PARALLEL RAY TRACING IN SCIENTIFIC VISUALIZATION

by

Carson Brownlee

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

December 2012

Copyright © Carson Brownlee 2012

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Carson Brownlee**

has been approved by the following supervisory committee members:

Name	Role	Date Approved
Charles D. Hansen	, Chair	**9/14/12**
Steven G. Parker	, Member	**9/14/12**
Peter Shirley	, Member	**9/14/12**
Claudio T. Silva	, Member	**9/19/12**
James Ahrens	, Member	**9/14/12**

and by **Alan Davis** , Chair of

the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Ray tracing presents an efficient rendering algorithm for scientific visualization using common visualization tools and scales with increasingly large geometry counts while allowing for accurate physically-based visualization and analysis, which enables enhanced rendering and new visualization techniques. Interactivity is of great importance for data exploration and analysis in order to gain insight into large-scale data. Increasingly large data sizes are pushing the limits of brute-force rasterization algorithms present in the most widely-used visualization software. Interactive ray tracing presents an alternative rendering solution which scales well on multicore shared memory machines and multinode distributed systems while scaling with increasing geometry counts through logarithmic acceleration structure traversals. Ray tracing within existing tools also provides enhanced rendering options over current implementations, giving users additional insight from better depth cues while also enabling publication-quality rendering and new models of visualization such as replicating photographic visualization techniques.

For my parents, advisors, coworkers and everyone who has provided guidance and encouragement throughout this endeavor.

CONTENTS

ABSTRACT . . . . **iii**

LIST OF FIGURES . . . . **vii**

LIST OF TABLES . . . . **x**

ACKNOWLEDGEMENTS . . . . **xi**

CHAPTERS

1. INTRODUCTION **1**

1.1 Motivation 1
1.1.1 Interactive Ray Tracing 2
1.1.2 Ray Tracing in Parallel Visualization Applications 3
1.1.3 Ray Tracing using OpenGL Interception 4
1.1.4 Physically-based Rendering 5
1.1.5 Thesis Statement 6
1.1.6 Thesis Contributions 6
1.1.7 Outline 7

2. RELATED WORK **8**

2.1 Parallel Ray Tracing in Scientific Visualization 11
2.2 OpenGL Interception in Scientific Visualization 12
2.3 Computational Photographic Visualization 13

3. PARALLEL RAY TRACING IN EXISTING VISUALIZATION TOOLS **15**

3.1 Rendering Methods 15
3.2 Ray Tracing Implementation 16
3.2.1 Data Distribution 17
3.2.2 Synchronization 18
3.2.3 Depth Buffer 19
3.2.4 Acceleration Structures 19
3.2.5 Color Mapping 19
3.2.6 VTK Factory Overrides 20
3.2.7 ParaView and VisIt 20
3.2.8 Advanced Rendering 21
3.3 Results 22
3.3.1 Datasets 22
3.3.2 Cluster Timings 23

3.3.3 Weak Scaling . . . 24
3.3.4 Strong Scaling . . . 25
3.4 Summary . . . 27

4. RAY TRACING THROUGH OPENGL INTERCEPTION . . . 35

4.1 Interception Implementation . . . 35
4.1.1 Intercepting OpenGL calls . . . 35
4.1.2 Asynchronous Rendering . . . 36
4.1.3 High Quality Rendering . . . 37
4.2 Results . . . 38
4.2.1 Datasets . . . 39
4.2.2 Scientific Visualization Programs . . . 39
4.2.3 Performance Scaling . . . 40
4.3 Summary . . . 44

5. COMPUTATIONAL PHOTOGRAPHIC METHODS . . . 52

5.1 Background . . . 52
5.2 Computational Methods . . . 55
5.3 Precomputation . . . 55
5.3.1 Computing the Refractive Index . . . 56
5.3.2 Octree . . . 57
5.4 Image Generation . . . 58
5.4.1 Emitting Photons from the Light Source . . . 58
5.4.2 Adaptively Tracing Rays through the Flow . . . 58
5.4.3 Reproducing the Cutoff . . . 59
5.4.4 Interferometry . . . 61
5.5 Filtering . . . 62
5.6 Interactive Cutoff Creation . . . 62
5.7 Multifield Flow Analysis . . . 63
5.8 Results . . . 64
5.9 Summary . . . 67

6. CONCLUSION AND FUTURE WORK . . . 77

6.1 Distributed Ray Tracing in Existing Visualization Tools . . . 77
6.2 Ray Tracing Through OpenGL Interception . . . 82
6.3 Computational Photographic Methods . . . 85
6.4 Summary . . . 86

APPENDIX: PUBLICATIONS . . . 88

REFERENCES . . . 90

LIST OF FIGURES

3.1 In the top row, we show renderings of the RM dataset with OpenGL on the left and Manta on the right using ambient occlusion and reflections. In the bottom row, VisIt is shown rendering a molecule plot with OpenGL on the left and an enhanced rendering with Manta on the right. 28

3.2 Manta rendering within ParaView on a single multicore machine using shadows and reflections, showing a dataset of the impact of an aluminum ball on an aluminum plate. 29

3.3 Manta rendering a version of the RM dataset within VisIt. Ambient occlusion provides additional insight to users by adding occlusion information from nearby geometry. 30

3.4 The datasets used for benchmarking. RMI: RM isosurface zoomed out (a), RMI: RM isosurface closeup (b), VPIC dataset with isosurface and streamlines (c) and a wavelet contour with 16 million triangles (d). 31

3.5 Frames per second from compositing using the binary-swap and IceT reduce compositors for a 1024^2 image from 2 to 128 nodes. 32

3.6 Weak scaling timings of an isosurface of the RMO dataset in ParaView. Geometry is added with each additional node so that the geometry per node remains roughly constant. 32

3.7 Strong scaling timings of an isosurface of the RMO dataset. 33

3.8 Strong scaling timings of an isosurface of the RMI dataset. 33

3.9 Strong scaling timings of the VPIC dataset. 34

3.10 Strong scaling timings of the wavelet dataset. 34

4.1 Sequential architecture of GLuRay. 46

4.2 Parallel architecture of GluRay. 47

4.3 GLuRay running within ParaView (a) and an external GUI (b). 47

4.4 Shading effects such as ambient occlusion shown in (b) add more depth cues compared to a standard phong shading technique (a). 48

4.5 Rendering of an impact dataset with ParaView using OpenGL (a) and a rendering using GLuRay demonstrating secondary effects such as reflections, shadows and ambient occlusion (b). 48

4.6 Magnetic fields from an astrophysics simulation colored with streamlines (a), the astrophysics dataset rendered with advanced effects (b), a wavelet contour (c), VPIC Plamsa simulation (d), RMI (e), and RMO (f) datasets used for benchmarking. 49

4.7 BVH build times for the wavelet dataset compared to Mesa rendering time for a single frame. 50

4.8 Gantt charts showing a rendering of a wavelet dataset with 8 million triangles. An overall view of setup time, geometry transfer, acceleration builds and rendering is shown in the top runs while a closeup of the rendering is shown in the bottom. The gold color displays the efficiency of the asynchronous renderer, while the blue displays the cost of BVH builds in relation to data loading and geometry specification through OpenGL, shown in brown. 50

4.9 Performance timings for increasing triangle counts for a wavelet dataset as seen in Figure 4.6(c). 51

4.10 Rendering times for the RMO dataset in ParaView using the Longhorn visualization cluster. Avg denotes the averaged render times across every node while Max designates the maximum render time across all nodes. . . . 51

5.1 2D illustration of the shadowgraph optical setup. 68

5.2 2D illustration of the schlieren optical setup. 68

5.3 A typical color filter used in schlieren optical setups. 69

5.4 2D illustration of the interferometry optical setup. 69

5.5 Infinite-fringe interferometry image computed using our method. 69

5.6 Illustration of the rendering pipeline. 70

5.7 A heptane dataset rendered using refractive indices calculated from temperature and pressure with a knife-edge cutoff (a), a simulated combustion dataset rendered using a schlieren knife-edge cutoff to enhance the flow (b), color filter (c), shadowgraph image (d), a circular cutoff (e), and using a complemented circular cutoff (f). 70

5.8 An illustration of a traversal through the octree. P_1 and P_2 are two rays traversing through the flow. P_1 is in a homogeneous region of the data and in a cell of the octree texture that will report a level number of 1 allowing P_1 to skip to the edge of that level. P_2, on the other hand, is at the lowest level of the acceleration structure and will only traverse to the next voxel. 71

5.9 Creating a custom color filter by painting on the schlieren image. The corresponding region on the color filter is looked up by calculating where that pixel lies on the color filter and coloring the filter red in this case. . . . 71

5.10 Demonstration of multifield data rendered using a schlieren knife-edge cutoff: (a) shows a combination of five different data fields and (b), (c), (d), (e), and (f) show individual renderings of scalar dissipation rate, heat release, vorticity, hydrogen oxygen mass fraction, and mixture fractions, respectively. 72

5.11 On the left, results of a combustion dataset of dimensions 480x720x100 seen in Figure 5.7 rendered with 10 iterations of progressive refinement per frame using cone filtering on a GeForce GTX 280 card at 512x512 resolution. Results of a coal fire with 5 iterations of progressive refinement per frame on a GeForce GTX 280 card at 512x512 resolution are shown on the right. 73

5.12 Comparison of volume rendering (a) with a line of sight schlieren approximation (b) and with our method (c). 74

5.13 Comparison of our method using a shadowgraph (a), a knife-edge cutoff (b), and a color filter (c). 74

5.14 Comparison of unfiltered film plane with 1, 10, 100, and 1000 samples per pixel (a, c ,e, g) and the corresponding images of the film plane filtered with a cone filter in (b, d, f, h). 75

LIST OF TABLES

4.1 Performance timings for various datasets across different applications with varying amounts of triangles specified in the millions. PV signifies ParaView and VI refers to VisIt. RMI and RMO are the Richtmyer-Meshkov datasets zoomed in and out, respectively. GLuRay achieves significant speedups in all runs tested with large polygon counts. 46

5.1 Video memory usage and octree construction time for various datasets. 76

ACKNOWLEDGEMENTS

This endeavor would not have been possible without the significant support from several people and institutions which guided me along the way. Principally, Charles Hansen proved an enthusiastic and patient advisor who introduced me to the field of scientific visualization while incorporating my interest in graphics and ray tracing. Through his expert guidance in the field of scientific visualization I was able to utilize my knowledge of ray tracing in research which could help scientists working in various fields that I initially knew very little about. Beyond giving valuable insight for my research, he also provided essential guidance on technical writing for academic papers and making it through graduate school.

Steven Parker proved a valuable asset as an initial advisor and lead designer and creator of many of the works at the University of Utah such as the Manta ray tracer. I would like to thank Patrick McCormick for mentoring me for two summers at Los Alamos National Laboratory. Through his mentorship I learned a great deal about working with scientists connected to the expiremental side of visualization. This collaboration led to two publications on the subject and a new interest in physically-based rendering for visualization. James Ahrens would prove instrumental in guiding work on integrating ray tracing into visualization systems and without his guidance much of this work would not be possible—as well as the considerable efforts of Li-Ta Lo, John Patchet, David Demarle and many others. Hank Childs at Lawrence Berkeley National Labs was an insightful mentor who helped incorporate my work into the VisIt visualization tool.

Karen Feinauer, Ann Carlstrom, and the rest of the staff at the School of Computing worked diligently to address administrative headaches throughout my program and their help and support is greatly appreciated. The contribution of Chris Johnson in founding the Scientific Computing and Imaging Institute (SCI) can not be understated. SCI presented an ideal atmosphere for student and faculty research that fostered collaboration

between researchers working in diverse fields. Chris provided all of those at SCI with a great building, an ideal work environment, talks, conferences, social events, and most importantly: a continuous source of caffeine.

Working with Kadi Bouatouch at the "Institut National de Recherche en Informatique et en Automatique" (INRIA) in France proved a great experience and I would like to thank all of those in Rennes that welcomed me to their group. Additionally, I had the great fortune of working with a number of collaborators at the University of Utah and other institutions including Vincent Pegoraro, Thomas Fogal, Christiaan Gribble, Thiago Ize, John Patchett, and David DeMarle who proved a great help in formulating ideas, implementating complex systems, or improving my technical writing. I would also like to thank the St. Catherine of Sienna Newman Center at the University of Utah for providing an enriching center for faith and fellowship with a welcoming group of fellow students.

Lastly, I appreciate the support of my family through a long graduate career who, though far away from home, were always there for me.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Advances in computational science have produced a wealth of data in applications ranging from aircraft design to combustion analysis [14, 26, 41, 42, 69]. For scientists working in these domains, exploring simulation data in a visual way allows for an intuitive analysis for feature extraction and debugging. Interactivity is of great importance for data exploration because it allows for rapid changes to camera views and parameters for analysis and simulation steering. As the size of simulations continues to increase, establishing rendering techniques that can display data accurately and that scale well with increasingly parallel architectures becomes essential for understanding complex simulations. Ray tracing presents a viable interactive rendering technique for emerging architectures; it scales well with increasing amounts of geometry and compute cores on compute clusters that are lacking hardware-acceleration, while also providing a straightforward method for generating physically-accurate images for analysis and validation. Integration of ray tracing into existing visualization tools allows scientists to use ray tracing without modifying their workflow by having to learn new tools and transfer data across programs. This dissertation presents such an implementation in the two most widely used open-source visualization tools: ParaView and VisIt [11, 47]. This allows for interactive ray tracing while working with many of the same visualization and analysis options present in the original programs. Such implementations present many engineering challenges for rendering modules which were initially designed for rasterization. This difficulty prompted the development of a ray tracing implementation which intercepts OpenGL API calls, allowing for a program-agnostic method of using ray tracing across many visualization tools without any source code modification to the visualization tool. These two different methods of ray tracing in widely used visualization

tools present the option of developing an integrated renderer which can have ray tracing specific parameters coded into the tools, or a program-agnostic option which requires no modifications to the original tools but does not allow for rendering enhancements exclusive to ray tracing. Additionally, to demonstrate the benefits of using ray tracing in enabling new visualization methods for domain scientists, a method for interactively rendering computational photography techniques for fluid flow visualization was developed.

1.1.1 Interactive Ray Tracing

Ray casting as first presented by Appel [3] presented a straightforward method for computing surface visibility by calculating ray paths from a virtual camera plane to scene objects. This techinque was extended by Whitted [82] to incorporate secondary rays to simulate reflections, refractions and shadow rays in what is commonly known as ray tracing. Kajiya et al. [43] proposed a more advanced illumination model, which better approximates the rendering equation by computing the irradiance at surface points from light reflecting off incident surfaces, when this is computed recursively, an approximation of global lighting is generated. In many common visualization tools, Phong shading is often utilized using only local illumination where surface color is computed from the direct light contribution from emissive light sources such as point lights. This contribution is a function of the surface normal and the angle of incidence between the viewer and the light. Such shading techniques ignore objects that occlude light and inter-reflection between surfaces. Tracing rays allows for the computation of complex light transfer in the form of shadows, inter-reflection, refraction, and scattering through participating media in a more accurate fashion than approximations through rasterization approaches [79]. Monte Carlo techniques using ray tracing allow unbiased representations of real-world lighting effects through sampling of light propagation. While physical simulation of light is often not as imperative as user comprehension in scientific visualization, realistic lighting models which incorporate global lighting effects have been shown to enhance user insight by providing additional cues for determining spatial proximity between adjacent objects [29]. To achieve advanced rendering effects, scientists must often use custom rendering software which provides ray tracing capabilities [66, 6]; however, end users are

often reluctant to learn new tools which are nontypical in their workflows.

In addition to accurate lighting computations, ray tracing provides an intuitive implementation of acceleration methods for achieving interactive rendering with massive polygonal models. Occlusion culling is gained implicitly by utilizing acceleration structures in a nearest-hit algorithm. Subpixel geometry is subsampled in screen-space, negating the necessity for geometry simplification at the cost of aliasing. Ray tracing has proven to scale well on increasingly parallel architectures. Parker et al. demonstrated a ray tracing system similar to Whitted's [82] with interactive frame rates for large data sizes on shared-memory systems [65]. Wald et al. later developed an interactive ray tracing solution for distributed-memory systems [79] using commodity desktop hardware and network infrastructures. Implementations of ray tracing on distributed-memory systems can split up data and rays by image-space or data-space or hybrid approaches incorporating both methods. The former relies on rays in spatially coherent portions of image space requesting similar regions of data as adjacent rays. Data is paged into local caches on demand during scene traversal. In highly efficient implementations, the same page faults used by the operating system can be remapped to network requests instead of disk reads [20]. Ize et al. [38] expanded upon this work by creating a distributed-memory ray tracing solution which paged BVH nodes to a local cache on each node in a ray-parallel fashion. This implementation approached 100 frames-per-second on two megapixel images of complex models.

1.1.2 Ray Tracing in Parallel Visualization Applications

To demonstrate that interactive ray tracing presents a viable rendering technique for visualization, this dissertation presents an implementation of ray tracing integrated into the VTK framework and a scaling study exploring the existing behavior of two widely used visualization applications, ParaView and VisIt, on large distributed-memory systems using CPU and GPU accelerated rasterization which is then compared to CPU ray casting in Chapter 3. These common visualization tools rely on software rasterization through Mesa3D and brute-force GPU rasterization without occlusion culling methods or advanced rendering features. Integrating a software ray tracing solution within VTK provides a

common code base for use across programs built on top of the VTK framework such as ParaView and VisIt. A depth-buffer is generated for compositing operations, allowing the ray tracing implementation to use the existing data-parallel work distribution and sort-last image compositing methods for running on parallel architectures. Interactive rendering performance is achieved for large datasets using basic lighting models while advanced rendering techniques such as reflections, shadows, and ambient occlusion are supported on shared-memory machines and are rendered interactively or off-line depending on the scene and the number of rays per-pixel needed. High quality offline renderings using multiple samples per-pixel and more realistic rendering parameters can be computed within the visualization tools for publication quality images without resorting to external rendering tools such as Maya.

Results are presented using the unmodified open source visualization tools ParaView and VisIt with scientific and synthetic datasets scaling into billions of triangles and up to 128 nodes of a GPU-accelerated rendering cluster. These results are then compared with integrated ray tracing solutions which were built-into the same visualization tools through modifications to the underlying VTK framework. These studies identify common bottlenecks and areas for improvement for rendering on distributed-memory systems where existing distribution methods are not always ideal for rendering implementations, since problem complexity is often very view-dependent. In many cases this solution provides increased rendering performance over the previous hardware or software based brute-force rasterization, especially on systems lacking hardware-acceleration.

1.1.3 Ray Tracing using OpenGL Interception

Integrating ray tracing into scientific visualization programs designed for rasterization algorithms is often a major engineering effort. Updates to the tools often require subsequent modifications to the ray tracing implementations. Additionally, making such modifications necessitates access to the source code of the initial tool. This dissertation presents an implementation of interactive software ray tracing which requires no source code modifications to underlying tools by intercepting OpenGL calls and mapping them to appropriate ray tracing calls to render out identical or improved renderings

to that of fixed-function OpenGL. Interactive performance is often achieved for camera manipulations by instantiating specified geometry with acceleration structures using current transform matrices and material properties specified through OpenGL.

In Chapter 4 we describe the implementation of our system, GLuRay, describing OpenGL interception, asynchronous rendering, and generating high quality images. To understand the trade-offs of our system for dealing with extremely large datasets, we have employed an in depth timing study of three different rendering methods for large polygonal data: software-based ray tracing, software-based rasterization, and hardware-accelerated rasterization. We use four different datasets: one synthetic, and three scientific. Through these studies we show that our system can handle large datasets of various types across multiple applications with vastly improved interactive rendering times, scalability, and enhanced quality over their built-in rendering engines.

1.1.4 Physically-based Rendering

The introduction of physically-based rendering in scientific visualization allows the replication of optical properties that scientists are used to seeing from experimental setups while additionally enabling validation against real-world photographs by replicating optical apparatuses. Shadowgraph, schlieren, and interferometry are common techniques used in experimental flow visualization to analyze features such as shock or heat transfer. These techniques rely on light refracting through a participating medium due to heterogeneous indices of refraction or phase-shifting. The refracted light is then filtered through a cutoff and shown on an image plane which gives a visual representation of light refraction, allowing the visualization of phenomena otherwise often invisible to the naked eye. In Chapter 5, this dissertation proposes a physically-based approach to simulate such experimental setups in an interactive and intuitive fashion by tracing light paths through time-varying scalar fields of computed flows on GPUs.

Recreating these experimental techniques computationally with the simulated physical constraints presents scientists used to schlieren photography a familiar and intuitive visualization method. Conversely, replicating these systems on the computer allows additional degrees of control in the visualization that would be difficult or impossible due to the physical configuration of experiments. This freedom allows for useful features such

as displaying silhouettes around edges or selectively culling ranges in the data. While methods have been developed for approximating schlieren images without refracting light [84, 75], they are not well suited for all data sets, such as shock waves or mixed materials with large changes in refractive indices, which results in divergent light paths.

Calculating light refracting through a flow presents a number of challenges. Light paths must be recomputed whenever the viewpoint changes, thus an interactive method for determining them at each frame is introduced. Graphics hardware is used to trace refraction through inhomogeneous datasets, employ acceleration structures for adaptively sampling data, computationally replicate schlieren cutoffs, and filter out noise. By utilizing these techniques, it is possible to simulate realistic light transport through a flow at interactive rates. To our knowledge this is the first technique to computationally replicate schlieren images by generating piecewise-linear light paths at interactive rates. For further study into the applicability of such methods in scientific visualization, this dissertation also explores interferometry visualization, interactive color filter editing, and an exploration of multifield data visualization. Interferometry allows a different view of data by tracking phase-shift through the flow producing visual bands. Custom color filters allow for exploring specific regions in the flow. Multifield data presents a difficult problem and an interesting exploration of the use of schlieren visualization as a new method for understanding complex data.

1.1.5 Thesis Statement

Ray tracing presents an efficient rendering algorithm for scientific visualization using common visualization tools and scales with increasingly large geometry counts while allowing for accurate physically-based visualization and analysis, which enables enhanced rendering and new visualization techniques.

1.1.6 Thesis Contributions

The main contributions of this work are:

- An integration of ray tracing into existing widely-used parallel visualization tools and a study of comparisons with native rendering methods.

- A method of ray tracing traditional rasterization programs through OpenGL interception which does not require source code modification to underlying programs.
- Physically-based interactive ray casting techniques to compute photographic visualization methods such as schlieren, shadowgraph, and interferometry.

1.1.7 Outline

This dissertation presents related work in Chapter 2, followed by our integration of ray tracing into common scientific visualization tools in Chapter 3 followed by our implementation of ray tracing through OpenGL interception in Chapter 4. Ray casting techniques for computing physically-based photographic visualizations from experimental methods are presented in Chapter 5. Finally, a conclusion and future work are given in Chapter 6.

CHAPTER 2

RELATED WORK

There have been many strategies developed to visualize large-scale data with post processing techniques. Transferring data to a GPU cluster for rendering is a well developed practice capable of displaying large datasets at very high frame rates by distributing data for rendering [25] and compositing the resulting images together using algorithms such as the binary-swap method [53]. To facilitate such techniques, many tools have been developed for rendering large-scale scientific data using polygonal representations. VAPOR was developed as a serial visualization tool by Clyne et al. for atmospheric scientists to explore data using isosurfaces, streamlines and volumetric representations [76]. Programs such as ParaView, VisIt and EnSight present visualization solutions across large heterogeneous cluster environments for data which are often too large to fit or be rendered efficiently on a single node [44, 48, 13]. Many of these systems utilize a client/server architecture with a single client interface utilizing multiple server nodes made up of render or data servers for rendering and analysis. ParaView and VisIt share a common code base called the visualization toolkit, VTK. These real-world applications often do not have the latest compositing or rendering algorithms implemented, nor do they have application specific optimizations. As node counts are scaled up, a larger percentage of time is often devoted to disk and network I/O than rendering a single image as shown by Childs et al. [16]; however, when interacting and exploring data, users may be rendering hundreds or thousands of frames and their rendering times for viewing isosurfaces were noninteractive. Our end goal is to utilize a rendering method which can achieve interactive rendering rates with software rendering on both GPU and compute clusters.

Massive polygon rendering presents challenges for traditional rasterization methods such as those used in VTK. OpenGL relies on hidden surface removal with a simple

Z-buffer test to determine visibility. This not only requires transforming vertex coordinates but also unnecessarily shading fragments that may be rendered back-to-front along the camera axis. When dealing with millions of triangles, many of which are likely obscured behind other triangles, these unnecessary transforms and shading operations degrade performance, resulting in a linear or greater decrease in speed in relation to the number of triangles. With rasterization there are two main methods of gaining better scaling performance: occlusion techniques and surface approximations. Occlusion methods include spatial subdivision schemes, which are used for occlusion culling hidden triangles that lie behind other triangles or are not in the view-frustum. A simple but common occlusion technique is back-face culling where triangles facing the other direction are simply ignored with a simple dot product calculation. For more advanced occlusion culling in scenes with higher depth complexity an acceleration structure's nodes are traversed front-to back and rendered in a multipass rendering process. If the current Z-buffer's values are less than the Z value of the next node, then that node is occluded and can be skipped. Implementations of these methods include the hierarchical Z-buffer [28] and the hierarchical occlusion map [87] which optimize performance through hierarchical Z-Pyramids. Prioritized-layered projection also provides an approximate version for instances where exact results are not required [45]. GPU optimized occlusion queries allow for querying faces of the acceleration structure, which may lie behind the current depth value stored in the Z-buffer in hardware [7].

