Performance Quantification of Visualization Systems

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Roman Valentin Bruder

aus Tübingen

Hauptberichter:	Prof. Dr. Thomas Ertl
Mitberichter:	Prof. Dr. Hank Childs
	Prof. Dr. Steffen Frey
Tag der mündlichen Prüfung:	16. Dezember 2021

Visualisierungsinstitut
der Universität Stuttgart

2022

Acknowledgments

With deep gratitude I acknowledge the following people, for without them this work would not have been possible. First of all, Thomas Ertl, who not only gave me the idea and opportunity to pursue a doctoral degree at VISUS, but also supervised me and left me the freedom to explore and realize some of my own ideas. Special thanks to Steffen Frey, who mentored, supported, and encouraged me throughout the years, from my master's thesis to numerous collaborations. I thank Hank Childs and Steffen Frey for their willingness to serve as reviewers for this thesis.

Thanks to my many coauthors and collaborators: Ruben Baur, Fabian Beck, Matthias Braun, Michael Burch, Hank Childs, Thomas Ertl, Steffen Frey, Florian Frieß, Moritz Heinemann, Melanie Herschel, Marcel Hlawatsch, Kuno Kurzhals, Houssem Ben Lamar, Mathias Landwehr, Matthew Larsen, Christoph Müller, Guido Reina, Christoph Schulz, Hagen Tarner, Gleb Tkachev, and Daniel Weiskopf. It was great working with you over the years and solving all the diverse research problems together! I thank all my colleagues at VISUS and VIS, research can be tiresome and frustrating at times—you have made it less so.

My work at the Visualization Research Center of the University of Stuttgart was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) within Project A02 of the SFB/Transregio 161 (project number 251654672). This included funding of my stay abroad at the University of Oregon in Eugene, Oregon, USA. Many thanks to Hank Childs who was an exceptional host there and the whole CDUX group for the warm welcome. Special thanks to Brent, Abhishek, and Kristi for a great time in Eugene.

For Annika and my family, thank you for all your support and patience over the years, without which all that I achieve means little.

Valentin

Table of Contents

Acknowledgments iii

Summary xii

Zusammenfassung xiv

1 Introduction 1
- 1.1 Research Questions . . . 2
- 1.2 Outline and Contributions . . . 5
  - 1.2.1 Overall Contributions . . . 8
  - 1.2.2 Awards . . . 9

2 Fundamentals 11
- 2.1 Visualization and Rendering . . . 11
  - 2.1.1 Visualization Pipeline . . . 12
  - 2.1.2 Rendering Pipeline . . . 13
  - 2.1.3 Foveated Rendering . . . 14
- 2.2 Volume Visualization . . . 15
  - 2.2.1 Volume Rendering . . . 15
  - 2.2.2 The Volume Rendering Integral . . . 16
  - 2.2.3 Raycasting . . . 17
- 2.3 Parallel and Distributed Visualization . . . 20
  - 2.3.1 Volume Raycasting on GPUs . . . 21
  - 2.3.2 In Situ Visualization . . . 22
- 2.4 Performance Analysis and Modeling . . . 23
  - 2.4.1 Volume Visualization and GPU Performance . . . 23
  - 2.4.2 Modeling of GPU and Scientific Visualization Workloads . . . 24
- 2.5 Data Sets and Hardware . . . 24

3 Runtime Performance Evaluation 29
- 3.1 Empirical Evaluation of GPU-Accelerated Interactive Visualizations . . . 30
  - 3.1.1 Measurement and Analysis . . . 31
  - 3.1.2 Case Study 1: Volume Raycasting . . . 33
  - 3.1.3 Case Study 2: Particle Visualization . . . 43
  - 3.1.4 Results and Recommendations . . . 52
  - 3.1.5 Future Directions . . . 54
- 3.2 Visually Comparing Performance Specifics . . . 55
  - 3.2.1 Multiple Perspectives Analysis System . . . 56

3.2.2 Application . . . 60
3.2.3 Future Directions . . . 60

4 Performance Modeling on GPU Systems 61
4.1 Load Balancing and Resolution Tuning for Interactive Volume Raycasting 62
4.2 Collection of Performance-Relevant Data . . . 64
4.2.1 Histograms of Volume Blocks (H and H_α) . . . 65
4.2.2 Depth Assessment (D_{front} and D_{back}) . . . 65
4.2.3 Early Ray Termination (D_{ERT} & D'_{ERT}) . . . 66
4.3 Hybrid Performance Model . . . 67
4.3.1 Machine Learning: Prediction of Sample Cost σ . . . 68
4.3.2 Analytical Model: Prediction of Frame Execution Time . . . 69
4.4 Prediction-Based Parameter Tuning . . . 70
4.4.1 Adaption of the Sampling Resolution . . . 70
4.4.2 Load Balancing . . . 71
4.5 Results . . . 72
4.5.1 Analysis and Comparison of a Sequence with a Single GPU . . . 73
4.5.2 Approximation and Prediction Accuracy . . . 76
4.5.3 Prediction Overhead . . . 76
4.5.4 Interaction Sequences . . . 77
4.5.5 Load Balancing . . . 78
4.5.6 Image Versus Ray Space Adaption . . . 79
4.6 Future Directions . . . 82

5 Performance Modeling on Distributed Memory Systems 83
5.1 A Hybrid In Situ Approach for Cost Efficient Image Database Generation 84
5.1.1 In Situ Visualization . . . 85
5.1.2 Hybrid In Situ Method for Image Database Generation . . . 90
5.1.3 Implementation Details . . . 95
5.1.4 Overview of Experiments . . . 96
5.1.5 Results . . . 98
5.1.6 Future Directions . . . 107
5.2 Performance Prediction to Support Render Hardware Acquisition . . . 108
5.3 Adaptive Encoder Settings for Interactive Remote Visualization . . . 111

6 Foveated Rendering to Improve Application Performance 115
6.1 Voronoi-Based Foveated Volume Rendering . . . 116
6.1.1 Method . . . 116
6.1.2 Results . . . 120
6.1.3 Discussion and Future Directions . . . 122
6.2 Foveated Encoding for High-Resolution Displays . . . 122

6.2.1 Method . . . 123
6.2.2 Results . . . 124
6.2.3 Future Directions . . . 125

7 Performance-Optimized Volume Rendering Applications 127
7.1 Volume-Based Large Dynamic Graph Analysis . . . 128
7.1.1 Static Volumetric Graph Representation . . . 129
7.1.2 Classes of Analytics Methods . . . 131
7.1.3 Evolution Provenance . . . 136
7.1.4 Implementation . . . 139
7.1.5 Application Examples . . . 140
7.1.6 Expert Evaluation . . . 146
7.1.7 Future Directions . . . 150
7.2 Space-Time Visualization of Gaze and Stimulus . . . 150
7.2.1 Method . . . 151
7.2.2 Examples . . . 157
7.2.3 Discussion . . . 160
7.2.4 Future Directions . . . 161

8 Conclusion 163
8.1 Summary . . . 164
8.2 Discussion . . . 166
8.2.1 Research Question 1 . . . 166
8.2.2 Research Question 2 . . . 167
8.2.3 Research Question 3 . . . 167
8.3 Outlook . . . 168

Author's Work 171

Bibliography 173

List of Figures

1.1 Thesis components and their relation . . . 3

2.1 Visualization pipeline . . . 12
2.2 Graphics rendering pipeline . . . 13
2.3 Illustration of object-order empty space skipping . . . 20
2.4 Renderings of artificial volume data sets . . . 25
2.5 Renderings of volume data sets from simulations . . . 25
2.6 Renderings of volume data sets from CT-scans . . . 27

3.1 Camera paths . . . 32
3.2 Execution time distribution for volume raycasting . . . 37
3.3 Distribution of Pearson correlation coefficients of volume raycasting . . . 38
3.4 Pearson correlation matrix for GPUs . . . 39
3.5 Speed-up of volume rendering on different GPUs . . . 40
3.6 Frame times for data sets using different camera paths . . . 42
3.7 Particle data sets . . . 46
3.8 Distribution of the execution times of particle rendering . . . 48
3.9 Distribution of Pearson correlation coefficients of particle rendering . . . 48
3.10 Pearson correlation matrix for particle data sets . . . 50
3.11 Pearson correlation matrix of camera paths . . . 51
3.12 Mean frame times of data sets rendered with different techniques . . . 51
3.13 Data Set Explorer . . . 56
3.14 Camera Path Explorer . . . 58

4.1 Process overview of the load balancing and resolution tuning approach . . . 63
4.2 Raycasting algorithm . . . 66
4.3 Load balancing distribution . . . 71
4.4 Frame times comparison . . . 74
4.5 Prediction accuracy of sample cost, sample count and ERT approximation . . . 75
4.6 Performance and quality comparison . . . 78
4.7 Load balancing performance . . . 80
4.8 Visual comparison of adaption in image space, ray space, and hybrid . . . 81

5.1 In situ processing types . . . 86
5.2 Notional organization of the viability of rightsizing . . . 90
5.3 Sequence diagram of our hybrid in situ system . . . 91
5.4 Compositing time prediction accuracy . . . 93
5.5 Load balancing approach . . . 94

5.6 Render time estimation accuracy 97
5.7 Baseline experiments 99
5.8 Parametric study: image count 102
5.9 Parametric study: image resolution 103
5.10 Nyx simulation renderings 105
5.11 Nyx weak scaling results 105
5.12 Differences in cost between hybrid and inline as a function of concurrency 107
5.13 Prediction accuracy for a GPU upgrade 109
5.14 Prediction accuracy across multiple clusters 110
5.15 CNN architecture for adaptive encoding 112

6.1 Sampling mask for foveated volume raycasting 117
6.2 Illustrations of foveated volume rendering 121
6.3 Illustration of the foveated encoding approach 123
6.4 Throughput comparison for foveated encoding 125

7.1 Volumetric representation using adjacency matrices 129
7.2 Scalability of the volumetric graph representation 130
7.3 Classes of analytics methods 131
7.5 Volume partitioning 134
7.6 Aggregation levels 135
7.7 Color mappings 136
7.8 Session graph example 138
7.9 Evolution provenance visualization 139
7.10 Timeline plot of the flight data graph 140
7.11 Volumetric representation of the flight data 142
7.12 Matrix view to visualize differences 143
7.15 Process of analyzing the software call graph 146
7.16 Exemplary provenance graph generated during the user study 150
7.17 Space time visualization application 151
7.18 Data processing pipeline 152
7.19 Illustration of the data input 153
7.20 Volume clipping 155
7.21 Transfer functions for optical flow and gaze density 156
7.22 Kite video example 158
7.23 Thimblerig video example 159
7.24 UNO game video example 160

List of Tables

2.1 Graphics Cards Used for Performance Measurements 25
2.2 Volume Data Sets Used for Performance Measurements 26

3.1 Performance Evaluation in Recent Selected Volume Rendering Papers . . . 34
3.2 Parameters of the Volume Rendering Benchmark 36
3.3 Performance Evaluations in Recent Particle Rendering Papers 44
3.4 Techniques and Required Shader Stages Used for Particle Rendering 45

4.1 Maximum Execution Times 77
4.2 Quality Impact of Adaption 79

5.1 The Four Types of Inefficiency for In Situ Processing 89
5.2 Node Configurations for the Nyx Simulation on Stampede2 106
5.3 Throughput of our Adaptive Encoding in Comparison to Flat Encodings . . 113

6.1 Foveated Volume Rendering Performance 120

7.1 Permitted Analytics Operations 137
7.2 Usefulness of Components 148
7.3 Relative Usage of the Widgets 149
7.4 Data Formats 154
7.5 Example Videos with Gaze Data 157

List of Abbreviations and Acronyms

AMR adaptive mesh refinement
API application programming interface
AOI area of interest
CNN convolutional neural network
CT computed tomography
CPU central processing unit
DDA digital differential analyzer
DDR double data rate
ERT early ray termination
ESS empty space skipping
FDR fourteen data rate
GPGPU general-purpose computing on graphics processing units
GPU graphics processing unit
GUI graphical user interface
HPC high performance computing
HVS human visual system
KRLS kernel recursive least squares
LBG Linde-Buzo-Gray
LOD level of detail
NLM United States National Library of Medicine
MSE mean-squared error
MPI message passing interface
MRI magnetic resonance imaging
OpenCL Open Computing Libray
OpenGL Open Graphics Library
OpenMP Open Multi-Processing
PNG portable network graphics
PRNG pseudo-random number generator
PSNR peak signal-to-noise ratio
RBF radial basis functions
RAM random access memory
ReLU rectified linear unit
RGB red, green, blue
RGBA red, green, blue, alpha
RMSE root-mean-square error
RLS recursive least squares
SIMT single instruction, multiple threads
SSIM structural similarity index measure
STC space-time cube
SU shader units
TACC Texas Advanced Computing Center
UDP User Datagram Protocol
UMAP Uniform manifold approximation and projection
UTCT University of Texas High-Resolution X-ray CT Facility
VRAM video random access memory

Units

fps frames per second

Summary

Visualization is an important part of data analysis, complementing automatic data processing to provide insight in the data and understand the underlying structure or patterns. A visualization system describes a visualization algorithm running on a specific compute architecture or device. Runtime performance is crucial for visualization systems, especially in the context of ever-growing data sizes and complexity. One reason for this is the importance of interactivity, another is to provide the opportunity for a comprehensive investigation of generated data in a limited time frame. Providing the possibility of changing the perspective beyond the original focus has been shown to be particularly helpful for explorative data analysis. Performance optimization is also key to save costs during visualization on supercomputers due to the high demand for their compute time. Being able to predict runtime enables a better resource planning and optimized scheduling on such devices.

The central research questions addressed in this thesis are threefold and build on each other: How can we quantify runtime performance of visualization systems? How to use this information to develop models for prediction, and ultimately: How to integrate both aspects in the application context? The goal is to gain a comprehensive understanding of the runtime performance of visualization systems and optimize them to save costs and improve the user experience.

Despite many works in this direction, there are still open questions and challenges on how to reach this goal. One of these challenges is the diversity of compute architectures used for visualization, including devices from mobile devices to supercomputers. Most visualization algorithms profit from running in parallel. However, this poses another challenge in performance quantification due to the usage of multiple heterogeneous parallel hardware hierarchies. Typically, visualization algorithms deal with large data, sparse regions, and interactivity requirements. Further, they can be fundamentally different in their rendering approaches. All these aspects make a reliable performance prediction difficult. This thesis addresses those challenges and presents research on performance evaluation, modeling, and prediction of visualization systems, and how to translate these concepts to improve performance-critical applications.

Assessing runtime performance plays a key role in understanding and improving it. A new framework for the extensive and systematic performance evaluation of interactive visualizations is introduced, to help gain a deeper understanding of runtime behavior and rendering parameter dependencies. Based on the current practice of runtime performance evaluation in literature, a database of performance measurements is created. A list of best practices on how to improve performance evaluation is compiled based on a statistical analysis of the data. Additionally, a frontend has been developed to visually compare the rendering performance data from multiple perspectives.

With a fundamental understanding of an application's runtime behavior, performance can be modeled, and the model used for prediction. New techniques for different hardware systems are introduced that are typically used for the visualization of large data sets: desktop computers featuring dedicated graphics hardware and high-performance distributed memory systems. For the former, a method to predict performance on-line is used to dynamically tune volume rendering during runtime to guarantee interactivity. For image database generation on distributed memory systems, a hybrid approach for dynamic load balancing during in situ visualization is introduced.

This work also explores how human perceptual properties can be used to improve the performance of visualization applications. Two novel techniques are introduced that adapt rendering quality to the human visual system by tracking the users gaze and changing the visualization accordingly. In this thesis, a special focus is set on volume rendering. Performance optimization makes it possible to use volume rendering to visualize data outside the typical use cases. Two visualization systems are presented that use volume rendering at their core: one for the interactive exploration of large dynamic graphs and one for the space-time visualization of gaze and stimulus data.

Overall, this thesis advances the state of the art by introducing new ways to assess, model, and predict runtime performance of visualization systems that can be used to improve usability and realize cost savings. This is demonstrated through several applications.

Zusammenfassung

Visualisierung ist ein wichtiger Bestandteil der Datenanalyse und ergänzt dabei automatische Datenverarbeitung. So ermöglicht Visualisierung oftmals einen besseren Einblick in die Daten, die zugrunde liegenden Strukturen oder Muster. Visualisierungssysteme beschreiben einen Visualisierungsalgorithmus, der auf einer bestimmten Rechenarchitektur oder einer bestimmten Hardware läuft. Die Laufzeitleistung ist für Visualisierungssysteme von entscheidender Bedeutung, insbesondere vor dem Hintergrund ständig wachsender Datenmengen und zunehmender Komplexität. Ein Grund dafür ist die Bedeutung von Interaktivität im Kontext von Visualisierung, ein anderer die umfassende Analyse generierter Daten in einem begrenzten Zeitrahmen zu ermöglichen. Die Möglichkeit, die Perspektive über den ursprünglichen Fokus hinaus zu ändern, hat sich als besonders hilfreich für die explorative Datenanalyse erwiesen. Leistungsoptimierung ist auch entscheidend für Kosteneinsparungen bei der Visualisierung auf Supercomputern, da dafür typischerweise ein erheblicher Teil an Rechenzeit benötigt wird. Die Vorhersage von Laufzeiten auf Supercomputern ermöglicht eine bessere Ressourcenplanung und optimierte Zeiteinteilung.

In dieser Arbeit werden drei zentrale Forschungsfragen behandelt, die aufeinander aufbauen: Wie kann die Laufzeitleistung von Visualisierungssystemen quantifiziert werden? Wie können diese Informationen genutzt werden, um Modelle für die Vorhersage der Laufzeitleistung zu entwickeln, und schließlich: Wie können beide Aspekte im Anwendungskontext umgesetzt werden? Ziel ist es, ein umfassendes Verständnis der Laufzeitleistung von Visualisierungssystemen zu erlangen und diese zu optimieren, um Kosten zu sparen und das Benutzererlebnis zu verbessern.

