be extended to support point, mesh or volume data, which could then be reconstructed by arbitrary implicit filters. Scalable rendering of complex objects featuring multiple piecewise

implicits with CSG operators could prove desirable. This could be accomplished either by extending the rasterization system and restricting the application to ray casting, or by attempting a full ray-tracing system, with an acceleration hierarchy, on the fragment shader or in CUDA. Finally, though applied here to general implicits, inclusion methods could potentially be employed in rendering arbitrary parametric or free-form surfaces.

7.5 Conclusion

Efficient rendering of implicit surfaces is a broad endeavor with numerous and often competing constraints. The systems we have presented here each benefit from different strategies in maximizing performance towards their respective applications and architectures. It is remarkable how even subtle changes in platform mandate wholesale changes in implementation. This is particularly the case in the octree volume isosurfacing systems in Chapters 3 and 4, and the arbitrary-form implicit renderers in Chapter 6.

Over the course of the research contained in this dissertation, techniques for interactive ray tracing on both CPU and GPU have improved significantly. Isosurface ray tracing of large structured data, which required a small supercomputer or cluster less than a decade ago, now operates efficiently on a multicore desktop. Octrees have not been popular structures for ray tracing for over a decade, yet they proved efficient in our work on CPUs, and more recently are experiencing resurgence as efficiency structures for ray tracing on the GPU. Ray tracing with bounding volume hierarchies was not popular until recently, but is now considered the state-of-the-art acceleration structure on both CPU and GPU. Our tetrahedral isosurface renderer marks the first application of the coherent BVH towards visualization. Similarly, interval and affine arithmetic have traditionally been considered accurate but slow methods of graphing implicit functions as surfaces in 3D. Our CPU and GPU algorithms employing IA and reduced AA are between three and four orders of magnitude faster than the best previously published implementations.

Moving forward, perhaps the biggest challenge lies in developing scalable isosurface rendering algorithms that perform well on both CPU (or Larrabee [112]) and GPU hardware. The choice of heavily threaded, explicit SIMD vector instructions of the CPU, or lightly threaded instruction parallelism with implicit SIMD on the GPU, dictates which algorithms work well on each platform. The costs of irregular memory accesses, so prevalent in ray tracing, also have a significant impact on performance. Currently, packet-based BVH and

kd-tree traversals, and variants, are the fastest structures for CPU ray tracing. Conversely, on the GPU, grids and increasingly full octrees seem to perform significantly better than the competition. While this does not necessarily impact the ability to perform ray intersection with implicit surfaces, it does change strategies for compression, traversal, and reconstruction of local support.

Extrapolating from current hardware trends, we believe that implicit surface ray tracing will continue to play a role in visualization and graphics in general. The lessons drawn from this work could have implications towards future rendering techniques: as computation becomes less of a bottleneck than memory access, rendering methods that employ multiresolution and compression will prove useful. In rendering higher-order geometry, it will be interesting to see whether brute-force or more sophisticated numerical methods will prevail. Similarly, rendering techniques currently stand at an interesting threshold where changes in geometric complexity tip the balance towards favoring rasterization or ray tracing methods. Ultimately, in visualization, we believe that irrespective of hardware platform, surface-based ray tracing methods hold promise for scalable rendering of massive data, and implicit surfaces allow for fast surface reconstruction from arbitrary filters in a wide range of scientific datasets.

APPENDIX A

OCTREE VOLUME HASHING AND TRAVERSAL

This appendix details structures and algorithms used in Chapters 3 and 4 of the dissertation.

A.1 Octree Volume Structure

An octree volume consists of the following structure: interior nodes are stored in an array indexed by depth, from root depth 0 to depth $d_{max} - 2$. "Cap" nodes exist at $d_{max} - 1$, and always contain scalars of the maximum depth. "Scalar leaves" are represented by child_scalars in the interior node structure.

```
struct OctreeData
{
  int max_depth;
  OctNode* nodes[max_depth];
  OctCap* caps;
  int child_bit_depth[max_depth];
};

struct OctNode
{
  T child_scalars[8];   //scalar leaves
  T child_mins[8];      //min/max tree
  T child_maxs[8];
  unsigned int child_start;  //base pointer to children
  char child_offset[8];      //offset from base
};

struct OctCap
{
  T child_scalars[8];
};
```

A.1.1 Cached Hash Array

When the max_depth=d_{max} of the octree is given, we compute an array that is subsequently used in hashing and ray traversal:

```
for(int d=0; d < max_depth; d++)
  child_bit_depth[d] = 1 << (max_depth - d - 1);
```

A.2 Octree Hashing

Our octree hash scheme consists of accelerated routines for point location and neighbor finding in canonical octree coordinates, $[0, d_{max}]$. While binary arithmetic on integers is not a new hashing scheme [32, 26], we propose caching the depth masks to avoid costly arbitrary left shifts, and then shifting by constants. The following functions may be considered as members of OctreeData in Appendix A.

A.2.1 Point Location

Point location algorithm. We use the precomputed array, child_bit_depth[] (Appendix A.1), to avoid arbitrary left-shift operations.

```
T point_locate(Vec3i dest, int depth, int index)
{
  for(;;)
  {
    OctNode& node = nodes[depth][index];
    int child_bit = child_bit_depth[depth];
    int child = (dest.x & child_bit!=0) << 2
              || (dest.y & child_bit!=0) << 1
              || (dest.z & child_bit!=0);

    if (node.child_offset[child] == -1)
    {
      return node.child_scalars[child];
    }
    else if (depth == max_depth - 2)
    {
      index = node.child_start + node.child_offset[child];
      child = (dest.x & 1)<<2 |
              (dest.y & 1)<<1 |
              (dest.z & 1);
      return caps[index].child_scalars[child];
    }
    index = node.child_start + node.child_offset[child];
    depth++;
  }
  return 0;
}
```