Model simplification becomes essential for GPU rasterization to scale with large geometry counts. Rendering large amounts of subpixel geometry is slow and often unnecessary when multiple polygons are shading a single pixel. One method for looking at large datasets remotely is to use levels of detail, LOD, techniques first proposed by Clark et al. [18, 52]. Coarser levels of detail can be rendered for distant parts of the scene [24]. For remote rendering, smaller amounts of data can be used by streaming in coarse data and sending finer levels of detail over time [23]. Areas of the data immediately around the camera are streamed in at high detail and areas in the distance are initially very coarse. Over time the coarse areas become refined as more data is streamed in. This works particularly well for data where large portions of the data are obscured behind other parts

of the scene, such as terrain rendering. The Gigawalk system combines occlusion culling, hierarchical level of detail (HLOD), and view frustum culling to view large static scenes [5]. Gigawalk reported a 30x increase in performance from the combination of techniques over using just view frustum culling alone. Their system did not support dynamic loading of out-of-core data sets and suffered from popping from the HLOD system. They also had a noticeable loss in fidelity due to the HLOD system they were using. This method was improved upon in the vLOD system by Chhugani et al. in both out-of-core rendering and better fidelity in the LOD system by using precomputed visibility sets [15]. These techniques have also been adapted to distributed-memory systems [83]. The GoLD system by Borgeat et al. attempted to reduce popping artifacts from LOD transitions by using a geomorphing technique; however, performance suffered as a result [8]. An inherent issue with these HLOD systems is how to represent clusters of data as they take up less and less screen space. At some point, even taken to the extreme this merely becomes rendering clusters of objects as a single triangle or patch which does not necessarily give a good representation of mixed material and surface properties. To ameliorate these problems, point representations of geometry or sample based techniques can be utilized. Point based techniques can represent triangles or clusters of triangles as points [50]. This method makes very good sense for representing subpixel triangles but it is difficult to match the same level of fidelity as a triangle representation [27]. Markus Gross discusses this technique in detail in his book [30].

Mitra and Chiueh [58] developed a parallel Mesa software rasterization implementation by running Mesa in parallel in the background through a serial interface. In order to provide scalability they utilized compositing operations for each running instance of Mesa. Nouanesengsy et al. [60] explored the current performance of Mesa software rasterization on large shared memory machines using various compositing methods. Their tests showed that nearly-linear speedups could be achieved with the number of threads using a hybrid sort-first and sort-last compositing step; however, running multiple instances of ParaView by spawning additional MPI processes failed to scale well in their tests and we focus on performance in existing real-world programs. Howison et al. [33] found that running with a single MPI process with 6 threads in a hybrid parallelism setup was significantly

faster on a 12 core node than an MPI only approach. Therefore, we focus our scaling studies (Chapter 3) on performance timings from a single running program instance, and use multithreaded hybrid parallelism to achieve scaling in our ray tracing implementation.

2.1 Parallel Ray Tracing in Scientific Visualization

Ray tracing on clusters for visualizing large-scale datasets is a well developed field with several benefits over brute-force rasterization methods without advanced techniques. Occlusion culling is gained through the use of visibility tests utilizing acceleration structures. Subpixel geometry is subsampled in screen-space, negating the necessity for geometry simplification at the cost of aliasing. Ray tracing performance has been shown to scale very linearly from one to hundreds of processors on large shared-memory machines [63]. Tracing rays scales well with the amount of geometry in the scene due to the logarithmic acceleration structures used [54, 79], for which we use a packet based traversal of a Bounding Volume Hierarchy, BVH [78]. Cluster based ray tracing methods for distributed-memory systems can split up data and rays by image-space or data-space. The former relies on rays in spatially coherent portions of the image space requesting the same data as their neighbors. When a node needs a new part of the scene, portions of data are paged in on demand. In highly efficient implementations, the same page faults used by the operating system can be remapped to network requests instead of disk reads [20]. Ize et al. expanded upon this work by creating a version of the Manta ray tracer which can run on distributed memory systems by paging in and out cache-aligned BVH nodes in a ray-parallel fashion [38]. Ray parallelism is efficient for out-of-core rendering but not possible within VisIt and ParaView's sort-last distribution and compositing, which distributes data independent of the view. This limitation can lead to suboptimal work distribution with respect to the view which limits rendering performance; however, we have found that this can still provide interactive frame rates for many of our tests while efficiently distributing data analysis and loading.

R-LODs presented a similar system that built LOD representations into a kd-tree, which led to large performance gains in some cases over not using an LOD system; however, their render times for a 128 million polygon dataset was still often subinteractive,

attaining interactive performance only when zoomed out when approximate representations of the data were used [85]. Another limitation of their system is that secondary rays could not always be handled, especially refraction and nonplanar reflections. Stephens et al. showed that the Manta ray tracer could achieve real-time frame rates with shadows on larger datasets such as the Boeing 777 by using large shared-memory systems [72]. Their system also displayed the speed at which semitransparent surfaces could be rendered with ray tracing which is often a difficult and slow process in rasterization methods which require depth-ordered rendering of scene geometry. Their method did not rely on LOD techniques and showed up to 93% efficiency when scaling up to 60 processors resulting in superior image quality and better performance than the Far Voxels technique. This was a clear advantage of using a software ray tracing system when geometry counts are very large, as out-of-core rasterization techniques on the GPU can be very complicated, slower, and be prone to approximation errors as seen in various point-based or voxel-based surface representations.

Current implementations and scaling studies of ray tracing employ a custom rendering framework as opposed to standard visualization tools such as ParaView or VisIt. Marsalek et al. recently implemented a ray tracing engine into a molecular rendering tool for advanced shading effects for the purpose of producing publication images [56]. We have integrated a ray tracing engine into a general purpose visualization tool that can use advanced rendering effects; however, we focus on ray tracing's logarithmic performance with large data rendering as the primary motivation for this dissertation. The Manta real-time ray tracing software provides an efficient rendering solution, but is not a full featured visualization package [72]. Thus, combining CPU rendering using Manta is ideal when combined with other cluster based visualization tools such as ParaView and VisIt, which handle geometry creation, analysis, data distribution, and image compositing within tools which users are accustomed to.

2.2 OpenGL Interception in Scientific Visualization

There have been many ray tracing implementations developed for high performance parallel rendering; however, these are often created for a specific paper and only exist as

a one-off implementation or an API which must be integrated into tools to be of much use to computational scientists. RTRT was developed for interactive ray tracing but was never expanded into an API for use as a rendering engine in other programs [79]. Wald et al. later introduced a ray tracing API called OpenRT which gave an API interface similar to OpenGL and was shown to scale well using commodity PC clusters [77]. The approach discussed in Chapter 4 intercepts OpenGL API calls from existing programs, instead of creating another API which requires developers to rewrite their visualization implementation. This enables ray tracing to be used in a program-agnostic fashion without having to develop custom renderers for each visualization tool.

OpenGL interception is a common method used for debugging and profiling OpenGL applications with programs such as glTrace [70]. WireGL and Chromium took OpenGL interception a step further by modifying the behavior of the OpenGL calls into a stream processing framework to support functions such as distributed rendering with sort-first and sort-last compositing operations [34, 35]. While there are implementations of Chromium that alter the rendering behavior of OpenGL, no ray tracing support has been integrated into such a system. The main contribution outlined in Chapter 4 is the presentation of a ray tracing implementation using OpenGL interception with interactive performance and advanced rendering effects for scientific visualization applications. Presently this implementation only translated fixed-function OpenGL commands and does not support shaders. Parker et al. developed RTSL [64], a shading language for Manta which was largely a superset of GLSL. This provides the potential to translate GLSL shaders for use with GLuRay; however, such an implementation was not explored for this dissertation. A scaling study is also presented in Chapter 4 which demonstrates GLuRay performance scaling over multiple nodes with ParaView using data-parallel work distribution.

2.3 Computational Photographic Visualization

Computational schlieren images of three-dimensional fluid flows have been computed noninteractively using a ray tracing method by Anyoji et al. [2, 73]. Such techniques produce an accurate image but are not ideal for data exploration. A nonphotorealistic method for producing schlieren-like images using line of sight ray traversals for visualization was

recently introduced [75], but without calculating light paths from refraction. In order to reproduce an accurate physically-based representation, tracing refracted light trajectories is necessary for flows with large variations in refractive index such as shock waves or flows with multiple materials. Ray tracing also allows for the reproduction of the optics used in an experimental setup. The inverse of the problem was achieved by Atcheson et al. [4] by using schlieren photographs to compute a three-dimensional scalar field; however, we focus on visualizing simulated flows.

Algorithms for computing caustics have been developed over the past two decades in the computer graphics community. Photon maps were introduced as a method for computing caustic and global illumination effects offline [39]. Photon maps were later extended to volumetric photon mapping to compute scattering effects and caustics through inhomogeneous media [31, 40]. Although these offline methods are not directly applicable to our work, they present filtering techniques for reducing noise in regions of low photon density as well as equations for computing light paths. Tracing light refraction through volumes at interactive rates was introduced with Eikonal rendering, which relied on precomputing wavefront propagation through a grid [36]. Eikonal rendering relies on a long precomputation step that is not feasible for schlieren systems where the light source changes relative to the volume whenever the camera rotates. Sun et al. presented a technique [74] that calculated single-scattering effects through a volume. Viewing rays were then computed as a separate pass for interactive light refraction. In a typical schlieren setup the film plane is directly facing the light source, so computing a separate pass for light scattering and viewing rays is unnecessary. Chapter 5 introduces a novel method for using ray tracing to reproduce both schlieren and interferometry imagery. Scattering effects can play a role in some flows but Chapter 5 focuses purely on refraction in media such as air.

CHAPTER 3

PARALLEL RAY TRACING IN EXISTING VISUALIZATION TOOLS

As computing power increases, the computational sciences are continuing to provide ever larger datasets that are challenging the brute-force rasterization architectures present in the most common open-source scientific visualization tools such as ParaView and VisIt. Ray tracing presents a rendering solution capable of attaining interactive rendering rates on supercomputing platforms, which often lack rendering hardware, through the efficient use of acceleration structures. Integrating ray tracing into existing widely used tools demonstrates the potential of using ray tracing as a primary rendering method for scientific visualization on emergent compute clusters for large-scale in situ renderings. We chose to implement our system on top of the Visualization Toolkit, VTK, to create an intermediary layer which could be implemented across multiple programs that use VTK such as ParaView and VisIt with limited modifications. In this chapter we show that our implementation achieves interactive rendering rates for large data sizes, scales within existing work, and data distribution methods on multicore machines and across distributed-memory clusters, and achieves high-quality renderings without requiring the use of external rendering tools.

3.1 Rendering Methods

Our approach to comparing visualization methods on large distributed systems was to evaluate three rendering techniques within two commonly used visualization applications (ParaView and VisIt): hardware-accelerated rasterization, software-based rasterization, and software-based ray tracing. Each method has its advantages and drawbacks.

Hardware-accelerated rasterization has proven to be fast for modest data sizes and is widely used and heavily supported. The disadvantages are the requirement for additional

hardware, small memory sizes on the GPU, and, due to the nature of rasterization, rendering times that scale linearly with the amount of geometry in the scene. Advanced rasterization techniques such as hierarchical level-of-detail methods, HLOD, are not currently implemented in widely used visualization tools such as ParaView or VisIt. Therefore, we do not consider them in our study. ParaView does support a single level of detail which can be toggled when the user interacts with the camera; however, this degrades the quality of the entire model and thus is not considered for our tests.

Software rasterization through Mesa is a build option for both ParaView and VisIt and is a common method used on supercomputers when GPU hardware is not available. It offers the same support for programs that would normally use hardware-acceleration methods. The main drawback of this method is speed, as Mesa remains single threaded and delivers very slow performance even for low geometry counts. A benefit over hardware-accelerated rasterization, however, is that it does not require additional graphics hardware and can utilize large system (CPU) memory.

Software ray tracing provides a rendering method that scales in $k * O(log(n))$ where k is image size and n is the number of polygons. This scaling performance assumes nonoverlapping polygons and a well-balanced acceleration structure. Because of the screen space dependent performance with logarithmic scaling to geometry, ray tracing provides efficient performance which scales well with increasingly large geometry counts, especially for subpixel geometry. Using an acceleration structure to test ray intersections also allows easy and straightforward visibility tests where only the nearest geometry needs to be shaded once for each pixel. Hardware ray tracing also exists, but we chose to focus only on software ray tracing. We have implemented ray tracing as a rendering mode for polygon data within ParaView and VisIt.

3.2 Ray Tracing Implementation

ParaView and VisIt are open-source visualization frameworks designed for local and remote visualization of a large variety of datasets. They are designed to run on architectures from a single PC desktop up to large cluster arrays using client/server separation and parallel processing. VisIt operates with a client/server architecture to run

multiple servers for data analysis and rendering on large clusters. Server/data separation allows ParaView to be broken into three main components: data servers, render servers, and a client [12]. This separation allows for varying numbers of servers to be used for data or rendering depending on the need. Much of the actual rendering code is based around the Visualization Toolkit (VTK) while much of the client/server model is unique to either ParaView or VisIt. This common base allows us to to implement ray tracing using the open-source ray tracer Manta in VTK and integrate the VTK ray tracer into ParaView and VisIt with only minor modifications specific to each program. Manta was chosen as a ray tracing engine for its open source distribution, real-time rendering performance, and use of advanced acceleration methods such as packet tracing and parallelized bounding volume hierarchy, BVH, builds.

3.2.1 Data Distribution

ParaView and VisIt both utilize client/server models for running on large, distributed-memory systems. VisIt launches a viewer client and multiple servers which are used for data loading and rendering. A single server processes is responsable for communicating with the viewer. ParaView's data server abstraction layer differs slightly by allowing for operations such as processing data on one node and sending the resulting geometry to another node or multiple nodes for rendering instead of each server processes devoted to loading, processing, and rendering a portion of the data. This allows for changing the data processing and rendering pipeline across heterogeneous architectures for balanced workload distribution when more or fewer rendering servers are needed than data servers. When rendering on multiple nodes, sort-last compositing is required to combine images from multiple nodes. Data-parallel data distribution is good for rasterization, but not necessarily optimal for ray tracing where render time is dependent more on work distributed over the viewing frustum. When zoomed out over an entire model, distributed cluster-based ray-tracing often produces a sufficiently balanced workload distribution; however, if a small portion of the data is taking up the majority of screen space then the majority of work is being done by a limited number of nodes which contain data within the viewing frustum. Despite this, we have found our solution within the existing work

distribution to be usable in practice as shown in section 4.2. Distributing the compositing work across a cluster is vital for efficient cluster utilization. For this we use a binary-swap [53] implementation and use the IceT compositing library with default settings, which can be enabled for both VisIt and ParaView. binary-swap is a parallel compositing algorithm that exchanges portions of images between processes to distribute the workload. Because binary-swap composites the entire image, empty portions of the scene are needlessly composited together. IceT allows for variably sized compositing windows, encapsulating only portions of the scene which contain rendered geometry. This has the potential to vastly decrease the amount of image data sent across the network for our applications as more nodes are used for rendering.

3.2.2 Synchronization

The Manta ray tracing library was originally designed for large shared memory systems. To achieve the highest possible frame rate possible, multiple rendering threads are launched on a single process and the renderer sequentially renders frames as fast as possible. VTK was designed for event driven rendering where one frame is rendered after user interaction. The threaded nature of Manta also presented a thread safety issue: Manta's state can only be accessed and modified at certain points in its pipeline through callbacks called transactions. In order to safely halt Manta between user interactions, synchronization was added through semaphores in the display phase. While each render thread renders the scene, the first thread displays the previously rendered frame and then continues ray tracing in order to utilize all threads. At the end of a rendering step, the threads are synchronized and state is updated. This is where state can safely be accessed and modified outside of Manta through transactions. The default Manta behavior results in a one frame delay between rendering and display of the current scene because of the display synchronization.

The rendering architecture of Manta was modified slightly to have the image display outside of the rendering stage as a separate synchronization step, which is only released upon a render event in VTK. This eliminates unnecessary rendering before or after a render event. A rendering call triggers Manta to release its rendering lock in the

display phase, process transactions, and render a frame, which is then copied back to the vtkRenderWindow instance. Due to the differences in how an image is stored between VTK and Manta, this requires an image conversion step. ParaView or VisIt then display the rendered image or send the image out for compositing if it is being rendered remotely through a cluster.

3.2.3 Depth Buffer

Sort-last rendering requires a depth buffer in order to determine an ordering for overlapping geometry in the compositing step. In order to work inside VTK, Manta required a depth buffer to be implemented, ray tracing typically does not require one. Depth values, or rather closest hit values, are typically kept per ray, which meant that for our implementation all that was needed was a separate buffer and a simple write operation for each ray. This results in additional memory overhead with one float per image pixel for the buffer.

3.2.4 Acceleration Structures

Ray tracing uses acceleration structures to compute hit points efficiently. This means that a new acceleration structure needs to be built with each change in geometry within VTK. Generating a new acceleration structure each time a new vtkActor is added or updated in the pipeline with a custom vtkActor override facilitates this. For very large scenes consisting of millions of triangles this can take several seconds of precomputation time. The amount of time also depends on the acceleration structure used. Grid based acceleration structures can be faster to update; however, we chose to use a Bounding Volume Hierarchy, BVH, as it gave similar performance to a kd-tree while benefiting from faster build times [37].

3.2.5 Color Mapping

In Manta, assigning a per-vertex color would require creating a material for each triangle which would entail a large memory overhead. There was also no support for per-vertex materials, thus, we chose to implement colors through a 1D colormap. In Manta, texture coordinates are weighted by the barycentric coordinates of the interior

point as a weight. This provides a smooth coloration and little overhead since texture coordinates are computed for each intersection point even for constant colors. Both VisIt and ParaView support texture colormaps which are implemented through the same texture colormap used for singular color values in Manta.

3.2.6 VTK Factory Overrides

The Manta context is created in VTK through a custom factory object which overrides many common VTK objects. A vtkRenderWindow overrides other windows to keep track of updates to image size. A vtkPolyDataMapper keeps track of updates to geometry which then sends a group of polygons, usually a triangular mesh, to the active vtkActor. vtkActors are overloaded to track material properties and to maintain acceleration structures needed for rendering. When an actor is updated, so is its corresponding acceleration structure. Unfortunately, because of the differences in how meshes are represented between Manta and VTK, there is currently additional memory overhead due to geometry duplication. Lights are overloaded with custom vtkLight objects which are stored as either directional or point lights. Ambient occlusion is treated as an ambient light source in Manta, and so it is added to the actor through custom options specific to the application. Finally, a custom vtkRenderer synchronizes the Manta display with VTK render calls and sends image data to the active window for compositing or display. This integration allows for Manta to behave identically to the OpenGL render engine for polygonal rendering within VTK. Figure 3.1 shows an isosurface of the Richtmyer-Meshkov instability rendered with OpenGL on the left and Manta on the right, using ambient occlusion with the same material and camera information provided by VTK.

3.2.7 ParaView and VisIt

VisIt and ParaView are both built on top of the VTK framework which allowed our VTK implementation to integrate into their rendering pipelines with little modification. Activating factory overrides to use the Manta ray tracer is handled through ParaView's plugin interface or Visit is commandline interface. These mechanisms allow for ease of activating or deactivating the Manta renderer; however, both implementations require closing other rendering windows before launching Manta. A Manta IceT rendering pass

was created for ParaView that sends image buffers directly to IceT for faster compositing. This modification has not yet been transfered over to Visit is implementation and instead, VisIt sends image data to the active render window which currently is not a custom Manta version for VisIt; this can limit performance in some cases. A custom implementation of some annotation objects, such as axes, were overridden to prevent them from sending significant amounts of geometry every frame; however, these could be reimplemented to maintain their geometry from frame to frame. In ParaView, most polygonal rendering modes are supported. Volume rendering is not currently supported in Manta-enabled VisIt or ParaView. While implementing Manta-based volume rendering for VisIt and ParaView is highly desired and implementations for this already exist within Manta, a VTK integration remains as future work.

3.2.8 Advanced Rendering

Ray tracing allows for additional scene and material properties from those previously found in VisIt and ParaView. These additional options include such materials as dielectrics or transparencies as well as multisampling and threading options to specify the number of threads the ray tracer will launch for rendering. The result of such rendering options is shown in Figures 3.1, 3.2, and 3.3. Figure 3.2 displays an aluminum ball hitting an aluminum plate with shadows and reflections. Figure 3.3 shows an isosurface of the RM dataset rendered with Manta within VisIt. Ambient occlusion provides additional insight to the viewer by shading by occlusion from neighboring polygons, pulling out features at the surface of the isosurface. Advanced shading effects such as shadows, ambient occlusion and reflections can only be rendered when running on a shared memory system because of the need to access global scene geometry. VTK has no way to handle fetching distributed data for such information in the middle of a rendering. Ambient occlusion will work with local data; however, due to data distribution regions bordering blocks of data will be missing occlusion information. This could be resolved by duplicating borders of data blocks for each node and only considering polygons a small distance from ray hit points. This would require rewriting the data distribution within ParaView and was beyond the scope of this work. In order to expose these options to the user, GUI elements

were added to ParaView. These options are expected to also be included with VisIt in a future release.

3.3 Results

We evaluated the rendering performance of various methods on Longhorn, an NSF XD visualization and data analysis cluster located at the Texas Advanced Computing Center (TACC). Longhorn has 256 4X QDR InfiniBand connected nodes, each with 2 Intel Nehalem quad core CPUs (model E5540) at 2.53 GHz and between 48-144 GB of RAM. Each node of Longhorn also has two NVidia FX 5800 GPUs. We used three datasets of varying sizes: a synthetic wavelet dataset, and a dataset from Los Alamos's plasma simulation code VPIC, and a timestep from a Richtmyer-Meshkov instability simulation rendered with two different views. We used ParaView 3.11 and VisIt 2.4 for all timings with three different rendering modes: Manta, an open-source ray tracer, Mesa, a software OpenGL rasterizer, and hardware-accelerated OpenGL. ParaView and VisIt were built with the IceT library and use Mvapich2 1.4. Additional code for timing was added for ParaView and VisIt. Mesa is not multithreaded nor the fastest available software rasterization package; however, it is the only one supported internally within ParaView and VisIt and is commonly used when GPUs are not available. The OpenGL implementation within VTK is also a brute-force implementation with no advanced acceleration methods used. This study is therefore not a comparison of the potential of these algorithms, but rather a real-world study of their existing performance within common visualization tools. To test the scaling of these packages we ran a series of weak and strong scaling studies up to 128 nodes on Longhorn.

3.3.1 Datasets

- **Richtmyer-Meshkov Instability** We created a polygonal model from an isosurface of time-step 273 of the Richtmyer-Meshkov instability (RM) simulation, resulting in 316 million triangles. To better understand the behavior of the data distribution in ParaView and VisIt, we have rendered both a zoomed out view of the RM dataset, RMO, seen in Figure 3.4(a) and a closeup view, RMI, in Figure 3.4(b). The closeup view should expose the behavior of sort-last rendering when the data being rendered

on screen belongs to only a small portion of the overall nodes.

- **VPIC Visualization** Using a single time-step from the VPIC plasma simulation, we calculated an isosurface and extracted streamtubes that combined, totaled 102 million polygons. A view of this data set can be seen in Figure 3.4(c).
- **Wavelet** The wavelet triangle dataset is a computed synthetic dataset source released with ParaView. We generated a 201^3 dataset and then calculated as many isosurfaces as needed to produce a certain quantity of triangles. The isosurfaces are nested within each other. Images produced with 16 million triangles are shown in Figure 3.4(d).

3.3.2 Cluster Timings

To test the performance of the three rendering modes with large-scale data we conducted a series of timing studies on the rendering cluster Longhorn that look at how performance behaves with varying numbers of cluster nodes using a single process per node of ParaView and VisIt. A window size of 1024^2 was used with offscreen rendering for both ParaView and VisIt. We investigate two types of performance: strong and weak scaling. A strong scaling study keeps the problem size constant while increasing resources to solve the problem. As the number of nodes increases, the data being loaded and rendered per node decreases. A weak scaling study keeps the problem size constant per node as the number of nodes increase. The total frame time is dominated by a combination of the rendering time and the compositing time where $t_{total} = t_{composite} + t_{render}$. An alternative method would be to have a frame delay, where the last frame is displayed and thus the total time would be $t_{total} = max(t_{composite}, t_{render})$. However, this would require a reworking of the VTK pipeline in order to be able to push rendered frames after user interaction was finished and was not implemented for this study.

In order to determine bottlenecks from compositing, Figure 3.5 displays total frames per second using binary-swap running in ParaView with an empty scene to show performance where the entire image from each node must be composited. With binary-swap, pixels generated by rendering empty portions of the scene are needlessly sent over the network. To show a more practical example, frames per second from the VPIC dataset

using IceT are presented. The IceT frames per second were calculated as the inverse of the reported composite and render times for the client node on processor zero subtracted from the maximum rendering time across all nodes. From these runs we can see that on the InfiniBand cluster used in our testing, the worst performance we could expect with binary-swap is over 20 fps on 128 nodes; however, in some cases IceT achieves nearly 100 fps by only compositing small portions of the image. Our goal is then to achieve rendering speeds which can match or exceed the maximum binary-swap compositing times.

3.3.3 Weak Scaling

In a weak scaling study the problem size scales with the processor resources. For our testing, we conducted a weak scaling benchmark studying rendering times for a scientific dataset. Figure 3.6 shows a weak scaling study conducted using the Richtmyer-Meshkov instability zoomed out, RMO, dataset and scaled up by adding additional isosurfaces to increase the number of polygons as more nodes were added. In total, the polygon count ranged from around 49 million triangles for a single node up to over 1.6 billion for 128 nodes. Hardware and software-based OpenGL times remain above 1 second while ray tracing times decrease with more nodes. This shows that the rendering time of the brute-force rasterization algorithm is fairly constant with geometry count regardless of the distribution of the data in the scene. Our ray tracing implementation, however, ignores occluded geometry and is bounded by $k * log(n)$. Empty space is easily skipped over by tracing rays in packets and testing the bound box of the scene. With weak scaling, geometry count, n, remains fairly constant while the effective rendered area, k, shrinks as the scene is broken up to be distributed among more nodes. It is not clear how the GPU performance could replicate this, except with the use of an intelligent hierarchical level-of-detail, HLOD, algorithm that averages subpixel geometry in a preprocessing step, such as with GigaVoxels [19]. The maximum rendering times are shown with the Max designation in Figures 3.6-3.10. These times display the average of the maximum times over all the nodes for each run. Ray tracing displays a large difference between the average and maximum render times as the data-parallel scene distribution is not ideal

for ray tracing and some nodes finish significantly faster than others. In cases where we achieve perfect work distribution the average and maximum average numbers would be identical and it is clear that increasing the number of nodes increases variance between render times. Manta displays superior performance on a single node as it uses a BVH to accelerate over large amounts of geometry and skip over occluded geometry.