Trotz vieler Forschungsarbeiten in diesem Themenbereich gibt es noch offene Fragen und Herausforderungen, wie dieses Ziel erreicht werden kann. Eine dieser Herausforderungen ist die Heterogenität der für die Visualisierung verwendeten Rechenarchitekturen, die von mobilen Geräten bis hin zu Supercomputern reichen. Die meisten Visualisierungsalgorithmen profitieren von paralleler Ausführung, dies stellt jedoch eine weitere Herausforderung bei der Leistungsquantifizierung dar, da mehrere heterogene parallele Hardware-Hierarchien verwendet werden. Typischerweise stellen Visualisierungsalgorithmen große Daten, die viel Leerraum enthalten können, dar und sollen dabei interaktiv sein. Außerdem können sie sich in ihren Rendering-Ansätzen grundlegend unterscheiden. All diese Aspekte erschweren eine zuverlässige Leistungsvorhersage. Die vorliegende Arbeit befasst sich mit diesen Herausforderungen und stellt Forschungsarbeiten zur Leistungsbewertung, Modellierung und Vorhersage von Visualisierungssystemen vor. Zusätzlich wird aufgezeigt, wie diese Konzepte zur Verbesserung von Anwendungen, in denen die Laufzeitleistung entscheidend ist, eingesetzt werden können.

Die Erfassung der Laufzeitleistung ist ein entscheidender Aspekt für deren Verständnis und Verbesserung. Es wird ein neues Framework für die umfassende und systematische Leistungsbewertung interaktiver Visualisierungen vorgestellt, um ein tieferes Verständnis des Laufzeitverhaltens und dessen Abhängigkeiten von Rendering-Parametern zu erlangen. Basierend auf der derzeitigen Praxis der Bewertung der Laufzeitleisung in der Literatur wird eine neu erstellte Datenbank mit Leistungsmessungen präsentiert. Auf der Grundlage einer statistischen Analyse dieser Daten werden Best-Practices zur Verbesserung der Leistungsbewertung zusammengestellt. Darüber hinaus wird ein Frontend vorgestellt, das einen visuellen Vergleich der Rendering-Leistungsdaten aus verschiedenen Perspektiven ermöglicht.

Mit einem grundlegenden Verständnis des Laufzeitverhaltens einer Anwendung kann die Leistung modelliert und das Modell für Vorhersagen verwendet werden. Es werden neue Techniken für verschiedene Hardwaresysteme eingeführt, die typischerweise für die Visualisierung großer Datensätze verwendet werden: Desktopcomputer mit dedizierter Grafikhardware und High-Performance-Distributed-Memory-Systeme. Für erstere wird eine Methode zur Online-Vorhersage der Leistung verwendet, um das Volumen-Rendering während der Laufzeit dynamisch anzupassen und durchgehend Interaktivität zu gewährleisten. Für die in situ Generierung von Bilddatenbanken auf Distributed-Memory-Systemen wird ein hybrider Ansatz zur dynamischen Lastverteilung während der Visualisierung vorgestellt.

In dieser Arbeit wird außerdem untersucht, wie menschlichen Wahrnehmungseigenschaften genutzt werden können, um die Leistung von Visualisierungsanwendungen zu steigern. Es werden zwei neue Techniken vorgestellt, die die Rendering-Qualität an das menschliche visuelle System anpassen, indem sie dem Blick des Benutzers folgen und die Visualisierung entsprechend verändern. In dieser Arbeit wird ein besonderer Schwerpunkt auf das Rendering von Volumina gelegt. Die in dieser Arbeit vorgestellten Techniken zur Leistungserfassung und -optimierung ermöglichen es, Volumen-Rendering zur Visualisierung von Daten außerhalb der typischen Anwendungsfälle zu nutzen. Zwei Visualisierungssysteme werden eingeführt, die Volumen-Rendering als Kernkomponente verwenden: eines für die interaktive Exploration großer dynamischer Graphen und eines für die Raum-Zeit-Visualisierung von Blick- und Stimulusdaten.

Insgesamt bringt diese Arbeit den Stand der Technik voran, indem sie neue Techniken zur Erfassung, Modellierung und Vorhersage der Laufzeitleistung von Visualisierungssystemen eingeführt, die zur Verbesserung der Benutzerfreundlichkeit und zur Realisierung von Kosteneinsparungen genutzt werden können. Dies wird anhand mehrerer Anwendungen demonstriert.

Chapter 1

Introduction

Today, people and organizations are confronted with a huge amount of data and information on a daily basis in both professional and private contexts. Dealing with this ever-increasing flood of data is a great challenge. Many methods and techniques have been developed by computer scientists to process, store, and analyze data produced by simulations, information systems, sensors, and individuals. Visual presentation of the data is a promising way to complement automatic processing and help in the extraction of relevant information.

Scientific visualization is concerned with visualizing data with given spacial dimensions. This typically means simulation or measurement data from different science disciplines, such as medicine, physics, molecular biology, or material sciences. In those contexts, interactive exploration of the data is often desired to provide additional insight and better understanding of the simulated process or structure, beyond the initial perspective. Simulation data evolves over time, therefore scientific visualization often needs to process a temporal dimension.

Typically, exploring scientific data requires powerful hardware to handle the multidimensional data in the short time interval between rendering of consecutive frames. Often, image or data elements can be processed independently from each other. This enables the usage of hardware capable of parallel execution such as a multi core central processing unit (CPU) or graphics processing unit (GPU). Modern systems often contain both. In high performance computing (HPC) environments, thousands of nodes are connected to form a supercomputer that provides a vast amount of processing power. Each node can be equipped with multiple CPUs, GPUs, and additional acceleration hard-

ware, forming several parallel layers in a hierarchy. However, those different hardware layers all add to the system's overall complexity and quantifying performance of visual computing algorithms that run in parallel on such systems becomes a challenging problem.

Today, there are two major approaches to scientific visualization for large scale simulation data that is typically generated on supercomputers. The classical approach is to save (parts of) the generated data to disk during simulation and then perform a *post hoc* visualization by loading the data into a visualization application afterwards. This approach has the advantage of keeping at least parts of the original data, providing full flexibility, but the disadvantage of requiring great amounts of disk space and I/O bandwidth to save it. This is especially the case for data with high resolutions in spacial and temporal dimensions, which is typically generated today. I/O bandwidth in particular has become a bottleneck in recent years because of a constant progress in the area of parallel compute performance, whereas the memory and interconnect bandwidth did not keep pace on leading-edge supercomputers. Therefore, it is often required to throw away and/or compress parts of the data when using the post hoc approach. In recent time, the *in situ* paradigm has gained a wider adaption in the high-performance scientific visualization community. In situ means that the visualization is performed while the simulation is running, which allows for a direct access of the simulation data in memory, obviating the need to write data to disk [43].

Performance plays a central role for both visualization paradigms. In the case of interactive post-hoc visualization, high frame rates are crucial for user satisfaction. For in situ visualization, short render times are critical to save compute time on supercomputers since access to compute time on those systems is both in high demand and expensive. Therefore, reliable performance assessment, modeling, and prediction is of great value to scientific visualization developers and researchers, as a basis to optimize algorithms, improve user experience, and evaluate techniques. However, performance quantification is a challenge since many factors influence runtime execution times, such as the visualization technique or variant thereof, the data set, parameters, and different hardware layers. The work presented in this thesis covers the different areas of performance quantification, i.e., assessment, modeling, and prediction, and integrates them into visualization applications from different domains to steer parameters and behavior.

1.1 Research Questions

This thesis takes a holistic approach to performance quantification of visualization systems. The structure is comprised of several components that are directly linked to performance (Figure 1.1).

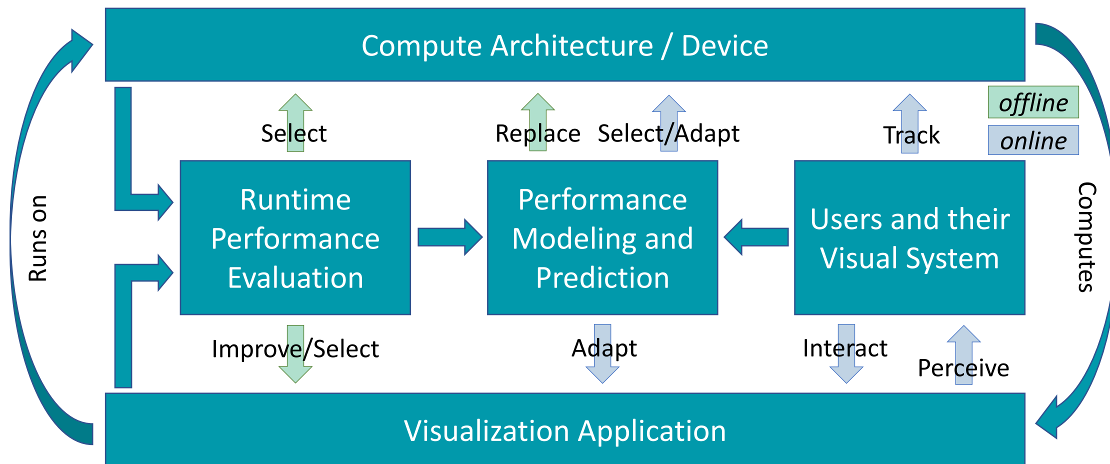


Figure 1.1: Overview of the thesis's components and their relation. A visualization application runs on a compute architecture / device. This results in a specific performance characteristic that can be evaluated. On this basis, a different compute architecture or application may be selected or the performance of the application optimized. The evaluation serves as a foundation for building models and predicting performance. Performance models can be used to dynamically adapt application parameters or hardware usage during runtime or offline. Finally, users interact with the application, thereby influencing the performance. The visualizations are perceived by the human visual system (HVS) that can be tracked and its properties exploited to improve performance.

Compute Architecture / Device. The visualization application runs on a specific compute architecture or device that has a fundamental impact on performance. Different compute architectures are used for visualization in practice, from small mobile devices to class-leading supercomputers. This thesis specifically investigates workstation systems featuring one or multiple GPUs and distributed memory systems from small visualization clusters featuring a few dozen nodes equipped with GPUs up to a leading-edge supercomputer with hundreds of pure CPU nodes. Some of the systems are equipped with eye-tracking or gaze-tracking devices.

Runtime Performance Evaluation (chapter 3). The evaluation of performance is the foundation of the components for modeling and prediction. Both, the application as well as the compute architecture need to be considered. Performance assessment enables a selection of the compute architecture that meets performance requirements. The same is true for the application, the underlying visualization algorithm, or a variant thereof. Typically, performance evaluation is the starting point for performance improvements.

Research Question 1

How can we improve the current practice in runtime performance evaluation of scientific visualizations?
Based on the systematic review of recent visualization papers on rendering volumes and particles, a general need for a better comparability and generalizability of runtime performance evaluations was identified. Recommendations for best practices when evaluating runtime performance to improve on those aspects, as well as a novel evaluation approach are presented.

Performance Modeling and Prediction (chapter 4 and chapter 5). With appropriate performance assessment, it is possible to derive performance models that can be used for prediction. Predictions can be used for different application scenarios, for instance to plan hardware upgrades. They can also be used during runtime, where accurate predictions enable the adaption of the application to meet pre-defined requirements, e.g., adapting the quality to guarantee a minimum number of frames per second (fps) for a fluid interaction. Additional use cases that benefit from performance prediction are the selection or adaption of the compute architecture. For instance, by dynamically distributing work among compute nodes to avoid idle times.

Research Question 2

How can we use performance modeling and prediction in the context of scientific visualization systems to improve performance?
This question focuses on concrete instances of performance modeling and prediction and their application. The techniques developed in this thesis propose several performance models in different (online and offline) scenarios and distinct objectives, from workstation environments to leading-edge supercomputers.

Users and Their Human Visual System (HVS) (chapter 6). By interacting with the visualization application, the user is a key influencing factor for runtime performance. Users perceive the produced renderings with their visual system, whose properties can be used to optimize performance (e.g., rendering less details where they cannot be perceived). Typically, special devices need to track the gaze of the user for HVS-based performance optimizations.

Visualization Applications (chapter 7). The visualization application is the central component that connects to all of the others others. It is also central for performance. The user interacts or steers the application and perceives the generated visualizations. Based on performance evaluation, modeling, and prediction, the application may be

selected, exchanged, and adapted. The latter can happen dynamically during runtime or offline with optimizations of the algorithms.

Research Question 3

How can we leverage performance evaluation to develop optimized visualization applications?
This question is concerned with two aspects: users and their HVS and performance optimized visualization applications. This thesis presents visualization applications for data that can be interpreted as a volume, whose performance optimizations enable interactive user interaction. Further, techniques introduced in this thesis use foveated rendering to improve application performance.

1.2 Outline and Contributions

This section outlines the thesis with short summaries of each chapter. For most of the presented publications, I am the first author and developed the software prototypes, framework extensions, data pre-processing, and conducted the data analysis myself. Some of the works presented in the following are collaborations and projects under my supervision. My supervisor Thomas Ertl and my mentor Steffen Frey were both involved as co-authors in all publications, they contributed their experience, advice, discussions, and ideas and helped analyzing and formulating many of the results.

Chapter 2—Fundamentals This chapter introduces the fundamentals of the thesis. That includes visualization and (foveated) rendering in general and direct volume rendering in particular, as the primary visualization technique used in most of the approaches introduced in this thesis. Further topics covered in this chapter include parallel rendering on GPUs, rendering in distributed environments, and in situ visualization, all of which are fundamental to the rest of this thesis. Finally, a brief discussion of performance analysis and modeling in visualization research completes the basics. An overview of the data sets and hardware used throughout the thesis is provided at the end.

Chapter 3—Runtime Performance Evaluation This chapter focuses on performance evaluation of scientific visualizations, covering an analysis of the current practice based on recent publications from volume and particle visualization techniques. We present a performance evaluation framework, where I took part in design and implementation. I also implemented the volume rendering plugin and orchestrated the execution of the benchmarks. Further, I analyzed millions of measurements with different configurations on up to eleven different GPUs [7]. Christoph Müller co-authored

this publication, implemented and discussed the different particle rendering techniques, mainly designed the architecture of our fully automated benchmarking framework, and implemented general parts of it. He also performed some of the statistical test presented. Hagen Tarner developed a web-based visual analytics system for a detailed analysis of the same data from multiple perspectives, focusing on the camera paths in particular [13]. For analysis of a second data set, I modified Intel's OSPRay framework to produce systematic benchmark measurements comparable with our own benchmark results. Further, I developed the projections of the camera paths and directions to 2D, as well as image comparisons and other data processing. Finally, I provided application use cases by investigating the data sets with the visual analytics system. Fabian Beck contributed his expertise in visual analytics design, advise, and discussion.

Chapter 4—Performance Modeling for Runtime Optimizations on GPU Systems This chapter presents a performance modeling and prediction approach for volume rendering on GPU systems featuring one or more graphics cards. I developed and implemented techniques for performance prediction during runtime of an interactive volume rendering application [2, 3]. The prediction is used for dynamic quality adaption to keep a consistent frame rate as well as for load balancing between different graphics cards in a workstation.

Chapter 5—Performance Modeling for Runtime Optimization and Cost Savings on Distributed Memory Systems Our novel techniques for performance predictions in distributed memory environments are presented in this chapter. First, a hybrid in situ visualization approach for cost efficient image database generation [6] is introduced. While I implemented the technique as an extension to the Ascent in situ framework and conducted all measurements, Matthew Larsen helped me with design and integration, and also implemented most of the Ascent integration into the Nyx simulation code. Besides general advise and ideas, Hank Childs provided suggestions for experiments, discussions and data analysis of the results. The performance prediction model for distributed volume rendering to support render hardware acquisition [14], is based on Gleb Tkachev's Master thesis, supervised by Thomas Ertl, Steffen Frey, Christoph Müller and me. In particular, I advised on the volume rendering and performance modeling. A technique for adapting encoder settings for streaming interactive visualizations to high-resolution displays is introduced [11]. The technique is based on the Master thesis work of Mathias Landwehr that I co-supervised together with Thomas Ertl, Florian Frieß and Steffen Frey. Florian Frieß integrated the work into his remote visualization framework for high-resolution displays. Besides general advise, I provided the rendering content for training and testing.

Chapter 6—Foveated Rendering to Improve Application Performance This chapter presents foveated rendering techniques to improve performance of scientific visualization applications. First, our Voronoi-based foveated volume rendering is discussed [8]. The work is based on the Bachelor thesis of Ruben Bauer who used my volume renderer as a basis for his implementation. The thesis was co-supervised by Thomas Ertl, Christoph Schulz, Steffen Frey and me. For the publication, I extended the foveated approach with a performance optimized Linde-Buzo-Gray stippling pattern that Christoph Schulz provided. Also, Daniel Weiskopf contributed his expertise. Further, an approach for foveated encoding for large high-resolution displays is discussed [10]. This publication was mainly developed by Florian Frieß, who did most of the implementation and measurements, while Matthias Braun contributed to the head-tracking implementation. Guido Reina provided his expertise, while I contributed derivation and discussion of the foveated region setup and quantization parameters.

Chapter 7—Performance-Optimized Volume Rendering Applications In this chapter, performance optimized volume rendering techniques are introduced. I developed a volume-based approach to large dynamic graph analysis [4]. Michael Burch proposed the initial idea and gave general advise, while Marcel Hlavatsch and Daniel Weiskopf contributed their expertise. The approach was later extended to include evolution provenance [1], where Houssem Ben Lamar and Melanie Herschel provided their expertise on data provenance in general and the specific evolution provenance model in particular. I also developed a space-time volume visualization of gaze and stimulus [5], based on the initial concept from Kuno Kurzhals who also provided the data and pre-processing. Daniel Weiskopf was also on hand to advise on this work.