A.2.2 Neighbor-Finding

Neighbor finding algorithm. The parent_trace array contains pointers to nodes, so we only need store 1-way pointers in the octree. As a simple optimization, one could use static polymorphism via templates for each X,Y,Z dimension, and thus only perform one or two equality tests per iteration to neighbors along designated axes.

```
T neighbor_find(Vec3i start, Vec3i dest, int depth,
                int parent_trace[])
{
  for(int up=depth; up >= 0; up--)
  {
    int child_bit = child_bit_depth[up];
    if ((dest.x & child_bit) == (start.x & child_bit)
         && (dest.y & child_bit) == (start.y & child_bit)
         && (dest.z & child_bit) == (start.z & child_bit)
      return point_locate(dest, up, parent_trace[up]);
  }
  //root node
  if ((dest.x & child_bit) == (start.x & child_bit)
       && (dest.y & child_bit) == (start.y & child_bit)
       && (dest.z & child_bit) == (start.z & child_bit)
    return point_locate(dest, 0, 0);
  return 0;
}
```

A.3 Single-Ray Octree Traversal

Pseudocode for a ray traversal through an interior node of an octree volume. For brevity, some operations are omitted; those are bracketed with a brief description. Traversals of scalar leaves and cap nodes operate similarly.

```
bool traverse(Ray ray,
     int depth, uint node_index,
     int parent_trace[], Vec3f cell,
     float tenter, float texit)
{
  OctNode& node = nodes[depth][node_index];
  parent_trace[depth] = node_index;
  int child_bit = child_bit_depth[depth];
  Vec3f center = Vec3f( cell | Vec3i(child_bit) );
  Vec3f tcenter = (center  ray.orig) / ray.dir;
  Vec3f penter = ray.orig + ray.dir * tenter;
  Vec3i child_cell = cell;
  Vec3i tc;
  tc.x = (penter.x >= center.x);
  tc.y = (penter.y >= center.y);
  tc.z = (penter.z >= center.z);

  int child = tc.x << 2 | tc.y << 1 | tc.z;
  child_cell.x |= tc.x ? child_bit : 0;
```

```
  child_cell.y |= tc.y ? child_bit : 0;
  child_cell.z |= tc.z ? child_bit : 0;

  Vec3i axis_isects;
  {perform 3-way minimum of tcenter such that axis_isects
  contains the sorted intersection with the X,Y,Z
  octant mid-planes}

  const int axis_table[] = {4,2,1};

  float child_tenter = tenter;
  float child_texit;
  for( {all valid axis_isects[i] while tcenter < texit} ; i++)
  {
    child_texit = min(tcenter[axis_isects[i]], texit);
    if (isovalue >= node.child_mins[child] ||
        isovalue <= node.child_maxs[child]){
       //traverse scalar leaf, cap or node
       if (node.child_offset == -1)
         if (traverse_scalar_leaf(...)) return true;
       else if (depth == max_depth -- 2)
         if (traverse_cap(...)) return true;
       else
         if (traverse(ray,depth+1,parent_trace,
            child_cell, child_tenter, child_texit))
            return true;
    }
    if (child_texit == texit)
       return false;
    child_tenter = child_texit;
    axisbit = axis_table[axis_isects[i]];
    if (child & axisbit){
       child &= ~axisbit;
       child_cell[axis_isects[i]] &= ~child_bit;
    }
    else{
       child |= axisbit;
       child_cell[axis_isects[i]] |= child_bit;
    }
  }
  return false;
}
```

A.4 Packet-Octree Traversal

Algorithm 1 Octree CGT algorithm

Require: *axes* $\vec{K},\vec{U},\vec{V}$; *packet P*; *octree volume OV*; *isovalue*
Ensure: *compute P intersection with OV*
  for all *depths* $i \in \{0..d_{max}\}$ **do**
    $d_{uv}[i] \Leftarrow [du_{min}, dv_{min}, du_{max}, dv_{max}]$ / $2^{d_{max}-i}$
    $k_0[i] \Leftarrow (P\ enters\ OV)_{\vec{K}}$ / $2^{d_{max}-i}$
    $k_1[i] \Leftarrow (P\ exits\ OV)_{\vec{K}}$ / $2^{d_{max}-i}$
    $e_{uv}[i] \Leftarrow [u_{min}, v_{min}, u_{max}, v_{max}]$ *at* $k_0[i], k_1[i]$
    $k[i] \Leftarrow k_0[i]$
    $k_{nextMC}[i] \Leftarrow k[i]+2$
  end for
  $d \Leftarrow 0$
  while $k[d] \leq k_1[d]$ **do**
    if $k[d] = k_{nextMC}[d]$ **then**
      $d \Leftarrow d-1$
      continue
    end if
    $traverseChild \Leftarrow false$;
    for all $u \in [u_{min}, u_{max}], v \in [v_{min}, v_{max}]$ of e_{uv} **do**
      $node \Leftarrow OV.lookup(vec3(k,u,v),d)$
      if $isovalue \in [node.min, node.max]$ **then**
        $traverseChild \Leftarrow true$
        break
      end if
    end for
    if $d = d_{cap}$ **then**
      clip e_{uv} to non-empty cap-level macrocells
    end if
    if $traverseChild = true$ **then**
      if $d = d_{max}$ **then**
        clip cell slice e_{uv} to active rays
        intersect P with slice $k[d_{cap}]$ at $e_{uv}[d_{cap}]$
        if all rays in P hit **then**
          return
        end if
      else
        $e_{uv}[d] \Leftarrow e_{uv}[d] + d_{uv}[d]$
        $k_{new}[d+1] \Leftarrow 2 * k[d]$
        $k[d+1] \Leftarrow k_{new}[d+1]$
        $k_{nextMC}[d+1] \Leftarrow k[d+1]+2$
        $d \Leftarrow d+1$
        continue
      end if
    end if
    $e_{uv}[d_{cap}] \Leftarrow e_{uv}[d_{cap}] + d_{uv}[d_{cap}]$
  end while

A.5 Transition Array Computation

This precomputation is executed per-frame, as described in Section 4.4 of Chapter 4.