In our tests the main performance bottleneck for the hardware-accelerated rasterization renderings appear to be due to a single display list being compiled over immediate mode rendering calls. This means that updates to a color require rebuilding the display list. Display lists can easily be disabled; however, this results in a drop in rendering performance. We found that a single display list resulted in poor performance with large data counts and in some cases would even crash the program requiring immediate mode rendering to be used for large polygon counts. Splitting up the display lists in our tests showed over 10x performance increases in some cases; however, results varied across different hardware and drivers. Using vertex and color buffers would result in significantly fewer draw calls per update which could drastically decrease rendering times. However, this would not affect the asymptotic runtime of the underlying algorithm and these methods are not used in released VisIt or ParaView versions.

3.3.4 Strong Scaling

Figure 3.7 shows strong scaling of the 316 million-triangle contour from timestep 273 of the Richtmyer-Meshkov instability simulation. At a single node the benefit of the BVH in Manta can clearly be seen as the ray tracer attains several times the performance of the brute-force OpenGL implementation, rendering in about .1 seconds with just two nodes.

VisIt is slightly slower than ParaView in many of our runs. Through testing we found this is likely because VisIt uses a texture map for colors instead of a single call to glColor as in ParaView and ParaView uses triangle strips whereas VisIt does not. Manta in VisIt does not scale as well in higher node counts as our ParaView implementation because our current VisIt implementation uses the current vtkRenderWindow rather than a custom Manta render window which limits performance to around .05 seconds from calls to glDrawPixels to copy the image buffer into the OpenGL context. Since the VisIt bottleneck results in a maximum of about 20 fps, which is the compositing limit of

binary-swap, this is not a significant issue that inhibits interactivity, and one that is likely to be fixed in a future update.

Figure 3.8 displays strong scaling numbers for a zoomed in view of the RM dataset, RMI, as seen in Figure 3.8. While this has little to no affect on the rasterization algorithm, in ray tracing this results in different scaling performance. While the dataset is broken up and distributed across nodes, only a few of those nodes actually have a portion of the visible data. The ray tracer was already culling out the nonvisible data when the entire dataset was rendered on a single node. Thus, roughly the same amount of visible data was being rendered on a single or small number of nodes. The average render times drop similarly to the RMO render times; however, the maximum render times are generally worse for the RMI renderings than for the RMO renderings at 8, 16, 32 and 64 nodes. This increase in maximum render time shows the effects of data-parallel rendering work distribution as a few nodes do more work. The maximum time eventually drops, which is likely when the data subsets per node were small enough to split up most of the visible geometry. Ize et al. [38] reported roughly linear scaling behavior through ray-parallel work distribution up to 60 fps until approaching a bottleneck introduced by sending pixels over the InifiniBand network at around 127 fps. A reworking of ParaView's work distribution could show similar scaling performance; however, our timings suggest that a data-parallel implementation within the existing framework scales to interactive rates. Figure 3.9 shows timings for the VPIC dataset which is made up of 102 million triangles. In this dataset, the GPU accelerated OpenGL render times for ParaView achieve below the .1 second mark needed for 10fps interaction, but only at 64 to 128 nodes when the effective triangle count per node is around a million triangles per node.

A contour of a wavelet was created to test performance with occluded geometry in a predictable fashion through 27 overlapping isosurfaces. While the RM and VPIC datasets contain a lot of overlapping geometry, each uses a single isosurface. Performance for the 16 million triangle wavelet dataset is shown in Figure 3.10. Manta rendering times are below .1 seconds on a single node and rendering time drops significantly with additional nodes; however, the rendering times appear to reach a bottleneck at eight nodes which is showing the bottleneck introduced from rendering a mostly empty scene and copying

image buffers from Manta into VTK. Some spikes in timings are seen at higher node counts, this is likely from a single node or a few nodes running slower due to the high maximum render time at 128 nodes for ParaView Manta.

3.4 Summary

We have demonstrated through our timing results that ray tracing implemented into the two most widely used open-source visualization tools in scientific visualization presents an alternative rendering implementation that often outperforms existing hardware and software rasterization implementations while also enabling advanced rendering effects. Rendering on compute clusters, which before may have utilized Mesa software rasterization and failed to achieve interactive rendering rates, can now be used interactively with our system, and in many cases, enhancing data exploration for computational scientists. This implementation is especially promising for in situ visualization where rendering is conducted on the same compute clusters as simulation that often lack hardware acceleration.

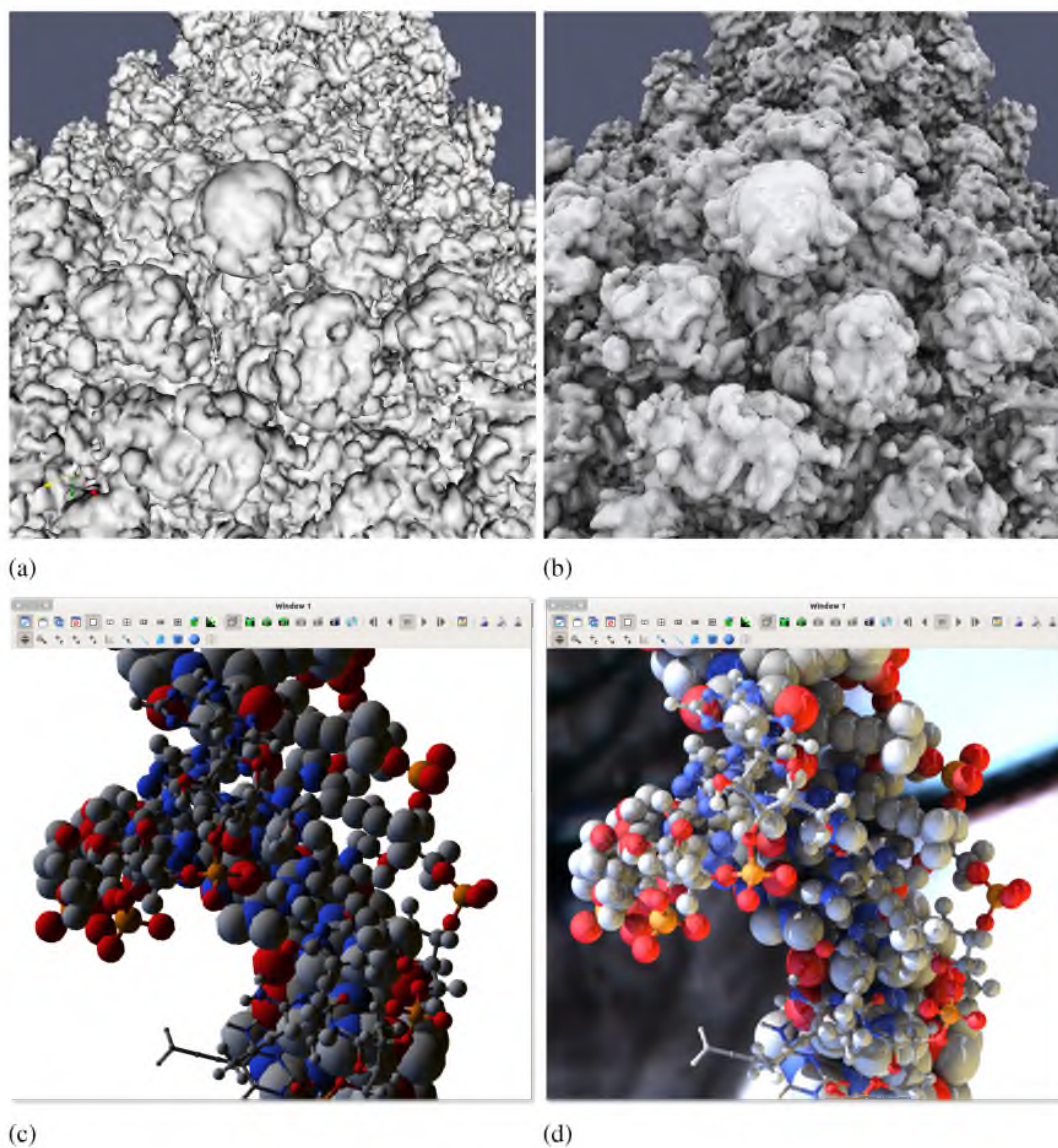


Figure 3.1. In the top row, we show renderings of the RM dataset with OpenGL on the left and Manta on the right using ambient occlusion and reflections. In the bottom row, VisIt is shown rendering a molecule plot with OpenGL on the left and an enhanced rendering with Manta on the right.

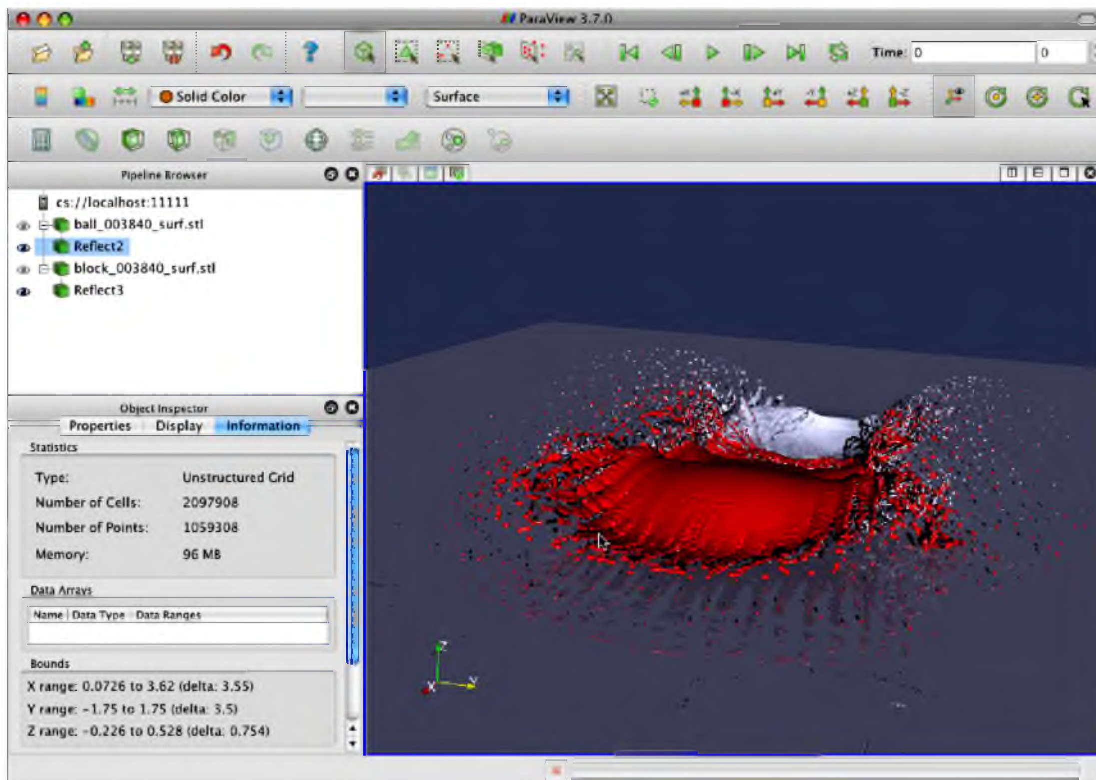


Figure 3.2. Manta rendering within ParaView on a single multicore machine using shadows and reflections, showing a dataset of the impact of an aluminum ball on an aluminum plate.

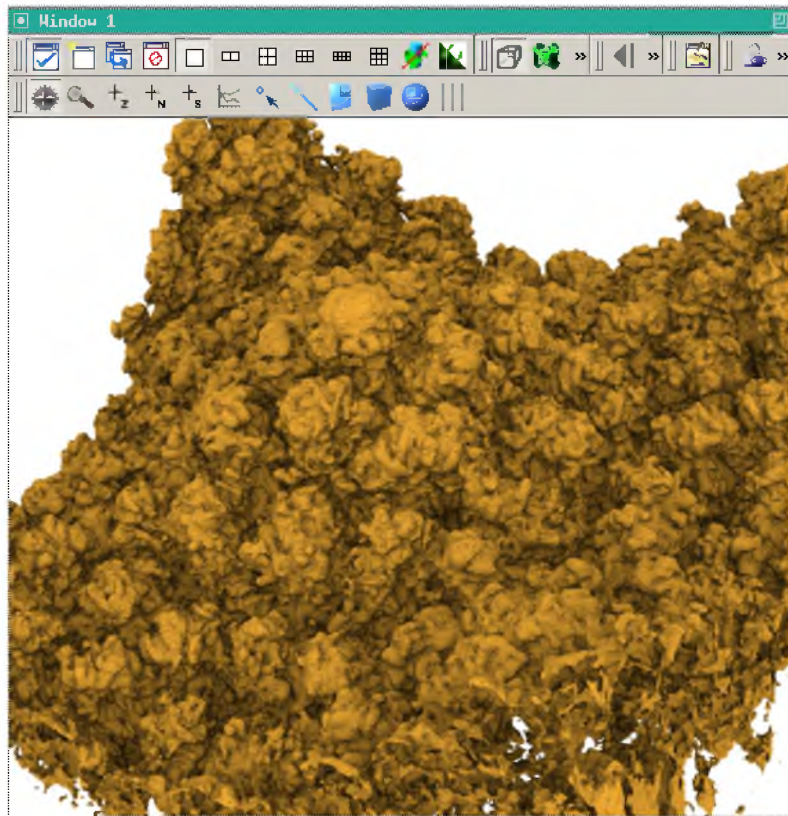


Figure 3.3. Manta rendering a version of the RM dataset within VisIt. Ambient occlusion provides additional insight to users by adding occlusion information from nearby geometry.

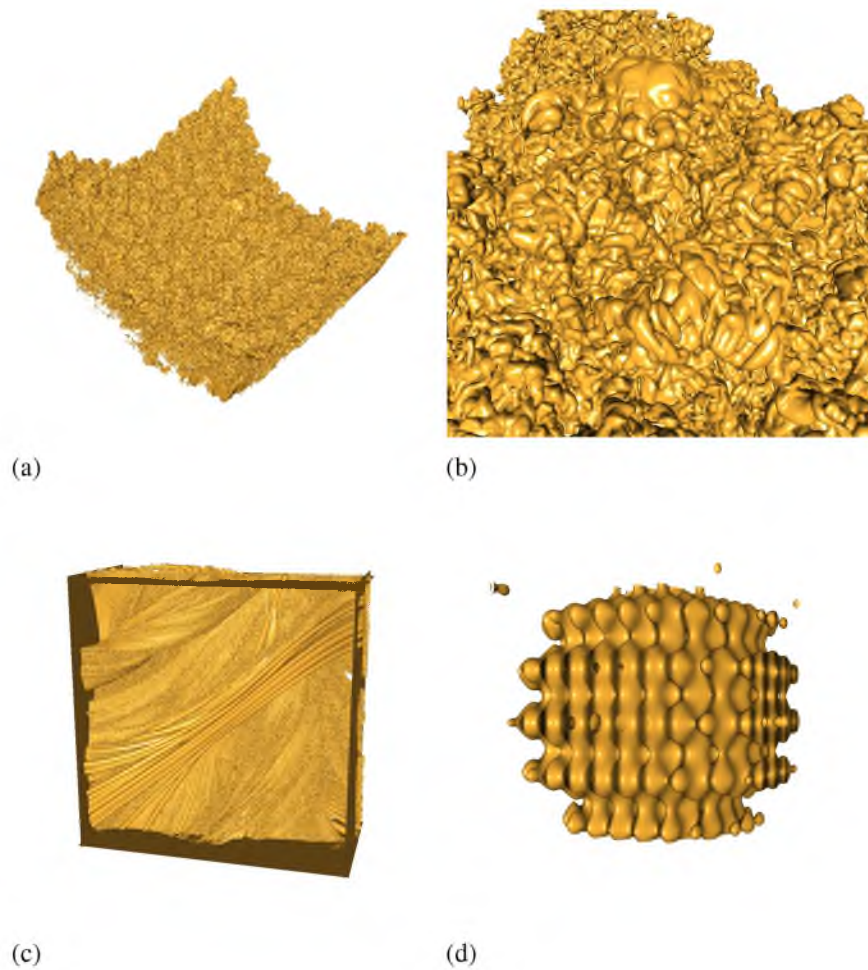


Figure 3.4. The datasets used for benchmarking. RMI: RM isosurface zoomed out (a), RMI: RM isosurface closeup (b), VPIC dataset with isosurface and streamlines (c) and a wavelet contour with 16 million triangles (d).

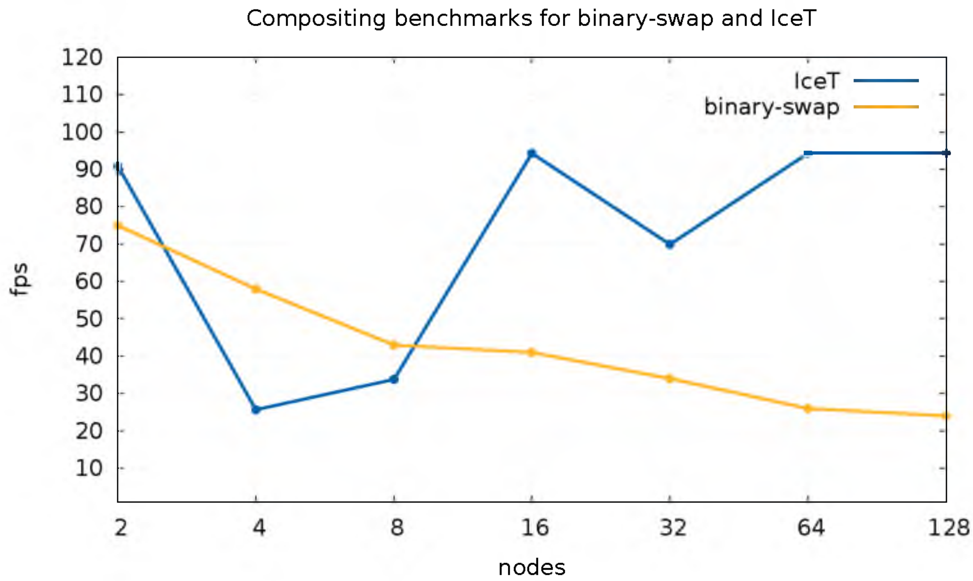


Figure 3.5. Frames per second from compositing using the binary-swap and IceT reduce compositors for a 1024^2 image from 2 to 128 nodes.

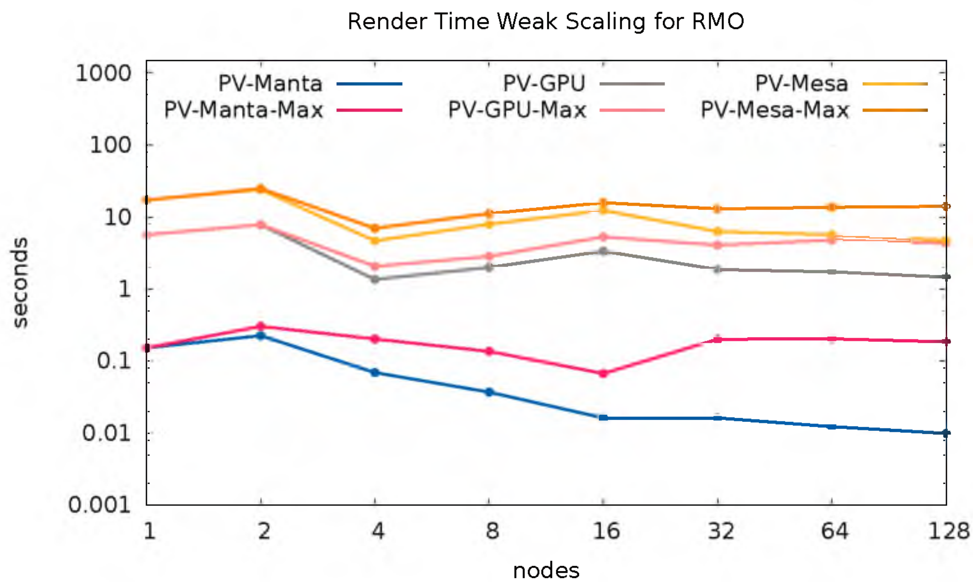


Figure 3.6. Weak scaling timings of an isosurface of the RMO dataset in ParaView. Geometry is added with each additional node so that the geometry per node remains roughly constant.

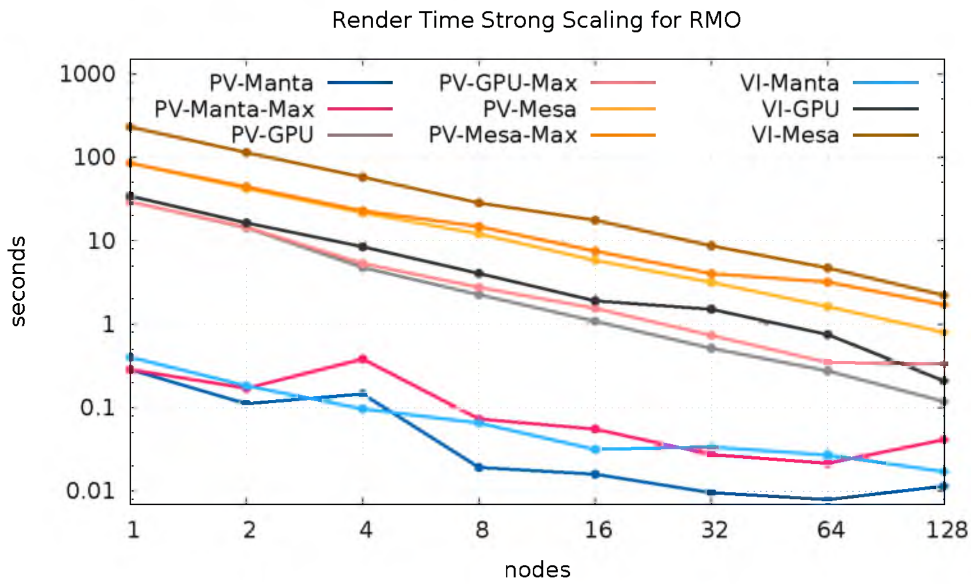


Figure 3.7. Strong scaling timings of an isosurface of the RMO dataset.

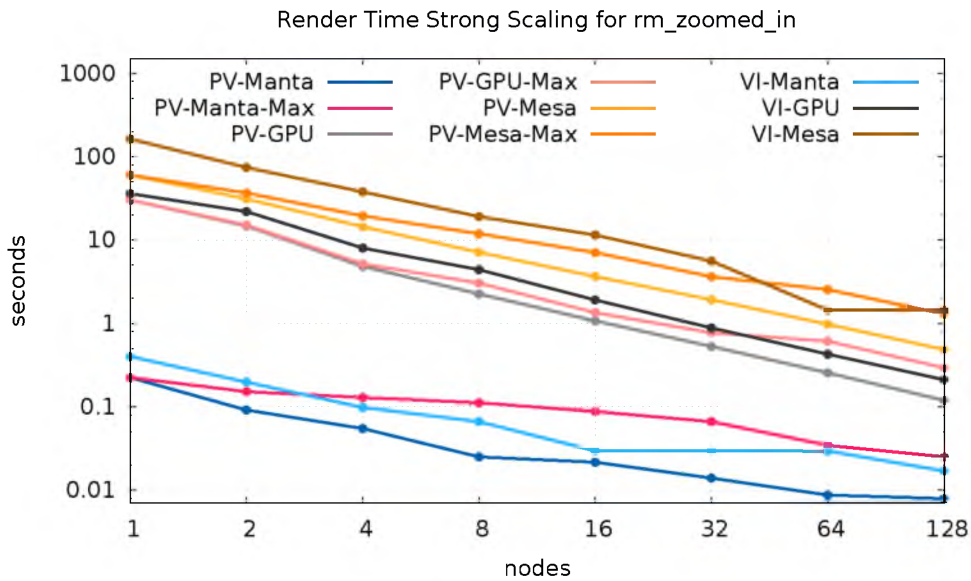


Figure 3.8. Strong scaling timings of an isosurface of the RMI dataset.

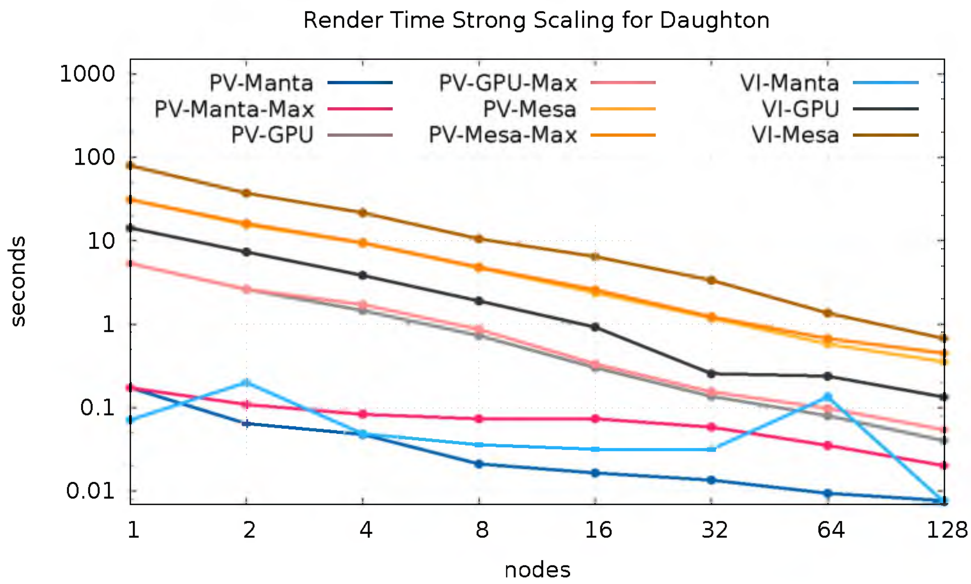


Figure 3.9. Strong scaling timings of the VPIC dataset.