The final chapter concludes this thesis with a summary and an overarching discussion relating to the research questions and also providing directions for future research.

Copyright This thesis reuses materials from several publications with kind permission of the respective copyright holder and first authors:

- [4, 7, 11, 10] are under copyright of IEEE,
- [2, 5] are under copyright of ACM,
- [1] is under copyright of Springer Nature,
- [8, 14] are under copyright of the Eurographics Association,
- [3] is licensed under Creative Commons license.

1.2.1 Overall Contributions

This thesis introduces various novel contributions in the field of visualization:

Performance evaluation methodology for scientific visualizations. This thesis introduces a systematic performance benchmark for scientific visualization algorithms, including millions of measurements for volume and particle rendering on different GPUs [7]. The selected parameters are based on a literature review on the current practice in performance evaluation. The results lead to recommendations for best practices in performance evaluation that can serve as a starting point for more elaborate models of performance quantification in the future. Further, a fine-grained approach for the evaluation of rendering performance is presented using a bottom-up instead of a top-down approach [13]. The approach focuses on camera configuration and comparison between different rendering techniques.

Performance modeling and prediction of rendering algorithms for runtime optimization and cost savings. The thesis contributes multiple techniques to model and predict performance in various application scenarios. For GPU-accelerated workstations, a novel approach to dynamically tune performance and quality based on user interaction is introduced, that can also be used for load balancing between multiple devices [2, 3]. For HPC, a hybrid in situ approach for image database generation is presented that dynamically shifts rendering load between nodes to save compute time that is particularly valuable on supercomputers [6]. Further, with the goal of supporting cluster hardware acquisition, a neural network-based approach is introduced that predicts the runtime performance of a distributed volume raycaster [14]. Finally, a technique that dynamically adapts encoding quality of image parts based on the content in remote rendering settings is introduced, the technique aims to provide the best quality under a given bandwidth limitation [11].

Foveated rendering techniques and performance optimized volume rendering applications. This thesis introduces two techniques related to foveated rendering. A novel approach based on Voronoi cells to accelerate volume rendering with the help of an eye-tracking device is presented [8], another approach uses gaze-tracking to reduce the throughput of remote rendering on high-resolution displays by adapting the encoding quality based on the HVS [10]. Further, two novel volume rendering techniques to visualize abstract data are introduced that are performance optimized to allow interactive exploration. The first one is used to visualize large dynamic graphs and also includes tracking and an interactive visualization of the evolution provenance [4, 1]. The second technique combines video data, eye-tracking recordings, and optical flow to a single, integrated visualization [5].

1.2.2 Awards

V. Bruder, M. Hlawatsch, S. Frey, M. Burch, D. Weiskopf, and T. Ertl. “Volume-based large dynamic graph analytics”. In: *Proceedings of the International Conference Information Visualisation (IV)*. Dec. 2018, pp. 210–219 — was awarded the best paper at the International Conference Information Visualisation (IV) 2018.

V. Bruder, C. Schulz, R. Bauer, S. Frey, D. Weiskopf, and T. Ertl. “Voronoi-Based Foveated Volume Rendering”. In: *Proceedings of EuroVis (Short Papers)*. The Eurographics Association, 2019, pp. 67–71 — was awarded the best short paper at EuroVis 2019.

F. Frieß, M. Braun, V. Bruder, S. Frey, G. Reina, and T. Ertl. “Foveated Encoding for Large High-Resolution Displays”. In: *IEEE Transactions on Visualization and Computer Graphics* 27.2 (Feb. 2021), pp. 1850–1859 — was awarded the best paper at the IEEE Symposium on Large Data Analysis and Visualization (LDAV) 2020.

CHAPTER 2

Fundamentals

This chapter covers the basics of this thesis and foundations of multiple of the presented approaches. First, a general introduction to visualization and rendering is given, with a focus on the visualization pipeline and the graphics rendering pipeline. Further, foveated rendering is briefly discussed. Most of the techniques introduced in this thesis use volume rendering as the core scientific visualization technique (subsection 2.2.1). Volume raycasting, as today's predominant technique for volume rendering in scientific contexts, is described in detail in subsection 2.2.3. As raycasting can be processed in parallel and on distributed systems, those are discussed as well, with a focus on raycasting on GPUs (subsection 2.3.1) and in situ visualization (subsection 2.3.2). This chapter concludes with an overview on performance analysis and modeling in the visualization domain (section 2.4). Data sets and hardware used in multiple experiments are listed in section 2.5.

2.1 Visualization and Rendering

Visualization generally means putting abstract data and relationships into a graphical respective visually comprehensible form. Visualizations typically have the goal to facilitate analysis, presentation, and communication of data by uncovering patterns and relationships that are not immediately obvious. It has been used in this capacity for centuries, for instance in astronomy (Johann Beyer, 1603), geography (Alexander von Humbold, 1817), and cartography (Charles Joseph Minard, 1869 [65]). With the advent of modern computer systems, these were increasingly used to generate

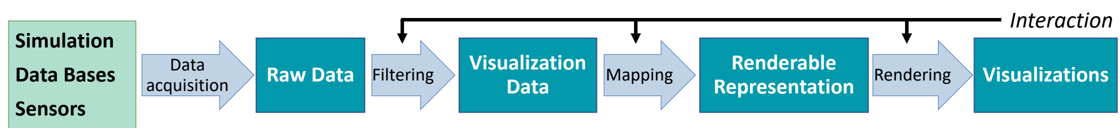


Figure 2.1: The visualization pipeline as described by Weiskopf [185].

graphical representations of data, initially on the basis of geometric descriptions [161]. Advances in simulations and sensor precision resulted in an increasing amount of non-geometric data, which led to the application of computational visualization to abstract data. McCormick et al. [134] introduce the term *visualization in scientific computation* and describe it as the transformation of symbolic information to geometric information. Today, this is referred to as *scientific visualization* and typically encompasses the visualization of measurement and simulation data with given spatial dimensions. The visualization of abstract data that lacks a spatial dimensions (i.e., it needs to be chosen) is referred to as *information visualization* [63]. Examples of such data are relationship graphs, text collections, networks and hierarchies, and multivariate data. This thesis is mainly concerned with scientific visualization in general and volume rendering in particular, one of the major algorithms used for rendering in this domain. However, some of the presented techniques can be classified as information visualization techniques, since they visualize data from performance measurements (section 3.2), dynamic graphs (section 7.1), and gaze (section 7.2).

In the following, the visualization pipeline is discussed, a step-wise process description of visualizing data. Further, the graphics rendering pipeline is introduced and how general-purpose computing on graphics processing units (GPGPU) can be used as a flexible alternative for image-order rendering. Finally, foveated rendering is described, a method that uses characteristics of the human visual system (HVS) to accelerate rendering.

2.1.1 Visualization Pipeline

Haber and McNabb [78] introduced the concept of the visualization process as a pipeline. Figure 2.1 shows a refined version of this so called *visualization pipeline* by Weiskopf [185]. First, the *raw data* is acquired from sources such as simulations, data bases, or sensors. Typically, the data acquisition is not considered part of the visualization pipeline. In the first stage of the actual pipeline, the raw data is filtered to transform it into *visualization data*. Examples for filtering are denoising, interpolation, or enhancement of data values. There can be multiple filtering operations or none at all. In the second stage, the visualization data is mapped to a *renderable representation*. Generally, this representation contains attribute fields such as geometry, color, opacity, and texture. For instance, assigning color and transparency to values in a scalar field

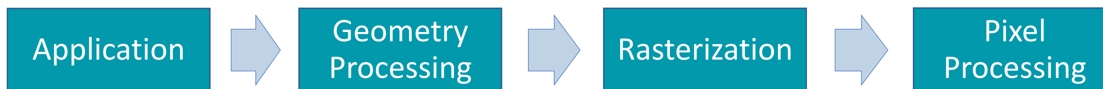


Figure 2.2: The graphics rendering pipeline [17]. The four stages may be pipelines themselves.

by using a transfer function is such a mapping. Finally, the rendering stage produces *visualizations* as an observable representation of the data. In the next section, rendering will be discussed in more detail.

A user can interact with all three steps of the pipeline. In the case of filtering, this could be the adaption of a threshold value for denoising. For mapping, an example may be the manipulation of a transfer function; and for rendering this could be the adaption of the camera angle and distance.

2.1.2 Rendering Pipeline

Figure 2.2 shows the four core stages of the rendering pipeline as used for rasterization-based, real-time rendering applications in visualization and computer graphics. Typically, these stages are further divided into sub-stages, forming small pipelines themselves. GPUs accelerate most of the stages in hardware, with some parts being fixed and others programmable. In contrast, the first stage typically runs on the CPU. It is called the *application* stage since it is driven by the application. Besides the general program flow, tasks in this stage include, for instance, data pre-processing, animation, and global acceleration techniques. At the end of the application stage, rendering primitives are sent to the next stage.

The *geometry processing* stage usually runs on the GPU and can be split into four sub-stages: vertex shading, projection, clipping, and screen mapping. In the vertex shading sub-stage, the positions of the vertices are computed and attributes such as surface normals and texture coordinates evaluated. Vertices are transformed into view space and then the view volume is projected into a unit cube. These operations typically take place in a programmable vertex shader. Optionally, the geometry processing stage also includes two other programmable vertex processing stages: tessellation and geometry shading. Both may generate new vertices and the stream output can be used instead of sending vertices further down the pipeline. In the clipping sub-stage, vertices outside the unit cube are discarded since they are not needed for further processing, while intersecting primitives get clipped. The last part of the geometry processing is mapping the clipped primitives to two dimensional screen coordinates.

The next stage of the rendering pipeline is called *rasterization*. Here, all pixels that are inside a primitive are determined. Finally, in the *pixel processing* stage, the color

of each pixel is determined using another shader program. Additionally, visibility of the fragment may be checked using depth testing, and per-pixel operations performed such as blending transparent primitives.

In recent years, the GPU has developed from an accelerator for a fixed-function version of the rendering pipeline described above to a fully programmable parallel accelerator. Using the GPU for tasks outside the graphics pipeline is known as GPGPU and can be done using platforms such as NVIDIA CUDA or the Open Computing Libray (OpenCL). In this thesis, OpenCL is primarily used in this capacity since it is platform and device portable and also runs on other parallel devices besides GPUs, such as multi-core or many-core CPUs. Importantly, those platforms allow access and forwarding of data generated on the GPU, i.e., data generated using GPGPU can be directly displayed using the traditional rendering pipeline without copying any data to the CPU.

An example of using GPGPU instead of the rendering pipeline for visualization, are image-order rendering techniques such as ray tracing. Here, instead of projecting primitives and rasterizing them, the color of each pixel is determined by shooting rays through each of them and determining the light transport between the scene's elements (objects, light sources, etc.). Volume raycasting is a special case of image based rendering and is discussed in more detail in subsection 2.2.3. In the case of image based rendering, the traditional rendering pipeline is typically only used to draw a single, screen-sized quad and texture it with the resulting image. Since the image never leaves the GPU in this case, there is no delay from transfers to the CPU. In the next section, foveated rendering is discussed as an extension to traditional rendering that improves performance by taking into account properties of the HVS.

2.1.3 Foveated Rendering

Humans perceive sharp, colorful details in a small region around the center of their gaze. This is called the foveal region. Outside this region, in the periphery, everything is perceived blurred and colorless. The ability to distinguish and recognize small objects and details is usually referred to as *visual acuity*. A large body of work has shown a fall-off in visual acuity towards the periphery of the human eye [169], it can be modeled by a hyperbolic function. This also roughly matches the average density distribution of photo receptors in the human macula and has been validated with low level vision tasks [36].

The acuity falloff in the periphery has been exploited to speed up many (object-order and image-order) computer graphics and visualization algorithms through the implementation of *foveated rendering* (e.g., [168, 75, 176]). With this technique, the image around the point of focus is rendered in high detail, while quality and details in the periphery are reduced. Most of the methods focus on object-order rendering or perception-driven acceleration of raytracing techniques (e.g., [139, 93, 184]). In their work from 1990,

Levoy and Whitaker [124] present a gaze-directed volume renderer for the Pixel-Planes 5 rendering engine. They reduce the sampling rate in the image as well as object space, thereby generating a performance increase of about a factor of five.

For remote rendering setups and pixel streaming, foveated video compression is a way to lower data throughput and therefore bandwidth requirements. For instance, Lee and Bovik [122] propose foveated video processing algorithms to reduce local bandwidth with foveation filtering. Using a foveated model, Chen and Guillemot [41] adapted the macroblock quantization adjustment in the Advanced Video Coding (H.264) compression standard. Their model enhances the spatial and temporal just-noticeable-distortion models in order to account for the relationship between eccentricity and visibility. Illahi et al. [92] propose a foveated streaming system in the domain of cloud gaming. Their approach adapts the video encoder based on the player's gaze. In their work on streaming panoramic videos in virtual reality applications, Zare et al. [194] propose to use a tiled based encoding in order to transmit their wide-angle and high-resolution spherical content to head-mounted displays. Using the High Efficiency Video Coding (HEVC) standard, they store the video content in tiles using two resolutions. Based on the user's gaze, the high or low-resolution tiles are transmitted.

2.2 Volume Visualization

As scientific computation is typically concerned with real-world phenomena, either through simulation or measurement, the according visualizations usually project three spatial dimension onto a 2D screen. To gain insight, the visualization is normally implemented to be an accurate, realistic, and detailed representation of the underlying data. A volume data set can be seen as a collection of scalar values taken from a continuous 3D signal. If the values are arranged on a regular grid, they are typically referred to as *voxels* and do not explicitly encode their location. If the volume data is arranged on a Cartesian grid, as is often the case for medical data from computed tomography (CT) scans or magnetic resonance imaging (MRI), the data set can be conveniently stored and processed using 3D textures on a GPU. Since an exact reconstruction of the 3D signal is not practical due to the computational demands, a filter is typically used in practice to approximate the signal. Today, the de facto standard is to reconstruct the signal using trilinear interpolation, an operation modern graphics processors accelerate in hardware.

2.2.1 Volume Rendering

Many different volume rendering techniques have been developed over the last decades. They can be classified into direct and indirect, as well as image space and object space techniques. In the case of indirect volume rendering, visualization is usually performed

using isosurfaces, i.e. surfaces that represent points of a constant scalar value. The isosurfaces can be generated as a triangle mesh in a pre-processing step, e.g. by using the Marching Cubes algorithm [129]. The rendering can then be done either using the classical graphics pipeline (subsection 2.1.2). Another method for rendering isosurfaces is shooting rays through the pixels of the image plane and analytically computing isosurface intersections [151].

In direct volume rendering, the volume rendering integral is discretely approximated. For object-order techniques, this means evaluating the contributions of each part of the volume to the image individually, for instance by using splatting [186], cell projection [157], or shear-warp techniques [113]. Image-order direct volume rendering approaches, today's predominant technique especially when using GPUs, determine the radiance that leaves the volume at each pixel of the image plane. The latter is used for the techniques presented in this thesis.

2.2.2 The Volume Rendering Integral

There are different optical models for light interaction with volume densities [133] that can be used as a basis for direct volume visualization. For this, the data is considered to consist of a semi-transparent material with basic physical properties regarding the interaction with light: emission, reflection, scattering, absorption, and shadowing. The most common model for scientific visualization, which is also as a basis for the rendering techniques in this thesis, is an emission-absorption model. That means, particles in a volume emit light by a factor q and simultaneously absorb incoming light with coefficient κ.

The *volume rendering integral* assumes an emission-absorption model that is evaluated along a viewing ray $\boldsymbol{R}(t)$. The rays are parameterized by the distance t to their origin, which is the position of the observer's camera. The model states that emitted radiant energy $q(t)$ at the distance $t = d$ from the camera is continuously absorbed along the viewing ray, until it reaches the camera. For a non-constant absorption κ along the ray, the remaining emitted energy q' reaching the camera can be expressed as:

$$q' = q \cdot e^{-\int_0^d \kappa(\hat{t})d\hat{t}}. \tag{2.1}$$

The second factor in Equation 2.1 is called the *transparency* T with the general form

$$T(d_1, d_2) = e^{-\int_{d_1}^{d_2} \kappa(\hat{t})d\hat{t}}. \tag{2.2}$$

Since radiant energy can be emitted from all positions t along a ray, the total energy Q can be obtained by integration:

$$Q = \int_0^\infty q(t) \cdot T(0, t)dt. \tag{2.3}$$

Considering a volume and a ray with the entry point t_0 and radiance $I(t_0)$ as well as an exit point D out of the volume, the total radiant energy $I(D)$ leaving the volume is defined by the volume rendering integral:

$$I(D) = I(t_0) \cdot T(t_0, D) + \int_{t_0}^{D} q(t) \cdot T(t, D) dt. \tag{2.4}$$

The volume rendering integral can be approximated numerically using a Riemann sum. For this, we can use the raycasting algorithm [123] that basically performs a front-to-back compositing, i.e. blending equidistant samples along the rays.

2.2.3 Raycasting

Raycasting [123] is an image-order algorithm, i.e., one or multiple rays are cast through each pixel of the image plane. The volume integral is evaluated by taking equidistant samples with distance Δ along the rays, commonly using trilinear interpolation as a reconstruction filter. That means, the scalar values of the eight neighboring voxels are considered and weighted according to their distance with respect to the sampling position. The resulting interpolated value is then typically mapped to red, green, blue, alpha (RGBA) values that represent emission (red, green, blue (RGB) color values) and absorption (alpha value). Usually, a modifiable transfer function is used for the mapping of scalar to RGBA values.