Algorithm 2 Transition Array Computation

Require: Pixel-width to voxel-width ratio, dP/dV
Per-ray camera offset along $\vec{U}$ axis, du_{camera}
Ensure: Array of $\vec{K}$-transition slices, $k_{transition}[]$
for all octree depths $d \in \{0..d_{max}-1\}$ **do**
$voxelWidth[d] \Leftarrow 2^{d_{max}-d} * dP/dV$
$t_{transition}[d] \Leftarrow voxelWidth[d]\ /\ du_{camera}$
$k_{transition}[d] \Leftarrow k_{origin} + t_{transition} k_{direction}$
end for

APPENDIX B

ARBITRARY-FORM IMPLICITS

This Appendix contains pseudocode and results from Chapter 6. Section B.1 provides code for the SIMD interval arithmetic method for the CPU using SSE; and Section B.2 has code for the GPU method.

B.1 SSE IA Pseudocode

Algorithm 3 Examples of SIMD Interval Arithmetic

```
struct interval4 {
  __m128 lo, hi;
};
interval4 add_i4(interval4 a, interval4 b) {
  return interval4( _mm_add_ps(a.lo, b.lo), _mm_add_ps(a.hi, b.hi) );
}
interval4 mul_i4(interval4 a, interval4 b) {
  __m128 lolo = _mm_mul_ps(a.lo, b.lo);
  __m128 lohi = _mm_mul_ps(a.lo, b.hi);
  __m128 hilo = _mm_mul_ps(a.hi, b.lo);
  __m128 hihi = _mm_mul_ps(a.hi, b.hi);
  return interval4( _mm_min_ps(lolo, _mm_min_ps(lohi, _mm_min_ps(hilo, hihi))),
                    _mm_min_ps(lolo, _mm_min_ps(lohi, _mm_min_ps(hilo, hihi))) );
}
interval4 abs_i4(interval4 a) {
  return interval4( _mm_max_ps(a.lo, _mm_max_ps(-a.hi), 0)), _mm_max_ps(-a.lo, a.hi) );
}
interval4 sqr_i4(interval4 a) {
  interval4 aa = abs_i4(a);
  return interval4( _mm_mul_ps(a.lo, a.lo), _mm_mul_ps(a.hi, a.hi) );
}
interval4 circle(interval4 x, interval4 y, interval4 z, float radius) {
  return sub_i4(add_i4(sqr_i4(x), add_i4(sqr_i4(y), sqr_i4(z))),
                radius*radius);
}
```

Algorithm 4 Ray-Implicit Traversal pseudocode.

```
template<int K, int U, int V, int DK>
void traverse(RayPacket r, Box domain,
              Implicit implicit, int d_stop) {
  (get t_enter, t_exit, t_kenter, t_kexit)
  __m128 validmask = intersectBB(r, domain);
  //validmask indicates rays that are active
  float full_tk = t_kexit - t_kenter;
  float full_u = _mm_mul_ps(r.dir[U], full_tk);
  float full_v = _mm_mul_ps(r.dir[V], full_tk);
  struct Stack {
    __m128 t_incr;
    __m128 u_incr, v_incr;
    float k_incr;
    char side;
  };
  Stack stk[maxDepth];
  for(int d=0;d<maxDepth;d++){
    float width = 1.f / (float)(1<<d);
    stk[d].t_incr = _mm_mul_ps(full_tk, width);
    stk[d].u_incr = _mm_mul_ps(full_u, width);
    stk[d].v_incr = _mm_mul_ps(full_v, width);
    stk[d].side = -1;
  }
  int depth = 0;
  float curr_k = DK==+1 ? domain.min[K]:domain.max[k];
  __m128 curr_t, curr_u, curr_v;
  curr_t = t_kenter;
  curr_u = _mm_add_ps(r.org[U],_mm_mul_ps(r.dir[U],curr_t));
  curr_v = _mm_add_ps(r.org[V],_mm_mul_ps(r.dir[V],curr_t));
  __m128 next_t, next_u, next_v;

  for(;;) {
    stk[depth].side++;
    next_k = DK==+1 ?
             curr_k + stk[depth].k_incr :
             curr_k - stk[depth].k_incr;
    next_u = _mm_add_ps(curr_u, stk[depth].u_incr);
    next_v = _mm_add_ps(curr_v, stk[depth].v_incr);
    next_t = _mm_add_ps(curr_t, stk[depth].t_incr);
    hitmask = and4(validmask, cmp_ge4(next_t, tenter));
    if (any4(hitmask)) {
      interval4 ibox;
      (fill ibox with curr and next k,u,v)
      interval4 F = implicit.evalute_interval4(ibox);
      if (any4(F.contains(0))) {
        if (!all4(cmp_ge4(sub4(F.hi,F.lo),INFINITY))){
          if (depth == maxDepth-1){
            //hit
            hit(r, curr_t);
            (compute normal);
            if (all4(r.hitmask))
              return;
          } else {
            //recurse
            depth++;
            continue;
          }
        }
      }
    }
    validmask = and4(validmask, cmp_le4(next_t, texit));
    if (none4(validmask))
      return;
    curr_k = next_k;
    curr_t = next_t;
    curr_u = next_u;
    curr_v = next_v;
    if (stk[depth].side & 1)
    {
      do{
        if (--depth == -1)
          return; }
      while(stk[depth] & 1);
      continue;
    }
  }
}
```

B.2 GPU IA/AA Pseudocode

Algorithm 5 Excerpt of GPU Interval Arithmetic.

```
typedef float2 interval;

interval iadd(interval a, interval b) {
  return interval( add(a.x, b.x), add(a.y, b.y) );
}
interval imul(interval a, interval b) {
  float4 lh = a.xxyy * b.xyxy;
  return interval(min(lh.x, min(lh.y, min(lh.z, lh.w))),
                  max(lh.x, max(lh.y, max(lh.z, lh.w))));
}
interval ircp(const float inf, interval i) {
  return ( (i.x <= 0 && i.y >= 0) ?
           interval(-inf, inf) : 1/i.yx );
}
```