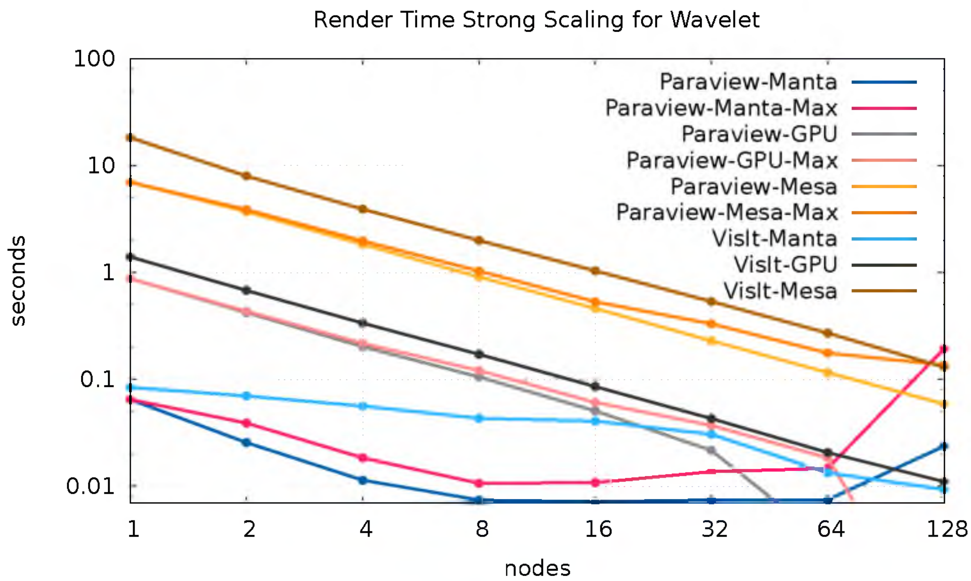


Figure 3.10. Strong scaling timings of the wavelet dataset.

CHAPTER 4

RAY TRACING THROUGH OPENGL INTERCEPTION

4.1 Interception Implementation

In the previous chapter we demonstrated that ray tracing integrated into common visualization tools presents a workable interactive rendering solution with many advantages over the existing rendering pipeline in terms of speed and rendering quality. Developing these implementations for each visualization tool is often a daunting development effort and updates to the tools often require considerable modifications to the the ray tracing implementation. Furthermore, some visualization tools such as EnSight are not open-source and thus, modifying the rendering back-end is infeasible. We therefore present a program-agnostic implementation of OpenGL using ray tracing, GLuRay, which does not require code modification by interpreting calls to the OpenGL library. With GLuRay, CPU ray tracing in visualization tools is a scalable, interactive, and high quality alternative rendering solution which can currently be used across a wide range of tools with no additional development effort by simply linking with a different library.

4.1.1 Intercepting OpenGL calls

GLuRay operates as a ray tracer that runs with existing OpenGL programs. Implementing GLuRay required creating a false OpenGL library and dynamically linking it with a host program at run-time using LD_PRELOAD or dlopen. This library maps calls from the rasterization algorithm present in OpenGL into a ray tracer.

In order to capture OpenGL API calls, an OpenGL implementation was created based on the official OpenGL specification. The open-source OpenGL debugging tool SpyGLass was used as a basis for the program [55]. Some calls are ignored with no direct mapping such as clearing the depth buffer, some are passed on to the system's implementation

library such as GLX calls, and some are sent to GLuRay's ray tracing implementation for tracking state or rendering. Function calls, which are mapped to ray tracing, include calls to modify transformation matrices, material properties, light properties, geometry information, and rendering attributes. Calls which are passed to the normal OpenGL library include calls such as glDrawPixels and glXSwapBuffers. When tracking these calls, it helps to think of OpenGL as a state machine. Each call either affects some given state or returns information about the given state. In OpenGL, this state affects how a polygon is drawn each time a draw call is made using either glVertex calls in immediate mode, glCallList for display lists, or a more modern glDrawArrays or equivalent call. When ray tracing, multiple draw calls need to be avoided as much as possible as the rendering time is $k \cdot O(log(n))$, where n is the amount of geometry and k is the screen size. For each draw call, the time complexity roughly linearly increases as k expands. Therefore, each draw call is recorded and not rendered until the system determines a draw is required for the entire scene. This is determined through calls to glXSwapBuffers, glFlush, glFinish, or glClear depending on the application. This has the potential to break certain behaviors such as depth-ordered blending modes; however, common uses of this are for transparency which can be handled by using transparent material properties with ray tracing and have not posed a problem for our current implementation.

In the serial implementation of GLuRay, rendering occurs as soon as a draw is required. Acceleration structures are built as needed which are instanced with their transforms, lighting information, material parameters, and geometry. When a render is requested by the host program, the rendered scene is drawn into the OpenGL context, and data is cleared as shown in Figure 4.1.

4.1.2 Asynchronous Rendering

To speed up interactive rendering, the option to add a one frame lag between rendering calls was added. This alleviates the idle time of GLuRay waiting for rendering calls from single-threaded applications. When a draw call is made, the previous frame is copied to the framebuffer and returns. While the next batch of OpenGL calls are being made, the multithreaded system is rendering and building acceleration structures for the previous

frame. For this we use a packet based Bounding Volume Hierarchy, BVH [78]. To further decrease the time to build acceleration structures, an approximate BVH can be used which builds faster but gives moderately slower runtime performance. This effectively gives a variable which can be changed depending on whether a system needs to be more interactive for updates to the underlying geometry or faster for changes to the camera. This system is shown in Figure 4.2.

4.1.3 High Quality Rendering

Ray tracing allows for advanced effects such as global illumination, accurate reflections, depth of field, soft shadows, transparency, and refraction to name a few. These techniques can be handled through other means; however, our implementation provides an intuitive implementation of the rendering equation using light rays which can be used for publication quality images. Manta supports path tracing; however, it was not used in our testing. Figure 4.3(a) shows GLuRay running within the visualization tool ParaView rendering a Richtmyer-Meshkov instability with ambient occlusion. Camera manipulations operate just as they would with OpenGL and material and light properties are updated whenever OpenGL state changes are made in the host program. Not all material properties can be provided through OpenGL alone, such as refractive indices of glass objects or ambient occlusion options. Such additional material properties proprietary to GLuRay are exposed through an external GUI application shown in Figure 4.3(b). Changes are applied globally to all objects in the current scene. Modifications are broadcast to running programs through TCP sockets over the localhost.

Figure 4.4 shows the comparison between a phong shading of a human skull and the same rendering with ambient occlusion added. Ambient occlusion provides depth cues not present in the phong shaded image by occluding light blocked by nearby geometry. Perceptual user studies have validated that more realistic lighting using approximations of global light can aid comprehension of complex features in data [29] when compared to purely local lighting algorithms such as phong shading. Figure 4.5 shows a side-by-side comparison of an OpenGL rendering within ParaView of an aluminum ball hitting an aluminum plate and GLuRay rendering the same dataset within the same program using

additional effects. Data loading, interaction, and rendering calls, were all done within the host program without noticeable differences to the user until special rendering modes were selected. Figure 4.6(b) shows an astrophysics simulation of magnetic reversal in a solar-type star rendered within VAPOR using GLuRay [10]. Ambient occlusion enhances streamlines while reflections and soft shadows add to the realism of the rendered image compared to only using local lighting as shown in Figure 4.6(a). Secondary rays are only supported on shared-memory systems or GLuRay's ray-parallel distributed mode which was implemented similar to Ize et al. [38] but as of this writing this implementation only works with replicated data on each node.

4.2 Results

We evaluated the rendering performance of GLuRay compared to hardware-accelerated OpenGL and software OpenGL rendering using Mesa on a single node of the rendering cluster Longhorn, an NSF XD visualization and data analysis cluster located at the Texas Advanced Computing Center (TACC). Longhorn has 256 4X QDR InfiniBand connected nodes, each with 2 Intel Nehalem quad core CPUs (model E5540) at 2.53 GHz and 48-144 GB of RAM. Each node of Longhorn also has 2 NVidia FX 5800 GPUs. We used datasets of varying sizes, including a synthetic wavelet dataset, a dataset from Los Alamos's plasma simulation code VPIC, a simulation of magnetic reversal in a solar-type star, and a timestep from a Richtmyer-Meshkov instability (RM) simulation rendered with two different views. All images were rendered at 1024x1024 resolution with the same settings and views across VisIt and ParaView, where applicable. ParaView, VisIt and Ensight are built with Mvapich2 1.4 which is provided on Longhorn. ParaView was run on Longhorn using $tacc_x run$, pvbatch, and offscreen rendering for the GPU and Mesa render timings. GLuRay was run using vglrun, except for the scaling study in which case pvbatch, $tacc_x run$ and offscreen rendering were used. All benchmarked timings for GLuRay have the same illumination model as OpenGL with local lighting only and no shadows computed. Benchmarks were conducted with up to 6000 frames and an initial warmup period. This shows an expected frame rate from camera exploration of an isosurface which is the focus of our study, but not necessarily what may be achieved

when exploring isosurface values or other updates which require rebuilding acceleration structures each frame.

4.2.1 Datasets

- **Astrophysics** The astrophysics dataset shows a sun-like star visualized with 48000 streamlines representing magnetic field lines [10]. Figure 4.6(a) shows a rendering of this dataset in VAPOR.
- **Wavelet** The wavelet triangle dataset is a computed synthetic dataset source released with ParaView. We generated a 201^3 dataset and then calculated as many isosurfaces as needed to produce a certain quantity of triangles. The isosurfaces are nested within each other. Images produced with 16 million triangles are shown in Figure 4.6(c).
- **VPIC Visualization** Using a single time-step from the VPIC plasma simulation, we calculated an isosurface and extracted streamtubes that combined, totaled 102 million polygons. A view of this dataset rendered in ParaView can be seen in Figure 4.6(d).
- **Richtmyer-Meshkov Instability** The Richtmyer-Meshkov instability simulation, RM, presents a commonly used scientific dataset. We created a polygonal representation with an isosurface from a single time-step resulting in 316 million triangles. To understand the behavior of the ray tracer we have rendered both a zoomed out view of the RM dataset, RMO, seen in Figure 4.6(f) and a closeup view in Figure 4.6(e), RMI. The closeup view shows a smaller portion of the overall data, however it also takes up more screen space.

4.2.2 Scientific Visualization Programs

- **VAPOR** VAPOR is a visualization program developed by NCAR and designed for oceanic, atmospheric, and solar research focusing on isosurfaces, volumes and streamlines. Rendering is done through vertex arrays and display lists. Version 2.0.2 was used in our timing study.
- **EnSight** EnSight is an in depth commercial visualization package featuring volume rendering, streamlines, glyphs, and contours to name a few of the rendering modes

supported. Rendering uses immediate mode rendering or display lists. Version 9.2 was used for our benchmarks.

- **ParaView** ParaView is a distributed visualization program built around VTK and designed for use on large cluster environments. Rendering is done through immediate mode rendering in OpenGL or display lists. For our tests we used the most recent available version when our tests were conducted, 3.11.0.
- **VisIt** VisIt is a distributed visualization program built around VTK for use on large clusters similar to ParaView. We utilized visit 2.4.0 for our tests.

4.2.3 Performance Scaling

We tested four visualization programs with three different rendering modes: software ray tracing using GLuRay, OpenGL software rasterization using Mesa, and hardware-accelerated OpenGL. Mesa is not multithreaded nor the fastest available software rasterization package; however, it is the only one supported as build options with ParaView and VisIt and is commonly used when GPUs are not available. The OpenGL implementation within these programs is a brute-force implementation with no advanced acceleration methods used. This comparison is therefore, not a comparison of the ultimate potential of rasterization versus ray tracing algorithms, but rather a real-world study of their existing performance in commonly utilized tools with real world problems.

To test the scaling of these methods, the synthetic wavelet dataset was scaled from 1 to 256 million triangles in ParaView and VisIt. The dataset is shown with 16 million triangles in Figure 4.6(c). Mesa manages over one fps in ParaView only when the triangle count remains under 2 million triangles. The hardware-accelerated OpenGL implementation retains interactive performance at 1 to 2 million triangles in ParaView and VisIt; however, performance degrades roughly linearly with triangle count. Slow GPU performance may be due to ParaView's rendering code which was built around immediate mode rendering and accelerated through a display list. This leads to a large number of function calls for the initial build and updates compared with using vertex buffers, which can specify large numbers of vertices with a single function call. We found rendering performance to be roughly a tenth of what it could be by using vertex buffer objects or multiple display lists

in our tests. Another issue is that display lists on Longhorn crash after about 32 million triangles. This could be fixed by splitting up the data across multiple display lists in a custom implementation; however, immediate mode rendering was used above 32 million triangles for the hardware-accelerated runs in our tests as this method worked with an unaltered code base. VisIt was slower than ParaView at lower geometry counts in our tests due to increased overhead for each frame, which accounts for the slightly slower GLuRay performance with VisIt. Additionally, VisIt uses a textured colormap which was set to interpolate between two identical colors for our tests whereas, ParaView uses a single solid color resulting in fewer OpenGL calls. GluRay shows sublinear performance degradation when the triangle count increases, scaling well into the hundreds of millions of triangles and only dropping below 5 fps past 128 million triangles in ParaView. Performance sometimes decreases with increased geometry counts. This may be a caching issue with some data sizes exhibiting better caching behavior than smaller triangle counts and differences in background space around the datasets which can be easily culled by the ray tracer.

Table 4.1 shows timings for various datasets over multiple applications. All times are averages of several render frame times after a few initial warmup frames and include all host program overhead which is the most accurate view of performance for an asynchronous renderer. Seven render threads were used on an 8-core node, leaving one main thread for processing OpenGL state changes and image display. GLuRay is faster in all cases we tested with and achieves better than 300x speedup over Mesa for the RMO dataset and a 62x speedup over the GPU implementation for VPIC in ParaView. GLuRay has at least a 90x performance increase over Mesa while speedup over the GPU ranges from 3.57x speedup rendering the astrophysics dataset in VAPOR, up to a 123.85x speedup over the GPU rendering the RMO dataset when zoomed out in ParaView. The RM dataset zoomed in and out achieved similar performance for each view when rendered with the GPU. GLuRay and Mesa, however, have differing results between the two views of the RM dataset. This is because the geometry took up a larger portion of the scene in GLuRay when zoomed in, and in Mesa this is likely showing that clipping triangles outside of the viewport gave a greater speedup for Mesa than for the GPU. GLuRay is

the only rendering method to achieve above 5-10 fps for interactive rendering for the 16M triangle wavelet dataset with 12.33 fps in ParaView and 9.69 fps in VisIt. GLuRay achieved 5.02, 6.94, and 2.55 fps rendering the VPIC datasets in ParaView, EnSight, and VisIt which was as much as a 62.75x performance improvement over OpenGL. Timings of the same dataset are different across different programs as each program incurs its own overhead as well as differing OpenGL calls per frame which affects the GPU, GLuRay, and to a lesser extent, the Mesa performance. In the case of the astrophysics dataset with VAPOR, only a 3.57x speedup was achieved over the GPU. This is likely due to VAPOR's use of vertex buffers instead of the glVertex calls used by the other programs which greatly accelerated the GPU rendering.

The time to render the first frame is usually a combination of data loading, geometry generation, passing data to GLuRay, and finally, acceleration structure construction and rendering. Approximate acceleration structure builds decrease rendering performance but speed up build times. The overhead from the acceleration structure builds using approximate builds for the wavelet dataset is shown in Figure 4.7. This overhead varies from less than a second at 1 million triangles to over a minute with 256 million triangles; however, this time is still less than rendering a single frame with software Mesa in all cases. Users of visualization programs typically generate an isosurface of data which is then explored through transformations to the camera. A duration of 10 seconds moving the camera with a render time of less than 0.1s per frame thus produces over a 100 rendered frames, making the time to process a single acceleration structure insignificant in overall runtime. Cases where geometry is animated, however, could need to build acceleration structures for each rendered frame which could limit performance for our program. When textures or color arrays are used in OpenGL, updating colormaps does not require rebuilding geometry. However, when colors are built into display lists those display lists must be rebuilt, which requires acceleration structures to be updated.

The overall build and runtime behavior of GLuRay is shown through Gantt charts in Figure 4.8, which illustrate a run from program start to end in ParaView with an 8 million triangle wavelet dataset and a closeup of rendering behavior from the same run at the bottom. There are nine rows in total, one for the main thread at the top and eight render

threads below. The empty space at the beginning shows program startup and idle GLuRay threads waiting on the host program to send data through OpenGL function calls. The brown line after data loading shows the host program making millions of OpenGL calls such as glVertex and glNormal, which copy geometry into GLuRay. The following blue bars display the construction of acceleration structures and then turn gold for rendering. A lazy system was used for BVH construction, resulting in some of the rendering time being used for additional BVH processing which can be seen in the initial setup phase. As shown in the Gantt chart, the time to build acceleration structures and setup the first frame in the rendering threads is roughly equivalent to the time the program spends specifying the geometry through OpenGL. Grey denotes time spent waiting for the render threads to finish in the main thread. In between each render call, global acceleration structures must be updated and images copied to the framebuffer. The global acceleration structure takes into account the transforms applied to each stored set of geometry, such as changes to the ModelView matrix applied to the geometry from glCallList in OpenGL. The closeup of rendering performance at the bottom shows that the rendering threads are well utilized with very little overhead for building acceleration structures in between rendering or downtime waiting for updates.

In order to benchmark strong scaling across multiple nodes, the 316 million triangle RMO dataset was rendered on 1 to 64 nodes using the Longhorn visualization cluster and the parallel visualization tool, ParaView, using the same camera position shown in Figure 4.6(f). For parallel rendering, ParaView uses sort-last compositing through the IceT library, which introduces an additional compositing step at the end of every frame. Howison et al. [33] used a sort-last compositing algorithm on the Jaguar supercomputer, where they found that compositing was their biggest bottleneck for a high resolution image. Their maximum achieved frame rate was 2 fps for a 21 million pixel image over 216,000 cores. Assuming this performance scales down to a 1 million pixel image, the maximum frame rate from compositing would be approximately 42 fps. We, therefore, aim to approach real-time rendering rates using our method for image sizes of 1024^2. Render times are reported from hardware-accelerated OpenGL and GLuRay in Figure 4.10. In comparison to the single node timings shown in Table 4.1, the scaling runs for ParaView

use the parallel version of ParaView, pvbatch, with offscreen rendering enabled and Longhorn's batch configuration script which decreases overhead from image display. To generate the correct image when needed by the compositor, rendering for GLuRay was modified to render upon calls to glCallList or glReadPixels, which IceT uses to gather the rendered scene for compositing. Enabling GLuRay's frame lag would result in a frame rate that is approximately equal to the maximum of the render time and the compositing time instead of the aggregate of the two. Therefore, in order to time just rendering, GLuRay was run without a frame lag such that asynchronous rendering was not utilized.

In our strong scaling study, GPU-accelerated average render times drop from 29.46 seconds to 0.28 seconds from 1 to 64 nodes, respectively, as the average triangle count drops from 316 million triangles to about 5 million triangles per node. Rendering times for GLuRay start at 0.21 seconds on a single node and decrease to 0.037 seconds for 64 nodes, which resulted in an overall frame rate of 18.47 fps on 64 nodes with compositing and other overhead within ParaView. The GPU render times at 64 nodes do not reach the performance of a single node with GLuRay, while GLuRay continues to increase in performance with each node added. GLuRay render times do not decrease as dramatically as the GPU render times with each node added because the acceleration structures used scale well with increasingly large amounts of geometry as seen in Figure 4.9. Therefore, decreasing the triangle count per node does not impact performance as much as OpenGL. In order to achieve better work distribution with GLuRay, view-dependent distribution of the data would be needed. Running GLuRay over programs which utilize sort-first data distribution, or a hybrid technique such as the one used by Nouanesengsy et al. [60], could provide better scaling behavior for ray tracing. Ize et al. used a paging approach which achieved up to 100 fps for a two megapixel rendering of the RM dataset using 60 nodes of a cluster using ray-parallel work distribution [38].

4.3 Summary

In this chapter we have shown that current rendering algorithms used in many common scientific visualization tools do not scale with increasingly large geometry counts and often fail to provide interactive rendering rates that facilitate data exploration

on machines lacking hardware acceleration and in some cases, even with hardware accelerated rendering. In the previous chapter we described an implementation of ray tracing integrated into the source code of two visualization tools. In this chapter we have shown that interactive performance with ray tracing in these and other tools can be achieved without code modification by using an OpenGL interception library. In some cases, GLuRay achieved over 300x the performance over software rasterization using Mesa and in many cases we gained significant speedups over even hardware accelerated display lists within several of the most commonly used visualization tools. This solution further enables the rendering of publication quality images without the need to export data to rendering tools, which are foreign to users and interrupt their workflow. GLuRay does not enable ray tracing specific features outside of the fixed-function OpenGL pipeline from the host programs that custom software integration may otherwise provide. However, GLuRay gives the benefits of fast, high quality rendering with no additional effort and presents a workable solution with little performance overhead where code modification is not possible or desired.

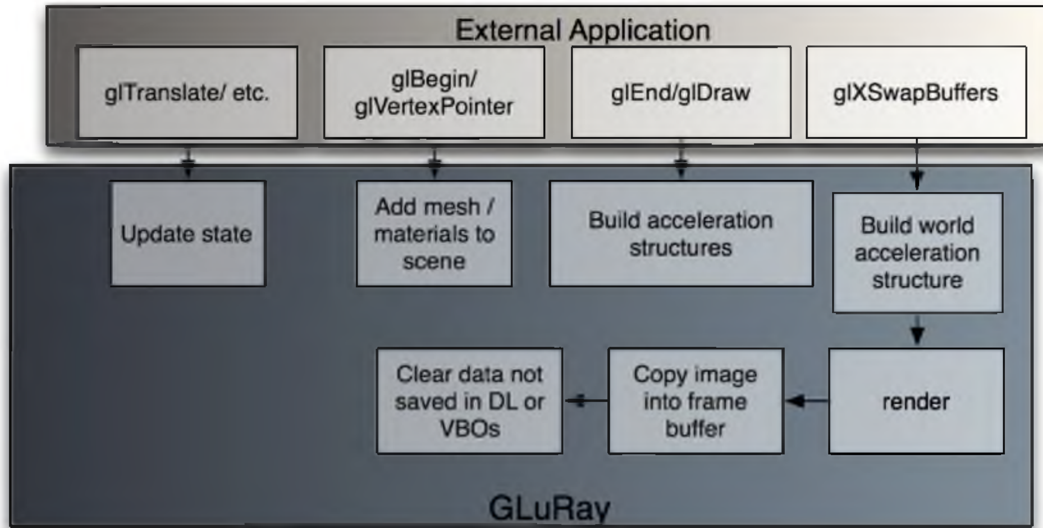


Figure 4.1. Sequential architecture of GLuRay.

DataSet	Triangles (M)	FPS	Mesa	GPU	Speedup vs. Mesa	vs. GPU
PV-Wavelet	16	12.33	0.13	1.22	94.85	10.11
VI-Wavelet	16	9.69	0.085	0.79	113.99	12.27
PV-VPIC	102	5.02	0.026	0.08	193.08	62.75
VI-VPIC	102	2.55	0.012	0.069	212.49	39.96
Ensight-VPIC	102	6.94	0.02	0.23	347.15	30.19
PV-RMI	316	2.53	0.001	0.03	180.71	93.70
PV-RMO	316	3.22	0.009	0.03	357.78	123.85
VAPOR-Star	86	2.39	0.03	0.67	95.60	3.57

Table 4.1. Performance timings for various datasets across different applications with varying amounts of triangles specified in the millions. PV signifies ParaView and VI refers to VisIt. RMI and RMO are the Richtmyer-Meshkov datasets zoomed in and out, respectively. GLuRay achieves significant speedups in all runs tested with large polygon counts.

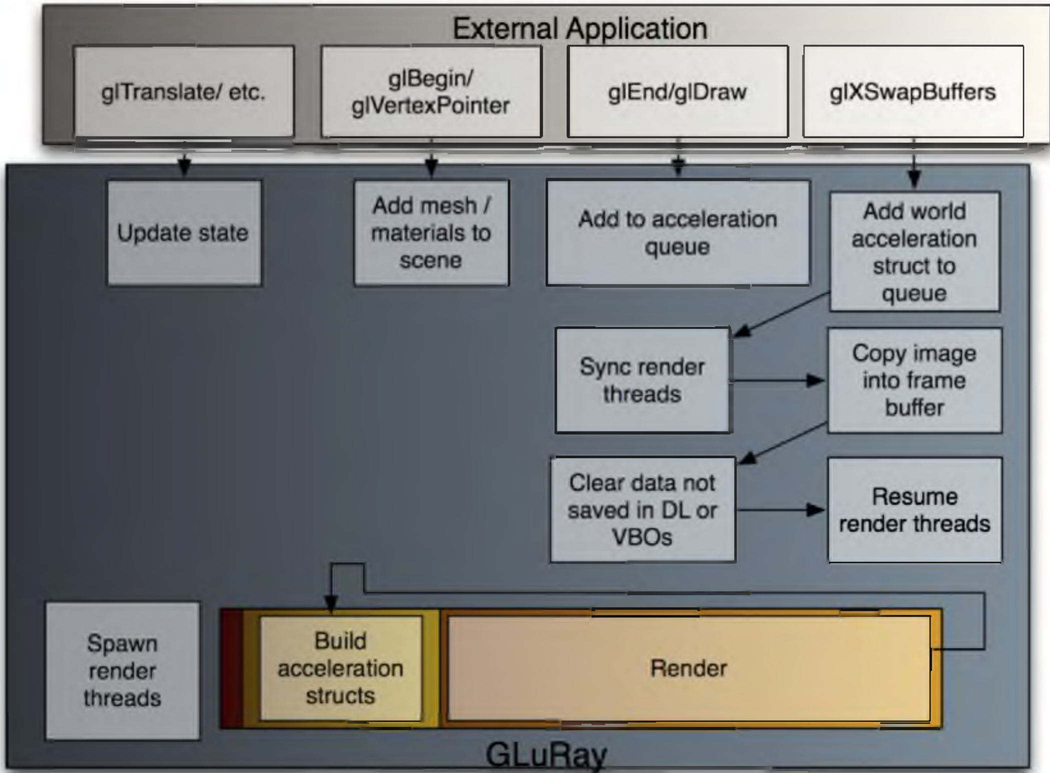
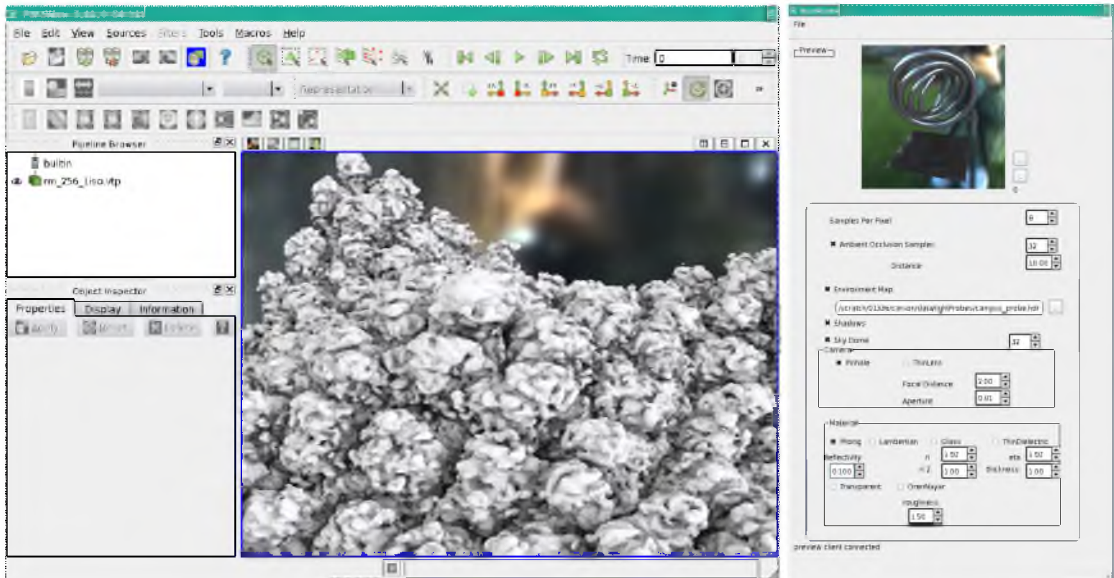


Figure 4.2. Parallel architecture of GluRay.