Using a Riemann sum and Equation 2.2, the accumulated absorption at t can be approximated after $N = \lfloor t/\Delta \rfloor$ samples along the ray:

$$T(0, t) \approx e^{-\sum_{i=0}^{N-1} \kappa(i \cdot \Delta)\Delta} = \prod_{i=0}^{N-1} e^{-\kappa(i \cdot \Delta)\Delta} = \prod_{i=0}^{N-1} T_i. \tag{2.5}$$

Here, a value of 1 indicates full transparency and a value of 0 full opacity. The *opacity* α on the other hand defines 1 as being fully opaque and 0 fully transparent, i.e. $T = 1 - \alpha$. Substituting this relation in Equation 2.5 (with $T_i = 1 - \alpha_i$) yields

$$T(0, t) = \prod_{i=0}^{N-1} (1 - \alpha_i). \tag{2.6}$$

If $\alpha^{(k)} = \prod_{i=0}^{k-1} (1 - \alpha_i)$ is the k-th iteration of Equation 2.6, it follows by induction:

$$\begin{aligned} k = 0 : \alpha^{(0)} &= 0 \\ k \to k + 1 : \alpha^{(k+1)} &= \alpha^{(k)} + (1 - \alpha^{(k)})\alpha_k. \end{aligned} \tag{2.7}$$

The emitted color along a ray segment i can be approximated similarly to the opacity:

$$Q_i = q(i \cdot \Delta)\Delta. \tag{2.8}$$

Algorithm 1 Front-to-back raycasting of a mapped volume $V_{rgb\alpha}$ along a ray R with sample distance Δ and a total depth D.

```
1: function RAYCASTING(V_rgbα, D, Δ)
2:     χ_α ← 0                                          ▷ initialize opacity
3:     χ_rgb ← (0, 0, 0)                                ▷ initialize color
4:     for all d ∈ {D_front ... D_back, Δ} do           ▷ sample ray with step size Δ
5:         rgb, α ← V_rgbα(R(d))                        ▷ color and opacity from mapped volume
6:         α ← 1 − (1 − α)^Δ                            ▷ adjust opacity contribution w.r.t. step size
7:         χ_α ← χ_α + α(1 − χ_α)                       ▷ blend opacity
8:         χ_rgb ← χ_rgb + α · rgb(1 − χ_α)             ▷ blend color
9:         if χ_α > 1 − ϵ then                          ▷ early ray termination
10:            break
11:        end if
12:    end for
13:    return(d, χ)                                     ▷ return depth and color at ray termination
14: end function
```

Finally, using Equation 2.6 and Equation 2.8, the volume rendering integral (Equation 2.4) can be approximated with

$$\hat{Q} = \sum_{i=0}^{N-1} Q_i \prod_{j=0}^{i-1} (1 - \alpha_j). \tag{2.9}$$

This leads to the front-to-back algorithm for direct volume raycasting (Algorithm 1) that is commonly used today and also the foundation of most of the techniques presented in this thesis. In line 6, the opacity contribution is adjusted according to the step size Δ that is given relative to the length of a voxel.

3D Digital Differential Analyzer

As an alternative to equidistant sampling along the viewing rays, the sampling distance may also be adjusted based on the content or the grid structure of the volume. For the latter, a 3D digital differential analyzer (DDA) [19] can be used that determines the next sampling position based on the distance to the next voxel along the ray. Using this method, each voxel the ray crosses is evaluated only a single time. This approach is similar to 3D line rasterization.

The technique may be slower than raycasting since neighboring rays diverge more easily which can lead to less profits from caching. However, the results are typically more accurate, especially when it is intended to render isolated voxels (e.g., the technique presented in section 7.1).

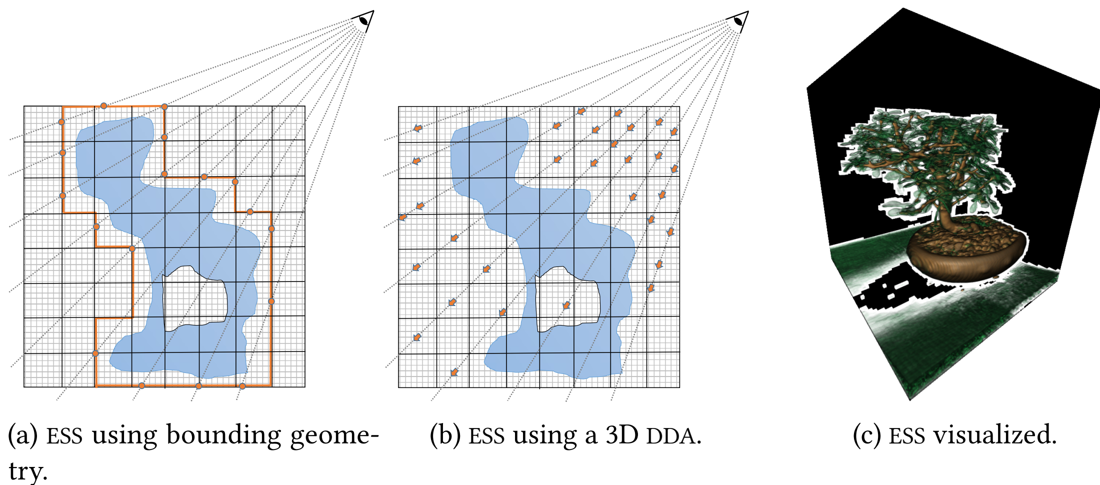
Raycasting Extensions

Many works have proposed extensions to the classic volume raycasting algorithm with the goal of shorter rendering times, higher sampling accuracy, and enriched visuals. One method to improve visuals is the use of lighting and shading, for instance using Phong-style illumination [153]. As a replacement for the normals that are typically used to calculate local illumination on opaque surfaces, the gradients can be evaluated on each sampling position along the rays. Other visual extensions to volume raycasting include for example the use of style transfer [39] and the enhancement of contours [46].

Many acceleration techniques for volume raycasting have been proposed over the years, the most straight forward one being early ray termination (ERT) [123]. As shown in Algorithm 1, ERT terminates the evaluation along the ray once the opacity approaches a value of 1. This early termination can be performed since the remaining samples would contribute practically nothing to the final color and opacity. Skipping the evaluation of those samples can save a substantial amount of processing power, depending on the data set and the transfer function.

Another commonly used acceleration technique in object-space is empty space skipping (ESS) [204, 44, 79]. Here, empty parts in the volume data (i.e., parts that do not contribute to the final image after applying the transfer function) get skipped during raycasting (see Figure 2.3c). There are many different approaches to accomplish this that depend on data properties, structure, and implementation of the raycasting. Typically, a coarse representation of the volume or a hierarchical data structure is used for processing [195]. One common approach is to render a proxy-geometry that represents a hull around the visible volume in a pre-raycasting pass and save the depth values of the front and back faces for each pixel in the image plane by using dual depth peeling [25]. Then, the depth values can be used as entry and exit points of the rays (Figure 2.3a). Another approach is to evaluate a coarse representation of the volume during runtime, for instance using a 3D DDA. Only a single lookup in memory is needed to determine if the brick in the coarse representation is empty, if it contains relevant data, the raycasting is processed with the regular fine-grained sampling. Compared to the approach using a proxy-geometry, this method needs additional memory lookups but no use of the graphics rendering pipeline (subsection 2.1.2) is required. Further, arbitrary empty blocks inside the volume can be skipped, which can only be achieved with additional render passes using the dual depth peeling method (Figure 2.3b).

Progressive rendering is a common image-space approach to accelerate rendering performance for raycasting. Progressive rendering refers to techniques that reuse pixel values of previous frames to improve rendering quality (e.g., [162, 154, 64]). Often, a low sampling rate in image space (which can be lower than the image resolution) is used during user interaction, resulting in high frame rates but a loss of image quality. If the camera is fixed, the image gradually refines over time by shooting more rays



(a) ESS using bounding geometry. (b) ESS using a 3D DDA. (c) ESS visualized.

Figure 2.3: Illustration of object-order empty space skipping (ESS): (a) A bounding geometry (orange) on a coarse grid defines ray entry and exit points. (b) A 3D digital differential analyzer (DDA) is used on a coarse grid to skip empty blocks. (c) shows the skipped blocks (black) for the Bonsai data set.

through different locations on the image plane and reusing samples from previous frames. Multiple rays per pixel typically also result in anti-aliasing on sharp edges, further improving the visual quality.

2.3 Parallel and Distributed Visualization

Many scientific visualization algorithms are computationally expensive on the one hand but can be trivially parallelized on the other, which can mitigate the high cost to some extend if enough parallel resources are available. Besides multi-core CPUs, GPUs have become a universal acceleration device for massive parallel computations in recent years, due to their many—today in the range of a few thousand—smaller, but more specialized cores. GPU architectures are primarily optimized for an aggregate throughput across all cores and less optimized for individual performance and thread latency [148, 15].

Another approach to minimize rendering time of scientific visualization algorithms is to distribute the work across multiple devices or nodes on a supercomputer. In particular, this is done for large data sets since today's GPUs are limited in video random access memory (VRAM) capacity compared to the cheaper (but also slower) random access memory (RAM) used by CPUs. Generally, there are two approaches to the rendering of scientific data on a distributed parallel systems. One is to distribute the data among the nodes such that each node renders a part of the final image. This is known as sort-first

rendering [140]. The other approach is to have each node render the full image but with only parts of the data. For visualization of large-scale simulation results, the decomposition of the data is usually done by the simulation code. In a second step, the images are then *composited* into a single, final image. This approach has been termed as sort-last rendering [140] and it has been shown to provide a better scalability than sort-first rendering, especially for large parallel jobs (e.g., [146, 191]). Compositing the images generated by each node into a single image is crucial for the efficiency of sort-last parallel rendering. Many sort-last rendering algorithms have been proposed with different advantages and drawbacks [141].

The typical technique used for communication between cluster nodes is the message passing interface (MPI). Traditionally, visualization on supercomputers has used post hoc processing, i.e., simulations save their data to disk and dedicated visualization programs read that data later. However, the post hoc paradigm is increasingly ineffective on supercomputers, as the ability to generate data on each new generation of supercomputer is increasing much faster than the ability to store and load data. As a result, I/O load times for visualization are becoming unacceptably large, as are I/O times for simulations performing frequent storage. In contrast to post-hoc visualization, the so called *in situ* visualization takes a different approach. Here, the visualization runs at the same time the simulation is running, thereby sharing the data directly in memory, avoiding the need to utilize the file system. Currently, in situ is increasingly the preferred processing paradigm on leading-edge supercomputers [42, 152] and has been used in many recent visualization works (e.g., [34, 58, 189, 149]).

In the following, volume rendering on GPUs is highlighted in more detail as an exemplary parallel rendering technique in the field of scientific visualization. The section is concluded by an in-depth discussion of in situ visualization, covering instantiations as well as hybrid and elastic approaches.

2.3.1 Volume Raycasting on GPUs

The performance advances of GPU architectures in recent decades, as well as the addition of general-purpose computing capabilities have made GPU-based raycasting the de-facto standard approach for interactive volume rendering in single-device environments [80, 32]. Typically, the contribution of each ray can be calculated independently from each other while at least clusters of neighboring rays are largely coherent. This makes volume raycasting by design a good fit for modern GPU's single instruction, multiple threads (SIMT) architecture. The first implementations on GPUs required multiple passes through the render pipeline [108], but this has since been reduced to a single pass with the advancement of GPU programming capabilities [167, 81].

Typically, the volume data set is uploaded as a 3D texture to GPU memory because this enables faster lookups due to optimized memory layout and hardware-accelerated

trilinear interpolation. Initially, an intersection test with the bounding box of the data set is conducted to determine the ray entry and exit points for each ray and if it needs to be evaluated at all. Alternatively, a pre-pass to determine the entry and exit points for a bounding geometry is carried out. Then, the rays are sampled with equidistant steps in the respective range. For each sample, a trilinear interpolation is used to reconstruct the scalar value at the respective position from the volume data. By using a transfer function, the scalar value is mapped to color and opacity values that are used to estimate the volume rendering integral (subsection 2.2.2). Illumination can be calculated on the fly, by estimating the gradients with central differences. The reconstruction of the volume signal can be improved at almost no cost by reusing the additional samples needed for the gradient estimation [45].

2.3.2 In Situ Visualization

There are many possible instances of in situ processing, varying over division of resources, access to data, and other factors [43]. There are two instances that are used most commonly. In the first instance, sometimes referred to as *inline* or tightly coupled in situ, the simulation code and visualization routines run on the same compute nodes, accessing the same memory and alternating usage of a node's cores. In this setting, visualization routines are typically integrated into the simulation via a library, and the simulation code invokes this library whenever visualization is required, effectively giving up control of the compute resources until the visualization routine returns from its function call. Popular products that use this form include Catalyst [22], Ascent [116], and LibSim [187]. In the second instance, sometimes referred to as *in transit* or loosely coupled in situ, the simulation code and visualization routines use distinct resources, often referred to as *simulation nodes* and *visualization nodes*. In this setting, the simulation code typically sends its data to the visualization nodes via network communication, and the visualization nodes will keep its own separate copy of the simulation data. Popular products that use this form include Damaris [55], Sensei [23], and Adios [67].

To date, there have been few true hybrid in situ approaches that blend between inline and in transit. Notably, Bennett et al. [28] used a hybrid in situ approach for S3D combustion simulations, using inline computations to reduce data such that modest in transit resources could be used to complete analysis tasks. In another notable work, Zheng et al. [202] introduced *PreDatA*, where compute nodes could do *local processing* before sending data to in transit nodes, although most of the calculations took place on in transit nodes.

Several works have focused on addressing inherent inefficiencies with in situ processing. With Flexpath [48], the authors focus on saving transfer costs by reducing data movements or optimizing the data placement based on network topology and other performance influencing factors. Damaris [54] considered the issue of variability, while

Kress et al. considered using in transit to reduce scalability in two separate studies [104, 105]. However, they use a fixed amount of visualization resources.

There are also several works on assessing resource usage of inline in situ and in transit analysis and visualization tasks, including quantitative formulations [200, 53]. Friesen et al. [66] discuss in situ experiments for the two instances with the Nyx simulation code and two in situ analysis suites while mainly considering overall execution time performance. Other works focus on general workflow optimization and orchestration [181, 136].

Those works mainly focus on performance instead of cost, i.e., reducing the overall time to solution often at the cost of additional node seconds by using additional resources. Further, either in transit or inline processing is used, but not both in a hybrid fashion. In the case of in transit, rightsizing is not considered although several works acknowledge the problem.

There have been several works exploring *elastic in situ* [56], as a promising direction to optimize visualization workloads. Elastic in situ describes techniques for resource adaptation over the execution time between simulation and visualization. Goldrush [201] is a system that identifies when simulation resources are idle and used them to perform analysis tasks. Landrush [69] extends this idea to use idle cycles on GPUs. Melissa [173] supports a design where a server processes data from multiple independent simulation groups that connect dynamically. Dirand's TINS system [52] approaches the problem from a task-based perspective, with resources being allocated for analytics when such tasks emerged.

2.4 Performance Analysis and Modeling

Performance analysis has been an integral part of visualization research since its inception. It is used and recommended for comparisons and to show the limits of the proposed algorithms [115]. However, there are still papers in the visualization domain that do not contain comprehensive experimentally validated results or provide only results with a limited informative value or generalizability [174, 62]. These shortcomings lead to a still active field of research in performance analysis for scientific visualization.

2.4.1 Volume Visualization and GPU Performance

Since interactivity in visualization is often crucial, as it allows for an exploratory approach that enables users to gain insight beyond the original focus, enhancing the performance of GPU-based volumetric raycasting in different forms has emerged into a significant field of research on its own [32]. However, even in this well-studied context the performance achieved in practice significantly varies with factors such as data sets,

parameters, and hardware in a way that is hard to assess beforehand. Bethel et al. [31] tested a variety of settings, algorithmic optimizations, and different memory layouts for the Intel Nehalem, AMD Magny Cours, and Nvidia Fermi architecture to assess and tune parameters for parallel raycasting. They found that optimal configurations vary across platforms, often in non-obvious ways. This type of detailed performance analysis is not only of interest for hardware acquisition or parameter tuning, but is also the basis for improving algorithm performance (e.g. [29, 118]).

2.4.2 Modeling of GPU and Scientific Visualization Workloads

Naturally, results of a thorough performance analysis can be used to model an application's performance. Many different approaches have been proposed, also in the field of scientific visualization and GPU performance modeling in general. Examples are generic performance models [88, 49, 102] and approaches specific to a selected hardware architecture (e.g., Nvidia GPUs of the 200-series [199]). Those approaches are typically based on micro-benchmarks (small pieces of code to test for a specific performance characteristic) and target more or less generic compute applications. There has also been work focusing on scientific visualization, especially in the domain of high-performance computing. For instance, Rizzi et al. [156] presented an analytical model to predict scaling behavior of parallel volume rendering on GPU clusters. Larsen et al. [117] took a more general approach and modeled the performance of in situ visualization in the context of high-performance computing. In their work, they investigated three rendering algorithms: ray casting of regular grids, raytracing of iso-surfaces using only primary rays, and a rasterization implementation that is based on sampling with barycentric coordinates. Other proposed performance modeling and prediction methods in the field of rendering and scientific visualization focus on object-order rendering algorithms (e.g., [188, 171]) or create a performance model for a visualization pipeline [35].

2.5 Data Sets and Hardware

In this thesis, several dedicated GPUs are used for runtime performance measurements. The selection encompasses multiple generations of the two major vendors of discrete graphics cards: AMD and Nvidia. For the evaluation of the presented volume rendering algorithms, two dozen volume data sets were used. They include measured data from computed tomography (CT) scans, simulation data, and artificially generated data sets.

Table 2.1 provides a list of all GPUs that were used for evaluation of the techniques presented in this thesis. The respective sections are referenced. The listed key specifications include the architecture, the number of shader units (SU) and the amount of video random access memory (VRAM).