Algorithm 6 Excerpt of GPU Reduced Affine Arithmetic.

```
typedef float3 raf;

raf interval_to_raf(interval i){
  raf r;
  r.x = (i.y + i.x);
  r.y = (i.y - i.x);
  r.xy *= .5; r.z = 0;
  return r;
}
float raf_radius(raf a){
  return abs(a.y) + a.z;
}
interval raf_to_interval(raf a){
  const float rad = raf_radius(a);
  return interval(a.x - rad, a.x + rad);
}
raf raf_add(raf a, raf b){
  return (a + b);
}
raf raf_mul{raf a, raf b){
  raf r;
  r.x = a.x * b.x;
  r.y = a.x*b.y + b.x*a.y;
  r.z = abs(a.x*b.z) + abs(b.x*a.z) +
        raf_radius(a)*raf_radius(b);
  return r;
}
```

Algorithm 7 Traversal algorithm with RAA.

```
float traverse(float3 penter, float3 pexit, float w,
   float max_depth, float eps, float nan, float inf){
  const float3 org = penter;
  const float3 dir = pexit-penter;
  interval t(0,1);
  raf F, it, ix, iy, iz;
  //rejection test
  ix = raf_add(org.x, raf_mul(it, dir.x));
  iy = raf_add(org.y, raf_mul(it, dir.y));
  iz = raf_add(org.z, raf_mul(it, dir.z));
  F = evaluate_raf(ix, iy, iz, w, nan, inf);
  if (raf_contains(F, 0)){
    int d=0;
    float tincr = .5;
    const int dlast = log2(length(dir)/epsilon);
    //main loop
    for(;;){
      t.y = t.x + tincr;
      (compute ix, iy, iz, F again for rejection test)
      if (raf_contains(F, 0)){
        if (d==dlast){ return t.x; /*hit*/}
        else{ tincr *= .5; d++; continue; }
      }
      t.x = t.y;
      //back-recursion
      float fp = frac(.5*t.x/tincr);
      if (fp < 1e-8){
        for(int j=0; j<=dlast; j++){
          tincr *= 2;
          d--;
          fp = frac(.5*t.x/tincr);
          if (d==-1 || fp > 1e-8) break;
        }
        if (d==-1) break;
      }
    }
  }
  return -1;  //no hit
}
```

B.3 Reference Implicits

Table B.1: Formulas of simple test surfaces used in Table 6.1.

sphere	$x^2+y^2+z^2-r^2$
steiner	$x^2y^2+y^2z^2+x^2z^2+xyz$
mitchell	$4(x^4+(y^2+z^2)^2)+17(x^2(y^2+z^2)-20(x^2+y^2+z^2)+17$
teardrop	$\frac{x^5+x^4}{2}-y^2-z^2$
4-bretzel	$\frac{1}{10}(x^2(1.21-x^2)^2(3.8-x^2)^3-10y^2)^2+60z^2-2$
klein bottle	$(x^2+y^2+z^2+2y-1)((x^2+y^2+z^2-2y-1)^2-8z^2)$ $+16xz(x^2+y^2+z^2-2y-1)$
tangle	$x^4-rx^2+y^4-5y^2+z^4-5z^2+11.8$
decocube	$((x^2+y^2-0.8^2)^2+(z^2-1)^2)((y^2+z^2-0.8^2)^2+$ $(x^2-1)^2)((z^2+x^2-0.8^2)^2+(y^2-1)^2)-0.02$
barth sextic	$4(\tau^2x^2-y^2)(\tau^2y^2-z^2)(\tau^2z^2-x^2)-(1+2\tau)(x^2+y^2+z^2-1)^2$ where τ is the golden ratio, $\frac{1+\sqrt{5}}{2}$
barth decic	$8(x^2-\tau^4y^2)(y^2-\tau^4z^2)(z^2-\tau^4x^2)(x^4+y^4+z^4-2x^2y^2-2x^2z^2-2y^2z^2)+$ $(3+5\tau)(x^2+y^2+z^2-w^2)^2(x^2+y^2+z^2-(2-\tau)w^2)^2w^2$, $\tau=\frac{1+\sqrt{5}}{2}$
superquadric	$x^{500}+\frac{1}{2}\lvert y\rvert^{35}+\frac{1}{2}z^4-1$

Table B.2: Formulas of more complicated implicit surfaces used in Table 6.1.

icos.csg	$ic(x,y,z) = 2 - (\cos(x+\tau y) + \cos(x-\tau y) + \cos(y+\tau z) + \cos(y-\tau z) + \cos(z-\tau x) + \cos(z+\tau x))\ , \qquad \tau = \frac{1+\sqrt{5}}{2}$ CSG condition (on inclusion intervals): $(0 \in ic)$ and $sphere_{inner} < 0$ and $sphere_{outer} > 0$
sesc.csg	CSG of superellipsoid (se) and sinusoid convolution (sc) $se(x,y,z) = x^6 + \frac{1}{2}(y^4+z^4)^4 - 20$ $sc(x,y,z) = xy + \cos(z) + 1.741\sin(2x)\sin(z)\cos(y) + \sin(2y)\sin(x)\cos(z) + \sin(2z)\sin(y)\cos(x) - \cos(2x)\cos(2y) + \cos(2y)\cos(2z) + \cos(2z)\cos(2x) + 0.05$ CSG condition (on inclusion intervals): $((sc > 0)$ and $(0 \in se))$ or $((se < 0)$ and $(0 \in sc))$
sin.blob	$1 + r_1(y+w) + \cos(r_2 z) + 4(\sin(4r_0 r_2 x)\sin(r_0 z)\cos(r_1 y) + \sin(2r_0 r_1 y)\sin(r_2 x)\cos(r_2 z) + \sin(2r_2 z)\sin(r_1 y)\cos(r_2 x)) - (\cos(2r_2 x)\cos(2r_0 y) + \cos(2r_0 r_1 y)\cos(2r_2 z) + \cos(2r_2 z)\cos(2r_2 x)))$ where $r_0 = 0.104,\ r_1 = 0.402,\ r_2 = -0.347$
cloth	$y - 0.5\sin(x+3w) - 0.1(1+.1\sin(xz))\cos(z+3w)$ where $w = [0, 2\pi]$ is a time-dependent variable
water	$\sin(\sqrt{((x+r_1)^2+z^2)} - w)/(10+\sqrt{((x+r_1)^2+z^2)}) + \sin(\sqrt{((x-r_1)^2+z^2)} - w + r_0)/(10+\sqrt{((x-r_1)^2+z^2)}) + \sin(2\sqrt{((z-r_1)^2+x^2)} - w - r_0)/(10+\sqrt{((z-r_1)^2+x^2)}) - \frac{y}{2}$ where $r_0 = 2.736,\ r_1 = 15,\ r_2 = -.830746$ and w is time-dependent.