(a) (b)

Figure 4.3. GLuRay running within ParaView (a) and an external GUI (b).

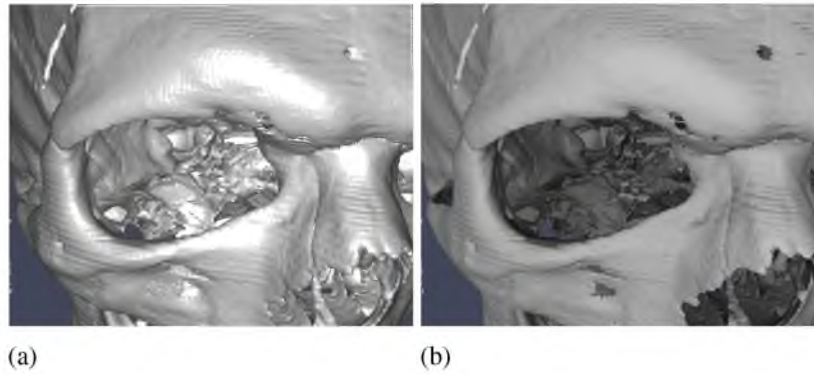


Figure 4.4. Shading effects such as ambient occlusion shown in (b) add more depth cues compared to a standard phong shading technique (a).

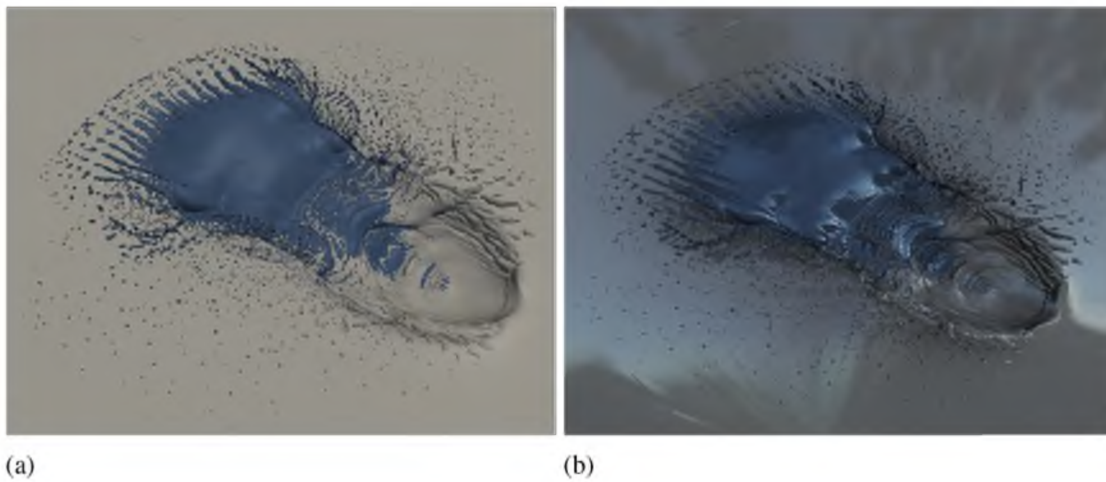


Figure 4.5. Rendering of an impact dataset with ParaView using OpenGL (a) and a rendering using GLuRay demonstrating secondary effects such as reflections, shadows and ambient occlusion (b).

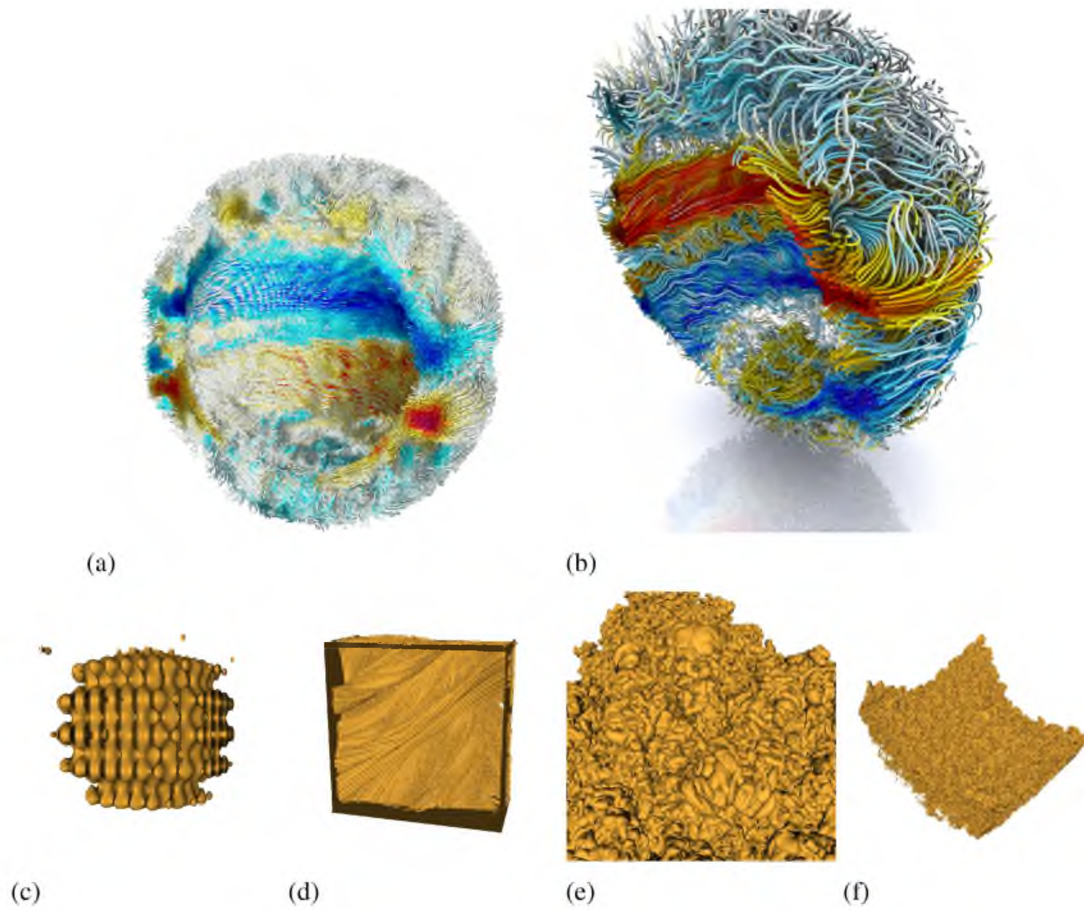


Figure 4.6. Magnetic fields from an astrophysics simulation colored with streamlines (a), the astrophysics dataset rendered with advanced effects (b), a wavelet contour (c), VPIC Plamsa simulation (d), RMI (e), and RMO (f) datasets used for benchmarking.

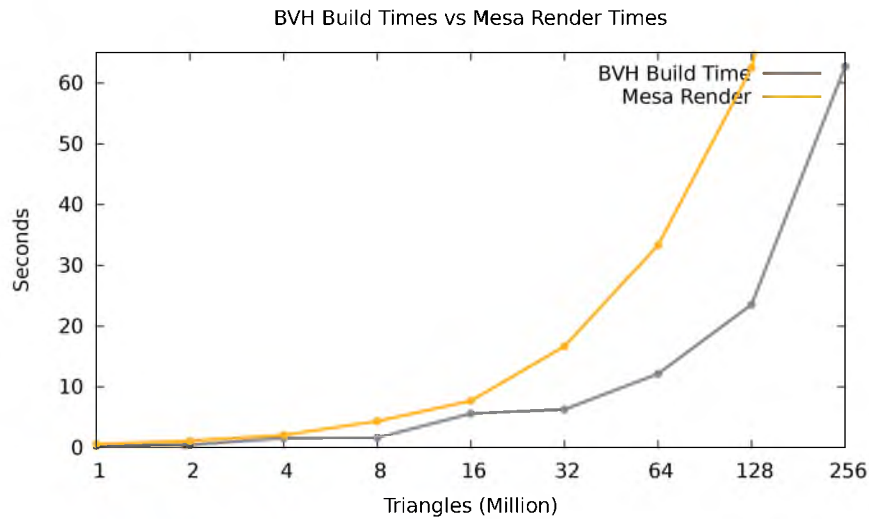


Figure 4.7. BVH build times for the wavelet dataset compared to Mesa rendering time for a single frame.

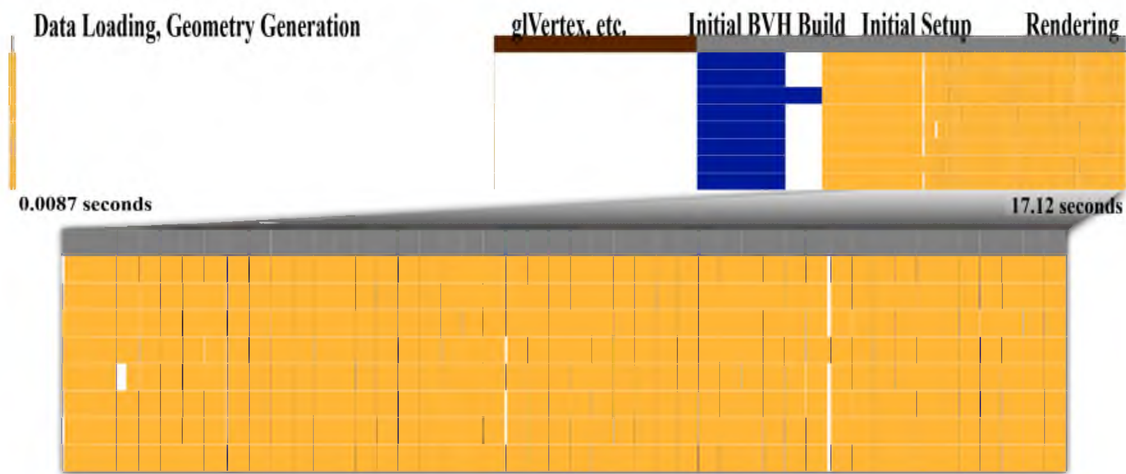


Figure 4.8. Gantt charts showing a rendering of a wavelet dataset with 8 million triangles. An overall view of setup time, geometry transfer, acceleration builds and rendering is shown in the top runs while a closeup of the rendering is shown in the bottom. The gold color displays the efficiency of the asynchronous renderer, while the blue displays the cost of BVH builds in relation to data loading and geometry specification through OpenGL, shown in brown.

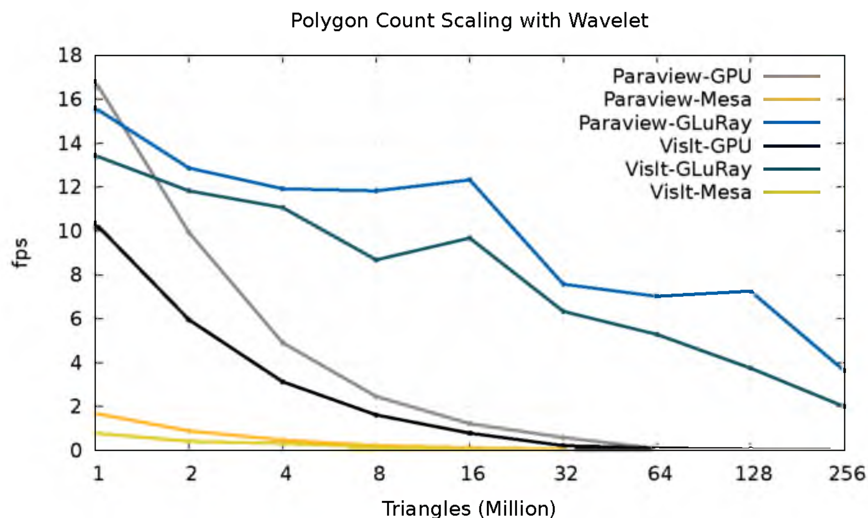


Figure 4.9. Performance timings for increasing triangle counts for a wavelet dataset as seen in Figure 4.6(c).

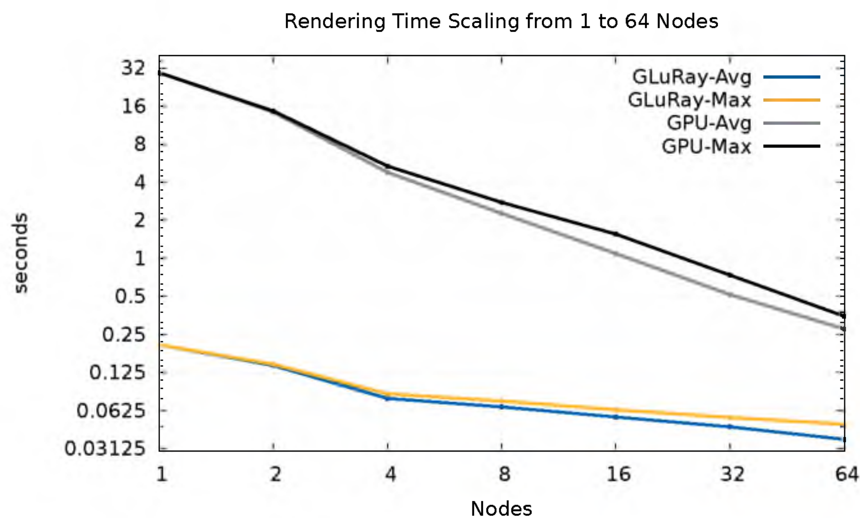


Figure 4.10. Rendering times for the RMO dataset in ParaView using the Longhorn visualization cluster. Avg denotes the averaged render times across every node while Max designates the maximum render time across all nodes.

CHAPTER 5

COMPUTATIONAL PHOTOGRAPHIC METHODS

In the previous chapters we have shown that ray tracing in visualization tools enables interactive rendering of large-scale data while also providing the utilization of lighting models which more closely simulate real-world lighting. Physically-accurate rendering allows for enhanced understanding of underlying data with enhanced depth-cues to users, but also enables the synthesis of visualization techniques used in experimental fluid flow such as schlieren, shadowgraph, and interferometry. These techniques trace light refracting or phase-shifting through inhomogeneities in media, giving an overall picture of changes in data. Previous interactive implementations of these techniques have used slice-based rasterization volume rendering techniques without calculating light paths, providing an inaccurate approximation of the technique. Tracing light through the media allows for physically-accurate reproduction of experimental techniques giving domain scientists working in fluid flow visualization a familiar rendering technique for simulated data. This chapter presents an implementation of ray casting several photographic techniques on GPUs achieving interactive rendering rates for rapid changes to rendering and viewing parameters with progressive refinement.

5.1 Background

Shadowgraph techniques have been used for centuries to look at flows that are not visible to the human eye, such as heat dissipation or shock waves [69]. Small changes in inhomogeneous media do not scatter light to a large degree but it was noticed that shining a bright light through the disturbance will produce a clear image of the flow by looking at the shadows formed from light refraction. In a shadowgraph system, this refracted light is imaged on a film plane. Figure 5.1 shows the optical setup of a typical shadowgraph

system. A light source is filtered through a slit apparatus, thus producing a small point light source. Nearly parallel rays are sent through the test area and focused onto a film plane. Light that was refracted in the test area will group together to produce bright areas in the film plane or disperse and create darker regions. Shadowgraphs only look at changes in the second derivative and are a poor indicator of the amount or direction of refraction. If all rays were refracted the same amount in the same direction, then the resulting image would be identical to a translated image of no refraction at all. Schlieren photographic techniques provide additional information by introducing a one dimensional cutoff that shifts intensity values based on the amount and direction of displacement at the focused cutoff region. In Figure 5.2, light rays traverse the flow from a light source similarly to the shadowgraph setup. In the schlieren system the light source is then refocused in a small area and a cutoff is inserted to reduce light from the light source. A vertical knife-edge is inserted at the center of the refocused light source. If no light is refracted, the knife-edge reduces the light source by half, resulting in a gray image. Refracted light causes shifts in the focused image of the original light source resulting in more or less of the focused light being blocked by the cutoff. If the focused image is shifted down the resulting region is darker, and if shifted up, then more of the original light gets through to the film plane. A knife-edge cutoff thus provides information about the amount of light shifted along a single axis. Another common type of cutoff is a circular cutoff that shades the image based on the amount of displacement without the directional information of the knife-edge. Color filters can also be used as a cutoff to produce colors based on the direction of displacement. An illustration of a color filter is shown in Figure 5.3. Whereas a knife-edge cutoff only gives information about the amount of displacement along one axis, color can give two dimensional information about the direction of displacement.

Interferometry differs from schlieren and shadowgraph images by looking at phase shift instead of refraction. When light travels through a disturbance and encounters a change in refractive index, the speed of that light changes resulting in a phase shift [71]. The idea behind interferometry is to directly measure this phase shift and display it, providing a picture of changes in refractive index. On the experimental side, this method allows for the direct calculation of refractive indices instead of looking at changes in

gradient values as in schlieren and shadowgraph images. The optical setup required is described in Figure 5.4. The setup starts with a beam of light typically generated by a laser because of the polarized parallel light at homogeneous frequencies. A reference beam is also created by splitting the light beam before hitting the inhomogeneity. This reference beam can be used to measure the phase shift of the main beam by comparison to the reference beam. The main light beam is sent through the inhomogeneity where differences in refraction will produce phase shifts. Where the phases line up, bright bands are created and where they conflict, dark bands emerge. In Figure 5.5 this is demonstrated by our computed image of a coal fire through the tight small bands in the center of the coal fire and larger dissipated fringes as the flame disperses.

In the perfect case, where the test beam and reference beam are perfectly aligned and no disturbances are intercepted, no fringes should appear. This is known as infinite-fringe interferometry. Finite-fringe interferometry puts the test beam and reference beam at slight angles, producing fringes even when there is no difference in the phases of the two beams. This is a more commonly used technique as finite-fringe interferometry allows for the determination of the phase shift from the images produced on the experimental side. The spacing of these beams produced through finite-fringe interferometry can be calculated as a measurement of χ by:

$$\chi = \frac{\lambda}{2\sin\frac{1}{2}\gamma} \tag{5.1}$$

where λ is the wavelength of the light and γ the angle of beam intersection [71]. The phase shift, ϕ can be calculated as a function over the refractive index field n as:

$$\phi = \frac{2\pi}{\lambda}\int_{z_1}^{z_2}(n-n_0)dz \tag{5.2}$$

where n_0 is the reference refractive index, which is the refractive index that the reference beam is hitting [84]. In many cases this is simply the refractive index of air. This equation assumes a line of sight traversal with no refraction; however, in the computed case the integration over the z axis is easily adapted to the integration over a bending light path

using a piecewise linear approximation [57]. A fringe is produced whenever a phase shift of 2π or an optical distance of λ is encountered for the infinite-fringe case [80]. For the finite-fringe case the phase shift and the angle between beams must be taken into account to calculate the resulting fringes.

5.2 Computational Methods

Our method for computing schlieren images relies on a number of acceleration techniques for tracing photons through inhomogeneous media. The overall series of steps used by our rendering pipeline are presented in Figure 5.6. The precomputation steps utilize the CPU while the photon tracing and filtering stages are done on the GPU using CUDA [61], which gives us the flexibility to arbitrarily store array values (a scatter operation) without relying on the framebuffer. This is important for our technique as the final photon positions cannot be predicted. The ray casting algorithm is ideally suited for the GPU, since each ray can run concurrently in its own thread and data locality can be exploited from nearby rays. This coherency benefits from CUDA's single instruction multiple thread (SIMT) architecture as many threads operate on the same data. The parallel nature of the computation benefits from the GPU's parallel architecture as long as the data can be stored on chip. CUDA's OpenGL interoperability also allows us to filter the resulting image and display to the screen without copying it back to the host CPU. Time-varying flow fields are rendered one frame at a time with precomputation being computed before each frame.

5.3 Precomputation

The precomputation stages are required to compute the refractive indices and gradient from related fields as well as construct an octree to accelerate ray traversals during runtime. These stages allow accurate and fast computations of light refraction through the flow at later stages of the pipeline. The precomputed data needed to be passed to the GPU for rendering consists of a 3D texture of refractive indices, a 3D texture of gradient values, and an array of randomly generated floats.

5.3.1 Computing the Refractive Index

In order to accurately simulate a schlieren photograph, it is important to use correct indices of refraction. The indices of refraction in a medium can be computed from a combination of several other scalar fields such as temperature, pressure, and humidity using Ciddor's method [17]. Ciddor's method has been adopted by the International Association of Geodesy (IAG) as a standard method for computing index of refraction. It will not be reproduced in its entirety here, due to the complexity of the method, but we provide a brief overview. Ciddor presents a method for computing accurate refractive indices from air [17]. The method is composed of a 10 step process that calculates the densities and compressibility of air at certain conditions in order to compute the refractive index. While it is beyond the scope of this dissertation to reproduce the entire derivation here the method is largely governed by Equation 5.3:

$$n_{prop} - 1 = (p_a/p_{axs})(n_{axs} - 1) + (p_w/p_{ws})(n_{ws} - 1) \tag{5.3}$$

This equation calculates the refractive index by multiplying the proportion of dry air by the refractivity of dry air and adding in the portion of wator vapor by the refractivity of pure wator vapor. In Equation 5.3, n_{prop} is the refractive index that is being calculated, p_{axs} is the density of dry air at 15 °C, and p_{ws} is the density of pure water vapor at 20 °C. p_a and p_w are the densities of the dry air and water vapor components. n_{axs} and n_{ws} are the refractivity of dry air and pure water vapor.

Figure 5.7(a) shows a heptane data set with indices of refraction computed from pressure and temperature fields. The resulting time-varying scalar fields of refractive indices, f, will later be used for computing light paths through the flow. The gradient of f, ∇f, is also computed as a preprocessing step using finite differences.

Alternatively, the Gladstone-Dale relation provides a method for computing the refractive indices from density fields [57]. Because we are working with gasses we use the abbreviated form:

$$n - 1 = Kp \tag{5.4}$$

K defines the Gladstone-Dale constant, p the sample density and n is the refractive index

we want to compute. For data with more than one material type, the Gladstone-Dale constant will need to be interpolated between the different materials using a mixture fraction field. As an example, one can vary the value from pure air to pure helium based on a provided mixture fraction. K varies by temperature and wavelength but with the temperature around 290 Kelvin and assuming our light has a constant wavelength of $0.633 \mu m$, we then know that K_h for helium is approximately $0.196 cm^3/g$ and K_a is $0.226 cm^3/g$ for air. If m_h is the volume fraction of helium in the mixture and m_a is the fraction of air, then n can be found by:

$$n = (K_h \cdot m_h + K_a \cdot m_a) \cdot p + 1 \tag{5.5}$$

5.3.2 Octree

Many flow datasets contain large regions of nearly homogeneous refractive indices but only changes in the refractive index are of interest to schlieren and shadowgraph imaging. A computational schlieren system can attain significant speedup by utilizing space-skipping techniques similar to empty space-skipping commonly employed in volume rendering as shown by Sun et al. [74]. Instead of skipping over empty-space in the data, we compute regions of nearly homogeneous refractive indices in the data, which determine how big of a step through the data can be taken before reaching a significantly large change in refractive index.

The octree is computed as a min-max octree with a tolerance value, t, that determines the level of each region and thus, the size of the area that can be skipped over. Only the octree level values, j, are stored in the resulting 3D texture, which has the same dimensions as f. The min-max octree structure is built to determine how large the homogeneous regions are but no intermediary nodes are stored in a texture so that lookups into the acceleration structure will not require a tree traversal. When traversing through the data, a lookup into the texture will return the octree level for a sample. For example, if a lookup returns a level $j = 2$, then the homogeneous region is of size 2^2x the texel size and this entire region can be skipped without encountering a refractive index value that is more than t from the current voxel's refractive index. This behavior is illustrated in Figure 5.8, which shows two different rays that lie in different levels of the octree. The distance to

the edge of the octree level is calculated to avoid overstepping homogenous regions.

5.4 Image Generation

The image generation stage computes light paths from the light source to the film plane. This process starts with generating parallel rays from the light source, which are then traversed through the refracting flow using a precomputed gradient and the acceleration structure discussed in Section 5.3.2. Finally, the rays are weighted by a cutoff for a schlieren image and projected to the film plane.

5.4.1 Emitting Photons from the Light Source

Photons are emitted along a grid to simulate the light source. Ideally the rays are parallel, but the behavior of any given optical setup can be replicated by making modifications to the ray tracer. The system relies on progressive rendering to show increased detail over time. Banding effects from the volume can be smoothed by using jittered sampling to alter the starting positions of rays. The cost of computing three random numbers for jittered sampling for each ray at each pass is prohibitive. Instead, an array of random floats is precomputed. This array can be any size; however, for this dissertation an array that is three times the size of the image is used so that each thread can access three different random numbers. At the start of each rendering pass, only three random numbers are generated and passed to all threads. Each thread then adds these numbers to their thread ids to obtain a unique lookup into the precomputed array of random floats. Thus, the system only needs to generate three random numbers at each pass instead of thousands or millions.