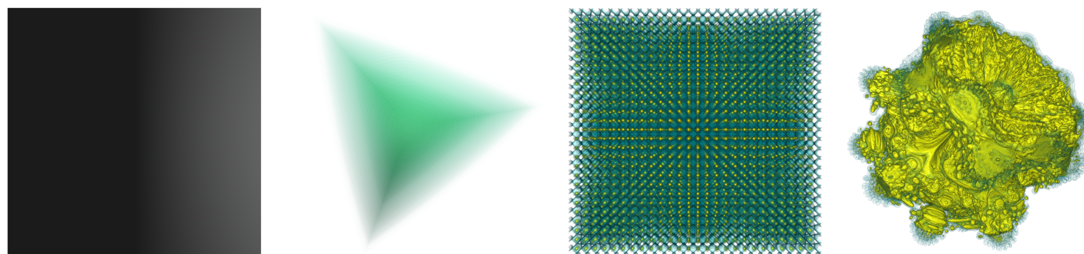


Figure 2.4: Renderings of volume data sets that are artificially generated and used in various experiments. The order is according to Table 2.2 (bottom).

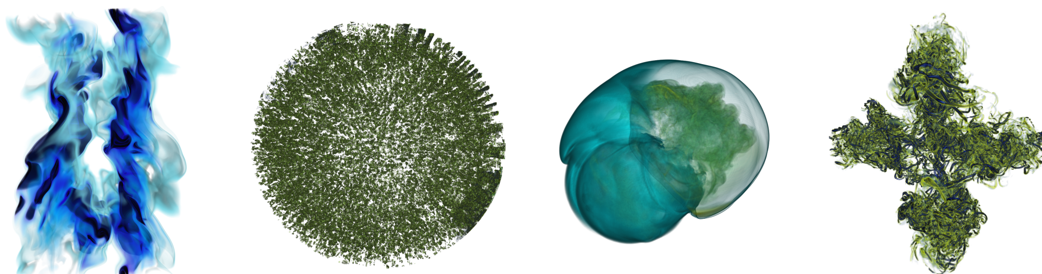


Figure 2.5: Renderings of volume data sets from simulations used in various experiments. The order is according to Table 2.2 (middle).

Table 2.1: Graphics Cards Used for Performance Measurements

Vendor	Model	Arch.	SU	VRAM	Used in
AMD	Radeon R9 290	GCN 2	2560	4 GB	3.1.2
AMD	Radeon R9 Nano	GCN 3	4096	4 GB	3.1.2
AMD	Radeon RX 480	GCN 4	2304	8 GB	3.1.2, 3.1.3
AMD	Radeon Vega FE	GCN 5	4096	16 GB	3.1.2, 3.1.3
Nvidia	GeForce GTX 480	Fermi	480	1.5 GB	5.2
Nvidia	GeForce GTX 680	Kepler	1536	4 GB	4.5 (A)
Nvidia	GeForce GTX 960	Maxwell	1024	4 GB	4.5 (B)
Nvidia	GeForce GTX 980	Maxwell	2048	4 GB	3.1.2, 4.5 (B)
Nvidia	Quadro M6000	Maxwell	3072	12 GB	3.1.2, 3.1.3, 5.2, 5.3, 6.2
Nvidia	GeForce GTX 1060	Pascal	1280	6 GB	6.2
Nvidia	GeForce GTX 1070	Pascal	2432	8 GB	6.1
Nvidia	Titan X (Pascal)	Pascal	3584	12 GB	3.1.2, 3.1.3, 4.5 (B), 7.1.5
Nvidia	GeForce GTX 1080 Ti	Pascal	3584	11 GB	3.1.2, 3.1.3
Nvidia	Quadro GP100	Pascal	3584	16 GB	3.1.2
Nvidia	Titan Xp	Pascal	3840	12 GB	3.1.2, 3.1.3
Nvidia	Tesla V100	Volta	5120	16 GB	3.1.2

Table 2.2: Volume Data Sets Used for Performance Measurements

Volume	Resolution	Size	Source/Creator	Used in
Bat	$1024^2 \times 720$	16 bit	UTCT	3.1.2
Bonsai	$256^2 \times 256$	8 bit	S. Roettger	3.1.2
Chameleon	$1024^2 \times 1024$	16 bit	UTCT	3.1.2, 4.5, 5.2, 5.3
Engine	$256^2 \times 128$	8 bit	General Electric	3.1.2
Field mouse	$1024^2 \times 975$	16 bit	UTCT	3.1.2, 4.5
Flower	$1024^2 \times 1024$	8 bit	University of Zurich	4.5
Foot	$256^2 \times 256$	8 bit	Philips Research	3.1.2
Foraminifera	$1024^2 \times 219$	16 bit	UTCT	3.1.2
Hazelnut	$512^2 \times 512$	8 bit	University of Zurich	3.1.2
Hoatzin	$1024^2 \times 729$	16 bit	UTCT	3.1.2, 4.5
Kingfisher	$1024^2 \times 885$	16 bit	UTCT	3.1.2, 4.5
Parakeet	$1024^2 \times 340$	16 bit	UTCT	3.1.2, 4.5
Skull	$256^2 \times 256$	8 bit	Siemens Medical	3.1.2
Stanford Bunny	$512^2 \times 361$	8 bit	T. Yoo et al. / NLM	3.1.2
VisFemale	$512^2 \times 512$	8 bit	NLM	3.1.2
Zeiss	$640^2 \times 640$	8 bit	Daimler AG	3.1.2, 4.5
Combustion	$480 \times 720 \times 120$	8 bit	J. Chen	6.1
Porous media	$2048^2 \times 256$	8 bit	—	3.1.2
Supernova	$432^2 \times 432$	8 bit	J. M. Blondin	3.1.2
Vortex Cascade	$529^2 \times 529$	8 bit	A. Beck & C.-D. Munz	6.1
Uniform box	$512^2 \times 512$	8 bit	—	3.1.2
Gradient box	$512^2 \times 512$	8 bit	—	3.1.2
Frequency box	$512^2 \times 512$	8 bit	—	3.1.2
Mandelbulb	$512^2 \times 512$	8 bit	D. White & P. Nylander	3.1.2

Top: data sets from CT-scans (Figure 2.6)
Middle: data sets from simulations (Figure 2.5)
Bottom: artificially generated data sets (Figure 2.4)

In Table 2.2, all volume data sets are listed. All are regular grids with the listed voxel resolution and scalar precision. Representative renderings of the data sets can be found in Figure 2.4 (artificially generated), Figure 2.5 (simulation data), and Figure 2.6 (CT scans).

Figure 2.6: Renderings of volume data sets from CT-scans used in various experiments. The order is according to Table 2.2 (top).

Chapter 3

Runtime Performance Evaluation

In general, the analysis of algorithmic and computational performance has always been a central aspect of computer science research. In the field of scientific visualization this is of a particular interest, since there is a high computational demand resulting from rendering algorithms and a strive for interactive frame rates. The typical approach to performance evaluation in scientific visualization papers dealing with interactive techniques is the performance measurement for several, often hand picked data sets under the assumption of representability. In many cases, authors compare their proposed techniques with comparable ones on one or two different hardware systems. The fact that there are numerous factors that can influence performance even for simple visualization algorithms raises the question to which extent such common evaluations are actually representative. Does using only a small subset of possible configurations such as a few data sets and a single computing device lead to missing influential performance characteristics? Which are the most important factors that need to be investigated closely to convey a comprehensive picture of rendering performance? Are there typical correlations of factors that might help finding a concise, yet complete description? This chapter discusses these questions from two directions.

In section 3.1, a top-down approach is used that incorporates descriptive statistics on thousands of different configurations from a systematic performance benchmark [7]. Independent and linear parameter behavior is discussed as well as non-obvious effects. A list for best practices when evaluating runtime performance of scientific visualization applications is compiled, which can serve as a starting point for more elaborate models of performance quantification.

In section 3.2, a more specialized and fine-grained approach for the evaluation of rendering performance is presented that takes multiple perspectives into account: camera position and orientation along different paths, rendering algorithms, image resolution, and hardware [13]. The approach comprises of a visual analysis system that shows and contrasts the data from these perspectives.

This chapter is partly based on these publications

- V. Bruder, C. Müller, S. Frey, and T. Ertl. "On Evaluating Runtime Performance of Interactive Visualizations". In: *IEEE Transactions on Visualization and Computer Graphics* 26.9 (2020), pp. 2848–2862 [7]
- H. Tarner, V. Bruder, T. Ertl, S. Frey, and F. Beck. "Visually Comparing Rendering Performance from Multiple Perspectives [in preparation]" [13]

3.1 Empirical Evaluation of GPU-Accelerated Interactive Visualizations

There are different forms of performance analysis ranging from pure analytical models to those that are solely based on measurements and hybrids of the two. For interactive visualizations, using empirical measurements to report performance is by far the most common approach. This is mainly because interactive applications can typically generate many measurements with a high frequency, thereby requiring only little alterations to the code. This way, empirical measurements can be used to capture performance characteristics of various performance-influencing factors for interactive applications in a reasonable amount of time.

In direct comparison with analytic performance modeling, there are further advantages of an empirical approach in the case of interactive visualizations. First, concrete performance numbers achieved in practice are reported. However, using measurements instead of theoretic estimates also raises the question of how portable performance numbers are to other systems, data sets, and configurations. We address this in part with our approach by performing and analyzing an extensive amount of measurements for different combinations of those performance influencing factors. In general, there are cases in which the assessment of portable performance numbers becomes very hard or even impossible. With increasing complexity of a system, it becomes difficult to take all possible influencing factors and their combined performance characteristics into account. In those cases, an empirical approach is often better suited. Finally, the performance impact of some factors is hard or even impossible to assess without executing the algorithm. An example is early ray termination (ERT), an acceleration technique for volume raycasting (see subsection 2.2.3). The performance impact of ERT

highly depends on the data set and employed transfer function. When using ERT, even small changes to the transfer function can substantially impact the number of samples along rays, for instance in the case where a large surface changes from transparent to fully opaque. Changes like this are very hard to capture in a model, for example if the performance is estimated based on the numbers and the cost per sample.

3.1.1 Measurement and Analysis

We compare the current practice of empirical performance evaluation in scientific visualization research to our approach of using extensive measurements. This includes investigating if the evaluation criteria typically used in publications are a generally valid approach to report performance. Further, we evaluate how to improve on the current practice to make performance analysis more expressive. We look at the common approach of performance evaluation for novel or improved techniques in scientific visualization. However, this approach does not focus on comparing against performance evaluation techniques in particular.

Our basis is the systematic measurement of an extensive amount of different parameter configurations and data sets on various systems (i.e., GPUs) for two scientific visualization techniques: volume raycasting and particle rendering. Both techniques are custom implementations that are embedded in a benchmarking framework handling program flow, automatic parameter changes and logging of the runtime measurements.

We investigate the performance of the two techniques in terms of frame times that are a crucial metric for interactive applications. There are also other performance measures we are not concerned with here, but which are relevant depending on the application scenario (e.g., power consumption, memory usage). Further, we restrict ourselves to scientific visualization applications in a single node environment that use GPUs without out-of-core-techniques. We presume that many of the concepts may be transferred to a broader field of visualization applications and usage scenarios.

For data analysis, we use descriptive statistics since a investigation of single frame times does not allow many conclusions to be drawn about performance behavior in general for the many different configurations we covered in our benchmark. Therefore, we mainly look at distributions, linear correlations of influencing factors, subsets of our data, and general patterns. Finally, we do not investigate single outliers and specific measurement points.

Experiment Design

For our experiment, we developed an extensible benchmarking framework allowing us to automatically evaluate all combinations of influencing factors (rendering parameters, resolution, data set, etc.) based on a declarative description. The factors that can be

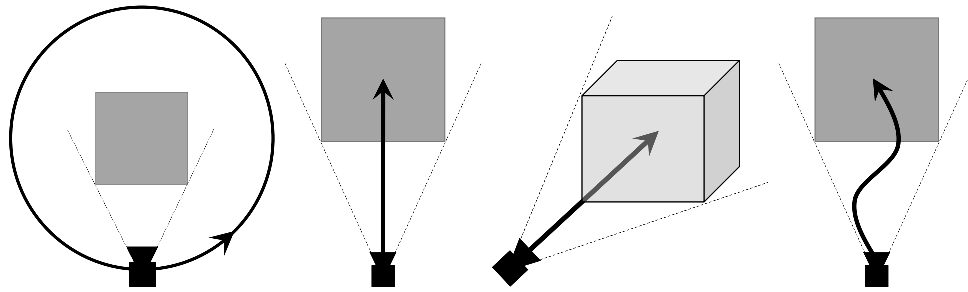


Figure 3.1: The camera paths used for measurements (left to right): orbit around the data set, straight path along the axes, diagonal, and sine curve. © 2020 IEEE [7].

measured also include a series of pre-defined camera paths ranging from commonly used orbits around the data set, over fly-throughs on straight or curved paths, to randomly selected views (Figure 3.1). Our framework provides a plugin mechanism, by means of which different kinds of compute/rendering application programming interface (API)s and devices can be added. For our experiment, we implemented an OpenCL 1.2 and a Direct3D 11 environment since both APIs support GPUs from different manufacturers. Refer to Table 2.1 for an overview on the GPUs we used in our experiment. For measuring the rendering performance, we used similarly equipped machines comprising of an Intel Xeon E5-2630 CPU running at 2.2 GHz and 64 GB of RAM per CPU for all GPUs except for the NVIDIA Titan Xp (Intel Core i9-7900X at 3.3 GHz) and the GeForce GTX 1080 Ti (Core i7-7700K at 4.2 GHz).

Analysis Process

We follow a top-down approach to analyze the generated data and gain insights into our large body of performance measurements with little prior knowledge. Accordingly, we start by looking at the overall distribution of obtained timings, investigating whether performance data of volume rendering and particle raycasting follow as known distribution. In a second step, we continue by investigating if we can find a linear relationship between factors that we changed during the experiment. Our focus on linear dependencies has several reasons. First, those dependencies may be a convenient and intelligible way to describe the overall influence of a factor. Second, we would expect factors such as the hardware device to have a linear influence on the results as long as all devices have the same capabilities (since we do not support out-of-core rendering). In general, we first compare correlation coefficients on a per feature level, i.e. we look at the means and ranges of the correlation factors for the different particularities of each feature.

We proceed our analysis process by looking into each of the influencing factors that we varied in more detail by further investigating correlations. Here, we focus on single values of the respective factor, i.e. examining correlation matrices. By calculating a linear

regression on the factors that behave linearly (as determined before), we additionally calculate concrete speed-ups. We conclude the analysis process with a closer look at interesting findings.

3.1.2 Case Study 1: Volume Raycasting

In a first case study, volume rendering performance is investigated. Volume visualization is a computationally demanding application, especially for high resolution data sets from state-of-the-art scanners and simulations (see subsection 2.2.1). We consider GPU raycasting techniques in particular since they are the de-facto standard for volume rendering in non-distributed environments.

Common Practice

Table 3.1 provides an overview on publications with a strong focus on GPU volume raycasting and performance-related aspects to assess the common practice of performance evaluation in the field during the last two decades. The majority of publications deal with real-time volume rendering in single node environments. After a review of their individual approaches for performance evaluation, we identified the following commonalities that are shared by a majority of the works.

Except Wu et al. [190], a single GPU was used in all papers (in almost all cases from Nvidia). On average, around five data sets were evaluated. Transfer functions were unspecified in most cases, but are often depicted in figures. Across different papers, the size of the data sets ranged from less than one Megavoxel to several Gigavoxels (with the exception of Hadwiger et al. [79], who used a data set with more than one Teravoxel). In most cases, camera positions and parameters were not specified explicitly. Five papers used orbital rotations around one or three axes [180, 170, 121, 167, 38], while three papers employed pre-defined user interaction sequences [183, 2, 64]. The size of the viewports were specified in most cases, with some exceptions [145, 95, 198, 198, 192, 84]. In most cases, a single resolution was measured, with a pixel count of less or around one Megapixel (1024×1024). In some papers, several viewports were evaluated [131, 180, 127, 167]. Other rendering parameters were given to some extent. For instance, raycasting step sizes are specified or described in roughly half of the papers listed. Algorithmic variations, such as acceleration techniques and illumination methods, were stated often, but not always. In terms of performance indicators, most authors provided at least a single fps value per data set. Although it is seldom stated explicitly, we assume these numbers to be average values of multiple measurements. Some authors additionally provided minimum and maximum frame rates [80, 167], speed-ups or even frame time diagrams. Memory consumption is also reported in some of the works.

Table 3.1: Performance Evaluation in Recent Selected Volume Rendering Papers

Author	Year	GPUs[d]	Viewports	Data Sets	Views
Wald et al. [179]	2021	N	1	4	–[a]
Waschk and Krüger [183]	2020	N	1	2	[b]
Morrical et al. [145]	2019	N	–[a]	4	–[a]
Hadwiger et al. [79]	2018	N	1	8	–[a]
Magnus et al. [131]	2018	N	3	8	–[a]
Wang et al. [180]	2017	N	3	3[c]	rot. x,y,z
Jönsson et al. [95]	2017	N	–[a]	4[c]	–[a]
Wu et al. [190]	2017	2×N	1	6	–[a]
Sans et al. [158]	2016	N	1	4	–[a]
Zhang et al. [198]	2016	I	–[a]	7	–[a]
Ament et al. [20]	2016	N	1	6	–[a]
Bruder et al. [2]	2016	N	1	6	> 500[b]
Zhang et al. [196]	2016	N	–[a]	6	–[a]
Ding et al. [51]	2015	N	1	4[c]	–[a]
Sugimoto et al. [170]	2014	N	1	2	rot. x,y,z
Hero et al. [85]	2014	N	1	5	–[a]
Frey et al. [64]	2014	N	1	4[c]	> 1000[b]
Lee et al. [121]	2013	N	1	5	360
Liu et al. [127]	2013	N	2	9	–[a]
Yang et al. [192]	2012	N	–[a]	6	–[a]
Jonsson et al. [94]	2012	N	1	5	1 or 2
Kronander et al. [106]	2012	N	1	8	–[a]
Schlegel et al. [159]	2011	N	1	7	–[a]
Hernell et al. [84]	2010	N	–[a]	1	–[a]
Zhou et al. [203]	2009	N	1	4	–[a]
Lundström et al. [130]	2007	N	1	3	–[a]
Ljung et al. [128]	2006	A	1	4	2
Stegmaier et al. [167]	2005	N	2	1	rot. y
Bruckner et al. [38]	2005	N	1	1	3×360
Krüger et al. [109]	2003	A	1	4	–[a]

[a] Property not mentioned in the paper.
[b] Interaction sequence with variable number of views.
[c] Time-series data sets.
[d] N = Nvidia, A = AMD/ATI, I = Intel.