REFERENCES

[1] AMANATIDES, J., AND WOO, A. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*. Eurographics Association, 1987, pp. 3–10.

[2] APPEL, A. Some Techniques for Shading Machine Renderings of Solids. *SJCC* (1968), 27–45.

[3] BERNARDON, F. F., CALLAHAN, S. P., COMBA, J. L. D., AND SILVA, C. T. An Adaptive Framework for Visualizing Unstructured Grids with Time-Varying Scalar Fields. *Parallel Computing* (2007).

[4] BIGLER, J., STEPHENS, A., AND PARKER, S. G. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 187–196.

[5] BLINN, J. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics 1*, 3 (July 1982), 235–256.

[6] BLOOMENTHAL, J. An Implicit Surface Polygonizer. *Graphics gems IV* (1994), 324–349.

[7] BOULOS, S., WALD, I., AND SHIRLEY, P. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Tech. Rep. UUCS-06-010, SCI Institute, University of Utah, 2006.

[8] BRONNIMANN, H., AND GLISSE, M. Cost-Optimal Trees for Ray Shooting. In *Proceedings of the Latin American Symposium on Theoretical Informatics* (2004).

[9] BUNYK, P., KAUFMAN, A., AND SILVA, C. Simple, Fast, and Robust Ray Casting of Irregular Grids. In *Proceedings of Scientific Visualization, Dagstuhl, Germany* (1997), pp. 30–36.

[10] CALLAHAN, S. P., BAVOIL, L., PASCUCCI, V., AND SILVA, C. T. Progressive Volume Rendering of Large Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006) 12*, 5 (Sept/Oct 2006), 1307–1314.

[11] CALLAHAN, S. P., COMBA, J. L. D., SHIRLEY, P., AND SILVA, C. T. Interactive Rendering of Large Unstructured Grids Using Dynamic Level-of-Detail. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2005) 12*, 5 (Sept/Oct 2005), 199–206.

[12] CAPRIANI, O., HVIDEGAARD, L., MORTENSEN, M., AND SCHNEIDER, T. Robust and Efficient Ray Intersection of Implicit Surfaces. *Reliable Computing 6* (2000), 9–21.

[13] CASTANIE, L., MION, C., CAVIN, X., AND LEVY, B. Distributed Shared Memory for Roaming Large Volumes. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1299–1306.

[14] CO, C. S., HAMANN, B., AND JOY, K. I. Iso-splatting: A Point-based Alternative to Isosurface Visualization. In *Proceedings of Pacific Graphics 2003* (Oct. 8–10 2003), J. Rokne, W. Wang, and R. Klein, Eds., pp. 325–334.

[15] COMBA, J. L. D., AND STOLFI, J. Affine Arithmetic and its Applications to Computer Graphics. In *Proceedings of SIBGRAPI* (1993), pp. 9–18.

[16] DE CUSATIS JUNIOR, A., DE FIGUEIREDO, L., AND GATTAS, M. Interval Methods for Raycasting Implicit Surfaces with Affine Arithmetic. In *Proceedings of XII SIBGRPHI* (1999), pp. 1–7.

[17] DE FIGUEIREDO, L. H., AND STOLFI, J. Adaptive Enumeration of Implicit Surfaces with Affine Arithmetic. *Computer Graphics Forum 15*, 5 (1996), 287–296.

[18] DE TOLEDO, R., LEVY, B., AND PAUL, J.-C. Iterative Methods for Visualization of Implicit Surfaces on GPU. In *ISVC, International Symposium on Visual Computing* (Lake Tahoe, Nevada/California, November 2007), Lecture Notes in Computer Science, SBC - Sociedade Brasileira de Computacao, Springer.

[19] DEMARLE, D., PARKER, S., HARTNER, M., GRIBBLE, C., AND HANSEN, C. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE PVG* (2003), pp. 87–94.

[20] DEMARLE, D. E., GRIBBLE, C., AND PARKER, S. Memory-Savvy Distributed Interactive Ray Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2004).

[21] DJEU, P., HUNT, W., WANG, R., ELHASSAN, I., STOLL, G., AND MARK, W. R. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. rep., The University of Texas at Austin, 2007. (Cond. accepted to ACM Transactions on Graphics).

[22] DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[23] DOI, A., AND KOIDE, A. An Efficient Method of Triangulating Equi-valued Surfaces by Using Tetrahedral Cells. *IEICE Trans Commun. Elec. Inf. Syst E-74*, 1 (1991), 213–224.

[24] FLOREZ, J., SBERT, M., SAINZ, M., AND VEHI, J. Improving the Interval Ray Tracing of Implicit Surfaces. In *Lecture Notes in Computer Science* (2006), vol. 4035, pp. 655–664.

[25] FRIEDRICH, H., WALD, I., AND SLUSALLEK, P. Interactive Iso-Surface Ray Tracing of Massive Volumetric Data Sets. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization* (2007), Eurographics.

[26] FRISKEN, S. F., AND PERRY, R. N. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools 7*, 3 (2002).

[27] FRYAZINOV, O., AND PASKO, A. GPU-based Real Time FRep Ray Casting. *Proceedings of Graphicon* (2007), 69–74.

[28] GAMITO, M. N., AND MADDOCK, S. C. Ray Casting Implicit Fractal Surfaces with Reduced Affine Arithmetic. *Vis. Comput. 23*, 3 (2007), 155–165.