5.4.2 Adaptively Tracing Rays through the Flow

Photons typically trace curved paths through a medium with spatially varying indices of refraction. Although the trajectory can be approximated using Snell's law, which is intended for refraction through discrete surfaces [67], it may produce undesirable artifacts when used to compute a ray moving through a compressible gas with no discernible surface. Snell's law also requires a significant amount of floating point arithmetic in three dimensions. In contrast, the ray equation of geometric optics based on Fermat's

principle presents a very fast and accurate approximation of the ray curve $x(s)$ through inhomogeneous materials [9].

$$\frac{d}{ds}\left(f\frac{dx}{ds}\right) = \nabla f \tag{5.6}$$

In order to simulate $x(s)$, Equation 5.6 is discretized using piece-wise linear approximations. The position x_i is updated according to the ray direction v_i, the refractive index f, and the step size Δs. The direction is updated according to the gradient of the scalar field of refractive indices ∇f.

$$x_{i+1} = x_i + \frac{\Delta s}{f} v_i \tag{5.7}$$

$$v_{i+1} = v_i + \Delta s \nabla f \tag{5.8}$$

The step size, Δs, can vary to adapt to the homogeneity of the refractive indices by using the acceleration structure computed in the precomputation step described in Section 5.3.2. The step size is modified to be the maximum of the base step size and the largest homogeneous region that can be skipped over. The homogeneous region will be 2^j times the size of a voxel, where j is the octree level stored at the current location x_i.

5.4.3 Reproducing the Cutoff

A typical setup may have a knife-edge in the center of the focal region to reduce any unaltered light by half. This allows both brighter and darker displaced regions to show up in the resulting image. This intensity value is accumulated from the number of photons that reach the film plane and a Monte Carlo Russian roulette style termination of photons leads to a realistic simulation of this process. A better solution is to assign an energy value instead, which can be weighted by the probability of being killed, significantly reducing noise and requiring fewer photons to be traced. If $\vec{d}$ is the resulting ray direction at the cutoff region, and $\vec{d_o}$ is the original ray direction from when the ray was first generated, then the resulting displacement is:

$$\vec{e} = (\vec{d} - \vec{d_o}) \tag{5.9}$$

$$e_x = \vec{e} \cdot \vec{camera_x} \tag{5.10}$$

$$e_y = \vec{e} \cdot \vec{camera_y} \tag{5.11}$$

where e_x and e_y are displacements along the camera axis $\vec{camera_x}$ and $\vec{camera_y}$. If $\vec{e}$ is the displacement from the original direction relative to the camera angle, then the resulting change in illumination I from a vertical knife-edge cutoff is:

$$\frac{\delta I}{I} = \frac{Kc_2}{\vec{e}} \int_{\delta 1}^{\delta 2} \frac{\partial p}{\partial z} dy \tag{5.12}$$

where c_2 is the focal distance of the lens projecting light onto the cutoff, K is the Gladstone-Dale constant, and the displacement is iterated over the focal region with the integral where $\delta 1$ and $\delta 2$ are the z coordinates of the ray entering and leaving the medium and p is the density [57, 71]. In the experimental setup, the focal distance or the cutoff can be altered in order to intensify the change in illumination. In a computer simulation the same effect can be achieved by replacing c_2, K, and the integration over the focal region by a scalar value, k. This value can be altered to correspond to an optical setup or modified to fit a desired range of intensities.

In the case of computing interferometry, the phase shift must be computed at each step in a piecewise linear fashion in order to approximate Equation 5.2. At each step we compute $n_{sum} = n_{sum} + (n - n_0) * \Delta s$. At the end of the traversal the intensity is computed as the following:

$$I = 0.5 - e_y \cdot k \quad \text{(horizontal knife-edge)} \tag{5.13}$$

$$I = 0.5 - e_x \cdot k \quad \text{(vertical knife-edge)} \tag{5.14}$$

$$I = 1 - |\vec{e}| \cdot k \quad \text{(circular cutoff)} \tag{5.15}$$

$$I = HSV(cos(\vec{d}, \vec{d_o}) \cdot k, 1, |\vec{e}| \cdot k) \quad \text{(color filter)} \tag{5.16}$$

The value k typically maps to the largest expected displacement as to yield normalized intensities without clamping [69]. The knife-edge can be flipped or rotated as desired and the circular cutoff can become a complement circular cutoff by complementing the equations. Where a circular cutoff will show regions with more displacement as darker, a complement cutoff shows regions with higher displacement as brighter. Once the intensities have been weighted according to the cutoff they are projected to the film plane and their values are accumulated. This leads to a potential race condition as different threads try to write to the same regions of the film plane at the same time. CUDA provides atomic operations that result in a slight speed decrease, but overall we find that this occurrence is sufficiently rare enough to ignore without introducing noticeable error for most instances. In cases where there is a great deal of refraction, synchronization may be necessary to avoid artifacts. For such cases we store values and the window coordinates in shared memory where each thread has its own separate index into a shared memory buffer. At the end of the CUDA kernel the threads synchronize and thread zero writes the values from shared memory out to the pixel buffer.

5.4.4 Interferometry

Interferometry relies on an unaltered reference beam separate from the beam traced through the inhomogeneity and does not have a physical cutoff. In the experimental domain this necessitates tracking wavelengths generated from the beam source by splitting the original source beam; however, the tracing of the reference beam is extraneous in the computational domain. Phase-shift is computed at each sample point over the inhomogeneity and bands appear from alterations in the phase-shift from the reference beam. This banding pattern is produced from the equation:

$$I = sin\left(\frac{2\pi}{\lambda} n_{sum}\right) \quad \text{(infinite-fringe interferometry)} \tag{5.17}$$

where λ is the wavelength of the light.

5.5 Filtering

Once a sufficient number of photons have been traced, the resulting image is filtered for noise and rendered to the screen. A simple Gaussian filter helps reduce noise while smoothing over gradations in luminance values. While a mean filter is better suited for reducing noise, it blurs out many of the small details.

Several methods exist for smoothing images generated with a limited number of photons. Jensen et al. [39] presented a cone filtering method weighting a given area by a sphere that encapsulates a set number of photons for use in photon mapping. Low density regions have a large filter width, while areas with high sample density have smaller filter width leading to a crisper image. This works well for caustics where large numbers of samples concentrate in a small area but may not always be the best approach for rendering high frequency schlieren images where dark crisp lines may be desirable.

In practice, only limited filtering is necessary as long as the photons are produced on a regular grid and the photons are given a weighted energy corresponding to the cutoff. The filter width should be decreased as more passes in the progressive rendering system are computed. This will lead to an initially blurry image but ultimately yield a better resolved image after sufficient passes of the renderer.

5.6 Interactive Cutoff Creation

In a schlieren optical setup if regions of the data need to be shaded in a certain way a custom cutoff can be created through a painstaking process involving a lot of trial and error. Fortunately this becomes easy on the computer. The ability to paint directly into a color filter and see the results in real-time becomes instrumental in pulling out regions of data. To eliminate the need to determine where to paint we have also enabled the user to paint on the color filter by clicking on a pixel of the resultant image. That pixel is then traced through the schlieren system to determine where it lies on the color filter and the appropriate region of the color filter is then colored in as demonstrated in Figure 5.9. This helps remove the guess work of where to color the filter. Additional aid to the user can be given by displaying a histogram of where light is hitting the color filter, thus showing where high frequencies of photons are gathering in specific regions, which the user can

then focus on and paint accordingly. These methods allow for high frequency informative color filters to be created very easily through interaction with the program.

5.7 Multifield Flow Analysis

The techniques presented in this chapter are focused on accurate reproduction of physical methods; however, the visualization of data with no direct mapping to a real world experiment is possible. One such example is the visualization of multifield data. Figure 5.10 demonstrates a schlieren image of a 5-field CFD combustion simulation that combines the scalar dissipation rate, hydrogen oxygen mass fraction, vorticity, heat release, and mixture fraction fields together [51]. This combination of data fields allows us to look at several possible features at once, such as the high frequency mixing inside the jet as well as the large features of the outer flame surrounding the jet. Since there is no real world correlation to this combination of data fields there is also no direct computation of refractive indices and so a mapping was created from the data values to refractive indices that could highlight the respective areas of interest in the data without drowning out other data fields.

The refractive index then becomes $n = f(r_1, r_2, r_3, r_4, r_5)$ where f is a function combining each resultant partial refractive index, r_i where $r_i = t_i(s_i)$ and s is the scalar field. Here, t_i is a function that maps each scalar value from a data field into a partial refractive index. This mapping should put the scalar value within some limits depending on the results desired. For our simulation a t_i that mapped each data field with the refractive index of air was chosen. In order to bring out features each t_i can be computed through a simple 1-dimensional transfer function specific to that field. For our example we used a simple average where $n = r_1 + r_2 + r_3 + r_4 + r_5$ and each $r_i = s_i/5$. Instead of specifying opacity values in a traditional transfer function used with volume rendering, the x-axis of our transfer functions specified the original normalized data values and the y-axis maps to the resultant altered refractive indices. As such, an unaltered transfer function starts out with a simple diagonal line. Because a schlieren image is altered according to the derivative of the data, setting uninteresting regions to the lowest value in the transfer function might not have the desired result. Instead, straight lines result in no

refraction and larger changes in the transfer function result in more pronounced regions in the resulting image.

As expected the final image is much higher frequency than any of the individual fields as it is a convolution of multiple fields. One issue with this is that it becomes hard to tell which feature corresponds to which data field. One possible way to ameliorate this would be to accumulate a color when sampling along the ray according to the data field being used, which is then weighted by the magnitude of the gradient such that the data field with the larger gradient will contribute more color. When the schlieren cutoff is applied the resulting image then attains both a global view of the flow through the schlieren cutoff as well as color information pertaining to each field. In practice we found that the colors become mixed together and unhelpful when dealing with more than three data fields at once.

5.8 Results

The method allows for high photon counts per second on approximately 256^3 sized datasets, as shown in Figure 5.11(a) and 5.11(b). An NVIDIA GeForce GTX 280 GPU with 1 GB VRAM was used for timings. Thirty-five million photons allow for a nearly interactive 13 fps on a 512x512 image with 10 samples per pixel (10 iterations of progressive refinement) on the combustion dataset, as demonstrated in Figure5.11(a) and 5.11(b). The frame rate varies based on the frequency of the data due to the adaptive step sizes through the volume and the size of the overall dataset. The frame rate is further influenced by the image size. This compares favorably as a visualization method to the images generated by Anyoji et al. [2], who reported rendering times of about 20 minutes. Figure 5.11(a) shows a moderate impact of using a cutoff with a shadowgraph performing slightly faster than a knife-edge cutoff and noticeably faster than the circular cutoff due to the normalization required in Equation 5.15.

Figure 5.12(a) shows a helium plume rendered using a traditional volume rendering technique that uses a one-dimensional transfer function over the density scalar field. This is compared against an approximation of schlieren imaging without computing refraction in 5.12(b), and our method shown in 5.12(c). The refractive indices were computed from

density measurements using the Gladstone-Dale relation, as shown in Section 5.3.1, with a Gladstone-Dale constant of $0.233 cm^3/g$ for air and helium due to a lack of mixture fractions. The volume rendered image using a transfer function provides a good indication of the shape of the flow by showing a discrete surface where the helium meets the surrounding air. The schlieren rendering in Figure 5.12(b) gives no indication of depth but gives a detailed rendering of the underlying changes in the flow by shading the degree of change in the flow rather than a set density value. This is similar to a technique of shading a volume based on the magnitude of a gradient except that the shading conforms to a cutoff value and alters according to the ray direction. The fringes of features are pulled out giving a silhouette to areas of the flow where large changes in the flow meet with orthogonal viewing rays. The technique also alleviates the need to tweak a transfer function as both large and small changes in the data are displayed and shaded according to their values, akin to an accumulative maximum intensity projection. A transfer function can still be used to pull out certain parts of the data using the technique, though the resulting image will no longer match the actual experimental schlieren image. Figure 5.12(c) gives further information and an accurate reproduction of what a real schlieren photograph would show by tracing refraction through the data. The bottom of the plume shows sharp features where the helium is emanating, resulting in significant changes in refractive indices. The rays cluster or disperse around the incoming helium resulting in sharp areas in the flow instead of the area clamping to white as seen in 5.12(b) without refraction. This becomes less severe towards the top of the plume, which shows that the helium is mixing with the air resulting in less light refraction. Edges of the flow are further enhanced as light in those areas bends around large changes in refractive indices.

Figure 5.13(a-c) shows a series of images from a simulation of the X38 aircraft on reentry comparing our method with various types of filters. Our method provides a clear image of the airflow around the body and bow of the plane, as well as vortices formed around the tail fins of the plane. The coloring over the density field shows distinct regions by showing differences in direction that a one-dimensional knife-edge cutoff might miss. Coloring a more detailed image such as the coal fire or heptane datasets as in Figure 5.7(c), results in more information but users may prefer to see only intensity variations.

Figure 5.14(a-h) shows a series of images of a combustion dataset rendered with and without filtering at different samples-per-pixel (spp). The left column is without filtering while the right column is filtered using a cone filter as described in Section 5.5. For the unfiltered images the root means square differences, RMS, between 1 and 10 spp, Figure 5.14(a) and Figure 5.14(c) is 7.75% while the RMS is between 10 and 100 spp, Figure 5.14(c) and Figure 5.14(e) is 2.78%. The RMS between 100 and 1000 spp, Figure 5.14(e) and Figure 5.14(g) is just 0.91% revealing a very small change beyond 100 spp. Filtering is very beneficial when rendering with a small photon count. To illustrate this, the RMS between 1 and 10 spp filtered, Figure 5.14(b) and Figure 5.14(d) is 0.83% while the RMS is between 10 and 100 spp filtered, Figure 5.14(d) and Figure 5.14(f) is 2.55%. The RMS between 10 to 100 spp filtered, Figure 5.14(f) and Figure 5.14(h) is only 0.176%. Recall that the cone filter expands its width to encapsulate a set number of samples. Thus, as the sample rate increases, the blurring cone decreases. In effect, above 100 spp, there is little or no blurring taking place.

The progressive rendering system displays a blurry image while rotating but a very crisp image with fine details when the mouse is released, which works very well in practice. The amount of time for the image to converge varies, but when generating the images and videos for this dissertation we found that typically, after approximately 1 second (at least 100 iterations or samples-per-pixel) there was little discernible improvement in image quality with additional time for 512x512 images as supported by the RMS terms.

Video memory usage and octree construction time on the CPU are listed in Table 5.1. The memory requirements are made up of the pixel buffer for a 512x512 image, the random number array, the gradient and the refractive indices. Single variables passed to the GPU and locally declared variables are ignored in the memory requirements. When working with large datasets memory usage can be mitigated by computing gradient values on the GPU at runtime as the gradient makes up a large portion of the memory usage. The octree construction times are for single-threaded calculation and memory allocation on the CPU.

5.9 Summary

In this work we have shown that in addition to interactive rendering of large-scale data, ray tracing solutions in scientific visualization allow for new methods of visualizing simulated data through manipulations to ray paths giving accurate simulations of real-world optical setups. Utilizing acceleration techniques on GPUs has allowed our simulation of schlieren, shadowgraph, and interferometry imaging to achieve interactive speeds with progressive refinement. These techniques provide a similar visualization to what scientists are used to seeing on the experimental side and differ from previous interactive computational schlieren rendering techniques by tracing light paths instead of line-of-sight approximations through inhomogeneous data. Tracing rays also allows an intuitive method of replicating optical setups, paving the way for validation of simulations by comparing accurate renderings against real-world photographs.

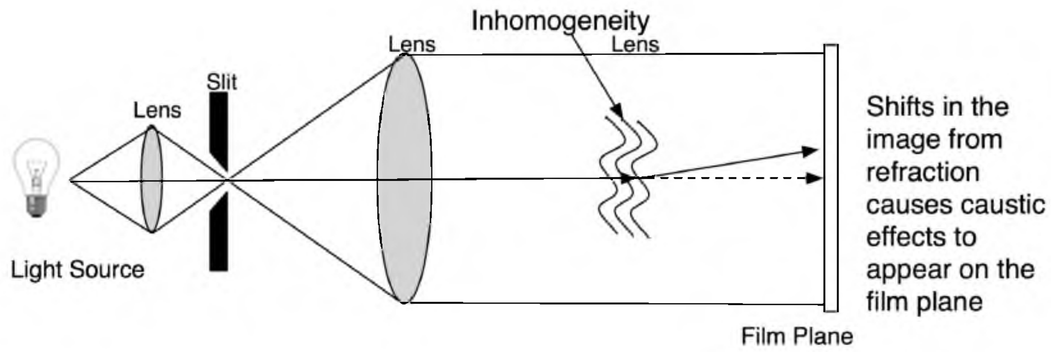


Figure 5.1. 2D illustration of the shadowgraph optical setup.

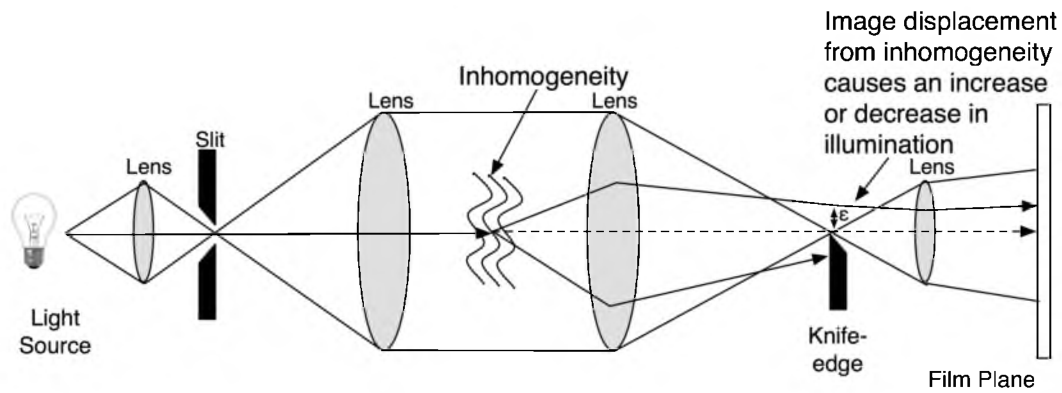


Figure 5.2. 2D illustration of the schlieren optical setup.

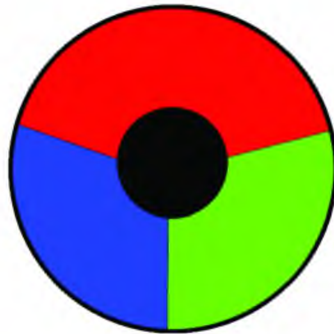


Figure 5.3. A typical color filter used in schlieren optical setups.

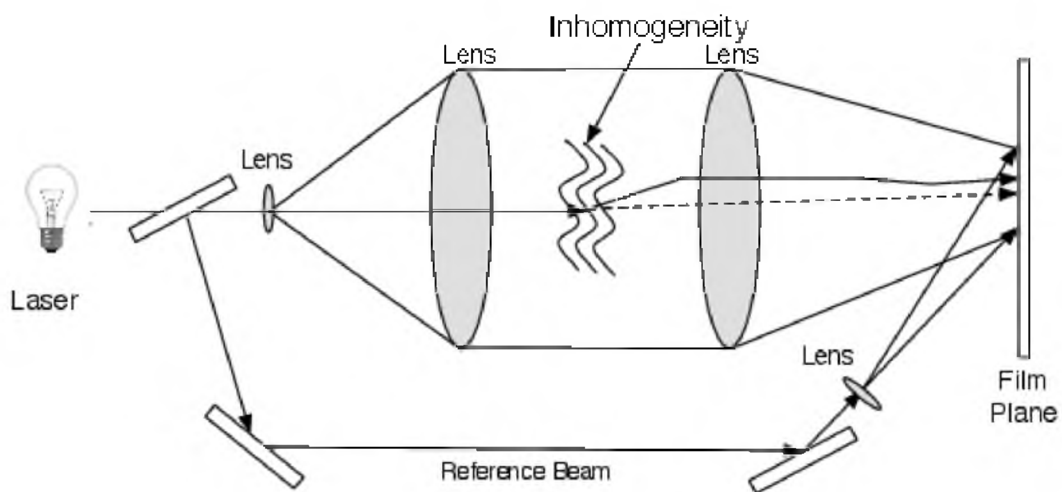


Figure 5.4. 2D illustration of the interferometry optical setup.

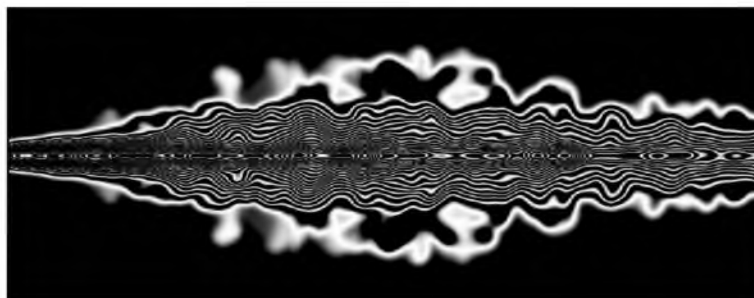


Figure 5.5. Infinite-fringe interferometry image computed using our method.

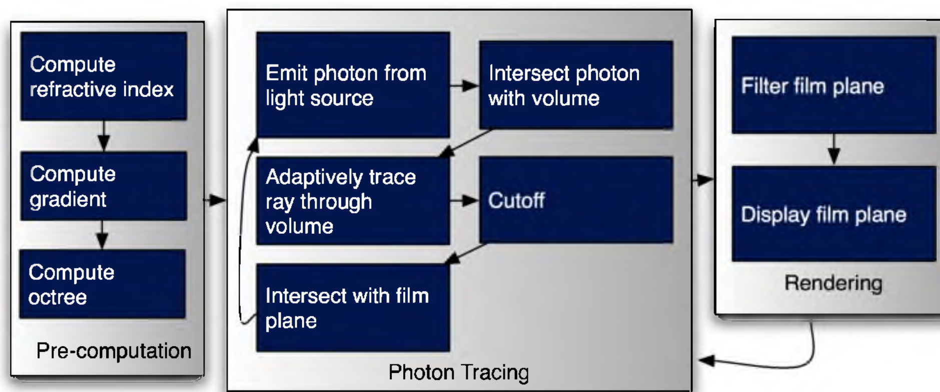


Figure 5.6. Illustration of the rendering pipeline.

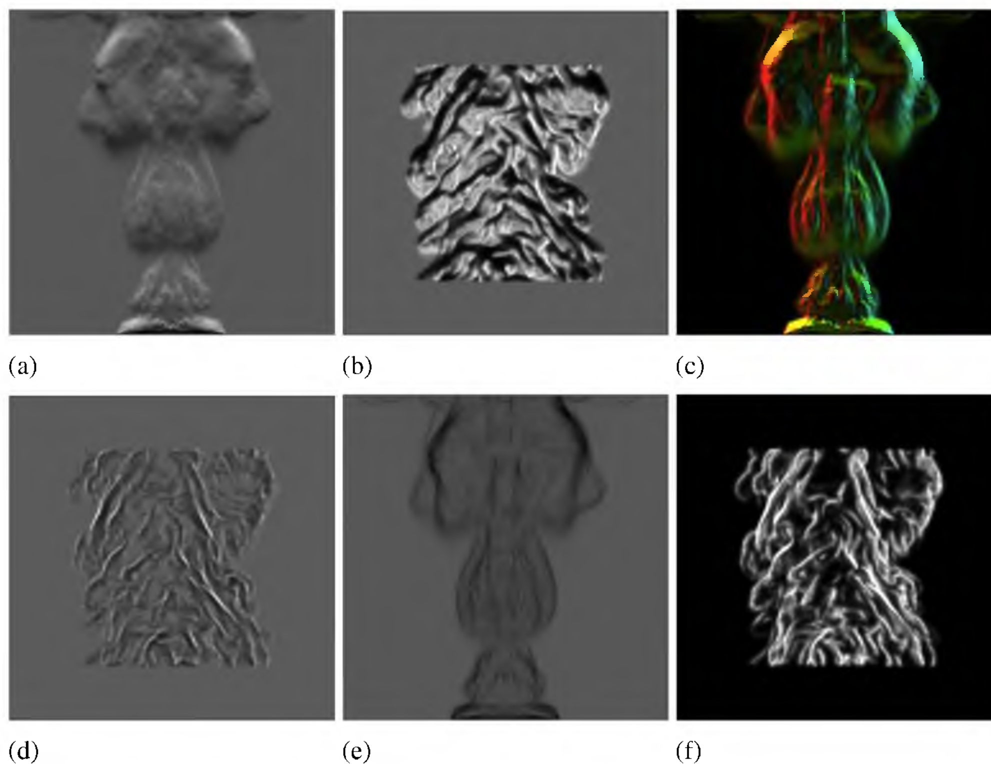


Figure 5.7. A heptane dataset rendered using refractive indices calculated from temperature and pressure with a knife-edge cutoff (a), a simulated combustion dataset rendered using a schlieren knife-edge cutoff to enhance the flow (b), color filter (c), shadowgraph image (d), a circular cutoff (e), and using a complemented circular cutoff (f).

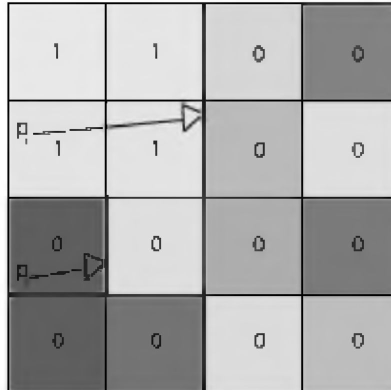


Figure 5.8. An illustration of a traversal through the octree. P_1 and P_2 are two rays traversing through the flow. P_1 is in a homogeneous region of the data and in a cell of the octree texture that will report a level number of 1 allowing P_1 to skip to the edge of that level. P_2, on the other hand, is at the lowest level of the acceleration structure and will only traverse to the next voxel.