In this first case study, we test the parameters usually evaluated in common practice: GPUs, viewports, data sets, and camera paths. We do not consider parameters that are specific to certain techniques such as illumination. Each parameter dimension was sampled at least as extensively as in the respective papers. In addition to the parameters directly derived from common practice, we measured different sampling sizes along the ray, two acceleration techniques, and transfer functions. Testing various GPUs helps in understanding device portability. Sampling rate parameters are numerical and have a well defined order, therefore providing means to test for scalability. Different data representations are covered by measuring data sets and transfer functions. Since the camera setup is not reported in many of the reviewed papers, we wanted to test how different camera paths influence performance and what can be considered most representative. Evaluating with and without ERT and object-order empty space skipping (ESS) provides us with the option to emulate accelerated rendering techniques.

Experiment and Implementation

We systematically analyze the performance of volume rendering, using our performance evaluation framework (see section 3.1.1) and our custom implementation of a GPU volume renderer. Our design choices are directly derived from the reviewed papers to reflect the current standard approach in the field. Accordingly, we employ a front-to-back volume raycaster with perspective projection, which usually results in a higher thread divergence compared to orthographic projection. It features local illumination based on gradients (calculated with central differences), and we can optionally enable ERT and ESS (see subsection 2.2.3). For our implementation, we make use of the OpenCL 1.2 API because it enables us to compare the same implementation across graphics cards from different vendors. Volume raycasting is a comparably simple algorithm and its parallelization is trivial, which minimizes the advantage of vendor-specific APIs with respect to kernel execution times. Finally, vendor optimized third party libraries are not used.

Table 3.2 lists the parameters that we varied in our volume rendering benchmark. The selection is based on our review of the common practice. In the following, we denote them as *factors* that influence performance. Further, we divide those factors into two classes: numerical (including viewport and the step size along rays) and categorical (the rest). We propose this distinction, because categorical factors do not have a well defined order, while we can naturally sort numerical ones. The categorical factors also include the binary values, such as the use of ERT or ESS. Besides the variations of sampling resolutions in object space and image space, we used 21 different data sets in our experiment. We test two different transfer functions, because in combination with the use of acceleration techniques, distinct transfer functions can be seen as different data shapes. The step sizes in Table 3.2 are relative to the voxel length. That means, a step size of 0.5 results in 2 samplings in the length of a voxel's edge.

Table 3.2: Parameters of the Volume Rendering Benchmark

Factor	Instances	Values
Device	11	see Table 2.1
Viewport	3	512^2, 1024^2, 2048^2
Step size	4	$0.25, 0.5, 1.0, 2.0$
Acceleration	2×2	ERT, ESS (on/off)
Camera path	7×36	$orbit_{x/y}, diagonal, path_{x/y/z/\sin z}$
Transfer function	2	see Figure 2.4–Figure 2.6 for examples
data set	21	see Table 2.2

We ran our application on various dedicated GPUs from NVIDIA and AMD (i.e., no integrated GPUs were measured), spanning multiple generations of different architectures. All of the GPUs used for the benchmark have sufficient video memory to store the test data sets. To study the impact of simple performance improvement techniques, we enabled respectively disabled ERT or ESS. We employ an ESS approach that is designed to be used in a single rendering pass using a 3D DDA (subsection 2.2.3). The time needed for data pre-processing is not part of the measurement. Further, the majority of the data sets we measured have also been used for performance evaluation in the reviewed papers (Table 3.1). Their resolutions range from $256^2 \times 128$ up to 1024^3 voxels. The spacing along the x- and y-axes is the same for all data sets, while the slice size along the z-axis differs for about half of the data sets. Notably, the tested data sets also include three generated cubic data sets: one with a uniform density value, another one with a density gradient from one corner to the opposite, and one featuring a high frequency pattern generated by using trigonometric functions. Exemplary renderings of all data sets (but the uniform box) are shown in Fig. 2.6. Although our implementation can handle and convert 8, 16, 32 and 64 bit data precision and sampling precision per voxel, we limit ourselves to 8 bit sampling precision for a better manageability of the data (every factor significantly raises the number of measurements due to the added dimension). In our pre-tests, they exhibited linear behavior for different data precision across most devices. The performance on AMD GPUs was affected a little less than on NVIDIA cards when increasing the scalar precision, with the exception of the AMD N9 Nano that was affected most of all cards.

To emulate different evaluation methods, we designed simple camera paths that can be sampled in arbitrarily small intervals. To keep the computational effort manageable, we used 36 samples (camera positions) per path, adding up to a total of 252 different camera positions. The employed paths are schematically depicted in Fig. 3.1. We use a full orbit around the x-axis and y-axis (as in [170, 180, 121, 167, 38]), three straight

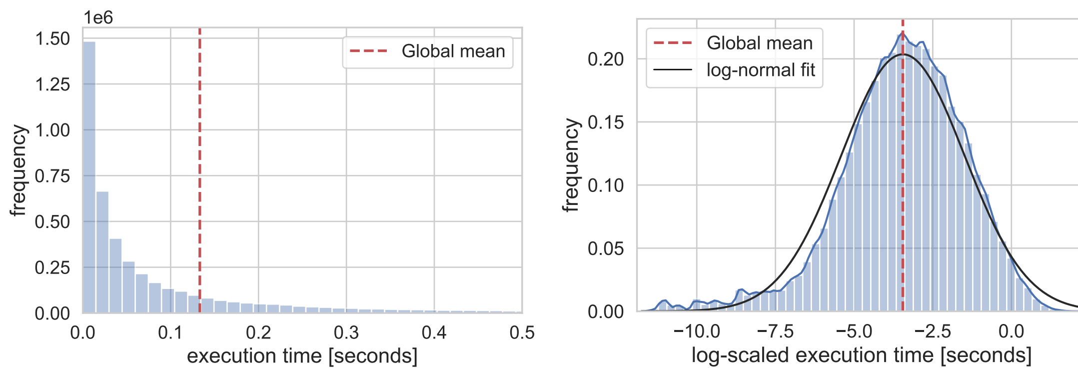


Figure 3.2: Overall distribution of execution time medians of the volume raycasting benchmark. The right plot has a logarithmic scaling.

paths into the volume data set along each axis, a diagonal path from one corner of the bounding box towards the opposite one, and one path along a scaled sine-curve in z-direction. All paths begin with a full view of the bounding box and (except the orbits) stop at its center. For the orbits, the view direction is always towards the center of the data set. We measured the kernel execution time of each configuration by using the queue profiling functionality of OpenCL, and rendered every configuration at least five times. In total, this amounted to over 25 million measurements covering more than five million distinct configurations.

Analysis of the Results

Performance Distribution. The overall distribution of all kernel execution times of our volume raycaster is shown in Figure 3.2 (left). The chart includes all measured configurations (using the median of the execution times for measurements with the same configuration). In general, the differences between runs with the same configuration are negligibly small (mean standard deviation of $0.002\,\mathrm{s}$). In some cases, the first run is slightly slower than the others (especially on AMD GPUs), which we attribute to caching effects of the kernel. Applying a logarithmic scaling on the measured execution times reveals a log-normal nature of the distribution (Figure 3.2, right). For verification, we performed a one-sample Kolmogorov-Smirnov test on the data, which rejected the null hypothesis of the sample coming from a log-normal distribution ($D = 0.78, p < 2.2 \cdot 10^{-16}$). On the one hand, there are some visible deviations in the tails (see Figure 3.2). On the other hand, it is a known problem of the used test to become very sensitive to even small deviations for large sample sizes as in our case (more than five million configurations). Although we cannot claim statistical support for the assumption that the performance of volume renderings is log-normal distributed, the visual inspection of the histogram shows that it is very close. The global distribution

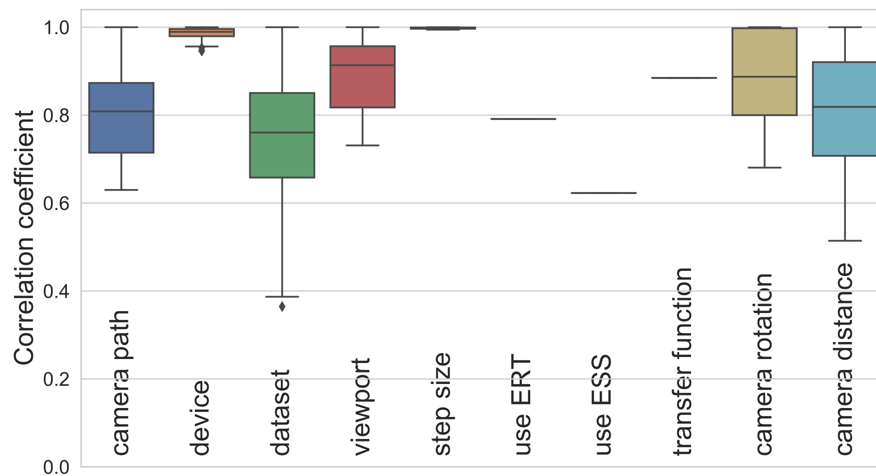


Figure 3.3: Pearson correlation coefficients for volume raycasting as boxplot for the observed performance influencing factors, camera rotation, and camera distance.

gives a general impression of the range and occurrence of the execution times. However, this representation does not provide any details about the influence of the different factors we varied in our measurements.

Linear (in)dependence of features. In a second step, we investigate the correlation of different factors with respect to the execution time (the target) with the other values or categories from the respective factor. For instance, we consider all timings obtained for the "NVIDIA Titan Xp" being one data set and all obtained for the "AMD Radeon Vega FE" being the other. Then, we compute the Pearson correlation coefficient between these two disjunctive data sets. The result gives us an idea of whether there is a linear correlation between the rendering performance of the two devices. If that is case, we can conclude that the devices behave similarly for all test cases except for a linear scaling factor and offset. We compute a correlation coefficient for all pairs of different values of a single factor, which yields a correlation matrix (e.g., Figure 3.4) and repeat this procedure for every factor we tested. That means, we calculate one such correlation matrix per factor. Figure 3.3 gives an aggregated overview of all those matrices: each single box in this boxplot represents the distribution of all correlation factors within one matrix. This yields one box per tested factor. High values in the matrix indicate a generally high linear correlation between all pairs of instances of the respective factor. Additionally, we calculated correlation matrices for camera rotation and distance that we added to the boxplot for a comparison to the camera path factor.

The means of the correlation coefficients indicate that the volume data set has the highest variance and lowest correlation (also the factor with the highest number of different

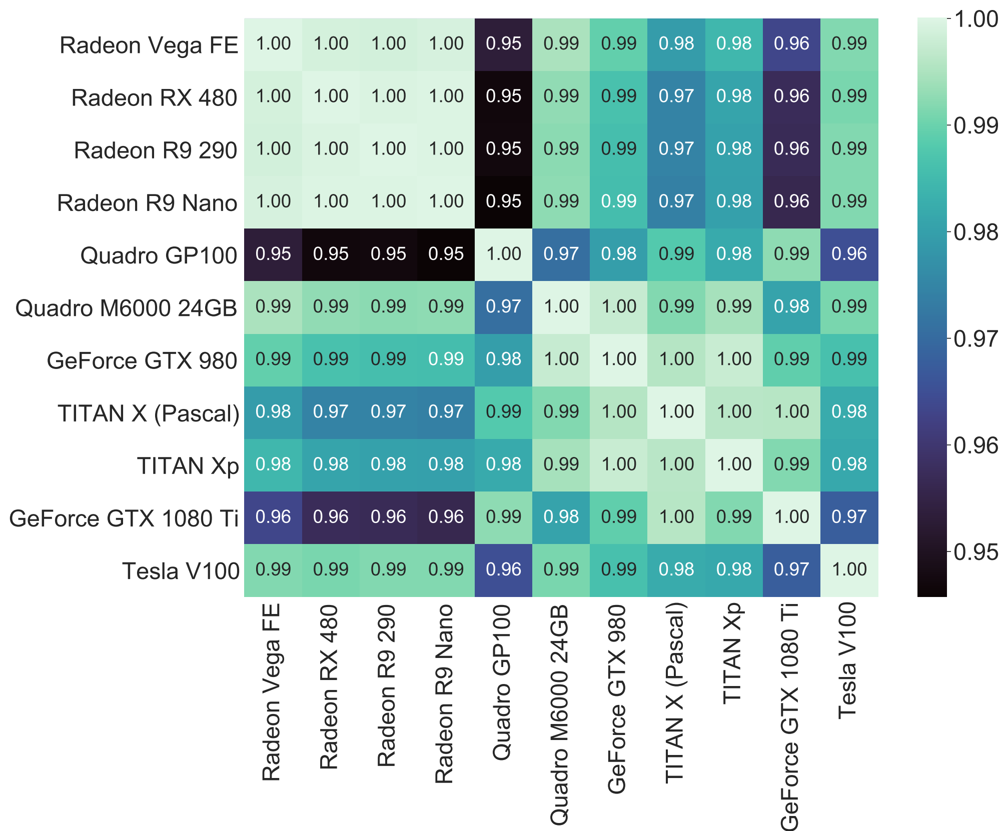


Figure 3.4: Matrix showing Pearson correlation coefficients among different measured GPUs for volume raycasting.

values in our measurement), followed by the camera path. While the step size has a very high correlation among different values, there is also some noticeable variation for the viewport. We attribute the variations when changing the viewport to performance deviations caused by oversampling of the smaller data sets (i.e., neighboring rays get very cheap due to caching effects). The binary factors for the acceleration techniques and transfer functions have a medium to high correlation. They are closely linked to the data set structure. Finally, all devices show a very predictable behavior in terms of performance.

We visualize the correlation matrices as a heat map for closer investigation. Figure 3.4 shows the correlation heat map for the *device* factor, i.e. the different GPUs. Notably, the linear correlations between the tested AMD GPUs are almost 1.0 in all cases, while correlations with the Nvidia cards are still above 0.95 in all cases.

The performance influencing factors we benchmarked, can further be classified into four categories:

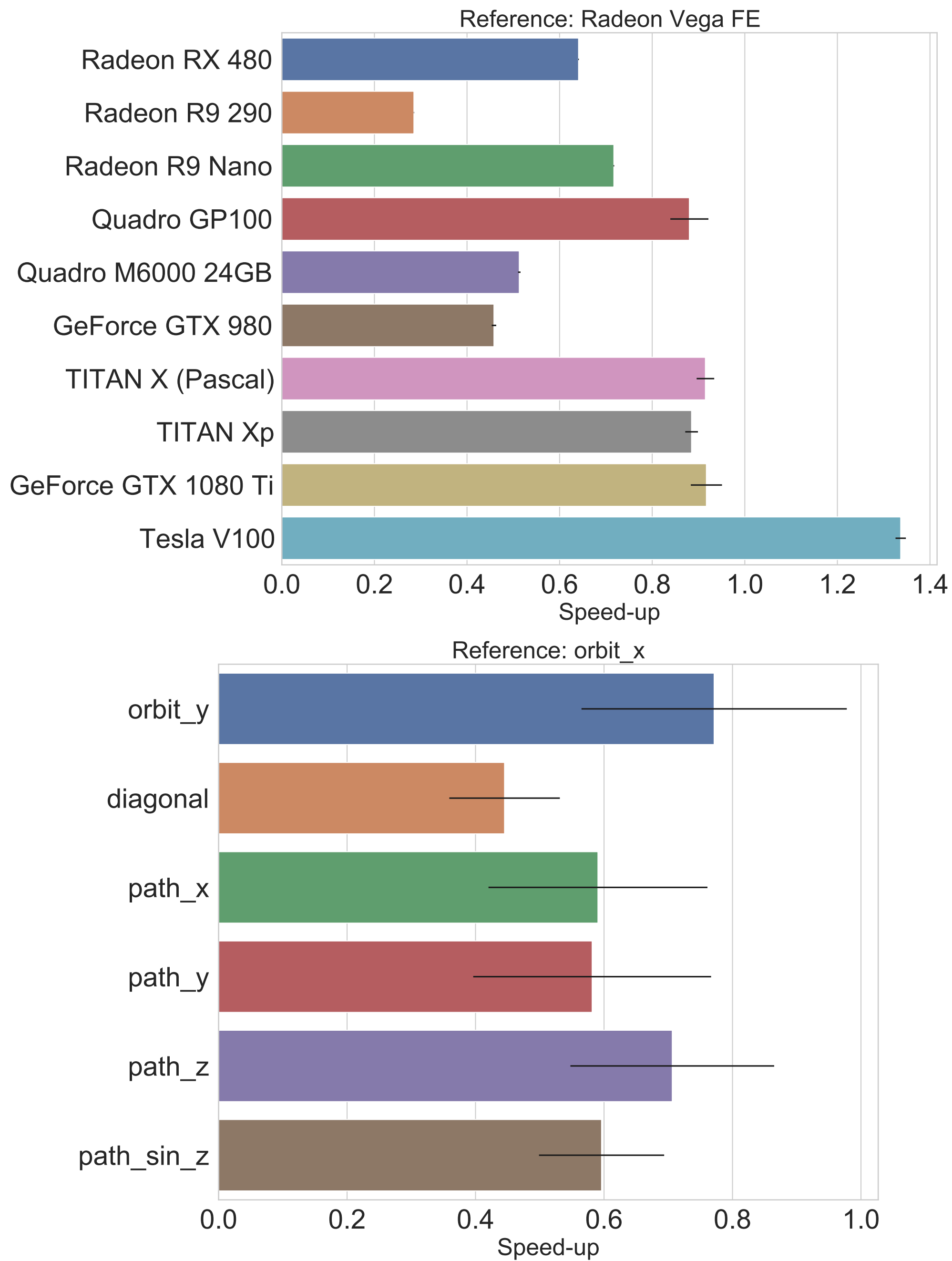


Figure 3.5: Speed-up of volume raycasting on different GPUs relative to the AMD Radeon Vega FE (top); and speed-up of different camera paths relative to an orbit around the x-axis (bottom). Error bars indicate the variance.