[29] GARGANTINI, I., AND ATKINSON, H. Ray Tracing an Octree: Numerical Evaluation of the First Interaction . *Computer Graphics Forum 12*, 4 (1993), 199–210.

[30] GARRITY, M. P. Raytracing Irregular Volume Data. *Computer Graphics (San Diego Workshop on Volume Visualization) 24*, 5 (November 1990), 35–40.

[31] GEORGII, J., AND WESTERMANN, R. A Generic and Scalable Pipeline for GPU Tetrahedral Grid Rendering. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1345–1352.

[32] GLASSNER, A. S. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications 4*, 10 (1984), 15–22.

[33] GOLDSMITH, J., AND SALMON, J. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

[34] GRIBBLE, C. *Interactive Methods for Effective Particle Visualization*. PhD thesis, University of Utah, 2006.

[35] GRIBBLE, C. P., IZE, T., KENSLER, A., WALD, I., AND PARKER, S. G. A Coherent Grid Traversal Approach to Visualizing Particle-based Visualization Data. Tech. Rep. UUSCI-2006-024, SCI Institute, University of Utah, 2006. (submitted for publication).

[36] GROSS, M., HAGEN, H., PFREUND, F., FRAUNHOFER, I., AND IRTG, G. Interactive SIMD Ray Tracing for Large Deformable Tetrahedral Meshes. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), pp. 147–154.

[37] GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. Interactive Rendering of Large Volume Data Sets. In *Proceedings of the conference on Visualization '02* (2002), IEEE Computer Society, pp. 53–60.

[38] HADWIGER, M., SIGG, C., SCHARSACH, H., BÜHLER, K., AND GROSS, M. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312.

[39] HANRAHAN, P. Ray tracing algebraic surfaces. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1983), ACM Press, pp. 83–90.

[40] HART, J. C. Sphere tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. *The Visual Computer 12*, 10 (1996), 527–545.

[41] HAVRAN, V. A Summary of Octree Ray Traversal Algorithms. *Ray Tracing News 12*, 2 (1999).

[42] HAVRAN, V. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[43] HAVRAN, V., HERZOG, R., AND SEIDEL, H.-P. On Fast Construction of Spatial Hierarchies for Ray Tracing. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[44] HEIDRICH, W., AND SEIDEL, H.-P. Ray-Tracing Procedural Displacement Shaders. In *Graphics Interface* (1998), pp. 8–16.

[45] HUTCHISON, B., HAINES, E., SAMET, H., AND JANSEN, E. Octree Traversal and the Best Efficiency Scheme. *Ray Tracing News 12*, 1 (1999).

[46] IGEHY, H. Tracing Ray Differentials. In *Computer Graphics (Proceedings of ACM SIGGRAPH)* (1999), pp. 179–186.

[47] JACKINS, C., AND TANIMOTO, S. Oct-trees and Their Use in Representing Three-Dimensional Objects. *Computer Graphics and Image Processing 14*, 3 (1980), 249–270.

[48] JOHNSON, C., AND HANSEN, C. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004.

[49] JU, T., LOSASSO, F., SCHAEFER, S., AND WARREN, J. Dual Contouring of Hermite Data. *ACM Trans. Graph. 21*, 3 (2002), 339–346.

[50] K3DSURF. The K3DSurf Project. http://k3dsurf.sourceforge.net/.

[51] KAJIYA, J. T. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH)* (1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 143–150.

[52] KALRA, D., AND BARR, A. H. Guaranteed Ray Intersections with Implicit Surfaces. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1989), ACM Press, pp. 297–306.

[53] KLEIN, T., STEGMAIER, S., AND ERTL, T. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04* (2004), pp. 186–195.

[54] KNISS, J., KINDLMANN, G., AND HANSEN, C. Multidimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 8*, 3 (2002), 270–285.

[55] KNISS, J., MCCORMICK, P., MCPHERSON, A., AHRENS, J., PAINTER, J., KEAHEY, A., AND HANSEN, C. Interactive Texture-Based Volume Rendering for Large Data Sets. *Computer Graphics and Applications, IEEE 21*, 4 (2001), 52–61.

[56] KNITTEL, G. The ULTRAVIS System. In *Proceedings of the 2000 IEEE symposium on Volume visualization* (2000), ACM Press, pp. 71–79.

[57] KNOLL, A., HANSEN, C., AND WALD, I. Coherent Multiresolution Isosurface Ray Tracing. Tech. Rep. UUSCI-07-001, University of Utah, School of Computing, 2007. The Visual Computer (to appear).

[58] KNOLL, A., HIJAZI, Y., KENSLER, A., SCHOTT, M., HANSEN, C., AND HAGEN, H. Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Computer Graphics Forum 28*, 1 (2009), 26–40.

[59] KNOLL, A., HIJAZI, Y., WALD, I., HANSEN, C., AND HAGEN, H. Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 11–18.

[60] KNOLL, A., WALD, I., AND HANSEN, C. Coherent Multiresolution Isosurface Ray Tracing. *The Visual Computer 25*, 3 (2009), 209–225.

[61] KNOLL, A., WALD, I., PARKER, S. G., AND HANSEN, C. D. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 115–124.

[62] KRÜGER, J., AND WESTERMANN, R. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003* (2003).

[63] LAMAR, E., HAMANN, B., AND JOY, K. I. Multiresolution Techniques for Interactive Texture-based Volume Visualization. In *VIS '99: Proceedings of the conference on Visualization '99* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 355–361.

[64] LANEY, D., MASCARENHAS, A., AND MILLER, P. Understanding the Structure of the Turbulent Mixing Layer in Hydrodynamic Instabilities. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1053–1060. Member-P. -T. Bremer and Member-V. Pascucci.

[65] LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes Using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–45.

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

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

[68] LEVOY, M., AND WHITTED, T. The Use of Points as Display Primitives. Tech. rep., CS Department, University of North Carolina at Chapel Hill, 2005.

[69] LIVNAT, Y., AND HANSEN, C. D. View Dependent Isosurface Extraction. In *Proceedings of IEEE Visualization '98* (Oct. 1998), IEEE Computer Society, pp. 175–180.