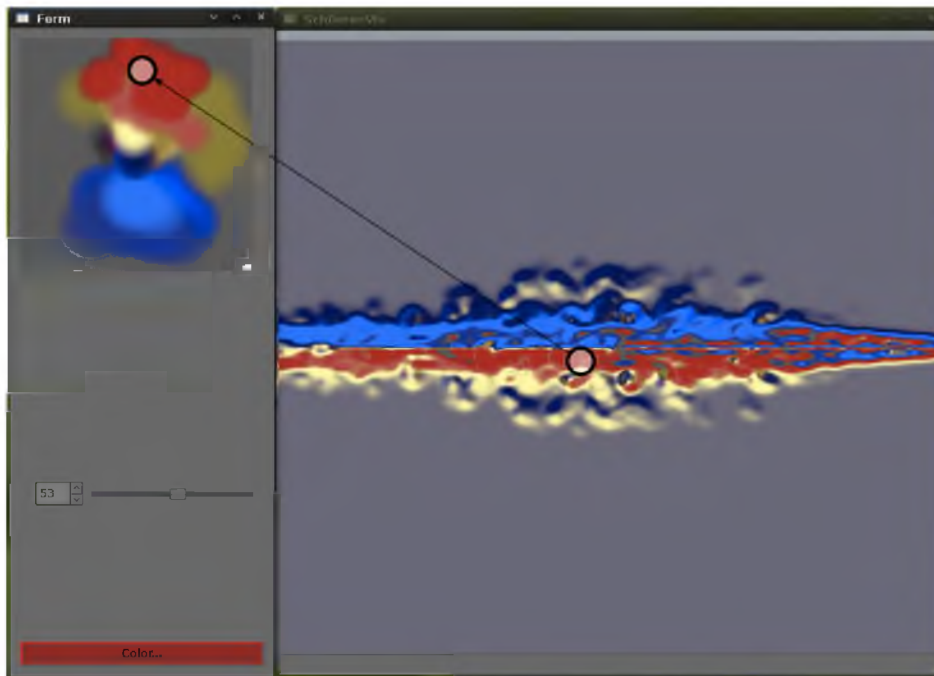


Figure 5.9. Creating a custom color filter by painting on the schlieren image. The corresponding region on the color filter is looked up by calculating where that pixel lies on the color filter and coloring the filter red in this case.

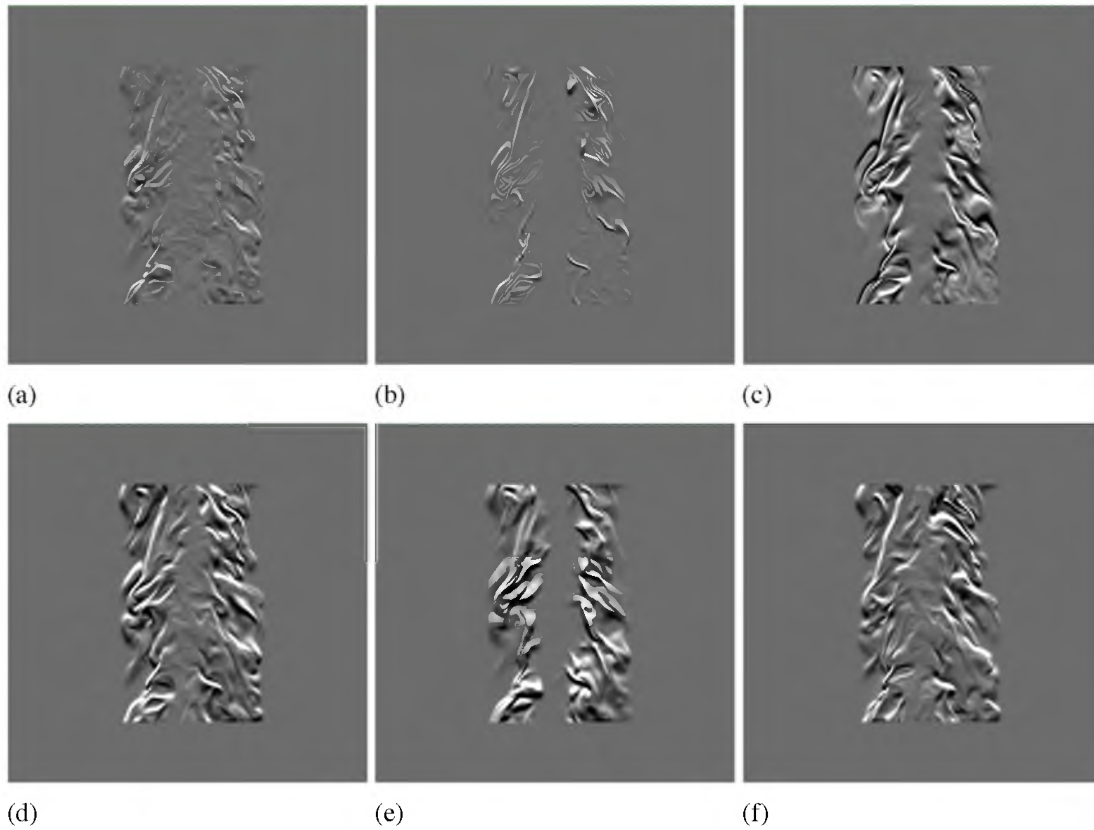
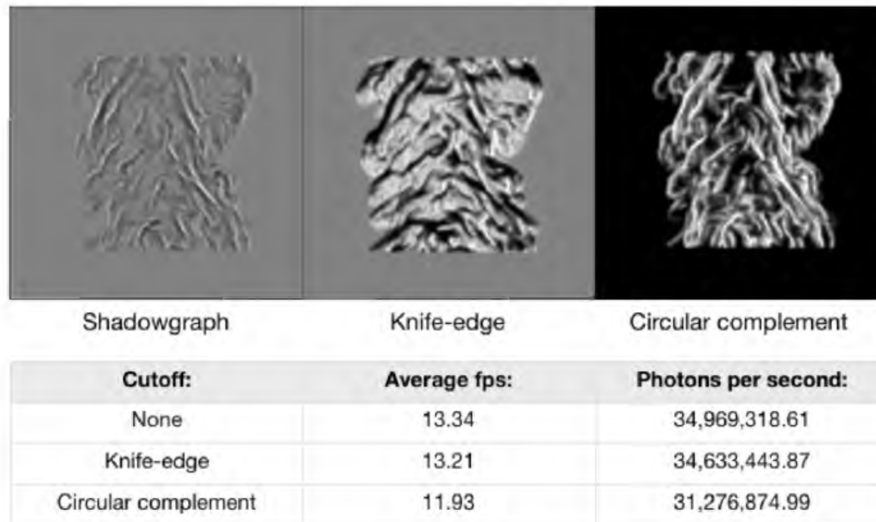
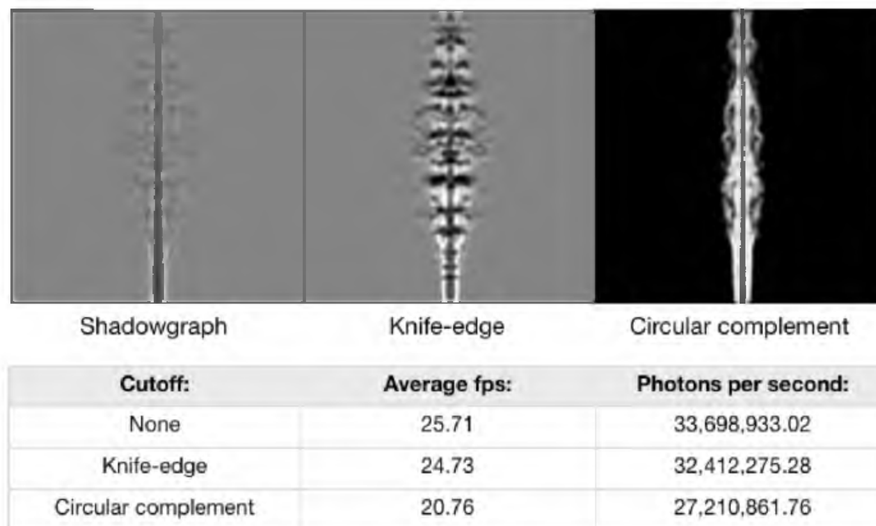


Figure 5.10. Demonstration of multifield data rendered using a schlieren knife-edge cutoff: (a) shows a combination of five different data fields and (b), (c), (d), (e), and (f) show individual renderings of scalar dissipation rate, heat release, vorticity, hydrogen oxygen mass fraction, and mixture fractions, respectively.



Cutoff:	Average fps:	Photons per second:
None	13.34	34,969,318.61
Knife-edge	13.21	34,633,443.87
Circular complement	11.93	31,276,874.99

(a)



Cutoff:	Average fps:	Photons per second:
None	25.71	33,698,933.02
Knife-edge	24.73	32,412,275.28
Circular complement	20.76	27,210,861.76

(b)

Figure 5.11. On the left, results of a combustion dataset of dimensions 480x720x100 seen in Figure 5.7 rendered with 10 iterations of progressive refinement per frame using cone filtering on a GeForce GTX 280 card at 512x512 resolution. Results of a coal fire with 5 iterations of progressive refinement per frame on a GeForce GTX 280 card at 512x512 resolution are shown on the right.

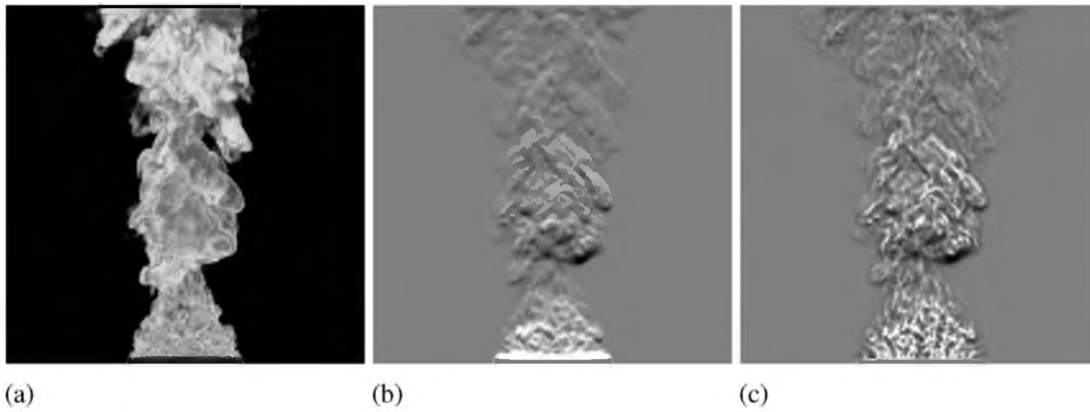


Figure 5.12. Comparison of volume rendering (a) with a line of sight schlieren approximation (b) and with our method (c).

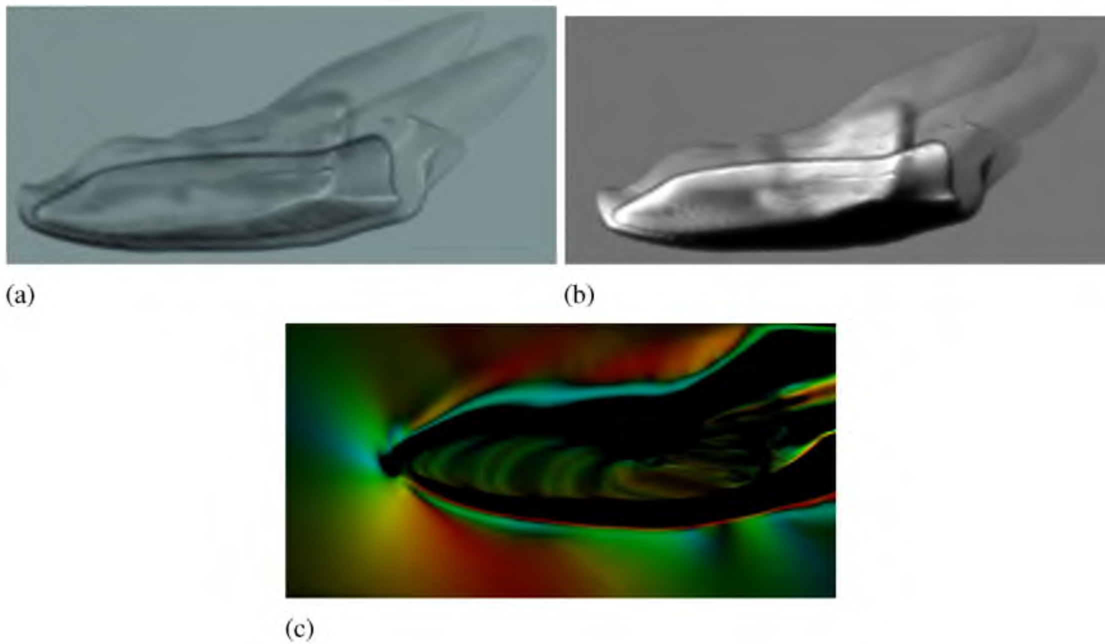


Figure 5.13. Comparison of our method using a shadowgraph (a), a knife-edge cutoff (b), and a color filter (c).

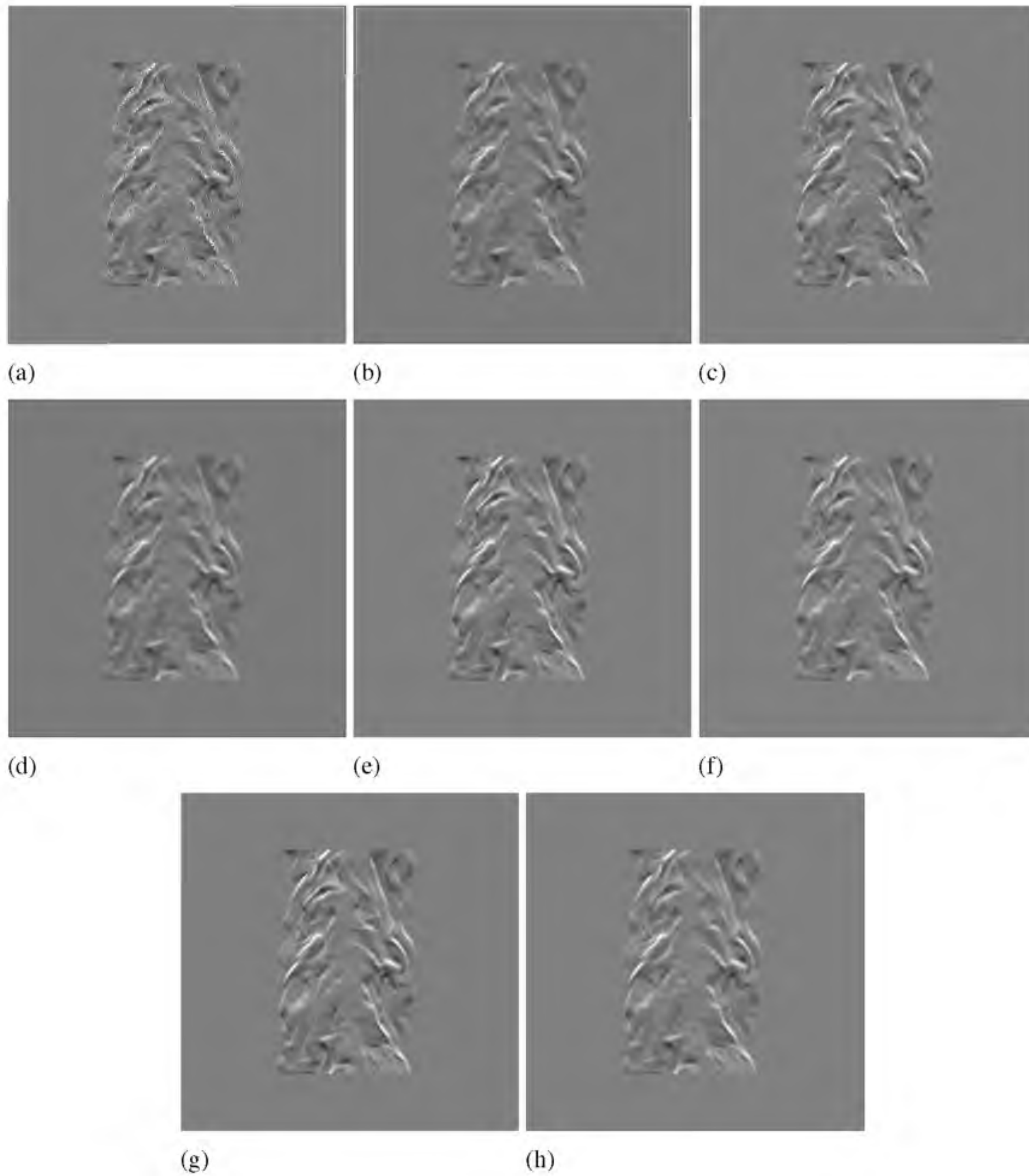


Figure 5.14. Comparison of unfiltered film plane with 1, 10, 100, and 1000 samples per pixel (a, c ,e, g) and the corresponding images of the film plane filtered with a cone filter in (b, d, f, h).

Table 5.1. Video memory usage and octree construction time for various datasets.

Dataset	Data Size (Megabytes)	Memory (Megabytes)	Octree Build (Seconds)
Combustion (480x720x100)	138.24	589.62	3.12
Heptane (293x293x293)	100.62	429.71	0.45
Helium (227x302x302)	82.81	354.05	0.40
Coal Fire (402x162x162)	42.20	181.45	0.59
X38 (256x256x256)	67.11	287.31	0.16

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this dissertation, implementations of ray tracing in scientific visualization tools using both software integration and OpenGL interception were presented as well as studies showing the scaling behavior of ray tracing in distributed-memory systems. These studies showed that not only did our implementations provide users with significantly improved rendering quality, but they also enabled interactive rendering of large datasets with software rendering in cases where the native programs failed to attain interactive rendering rates even with hardware acceleration. Interactivity is an important feature for data exploration of simulations which are producing increasingly large data sizes; developing ray tracing solutions which can provide interactive visualization for large data and scale with increasingly parallel compute clusters while also allowing advanced illumination models demonstrates that ray tracing is a promising rendering algorithm for current and emergent architectures. We demonstrate two methods of using ray tracing solutions within existing tools: through custom source code modifications to VTK and through OpenGL interception, which does not require program-specific source code changes. By using common visualization tools, we have shown that ray tracing can provide a working solution within users' existing workflows without having to resort to external stand-alone ray tracing programs. Additionally, novel methods of interactively computing physically-based photographic visualization techniques were presented to demonstrate that the capabilities present in ray tracing can facilitate new methods of scientific visualization.

6.1 Distributed Ray Tracing in Existing Visualization Tools

With this dissertation we have shown timings of widely used visualization tools on real-world datasets with weak-scaling and strong-scaling studies on distributed-memory

systems. In order to explore alternative rendering algorithms, we have integrated a software ray tracing solution into common tools which has demonstrated superior rendering performance with large polygon counts over the built-in OpenGL hardware rendering and Mesa software rendering methods. VisIt and ParaView have shown rendering times decrease by as much as 100x in our tests compared to brute-force rasterization and can achieve interactive rendering performance for large geometry counts on large cluster environments without the need for specialized hardware acceleration. Furthermore, our ray tracing solution provides scientists enhanced rendering quality. By integrating such systems into popular visualization tools, scientists may use the system without having to resort to external ray tracing tools. Using external stand-alone programs inhibits user adoption due to the complexities of exporting data and learning new tools but also disrupts the workflow of users already using common visualization and analysis tools such as ParaView and VisIt. Our Manta plugin is currently available in the source version of ParaView and was released in the binary distributions for Linux and Mac OS in version 3.10. Our VisIt implementation is expected to be released to the public in VisIt 2.5.2.

Current trends in super computers at the petascale have shown compute and memory bandwidth far out pacing I/O bandwidth. Whitlock et al. [81] demonstrated that I/O to compute is expected to be a thousandth of a percent in the upcoming Sequoia supercomputer. In order to reach exaflop levels of performance, concurrency will increase a factor of 40,000 to 400,000 times current petascale machines [1]. Memory, on the other hand, is only expected to increase about 100 to 200 times current levels resulting in a factor of 100 decrease in memory-per-compute thread. Postprocessing routines including rendering and analysis techniques can be conducted at varying stages of the data pipeline and with different representations of the data. Visualization and analysis of large-scale data on such emergent architectures presents a number of challenges over existing methods currently used for smaller-scale datasets. Saving simulation results at full resolution for later rendering and analysis on desktop machines is increasingly infeasible. Simulations run at the petascale are also increasingly difficult to save to disk as increases in compute power out-pace growth in disk space. There are a number of different methods designed to provide feasible solutions to these issues.

- Out-of-core

  Out-of-core postprocessing methods read in sections of larger out-of-core datasets on smaller machines for noninteractive processing. This allows for visualization and analysis of large data on a single machine without necessary modification of frameworks for large-scale systems. This technique is increasingly infeasible for increasingly large data where exascale runs may not even fit on disk and are generated on supercomputers with millions of compute threads.

- Rendering Cluster-based Visualization

  Rendering cluster-based visualization is a commonly used technique where a rendering cluster can be used for postprocessing. As visualization and analysis typically requires less processing than simulation, the rendering cluster is often significantly smaller than the larger compute cluster used to produce the original dataset. As data sizes increase disproportionately to disk and network IO, transferring full resolution datasets to external rendering clusters becomes prohibitive.

- Multiresolution

  Multiresolution techniques reduce data by calculating coarser representations of most of the data, with finer representations of particular portions of the data which are of interest to the user. This technique has the advantage of saving considerably on I/O by discarding finer resolutions of the data, but it must be known a priori which sections of the data can be safely down-sampled without losing necessary information for later analysis.

- In Situ

  Recent trends in supercomputers have shown that they are increasing in FLOPs faster than I/O bandwidth for writing out and reading in stored datasets. Peterka et al. [68] demonstrated postprocessing on a volume rendering application where data was saved and loaded for postprocessing where I/O made up over 90% of overall runtime. As saving out a full resolution dataset becomes increasingly infeasible, data must then be processed and compressed in some fashion on the fly while the simulation is running. In situ techniques couple visualization and analysis with running simulation code and utilizes the same supercomputing machine or a

> subset of the same resources used for simulation. This has the benefit of giving the postprocessing code access to the full resolution dataset while it is already in system memory. Features can be analyzed and extracted from the full resolution dataset or an image generated, drastically reducing overall I/O to disk. Visualization done at simulation time also has the benefit of simulation steering where simulations can be altered and rerun based on the output of the last rendered image.

In situ processing is still seldom utilized for a variety of reasons. Running routines in situ over a running simulation requires a considerable software developement effort. To ease the combination of simulation and analysis code, common visualization tools such as ParaView and VisIt have been adapted to support in situ processing. Moreland et al. [59] developed a coprocessing framework within VTK and ParaView; however, this implementation still has drawbacks. Data adapters must be developed to translate simulation data into VTK formats. Furthermore only portions of ParaView were built with the coprocessing library, such as GUI dependent functions. An inherent problem with all in situ implementations exists in which only a single timestep of the simulation is available at any given time, breaking any time dependent filters. Whitlock et al. [81] developed a similar coprocessing system for VisIt. In their implementation, data must be mapped and accessible through application specific callbacks. This system additionally allowed for the computational steering through a GUI interface in VisIt, which further complicates simulation implementations.

Interactive in situ visualization is often hampered by current rendering and compositing algorithms which do not scale. The work of Moreland et al. did not achieve above 1fps in rendering time alone at 500 cores even when data sizes were less than a million polygons per core. Additionally, running marching cubes to generate isosurfaces often took several seconds which inhibited interactive isosurface value editing. The implementations are also often geared toward rendering a single frame, where interactive visualization often requires the full resolution dataset to remain loaded across the simulation for data exploration. In large simulations with runs taking over large numbers of cores this could make interactive exploration prohibitively expensive. Yu et al. [86] developed an in situ visualization system for combustion simulations which achieved interactive rates for modestly sized

volume rendering; however, their naive compositing implementation impaired interactive rates.

While the scaling performance is currently dependent on the view and data distribution, integrating view-dependent data partitioning into VTK or using a system such as Chromium [35] could potentially alleviate a lot of compositing work and suboptimal data distribution present in ParaView and VisIt. Much work still needs to be done to accommodate visualization, including maximizing single node performance for other parts of the visualization pipeline such as reading, isosurfacing, calculator operations, and building acceleration structures. GPU accelerated ray tracing is another avenue of research which was not considered for this dissertation, but it is worth further study. The design of Manta has differing data representation and frame behavior resulting in wasted memory and a total frame time of the aggregate of compositing and rendering times instead of the maximum of the two. Solving these issues could result in decreased memory usage and increased rendering time.

The existing data flow of running a simulation as a separate entity and saving results independent of analysis for future postprocessing is no longer a feasible luxury. Predetermined coupling of simulation, analysis and visualization is increasingly necessary as abstracted modular steps in the dataflow pipeline consume disk space and memory that is no longer keeping pace with computing power. Increases in concurrency at factors of 400,000 also challenge existing sort-last compositing algorithms as compute becomes very cheap but data movement will prove prohibitively expensive. Even at the petascale, the limits of the human visual system and the limited set of information in a 2M pixel image are well below the amount of data generated and this problem of human perception will increase significantly at the exascale.

Visual analytics may prove a necessity for understanding datasets at the exascale for both data compression of large time-varying data and complexity reduction for human comprehension. For interactive debugging purposes, exploration of exascale sized data may prove infeasible for a user without analytically computed areas of interest and simplifications for determining faults. Data compression techniques allow saving results to disk for later postprocessing. Time-dependent feature extraction techniques [49] reduce

large time-varying datasets into a single manageable visualization. Topological feature extraction allows for reduction of complexity and data sizes of large scalar fields [32].

For interactive visualization, brute-force rendering algorithms utilized by programs such as VisIt and ParaView are no longer justifiable at the exascale. As data sizes increase drastically, but image resolutions remain roughly fixed, it no longer makes sense to have each node render full resolution versions of local data. In sort-last compositing, it is a severe waste of network bandwidth to have hundreds of thousands of cores rendering and sending imagery which is occluded or imperceptible to the user. Generating polygonal representations of implicit surfaces which are already stored in memory for in situ visualization such as generating isosurfaces in volume rendering also makes increasingly less sense as the memory per thread decreases. Knoll et al. demonstrated that high definition images of highly detailed direct volume rendered images can already be done interactively on a single machine [46], making a data streaming framework utilizing a few select compute nodes paging in multiresolution versions of the overall dataset for rendering seem like a very viable solution. When a single machine can render imagery which stretches the limits of human perception it becomes less of a question of how to render exabytes of data but rather why render it at all. This does not apply to rendering out image sizes beyond human perception for off-line exploration. Interactive visualization would still require maintaining the full dataset across the network in order to be able to stream in the highest resolution of data when users zoom into them. This may be solved in part through temporary storage in faster nonvolatile SSD drives. In cases where the simulation can be selectively rerun at higher resolutions for a few select nodes, adaptively rerunning portions of the simulation may make more sense.