1. **Hardware**, in our case GPUs.
2. Numeric **sampling parameters** in image space (viewport resolution) and object space (step size along the ray).
3. **Camera parameters**, in our case covered by different camera paths.
4. **Data structure**, a combination of data set and transfer function.

One can argue that different camera parameters are also a form of distinctive data representations, respective data structures. That means, by changing the camera perspective, we "generate" a new structure, through processing only parts of the original data and changing the behavior or impact of ERT and ESS. Therefore, a reduction of the four categories to three (adding camera parameters to data structure) is also possible.

In terms of those categories, the determined correlation coefficients imply that using different hardware results in comparably predictable behavior. The same holds for the sampling factors, with some limitations for the viewport. The camera parameters show significantly bigger deviations, while factors related to the volume data structure have the lowest correlation among one another. We conclude that the data set structure is the most important factor for performance quantification in our volume raycasting application.

To validate this conclusion, we separate the data based on all factors related to the volume data structure as discussed above: data set, transfer function, acceleration techniques and camera. We then apply a linear regression on a training subset of the data. When testing the resulting model, we achieve an average coefficient of determination of $R^2 = 0.894$ across all data sets, with a minimum of $R^2_{min} = 0.756$ and a maximum of $R^2_{max} = 0.940$. In order to generate a reference, we used random forest regression. Using this more advanced machine learning technique, we were able to achieve a coefficient of determination of $R^2 = 0.982$. We deliberately chose linear regression to show and understand linear relations in the data, which may not be as obvious with advanced machine learning techniques such as random forests or neural networks. Figure 3.5 shows the linear relation of all tested GPUs and camera paths. The error bars indicate uncertainty and are calculated using $error = (1 - r) \cdot s$, with s denoting the slope of the regression and r the correlation coefficient.

Further investigation of interesting findings. Based on the findings described above, we were interested in two additional aspects: a detailed investigation of the performance of different volume data sets, which showed the highest variation among all features, and the influence of the camera parametrization on performance. To accomplish the former, we measured the performance of a stack of down-sampled data sets (i.e., aggregating $2^3, 3^3, 4^3, \ldots$ neighboring voxels). As an example, we used the Chameleon data set with an original resolution of 1024^3 voxels and created seven down-sampled variants, the smallest one having a resolution of 128^3 voxels. Investigating the correlation matrix of those data sets showed a minimum coefficient of 0.85 (between

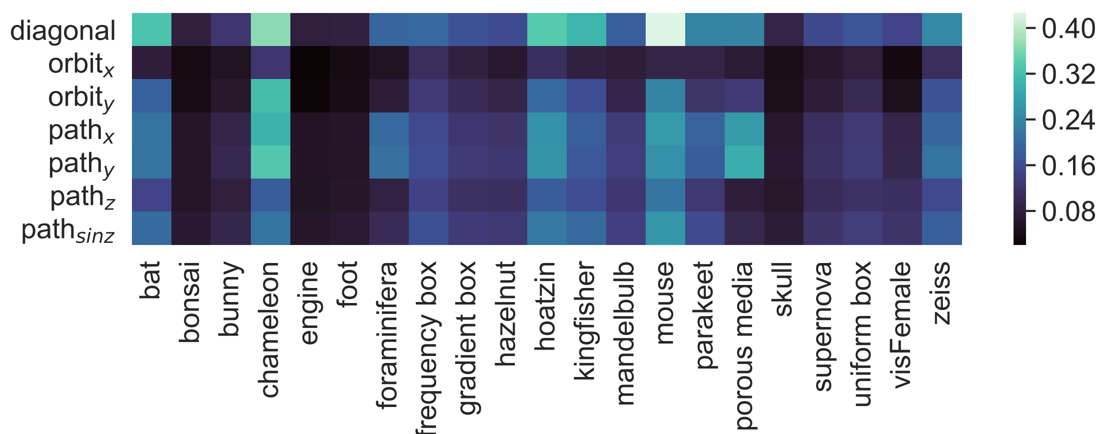


Figure 3.6: Mean frame time (in seconds) for rendering the volume data sets using different camera paths.

the highest and the lowest resolutions), with a mean value of 0.97 for all combinations. Another example using the Zeiss data set with an original resolution of 640^3 and six down-samplings showed similar results, with a mean correlation coefficient of 0.99 a minimum of 0.96. The results suggest that the structure of the data set is mainly responsible for the variance in performance behavior. However, there might be cases where a high resolution data set has distinct performance characteristics from a down-sampled version of the data. For instance, this could be the case if fine structures or noise in the original data get averaged out in the down-sampling process and therefore result in faster rendering times.

Figure 3.6 shows a heat map for the execution time means of all data sets when using a specific camera path. The performance deviation are comparably high for several combinations, which motivated us to further investigate the performance behavior for different camera parameters. We calculated Pearson correlation coefficient matrices for solely the rotation respectively the distance to the center of the bounding box of the volume data set. For the former, we used the samples from the camera path doing a full orbit around the y-axis. For the latter, we took the samples of the straight camera paths along the three major axes. Different positions among the orbit showed a mean correlation of 0.88, with the minimum being 0.68 (Figure 3.3). Besides the structure, shape, and spacing of a data set, memory access patterns and caching influences performance during rotation around a data set in texture-based volume rendering [170, 180].

For the distance to the volume (i.e., zoom into the data set), the mean of the correlation coefficients was lower compared to the one for rotation, at 0.83. Notably, the lowest factor was 0.54: It is the correlation factor between the camera configuration with the

shortest and the one with with the largest distance to the center. Overall, correlation to other camera configurations decreases with smaller distances to the center. This suggests a higher divergence in performance behavior between zooming into a volume than circling around it.

Finally, we performed several two-sample, two-sided Kolmogorov-Smirnov tests to analyze if the intuitive choice of rotating the camera around the data set is representative for a wider range of views in the sense that the sample comes from the same distribution as all views. For both paths that orbit around the data set, the test rejected the null hypothesis ($D = 0.09, p < 2.2 \cdot 10^{-16}$ for $orbit_x$ and $D = 0.04, p < 2.2 \cdot 10^{-16}$ for $orbit_y$) whereas it rejected it for none of 100 randomly chosen samples of the same size (approximately 1.1 m), with p-values ranging between 0.051 and 0.997 (mean: 0.644, median: 0.686).

3.1.3 Case Study 2: Particle Visualization

In a second case study, we examine particle rendering in the form of several different techniques of raycasting sprites. Those techniques are often used for visualizing particle-based data sets such as molecular dynamics simulations.

Common Practice

Table 3.3 shows a summary of how runtime performance evaluation has been done in publications on particle-based visualization over the last two decades. It can be seen that the authors mostly focused on varying the data sets, while in most cases only one GPU and one viewport resolution was tested. Information about the camera was rarely mentioned, especially in older publications. More recently, the camera was oftentimes adjusted such that the data set fits the available screen area [107, 73, 96, 70]. Other authors complemented this overview rendering with a close-up [155, 61, 120], sometimes including the actual renderings for reference [100, 178, 71]. In three cases [74, 125, 120], the authors used knowledge about their algorithm and placed the camera in the assumed worst-case position. Performance measurements for a larger number of camera positions are rarely reported. If so, rotations around the data sets are used [125], also complemented by fly-through paths [147]. As an exception, Hermosilla et al. [83] report average frame rates from random camera positions on a sphere around the data set.

The predominant measure used in basically all cases are fps, similar as for volume rendering. An exception is the work of Gumhold [76], who reported the number of ellipsoids rendered per second along with the number of fragments filled per second for a series of different data set sizes. Frame rates were sometimes complemented by detailed performance information regarding individual steps of the algorithm [107, 74, 126, 100,

Table 3.3: Performance Evaluations in Recent Particle Rendering Papers

Authors	Year	GPUs[b]	Viewports	Data Sets	Views
Ibrahim et al. [90]	2021	N	1	9	3
Gralka et al. [70]	2020	2×N, 2×A	1	5	1
Müller et al. [147]	2018	H	1	5	5×?[c]
Ibrahim et al. [91]	2018	N	1	6	2
Hermosilla et al. [83]	2017	N	1	9	512
Jurčík et al. [96]	2016	N	1	4	1
Skånberg et al. [165]	2016	N	1	4	1
Grottel et al. [72]	2015	N	–[a]	3	–[a]
Wald et al. [178]	2015	4×I	1	7	2
Guo et al. [77]	2015	4×N	1	2	7
Knoll et al. [100]	2014	2×I, N	1	8	2
Le Muzic et al. [120]	2014	N	1	1	1
Grottel et al. [73]	2012	N	1	6	1
Lindow et al. [125]	2012	N	1	7	rot.
Chavent et al. [40]	2011	4×N	1	12	1
Lindow et al. [126]	2010	2×N	1	5	–[a]
Grottel et al. [74]	2010	N	1	5	4
Krone et al. [107]	2009	N	1	10	1
Falk et al. [61]	2009	N	1	6	2
Lampe et al. [114]	2007	N	1	6	–[a]
Gribble et al.[71]	2007	O	2	6	1
Tarini et al. [172]	2006	A	–[a]	≥ 2	–[a]
Reina & Ertl [155]	2005	3×N	–[a]	4	≥ 2
Klein & Ertl [99]	2004	N	1	1	–[a]
Gumhold [76]	2003	N, A	–[a]	1	–[a]

[a] Property not mentioned in the paper.
[b] A = ATI/AMD GPU, H = HoloLens, I = Intel many-core CPU or Xeon Phi, N = NVIDIA GPU, O = Opteron CPU.
[c] Multiple camera paths with variable number of frames.

Table 3.4: Techniques and Required Shader Stages Used for Particle Rendering

Technique	Active shader stages*	Unique factors
Screen-aligned quad	VS, **GS**, PS	
Ray-aligned quad	VS, **GS**, PS	
Instanced quad	**VS**, PS	color conversion
Ray-aligned quad	VS, **HS, DS**, PS	
Ray-aligned polygon	VS, **HS, DS**, PS	corners (4–8, 16, 32)
Adaptive polygon	VS, **HS, DS**, PS	allowed corners (4–16)

* Bold face marks the shader stage used to compute the sprites.
V = Vertex, G = Geometry, H = Hull, D = Domain, P = Pixel shaders.

77, 40]. These were typically reported in milliseconds. Relative speed-ups [126, 71] are less common and given if a new technique was compared to an existing one [114, 40, 125, 100, 72, 96]. Besides timings, bandwidth [114, 74] and memory requirements [125, 100, 178] were reported. Müller et al. [147] included data from pipeline statistics queries such as the number of shader invocations or the geometry load.

While some authors provided quite detailed explanations about certain performance characteristics and their causes [100, 178, 90] or compared several variants of their technique [61, 107, 73, 165, 70]. Only a couple of works of the investigated ones present systematic performance studies: Grottel et al. [74] investigated different techniques to transfer data from main memory to the GPU, data quantization, two culling techniques and deferred shading, and the combinations thereof. The reported results did not only include frame rates but also statistics such as the visible data after culling. Müller et al. [147] compared the performance of different shader-based methods of rendering spherical glyphs on the Microsoft HoloLens.

For our benchmark, we derive similar factors from the literature review as in the volume rendering case: GPUs, viewports, data sets, and camera paths. This also allows us to compare across experiments. We focus on evaluating the different techniques and variants used for particle rendering to provide a better understanding of the differences. Additionally, we test several rendering specific parameters.

Our Test Implementation

Today, the predominant technique for rendering particle data sets as spherical glyphs is computing the ray-sphere intersections on sprites. Since we need rasterization for this, we base our test implementation on the Direct3D 11 API. In our experiment, we evaluated several variants of this GPU-based raycasting, which are implemented via modified pixel shaders. Our tests include a series of techniques for generating the

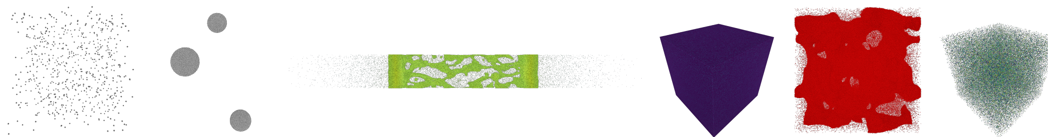


Figure 3.7: The particle data sets from molecular dynamics simulations used in our tests. From left to right: small test run (1000 particles), three droplets (79 509 particles), formation of a liquid layer (2 M particles), laser ablation (6 185 166 particles), and a set of 10 M nanoparticles. The last rendering shows one of the randomly generated data sets with 100 000 particles. © 2020 IEEE [7].

sprites. Table 3.4 shows an overview of all rendering techniques along with specific factors that might influence the rendering performance of the technique. They differ in the usage of quads or polygons, the alignment of quads, the use of instancing, the active shader stages, and the particular shader stage used for computing the sprites.

The *Instanced quad* approach differs from all others in that it does not obtain its data (i.e., position, radius, and color of the particles) from a vertex buffer. Instead, data is obtained from a structured resource view, which is accessed based on the instance ID. The vertices for the instances are not stored in a vertex buffer, but computed on-the-fly from the vertex ID. The ray-sphere intersection itself is always computed in the pixel shader. For all tests using particles without a color, but with an intensity value, we also tested the transfer function lookup in the vertex and pixel shaders regardless of the technique. Further, we tested conservative depth output enabled and disabled for all techniques. Conservative depth output allows the GPU to perform early z-culling even as we are writing the correct depth of the ray-sphere intersection point computed in the pixel shader. This way, many fragments in dense data sets can be discarded before invoking the pixel shader. However, the effect varies with the position of the camera because the order in which particles are emitted depends on the fixed layout of the vertex buffer, not on the view. Therefore, the amount of overdraw and the order in which it happens—and in turn the number of fragments discarded early—are view-dependent.

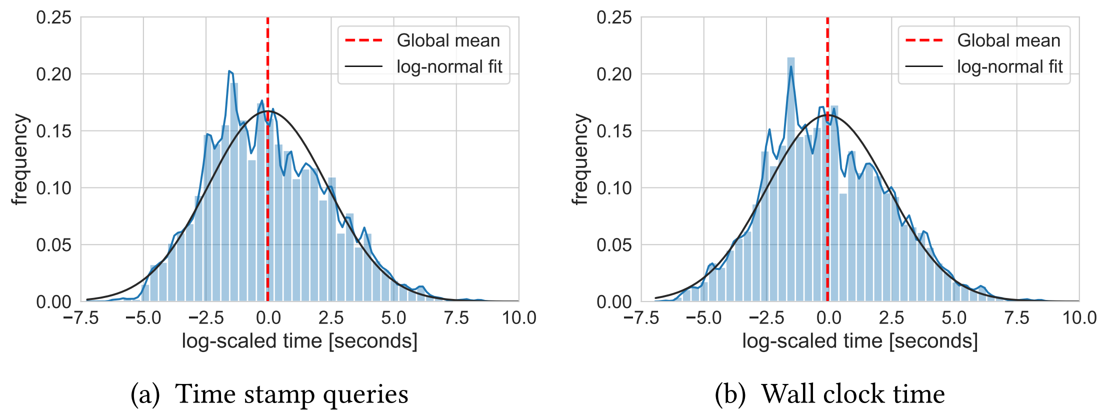
We used quadratic viewports of 512^2, 1024^2 and 2048^2 pixels (the same as in the volume rendering case). The camera was moved in ten steps along different paths: $diagonal_{x/y/z}$, $orbit_{x/y}$, $path_{x/y/z}$ and $path_{\sin x/y/z}$. We measured the particle rendering on the GPUs listed in Table 2.1. Our tests included five real-world data sets (cf. Fig. 3.7) and 21 artificial ones with uniformly distributed particles in a 10^3 bounding box. The artificial data sets were generated in two series: the first one contains $10^k, k \in \{3, 4, 5, 6\}$ particles, each having approximately the same fraction of the bounding box filled (i.e., the size of the particles decreases as the number of particles increases). The second series always comprises 100 000 particles but their size increases over three steps, i.e.,

more of the bounding box is filled increasing the overdraw that occurs during raycasting. All artificial data sets were tested with 8 bit RGBA coloring, 32 bit RGBA coloring, and intensity only.

For each test configuration, we obtained mainly two data points for comparison: GPU timings measured with a timestamp query injected into the command stream and the wall clock time measured with the high-resolution timer on the CPU. Additionally, we obtained the number of shader invocations for each configuration by issuing a query for pipeline statistics. Before the actual test run, we performed a couple of pre-warming renderings, which are not included in the result. These serve mainly two purposes: the first frame after switching shaders is usually particularly slow and therefore should be excluded from the result for not being representative. At the same time, the system computes from the pre-warming renderings how many frames to render for a minimum runtime of 100 ms. The actual measurements are then performed in three separate steps. First, we render eight frames and obtained the timestamp query results for them. Unless denoted otherwise, the GPU times reported subsequently are the medians of these eight measurements. The difference between the minimum and maximum time for one measurement ranges between 0 ms and 1385 ms (mean 0.16 ms). Second, we obtain the pipeline statistics by rendering the same configuration again. And finally, we render the number of frames computed beforehand to measure the wall clock time. As we completely rely on rendering to off-screen targets, which lacks the buffer swap as synchronization point, we stalled the CPU by waiting for an event query injected into the command stream after the last frame. Wall clock times are the time between the first frame and the point when the event query returned, divided by the number of frames rendered during this period.