[70] LIVNAT, Y., SHEN, H.-W., AND JOHNSON, C. R. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 73–84.

[71] LIVNAT, Y., AND TRICOCHE, X. Interactive Point Based Isosurface Extraction. In *Proceedings of IEEE Visualization 2004* (2004), pp. 457–464.

[72] LOOP, C., AND BLINN, J. Real-time GPU Rendering of Piecewise Algebraic Surfaces. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM Press, pp. 664–670.

[73] LORENSEN, W. E., AND CLINE, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH) 21*, 4 (1987), 163–169.

[74] MALLAT, S. *A Wavelet Tour of Signal Processing*. Academic Press, 1999.

[75] MARMITT, G., FRIEDRICH, H., KLEER, A., WALD, I., AND SLUSALLEK, P. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)* (2004), pp. 429–435.

[76] MARMITT, G., AND SLUSALLEK, P. Fast Ray Traversal of Unstructured Volume Data Using Plucker Tests. Tech. rep., Saarland University, 2005. submitted for publication.

[77] MARMITT, G., AND SLUSALLEK, P. Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS)* (2006), pp. 235–242.

[78] MESSINE, F. Extentions of Affine Arithmetic: Application to Unconstrained Global Optimization. *Journal of Universal Computer Science 8*, 11 (2002), 992–1015.

[79] MEYER, M., GEORGEL, P., AND WHITAKER, R. Robust Particle Systems for Curvature Dependent Sampling of Implicit Surfaces. *Shape Modeling and Applications, 2005 International Conference* (2005), 124–133.

[80] MEYER, M., NELSON, B., KIRBY, R., AND WHITAKER, R. Particle Systems for Efficient and Accurate High-Order Finite Element Visualization. *Transactions on Visualization and Computer Graphics 13*, 5 (2007), 1015–1026.

[81] MIDONICK, H. O. *The Treasury of Mathematics*. Philosophical Library, 1965.

[82] MITCHELL, D. Robust Ray Intersection with Interval Arithmetic. In *Proceedings on Graphics Interface 1990* (1990), pp. 68–74.

[83] MOORE, R. E. *Interval Analysis*. Prentice Hall, 1966.

[84] NELSON, B., AND KIRBY, R. M. Ray-Tracing Polymorphic Multi-Domain Spectral/hp Elements for Isosurface Rendering. *IEEE Transactions on Visualization and Computer Graphics (Proceedings IEEE Visualization 2005) 12*, 1 (2005), 114–125.

[85] NIELSON, G., AND HAMANN, B. The Asymptotic Decider: Removing the Ambiguity in Marching Cubes. In *Proceedings of Visualization '91* (1991), G. Nielson and L. Rosenblum, Eds., IEEE Computer Society Press, pp. 83–91.

[86] NIELSON, G. M. Dual Marching Cubes. In *VIS '04: Proceedings of the conference on Visualization '04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 489–496.

[87] OHTAKE, Y., BELYAEV, A., AND ALEXA, M. Sparse Low-Degree Implicit Surfaces with Applications to High Quality Rendering, Feature Extraction, and Smoothing. *Proceedings of the third Eurographics symposium on Geometry processing* (2005).

[88] PAIVA, A., LOPES, H., LEWINER, T., AND DE FIGUEIREDO, L. H. Robust Adaptive Meshes for Implicit Surfaces. In *19th Brazilian Symposium on Computer Graphics and Image Processing* (2006), pp. 205–212.

[89] PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. Interactive Ray Tracing for Volume Visualization. *IEEE Computer Graphics and Applications 5*, 3 (1999), 238–250.

[90] PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98* (October 1998), pp. 233–238.

[91] PASCUCCI, V. Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *Eurographics - IEE TCVG Symposium on Visualization (2004)* (2004), pp. 293–300.

[92] PASKO, A. A., ADZHIEV, V., SOURIN, A., AND SAVCHENKO, V. V. Function Representation in Geometric Modeling: Concepts, Implementation and Applications. *The Visual Computer 11*, 8 (1995), 429–446.

[93] PERLIN, K., AND HOFFERT, E. M. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1989), ACM Press, pp. 253–262.

[94] PESCO, S., LINDSTROM, P., PASCUCCI, V., AND SILVA, C. T. Implicit Occluders. In *IEEE/SIGGRAPH Symposium on Volume Visualization* (2004), pp. 47–54.

[95] POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007). (Proceedings of Eurographics), to appear.

[96] POV-RAY. http://www.povray.org.

[97] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes in C (2nd ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[98] REIMERSS, M., AND SELAND, J. Ray Casting Algebraic Surfaces Using the Frustum Form. In *Computer Graphics Forum* (2008), vol. 27, Blackwell Synergy, pp. 361–370.

[99] RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH 2005).

[100] ROMEIRO, F., VELHO, L., AND DE FIGUEIREDO, L. H. Hardware-assisted Rendering of CSG Models. In *SIBGRAPI* (2006), pp. 139–146.

[101] RÖSSL, C., ZEILFELDER, F., NÜRNBERGER, G., AND SEIDEL, H.-P. Reconstruction of Volume Data with Quadratic Super Splines. *IEEE Transactions on Visualization and Computer Graphics 10*, 4 (2004), 397–409.

[102] RUDIN, W. *Principles of Mathematical Analysis*, 3rd ed. McGraw-Hill, New York, 1976.

[103] RUIJTERS, D., AND VILANOVA, A. Optimizing GPU Volume Rendering. *Winter School of Computer Graphics, Pilzen* (2006).

[104] RUSINKIEWICZ, S., AND LEVOY, M. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. of ACM SIGGRAPH* (2000), pp. 343–352.

[105] SAMET, H. Implementing Ray Tracing with Octrees and Neighbor Finding. *Computers and Graphics 13*, 4 (1989), 445–60.

[106] SAMET, H. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, 1990.

[107] SANJUAN-ESTRADA, J. F., CASADO, L. G., AND GARCIA, I. Reliable Algorithms for Ray Intersection in Computer Graphics Based on Interval Arithmetic. In *XVI Brazilian Symposium on Computer Graphics and Image Processing, 2003. SIBGRAPI 2003.* (2003), pp. 35–42.