6.2 Ray Tracing Through OpenGL Interception

We have shown that current rendering algorithms utilized in many scientific visualization tools do not achieve sufficient performance to interactively render large polygonal models in many cases. With GLuRay, we have proven that by intercepting OpenGL calls and using an optimized software ray tracer, we can achieve significant improvements in rendering performance in some of our tests using millions of polygons over several

common scientific visualization programs which otherwise fail to achieve interactivity on the Longhorn visualization cluster. Through a strong-scaling study, we have shown that GLuRay's performance scales on a distributed-memory cluster using ParaView's data-parallel work distribution and sort-last compositing.

Since interactive rendering for gigatriangle sized datasets is already possible using GLuRay with current systems, we believe frame rates will improve on future machines as the number of cores per node increases. For users who do not need increased performance, we have also presented advanced rendering for publication quality images and enhanced insight within existing tools without the need for learning additional rendering programs. The main limitations of our approach include decreased performance from building acceleration structures each frame with dynamic data, memory consumption, and a lack of support for shaders.

Ray tracing by intercepting OpenGL calls produces a few limitations. GLuRay usually relies entirely on the data distribution of the host program and since that host program typically relies on data-parallel distribution, there is no way to access other portions of the scene from remote nodes for secondary effects such as shadows at run-time. Adjacent areas can be duplicated across nodes for some effects such as distance-limited ambient occlusion; however, this is not common in the programs we have tested with. Data distribution in distributed-memory systems may be solved by implementing distribution through GLuRay in the background similar to such programs as Pomegranate or Chromium [22, 35]. Currently a ray-parallel work distribution similar to Ize et al. [38] was implemented for GLuRay but as of this writing only supports replicated data on each node and has been tested with ParaView, where data distributed by ParaView is sent to each node and only node 0 sends image data to ParaView. An out-of-core solution where nodes can page in data as needed is in developement. Stephens et al. showed Manta scaling very well on a large shared-memory system using transparency and other effects [72]. The main limitation of GLuRay is the memory overhead incurred by storing geometry and building acceleration structures. For a dataset with n polygons a typical BVH will be bounded by $2(n-1)$ BVH nodes. In clusters where memory is at a premium and compute is cheap, a slower but less memory intensive implementation may be ideal; however, system memory

is often much larger than that found on GPUs. The additional time to build acceleration structures is also a concern, but in exploratory visualization, a user will typically generate an isosurface to be interactively viewed resulting in many renderings for each update to geometry.

GLuRay is not a full mapping of OpenGL. Shaders are not supported yet and multipass rendering can significantly slow down the running system. Multiple passes are often used for effects such as shadows. In our testing many of the scientific visualization packages do not use such techniques and if they did, such systems could likely be turned off and their intended purpose replicated through the ray tracer in a single pass for performance considerations. There are many operations within OpenGL which may break the current program architecture and are not currently supported, such as blending functions or state changes beyond geometry, texture or materials within display lists. Programs which use OpenGL to render out GUI elements could prove problematic; however, none of the production level visualization tools we tested with use this method. None of these shortcomings have proven problematic for the generally simplistic rendering implementations within the tools we have tested with.

There is significant future work which could benefit ray tracing through OpenGL. Knoll et al. [46] recently demonstrated that direct volume rendering through CPU ray casting presents a very efficient approach for volume and isosurface rendering with large speed advantages compared to out-of-core GPU rendering. Supporting volume rendering and shaders would be highly beneficial and would be interesting future work. GPU ray tracing using an implementation such as Optix could provide increased performance on machines with hardware acceleration [62]. Another avenue of future work that was not explored in this dissertation is the use of GLuRay to optimize OpenGL code using the existing rasterization pipeline. Similar work was conducted with Chromium looking at optimizing redundant immediate mode OpenGL API calls across subsequent frames [21]. Through our own tests large speedups could be attained by splitting up display lists or implementing display lists as vertex buffer objects through OpenGL interception. Various culling methods such as frustum or occlusion culling could also be implemented through such a system for better scaling performance. Much work still needs to be done

to accommodate visualization, including maximizing single node performance for other parts of the visualization pipeline such as IO, isosurfacing, calculator operations, and implicit geometry rendering without using additional memory for geometry generation and storage. GPU accelerated ray tracing is another avenue of future research which was not considered for this dissertation but worth further study. Using GLuRay to provide acceleration techniques for poorly-optimized OpenGL code is another avenue of research worth exploring. Although we have shown the capability of GLuRay to scale when running ParaView on a cluster, an in depth study of render times in cluster environments would be worthwhile to determine the compositing and data-distribution impact of programs intended for rasterization.

6.3 Computational Photographic Methods

We have demonstrated that reproducing light paths for computing schlieren photographs is possible at interactive frame rates by intelligently combining various acceleration techniques and exploring the computational resources of modern graphics hardware. This method provides scientists with an accurate tool for simulating familiar visualization techniques in a computational environment, which requires far less resources and time than an experimental setup with physical constraints and complicated optics. The approach also opens the door for making a sufficiently accurate reproduction of real-world photographs that can be used to validate simulation data by simulating optical apparatuses in a straightforward manner.

Reproducing an exact replication of schlieren photographs' error presents several challenges. One source of error comes from one of the many cutoffs used and the artifacts they may produce. It is not clear to what degree these artifacts contribute to the overall image but the various cutoffs used may present undesirable refraction themselves [69]. Additionally, the light source could be faithfully reproduced as well as the amount of luminance over the length of the exposure.

Combining multiple data fields into a single refractive index has been explored; however, there are other possible methods that might be used. One such method of exploring multifield data may be to accumulate a color value per field similarly to a

volume renderer. Refraction could then be based on another field with a simple grayscale modifier such as a knife-edge cutoff applied such that one field produces a color value while the other produces a shadowgraph or schlieren effect and the resulting image is a convolution of the two techniques. Yet another method would be to have each data field traced independently with separate cutoffs applied and then the resulting images combined later on. These are merely some of the proposed methods for such a complex topic which might be usefully investigated.

The system assumes a constant wavelength across photons. Visible light waves have wavelengths across the visible spectrum and will refract differently producing various effects such as chromatic aberration. This is especially important for interferometry where a light source with uniform wavelength should be chosen. Finally, only purely refractive flows have been investigated so far, but simulating scattering effects, emission, absorption and polarization may also be necessary depending on the materials used in the simulation. Some materials, such as fire, may even need emissive calculations. Future work could explore all of the above issues for faithfully reproducing an experimental setup.

6.4 Summary

In Chapter 3, an integration of ray tracing into widely used parallel visualization tools was presented as well as weak- and strong-scaling studies on distributed-memory systems. By using an efficient CPU ray caster, orders of magnitude improvements were observed over existing brute-force OpenGL implementations, and in many cases over both CPU and hardware-accelerated rasterization algorithms. Through a series of weak- and strong-scaling studies, it was proven that CPU ray casting provides an interactive visualization experience which can work within existing data-parallel work distribution systems with varying, yet promising, scaling results related to the view-dependent nature of ray tracing. Implementation within the intermediary VTK library allowed the use of a common code base across multiple VTK-based programs with little modification. By integrating ray tracing in widely-used tools we have presented a working solution to users' existing workflows. This enables easy adoption of our solutions for existing users of these tools.

In Chapter 4, a program-agnostic ray tracing solution called GLuRay was presented that could run on top of existing visualization tools without modification by intercepting calls to the OpenGL rendering API. This solution could trap for both rendering calls and MPI calls for changes to the underlying rendering algorithm and distribution methods across programs. Mappings between OpenGL and the utilized Manta ray tracing library and acceleration structure generation proved minimal compared to data loading, specification, and Mesa based render times. GLuRay also allowed for advanced rendering models to compute secondary ray bounces, enabling the generation of publication quality images within existing programs. This allows for the generation of high quality images without interrupting users workflows, since currently, users often must export their data to external rendering programs to generate publication images which also often requires additional training to use.

A new method for interactively rendering photographic visualization techniques relying on accurate computation of light refraction was presented in Chapter 5, improving upon methods which computed straight ray paths through the medium. A temporally progressive rendering step allows for real-time exploration of data and quickly converging high quality renderings. User editable filters were presented allowing for new exploration techniques utilizing physically-based techniques that scientists are used to from experimental apparatuses. Multifield data exploration using schlieren techniques were also presented, which investigated using a physically inspired method for visualizing complex data with no real-world basis.

APPENDIX

PUBLICATIONS

- Book (To Appear): *High Performance Visualization: Enabling Extreme-Scale Scientific Insight* Editors: Wes Bethel, Hank Childs, Charles Hansen Chapter Title: Rendering, Authors: Charles Hansen, Wes Bethel, Thiago Ize, Carson Brownlee
- Carson Brownlee, Thomas Fogal, and Charles D. Hansen. *GLuRay: Ray Tracing in Scientific Visualization Applications using OpenGL Interception*, Proceedings of Eurographics Parallel Graphics and Visualization 2012, 41-50, 2012.
- Carson Brownlee, John Patchett , Li-Ta Lo , David DeMarle , Christopher Mitchell , James Ahrens , and Charles D. Hansen. *A Study of Ray Tracing Large-scale Scientific Data in Parallel Visualization Applications*, Proceedings of Eurographics Parallel Graphics and Visualization 2012, 51-60, 2012.
- M. Schott, T. Martin, A.V.P. Grosset, C. Brownlee. *Combined Surface and Volumetric Occlusion Shading*, Proceedings of Pacific Vis 2012, 169-176, 2012.
- Ize, Thiago, Brownlee, Carson, Hansen, Chuck. *Real-time ray tracer for visualizing massive models on a cluster*. Proceedings of Eurographics Parallel Graphics and Visualization 2011, 61-69, 2011.
- Carson Brownlee, Vincent Pegoraro, Siddharth Shankar, Patrick McCormick, Charles Hansen. *Physically-Based Interactive Flow Visualization Based on Schlieren and Interferometry Experimental Techniques*. ACM Transaction on Visualization and Computer Graphics, 17(11), 1574-1586, 2011.
- Carson Brownlee, Vincent Pegoraro, Siddharth Shankar, Patrick McCormick, Charles Hansen. *Physically-Based Interactive Schlieren Flow Visualization*. Proceedings of IEEE Pacic Visualization, 145-152, 2011. Best Paper Award.
- Vincent Pegoraro, Carson Brownlee, Peter S. Shirley, Steven G. Parker. *Towards interactive global illumination effects via sequential Monte Carlo adaptation*. IEEE

Symposium on Interactive Ray Tracing, 107-114, 2008.

- Patrick McCormick, Erik Anderson, Steven Martin, Carson Brownlee, Jeff Inman, Mathew Maltrud, Mark Kim, James Ahrens and Lee Nau. *Quantitatively driven visualization and analysis on emerging architectures*. SciDAC (2008), Journal of Physics: Conference Series 125, 125(1), 12-95, 2008.
- Christiaan P. Gribble, Carson Brownlee, and Steven G. Parker. *Practical Global Illumination for Interactive Particle Visualization*. Computers & Graphics (2007), vol. 32, 14-24, 2007.

REFERENCES

[1] AHERN, S., SHOSHANI, A., MA, K.-L., CHOUDHARY, A., CRITCHLOW, T., KLASKY, S., AND PASCUCCI, V. Scientific discover at the exascale: Report from the doe ascr 2011 workshop on exascale data management, analysis, and visualization. Technical report, 2011. ASCR.

[2] ANYOJI, M., AND SUN, M. Computer analysis of the schlieren optical setup. In *Proc. of SPIE* (2007), vol. 6279, 62790M.

[3] APPEL, A. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, 37–45.

[4] ATCHESON, B., IRKHE, I., HEIDRICH, W., TEVS, A., BRADLEY, D., MAGNOR, M., AND SEIDEL, H.-P. Time resolved 3d capture of non-stationary gas flows. *ACM Transaction on Graphics 25*, 5 (December 2008), 132.

[5] BAXTER, III, W. V., SUD, A., GOVINDARAJU, N. K., AND MANOCHA, D. Gigawalk: Interactive walkthrough of complex environments. In *Proceedings of the 13th Eurographics Workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, 203–214.

[6] BIGLER, J., STEPHENS, A., AND PARKER, S. Design for parallel interactive ray tracing systems. In *Interactive Ray Tracing 2006, IEEE Symposium on* (September 2006), 187–196.

[7] BITTNER, J., WIMMER, M., AND PURGATHOFER, H. P. W. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (2004).

[8] BORGEAT, L., GODIN, G., BLAIS, F., MASSICOTTE, P., AND LAHANIER, C. Gold: Interactive display of huge colored and textured models. *ACM Trans. Graph. 24* (July 2005), 869–877.

[9] BORN, M., WOLF, E., AND BHATIA, A. B. *Principles of Optics (7th edition)*, 7 ed. Cambridge University Press, New York, 1999.

[10] BROWN, B. P., MIESCH, M. S., BROWNING, M. K., BRUN, A. S., AND TOOMRE, J. Magnetic Cycles in a Convective Dynamo Simulation of a Young Solar-type Star. *Astrophysical Journal 731* (2011), 69.

[11] BROWNLEE, C., PATCHETT, J., LO, L.-T., DEMARLE, D., MITCHELL, C., AHRENS, J., AND HANSEN, C. D. A study of ray tracing large-scale scientific data in two widely used parallel visualization applications. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '12)* (2012), 51–60.

[12] CEDILNIK, A., GEVECI, B., AHRENS, J., AND FAVRE, J. Remote large data visualization in the paraview framework. *Eurographics Symposium on Parallel Graphics and Visualization* (2006), 162–170.

[13] CEI. Cei-creators of ensight visualization software, 2010. http://www.ensight.com/.

[14] CHEN, J. H., CHOUDHARY, A., DE SUPINSKI, B., DEVRIES, M., HAWKES, E. R., KLASKY, S., LIAO, W. K., MA, K. L., MELLOR-CRUMMEY, J., PODHORSZKI, N., SANKARAN, R., SHENDE, S., AND YOO, C. S. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science and Discovery 2* (2009), 1–3.

[15] CHHUGANI, J., PURNOMO, B., KRISHNAN, S., COHEN, J., VENKATASUBRAMANIAN, S., JOHNSON, D. S., AND KUMAR, S. Vlod: High-fidelity walkthrough of large virtual environments. *IEEE Transactions on Visualization and Computer Graphics 11* (2005), 35–47.

[16] CHILDS, H., PUGMIRE, D., AHERN, S., WHITLOCK, B., HOWISON, M., PRABHAT, WEBER, G. H., AND BETHEL, E. W. Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications 30* (2010), 22–31.

[17] CIDDOR, P. E. Refractive index of air: New equations for the visible and near infrared. *Applied Optics 35* (1996), 1566.

[18] CLARK, J. H. Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19* (October 1976), 547–554.

[19] CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (2009), I3D '09, 15–22.

[20] DEMARLE, D. E., GRIBBLE, C., AND PARKER, S. Memory-savvy distributed interactive ray tracing. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization* (2004), 93–100.

[21] DUCA, N., KIRCHNER, P., AND KLOSOWSKI, J. Stream caching: Optimizing data flow within commodity visualization clusters. In *Workshop on Commodity-Based Visualization Clusters* (October 2002).

[22] ELDRIDGE, M., IGEHY, H., AND HANRAHAN, P. Pomegranate: A fully scalable graphics architecture. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), SIGGRAPH 00, ACM Press/Addison-Wesley Publishing Co., 443–454.

[23] ENGEL, K., SOMMER, O., AND ERTL, T. Remote 3d visualization using image-streaming techniques. *Advances in Intelligent Computing and Multimedia Systems* (1999), 91–96.

[24] ERIKSON, C., MANOCHA, D., AND BAXTER, III, W. V. Hlods for faster display of large static and dynamic environments. In *I3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics* (2001), 111–120.

[25] FAN, Z., QIU, F., AND KAUFMAN, A. E. Zippy: A framework for computation and visualization on a gpu cluster. *Computer Graphics Forum 27*, 2 (2008), 341–350.

[26] GAITHER, K. Visualization's role in analyzing computational fluid dynamics data. *IEEE Computer Graphics 24*, 3 (2004), 13–15.

[27] GOBBETTI, E., AND MARTON, F. Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Comput. Graph. 28*, 6 (2004), 815–826.

[28] GREENE, N., KASS, M., AND MILLER, G. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1993), ACM, 231–238.

[29] GRIBBLE, C. P., AND PARKER, S. G. Enhancing interactive particle visualization with advanced shading models. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization* (New York, NY, USA, 2006), APGV '06, ACM, 111–118.

[30] GROSS, M., AND PFISTER, H.-P. *Point-based Graphics*. Elsevier Sciences Ltd., 2007.

[31] GUTIERREZ, D., SERON, F. J., ANSON, O., AND MUNOZ, A. Chasing the green flash: A global illumination solution for inhomogeneous media. *Spring Conference on Computer Graphics 2004* (2004), 97–105.

[32] GYULASSY, A., AND NATARAJAN, V. Topology-based simplification for feature extraction from 3d scalar fields. In *Visualization, 2005. VIS 05. IEEE* (October 2005), 535–542.

[33] HOWISON, M., BETHEL, E., AND CHILDS, H. Mpi-hybrid parallelism for volume rendering on large, multi-core systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (May 2010).

[34] HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. Wiregl: A scalable graphics system for clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, 129–140.

[35] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. Chromium: A stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th Annual Conference*

on Computer Graphics and Interactive Techniques (New York, NY, USA, 2002), SIGGRAPH '02, ACM, 693–702.

[36] IHRKE, I., ZIEGLER, G., TEVS, A., THEOBALT, C., MAGNOR, M., AND SEIDEL, H.-P. Eikonal rendering: Efficient light transport in refractive objects. *ACM Trans. on Graphics (Siggraph'07)* (2007), 59:1–9.

[37] IZE, T. *Efficient Acceleration Structures for Ray Tracing Static and Dynamic Scenes*. PhD thesis, University of Utah, 2009.

[38] IZE, T., BROWNLEE, C., AND HANSEN, C. D. Revisiting parallel rendering for shared memory machines. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2011), 61–69.

[39] JENSEN, H. W. Global illumination using photon maps. In *Proceedings of the Seventh Eurographics Workshop on Rendering* (1996), 21–30.

[40] JENSEN, H. W., AND CHRISTENSEN, P. H. Efficient simulation of light transport in scenes with participating media using photon maps. *Proceedings of ACM Siggraph 98* (1998), 311–320.

[41] JOHNSON, C., ROSS, R., AHERN, S., AHRENS, J., BETHEL, W., MA, K.-L., PAPKA, M., VAN ROSENDALE, J., SHEN, H.-W., AND THOMAS, J. Visualization and knowledge discovery: Report from the doe/ascr. *Workshop on Visual Analysis and Data Exploration at Extreme Scale* (October 2007).

[42] JOHNSON, F. T., TINOCO, E. N., AND YU, N. J. Thirty years of development and application of cfd at Boeing Commercial Airplanes, Seattle. *Computers and Fluids 34* (2005), 1115–1117.

[43] KAJIYA, J. T. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1986), SIGGRAPH '86, ACM, 143–150.

[44] KITWARE INCORPORATED. Paraview - open source scientific visualization, January 2010. http://www.paraview.org/.

[45] KLOSOWSKI, J. T., AND SILVA, C. T. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics 6*, 2 (2000), 108–123.

[46] KNOLL, A., THELEN, S., WALD, I., HANSEN, C. D., HAGEN, H., AND PAPKA, M. E. Full-resolution interactive cpu volume rendering with coherent bvh traversal. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium* (Washington, DC, USA, 2011), PACIFICVIS '11, IEEE Computer Society, 3–10.

[47] LABORATORY, U. A. R. Arl dsrc - data anaylsis, May 2012.

[48] LAWRENCE LIVERMORE NATIONAL LABORATORY. *VisIt Visualization Tool*, 2010. https://wci.llnl.gov/codes/visit/.

[49] LEE, T.-Y., AND SHEN, H.-W. Visualizing time-varying features with tac-based distance fields. In *Visualization Symposium, 2009. PacificVis '09. IEEE Pacific* (April 2009), 1–8.

[50] LEVOY, M., AND WHITTED, T. The Use of Points as a Display Primitive. *Computer Science Department, University of North Carolina at Chapel Hill Technical Report 85-022* (1985), 132–142.

[51] LIU, S., HEWSON, J. C., CHEN, J. H., AND PITSCH, H. Effects of strain rate on high-pressure nonpremixed n-heptane autoignition in counterflow. *Combustion and Flame 137* (May 2004), 320–339.

[52] LUEBKE, D. A developer's survey of polygonal simplification algorithms. *Computer Graphics and Applications, IEEE 21* (2001), 24 –35.

[53] MA, K.-L., PAINTER, J. S., AND HANSEN, C. D. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications 14* (1994), 59–68.

[54] MA, K.-L., AND PARKER, S. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Computer Graphics and Applications 21* (2001), 72–83.

[55] MAGALLON, M. E. spyglass: an opengl call tracer and debugging tool, 2011. http://spyglass.sourceforge.net/.

[56] MARSALEK, L., DEHOF, A., GEORGIEV, I., LENHOF, H.-P., SLUSALLEK, P., AND HILDEBRANDT, A. Real-time ray tracing of complex molecular scenes. In *Information Visualization: Information Visualization in Biomedical Informatics (IVBI)* (2010).

[57] MERZKIRCH, W. *Flow Visualization.* Academic Press, 1987.

[58] MITRA, T., AND CHIUEH, T. C. Implementation and evaluation of the parallel mesa library. In *Parallel and Distributed Systems, 1998. Proceedings.* (December 1998), 84–91.

[59] MORELAND, K., FABIAN, N., MARION, P., AND GEVECI, B. Visualization on supercomputing platform level ii asc milestone (3537-1b) results from sandia. Technical report, 2010. Sandia National Laboratories.

[60] NOUANESENGSY, B., AHRENS, J., AND WOODRING, J. Revisiting parallel rendering for shared memory machines. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2011), 31–40.

[61] NVIDIA. Cuda programming guide, January 2009.

[62] NVIDIA. Nvidia optix ray tracing engine programming guide, 2010.

[63] PARKER, S. Practical parallel rendering. A. K. Peters, Ltd., Natick, MA, USA, 2002, ch. Interactive ray tracing on a supercomputer, 187–194.

[64] PARKER, S., BOULOS, S., BIGLER, J., AND ROBISON, A. Rtsl: A ray tracing shading language. In *IEEE Symposium on Interactive Ray Tracing, 2007. RT '07.* (September 2007), 149 –160.

[65] PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. Interactive ray tracing for volume visualization. *Visualization and Computer Graphics, IEEE Transactions on 5*, 3 (July 1999), 238–250.

[66] PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. Interactive ray tracing for isosurface rendering. In *Visualization '98. Proceedings* (October 1998), 233–238.

[67] PEGORARO, V., AND PARKER, S. G. Physically-Based Realistic Fire Rendering. In *Proceedings of the 2nd Eurographics Workshop on Natural Phenomena* (2006), 51–59.

[68] PETERKA, T., YU, H., ROSS, R., AND MA, K.-L. Parallel volume rendering on the ibm blue gene/p. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '08)* (2008), 73–80.

[69] SETTLES, G. *Schlieren and Shadowgraph Techniques, Visualizing Phenomena in Transparent Media.* Springer, New York, 2001.

[70] SGI, AND MILES, J. Gltrace, 1997. http://reality.sgi.com/opengl/gltrace/.

[71] SMITS, A. J., AND LIM, T. T. *Flow Visualization: Techniques and Examples.* Imperial College Press, London, 2000.

[72] STEPHENS, A., BOULOS, S., BIGLER, J., WALD, I., AND PARKER, S. G. An application of scalable massive model interaction using shared memory systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2006), 19–26.

[73] SUN, M. Computer modeling of shadowgraph optical setup. In *Proceedings of SPIE* (2007), vol. 6279, 62790L.

[74] SUN, X., ZHOU, K., STOLLNITZ, E., SHI, J., AND GUO, B. Interactive relighting of dynamic refractive objects. *ACM Transaction on Graphics 27*, 3 (2008), 35:1–9.

[75] SVAKHINE, N. A., JANG, Y., EBERT, D., AND GAITHER, K. Illustration and photography inspired visualization of flows and volumes. *IEEE Visualization 2005* (2005), 687–694.

[76] UCAR. Vapor, 2009. http://www.vapor.ucar.edu/.

[77] WALD, I., BENTHIN, C., DIETRICH, A., AND SLUSALLEK, P. Interactive ray tracing on commodity pc clusters. *Lecture Notes in Computer Science* (2003), 499–508.

[78] WALD, I., BOULOS, S., AND SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (2007).

[79] WALD, I., SLUSALLEK, P., AND BENTHIN, C. Interactive distributed ray tracing of highly complex models. In *Proc. of Eurographics Workshop on Rendering* (2001), 274–285.

[80] WEINBERG, F. J. *Optics of Flames*. Butterworths, London, 1963.

[81] WHITLOCK, B., FAVRE, J. M., AND MEREDITH, J. S. Parallel in situ coupling of simulation with a fully featured visualization system. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (2011), 101–109.

[82] WHITTED, T. An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June 1980), 343–349.

[83] XIONG, H., PENG, H., QIN, A., AND SHI, J. Parallel strategies of occlusion culling on cluster of gpus. *Computer Animation and Virtual Worlds 18*, 3 (2007), 165–177.

[84] YATES, L. A. Images constructed from computed flow fields. *American Institute of Aeronautics and Astronautics (AIAA) 31*, 10 (1993), 1877–1884.

[85] YOON, S.-E., LAUTERBACH, C., AND MANOCHA, D. R-lods: Fast lod-based ray tracing of massive models. *The Visual Computer 22* (2006), 772–784. 10.1007/s00371-006-0062-y.

[86] YU, H., WANG, C., GROUT, R. W., CHEN, J. H., AND MA, K.-L. In situ visualization for large-scale combustion simulations. *IEEE Comput. Graph. Appl. 30* (May 2010), 45–57.

[87] ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, III, K. E. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., 77–88.