[108] SCHAEFER, S., AND WARREN, J. Dual Marching Cubes: Primal Contouring of Dual Grids. In *Proceedings of Pacific Graphics* (2004), pp. 70–76.

[109] SCHEIDEGGER, C. E., FLEISHMAN, S., AND SILVA, C. T. Triangulating Point Set Surfaces with Bounded Error. In *SGP '05: Proceedings of the Third Eurographics Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, 2005), Eurographics Association, pp. 63–72.

[110] SCHREINER, J., AND SCHEIDEGGER, C. High-Quality Extraction of Isosurfaces from Regular and Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1205–1212. Member-Claudio Silva.

[111] SCHWARZE, J. Cubic and Quartic Roots. In *Graphics Gems*, A. Glassner, Ed. Academic Press, 1990, pp. 404–407. ISBN: 0122861663.

[112] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph. 27*, 3 (2008), 1–15.

[113] SHANNON, C., AND WEAVER, W. *The Mathematical Theory of Communication*. University of Illinois Press, 1963.

[114] SINGH, J. M., AND NARAYANAN, P. Real-Time Ray Tracing of Implicit Surfaces on the GPU. Tech. rep., International Institute of Information Technology, Hyderabad, India, 2007.

[115] SRAMEK, M. Fast Surface Rendering from Raster Data by Voxel Traversal Using Chessboard Distance. *Proceedings of IEEE Visualization 1994* (1994), 188–195.

[116] STANDER, B., AND HART, J. Guaranteeing the Topology of an Implicit Surface Polygonization for Interactive Modeling. *International Conference on Computer Graphics and Interactive Techniques* (2005).

[117] STOLL, C., GUMHOLD, S., AND SEIDEL, H. Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU. *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (Sept. 2006)*, 141–150.

[118] STOLTE, N., AND CAUBET, R. Fast High Definition Discrete Ray Tracing Implicit Surfaces. *5th DGCI-Discrete Geometry for Computer Imagery* (1995), 61–70.

[119] STOLTE, N., AND KAUFMAN, A. Voxelization of Implicit Surfaces Using Interval Arithmetics. *Graphical Models 63*, 6 (2001), 387–412.

[120] SUNG, K. A DDA Octree Traversal Algorithm for Ray Tracing. In *Eurographics '91* (Sept. 1991), W. Purgathofer, Ed., North-Holland, pp. 73–85.

[121] TOTH, D. L. On Ray Tracing Parametric Surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), ACM Press, pp. 171–179.

[122] VAN WIJK, J. Ray Tracing Objects Defined by Sweeping a Sphere. *Computers & Graphics 9* (1985), 283–290.

[123] VARADHAN, G., KRISHNAN, S., ZHANG, L., AND MANOCHA, D. Reliable Implicit Surface Polygonization Using Visibility Mapping. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing* (Aire-la-Ville, Switzerland, 2006), Eurographics Association, pp. 211–221.

[124] VELASCO, F., AND TORRES, J. C. Cell Octree: A New Data Structure for Volume Modeling and Visualization. *VI Fall Workshop on Vision, Modeling and Visualization* (2001), 665–672.

[125] WÄCHTER, C., AND KELLER, A. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149.

[126] WALD, I. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[127] WALD, I., BOULOS, S., AND SHIRLEY, P. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), (article no. 3).

[128] WALD, I., FRIEDRICH, H., KNOLL, A., AND HANSEN, C. Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS* (2007), 1727–1734.

[129] WALD, I., FRIEDRICH, H., MARMITT, G., SLUSALLEK, P., AND SEIDEL, H.-P. Faster Isosurface Ray Tracing Using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–573.

[130] WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. Ray Tracing Animated Scenes Using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493. (Proceedings of ACM SIGGRAPH).

[131] WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics).

[132] WESTERMANN, R., KOBBELT, L., AND ERTL, T. Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces. *The Visual Computer 15*, 2 (1999), 100–111.

[133] WHITTED, T. An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (1980), 343–349.

[134] WILHELMS, J., AND VAN GELDER, A. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics 11*, 3 (July 1992), 201–227.

[135] WITKIN, A. P., AND HECKBERT, P. S. Using Particles to Sample and Control Implicit Surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM Press, pp. 269–277.

[136] WOODWARK, J., AND QUINLAN, K. The Derivation of Graphics from Volume Models by Recursive Subdivision of Object Space. In *Proceedings Computer Graphics'80 Conference, Brighton, UK* (1980), pp. 335–343.

[137] WYLIE, B., MORELAND, K., FISK, L. A., AND CROSSNO, P. Tetrahedral Projection Using Vertex Shaders. In *Proceedings of IEEE Volume Visualization and Graphics Symposium* (October 2002), pp. 7–12.

[138] WYMAN, C., PARKER, S., SHIRLEY, P., AND HANSEN, C. Interactive Display of Isosurfaces with Global Illumination. *IEEE Transactions on Visualization and Computer Graphics 12* (2006), 186–196.

[139] WYVILL, G., MCPHEETERS, C., AND WYVILL, B. Data Structure for Soft Objects. *The Visual Computer 2* (1986), 227–234.

[140] YAGEL, R., COHEN, D., AND KAUFMAN, A. Discrete Ray Tracing. *Computer Graphics and Applications, IEEE 12*, 5 (1992), 19–28.

[141] YOON, S.-E., LAUTERBACH, C., AND MANOCHA, D. R-LODs: Fast LOD-based Ray Tracing of Massive Models. *The Visual Computer (Proceedings of Pacific Graphics 2006) 22*, 9-11 (2006), 772–784.

[142] YU, L., JIN, X., ZHAO, Y., FENG, J., AND PENG, Q. Fast Tessellation for Implicit Surfaces. *Proceedings of the 8th International Conference on CAD/CG, Macao* (2003), 283–287.

[143] ZHOU, Y., AND GARLAND, M. Interactive Point-Based Rendering of Higher-Order Tetrahedral Data. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1229–1236.