Discussion of our Measurements

Performance distribution. We start our analysis by looking at the histogram of GPU (Figure 3.8a) and CPU (Figure 3.8b) timings from all measurements. The applied logarithmic scale shows that the distribution vaguely resembles a log-normal one, but the fit is not as good for the volume rendering case. Given the histogram and the sample size of more than 3.3 million measurements, it is not surprising that a one-sample Kolmogorov-Smirnov test rejected a log-normal distribution ($D = 0.25, p < 2.2 \cdot 10^{-16}$). Figure 3.8 also shows that there are small differences in the distribution of GPU time stamp queries and wall clock measurements, but both histograms have approximately the same spikes and a similar mean. These spikes could either result from particle rendering generally not being log-normal distributed or from our test cases not sampling the parameter space sufficiently well. We reckon that the selection of the data sets is an important factor here, because we mainly sample based on orders of magnitude and not evenly distributed numbers of particles. Data set sizes of 100 000 particles are over-represented in the results and account for more than half of the observations in



(a) Time stamp queries

(b) Wall clock time

Figure 3.8: Distribution of the logarithmically scaled timestamp queries (a) and wall clock times (b), of the particle renderings.

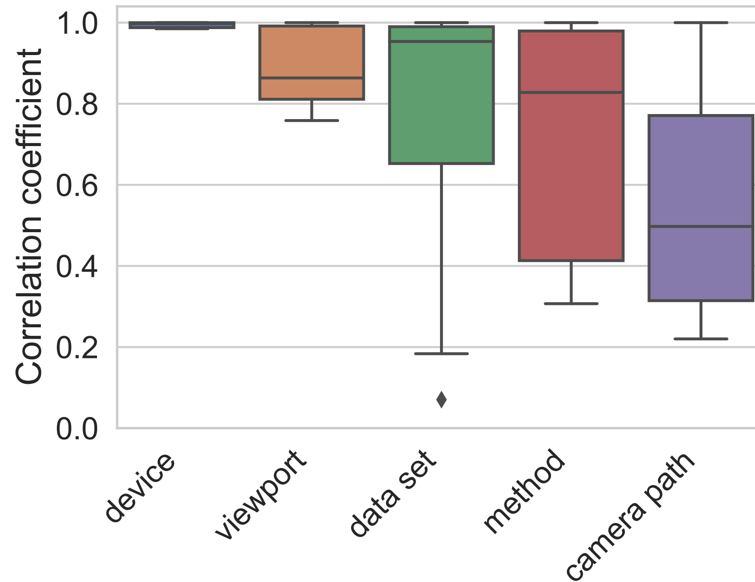


Figure 3.9: Pearson correlation coefficient for five selected factors influencing the rendering speed of the particle data sets.

Figure 3.8, because the series of tests for different sphere sizes uses this number of particles. The other artificial data sets comprising 1000 and 10 000 particles further add to a large imbalance of data sets below 1 million particles, which has a notable influence on the shape of the histogram.

Linear (in)dependence of features. We investigate the mean correlation between the different levels of factors, such as the device, the viewport, etc. (see Figure 3.9), akin to the analysis of the volume rendering data (subsection 3.1.2). Again, we observe a strong linear relationship between all devices, i.e. the behavior of the GPUs is generally the same for all tests. The correlation between the data sets is generally higher than for the volume rendering case, but there is a lot of variation and there are some outliers: Both effects can be explained by the large number of artificial data sets in the test. The performance behavior of these seems to be largely the same with respect to the other factors with correlation factors close to one. One notable exception are the data sets containing spheres with very small radii that result in particles being represented by a single pixel in the majority of views. Their correlation factor with the other artificial data sets is only around 0.72 (see Figure 3.10). The data sets from simulations naturally exhibit more variation, most notably the 79 509 particles that form three drops and therefore differ from all other data in that a large fraction of the bounding box is actually empty. This data set has a correlation factor of around 0.43 with most other data sets, except for the laser ablation (0.18) and the liquid layer formation (0.05). The latter has a Pearson correlation of around 0.39 with most other data sets and is special in the sense that it is the only data set with a strongly non-cubic bounding box, i.e. the path along the z-axis is much longer than along the other two. The two factors showing the least correlation on average, and in turn the greatest variance, are the rendering method and the camera path. While the methods based on sprites aligned with the view ray mostly have correlation factors above 0.9 among each other, the screen-aligned one lies around 0.7 depending on the specific method. The camera paths exhibit similar results (Figure 3.11). Paths along one axis (straight and sine) reach 0.96 or higher, while the two orbits yield around 0.9 with the other methods (comparing the orbits yields a 1.0). We found that the underlying cause for both of these observations is that the screen-aligned sprites can cause significantly more overdraw if the camera is close to a sphere. This can also be measured via the number of pixel shader invocations that is an order of magnitude above the one for the ray-aligned sprites. While the ray-aligned quads can be clipped against the front plane (causing visual artifacts), the screen-aligned ones become larger as a sphere comes closer and are only clipped at once if they reach the front plane. Furthermore, depending on the radius of the sphere, the camera position and the clipping planes, the method might also generate sprites for back faces of spheres, which generate no fragment in the end. This effect cannot occur if the camera is outside of the bounding box, which is the case for the orbit paths.

Further investigation of interesting findings. We assumed that the radius of the spheres plays an important role for this effect and further investigated the rendering times of all methods depending on the data sets. Figure 3.12 reveals that this is the case. The high overdraw of screen-aligned quads becomes only a relevant factor if the number

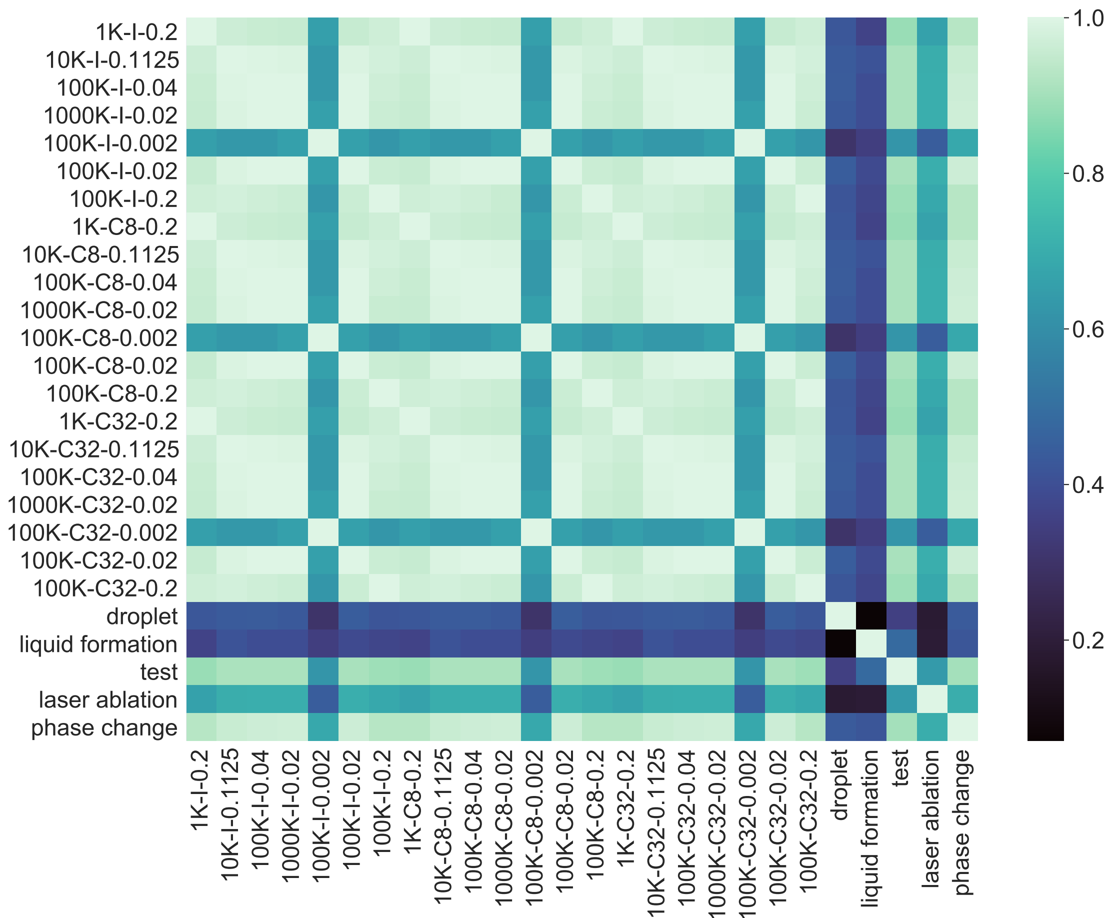


Figure 3.10: Matrix of the Pearson correlation factors between timings for different particle data sets.

of particles is high or, in case of the artificial data sets, if the radius is—compared to real-world applications—unnaturally high. The matrix also shows that the generation of view-aligned quads performs almost the same for all data sets, only if the number of particles becomes very large, tessellation-based methods become slightly slower.

Given this observation, it is clear that an orbit path cannot be representative for the entirety of the views. We performed a two-sample Kolmogorov-Smirnov test, which rejected that both orbit paths have the same distribution as population of all views ($D = 0.03, p < 2.2 \cdot 10^{-16}$ for $orbit_x$ and $orbit_y$). As with the volume rendering data, we tested 100 random samples of the same size, for most of which the test did not reject the null hypothesis (mean p-value 0.605, median p-value 0.658).

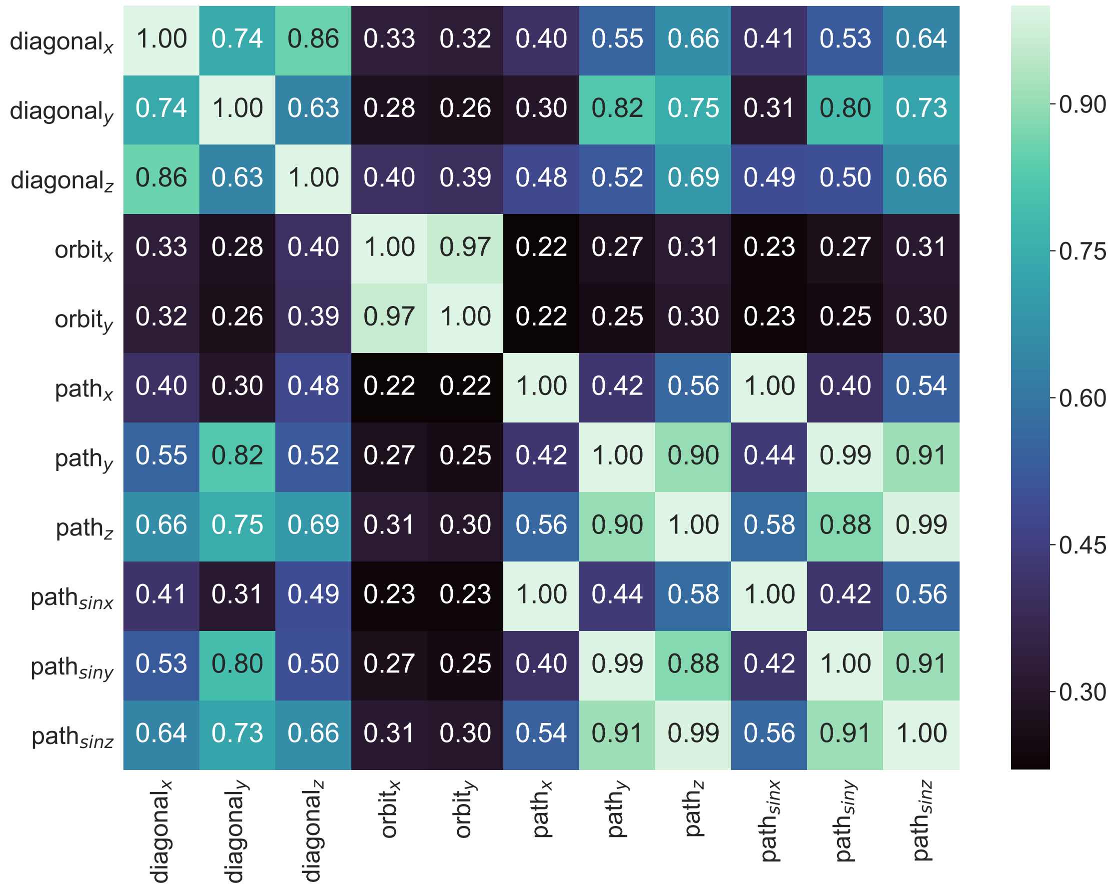


Figure 3.11: Matrix of the Pearson correlation factor between the timings for different camera paths through the particle data sets.

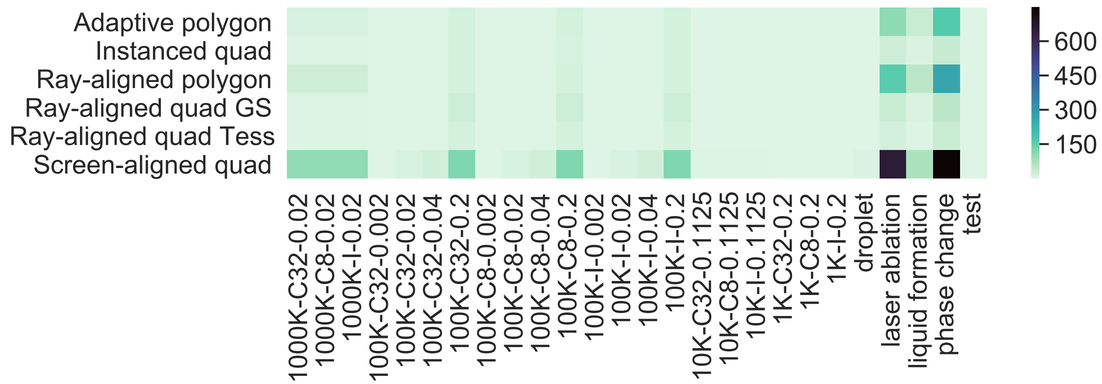


Figure 3.12: Mean frame time (in milliseconds) for the particle data sets rendered using different sprite-based sphere rendering techniques.

3.1.4 Results and Recommendations

One rather obvious conclusion we draw from our experiments is that specifying minimum and maximum values as well as percentiles along with average fps or frame times is desirable, although seldom done in practice. For instance, when performing a fully accelerated volume raycasting of the "Mandelbulb" on an NVIDIA Titan Xp, the mean of 0.0456 s implies a decently interactive frame rate of 21 fps, but the 75$^{\text{th}}$ percentile is actually 0.0746 s and the 90$^{\text{th}}$ percentile only 0.1227 s. However, none of the reviewed papers give percentiles, many of them only report average fps.

Given the amount of measurements we obtained in two different areas of visualization, we believe it is reasonably safe to conclude that the typical approach of testing only one device (done by $\approx 83\%$ of the reviewed papers) seems to be valid for most cases. There might be exceptions in case of different device types (CPU vs. GPU) or disruptive technology changes enabling different kinds of algorithms. However, we found a linear behavior even across different memory technologies, architectures and vendors for the same algorithm.

Authors from both application areas oftentimes use different data sets for their evaluation: $\approx 89\%$ in the reviewed volume rendering papers and $\approx 87\%$ in the particle rendering ones. In our studies, we could confirm that this is important due to their generally significant impact on performance. However, in the case study on particle rendering, we found that measurements with artificially generated data sets must be handled with care. Real-world data usually have complex internal shapes influenced by many parameters, making it difficult to decide which parameters should be used in which way to generate test data. Further, designing data sets with a specific property in mind that is to be tested, can cause an undesired bias in the distribution of the results. In hindsight, we would recommend separating the corpus of data sets for describing general performance characteristics and the ones for closer investigation of a priori known effects. However, more realistic performance characteristics when using artificial data could possibly be achieved with generative data models [160]. By using them, a potential scarcity of data sets could be circumvented, there is even the possibility of finding a small set of generated data sets that represents most of the common performance characteristics.

The camera parameters proved to have an equally high impact in both of our tests. However, evaluations in papers oftentimes only use a low number of view points. About 58% of the papers that actually report the number of views (which are only half the papers) use less than eight different camera configurations. Therefore, it is particularly desirable that a systematic performance evaluation explicitly states the respective configuration, and that a variety of different camera configurations are considered. Almost all reviewed papers that report on evaluating more than eight views perform some form of intuitive camera paths, e.g., rotations around the bounding box or recorded

user interactions. We found that using those intuitive choices might be insufficient. Based on our results we assume that testing a number of randomly chosen camera poses—filtered in a way that a reasonable portion of the data is visible—is a sensible approach. Although random camera parameters are typically not a realistic usage scenario, using them for measurement has the advantage that the overall distribution and performance characteristics can be covered with less samples, thus freeing resources for measuring other parameters.

Best Practices

From our results, we derived the following set of best practices for performance evaluation of interactive scientific visualization techniques in single node environments:

- Report performance distributions instead of a single fps value (or report at least frame rates for different percentiles).
- Report all important factors that have an influence on the performance. Our literature review shows that factors like the viewport size seem to be so obvious to the authors that they are often missing in the exposition.
- Generally, one system for testing seems sufficient to report performance properties, if the software is not tailored towards special hardware characteristics like specific memory hierarchies.
- Multiple (meaningful) random camera configurations should be used for measurements. While these might not be representative for the actual interactions of a user in a specific scenario, they allow for a comprehensive general assessment. Typically, they yield much more expressive results in comparison to using just a single view point or a single path.
- A large variety of real-world data sets and/or different shape-defining properties (such as transfer functions in volume rendering) should be measured. This is particularly important when acceleration techniques are used, as their performance results typically strongly depends on data characteristics.

This list is a rough guideline addressing several properties that we discovered during our data analysis. Although we believe that they are a good starting point, the importance and portability may vary depending on the specific visualization application and domain. We consciously tried to keep these recommendations as general as possible, while also providing a guideline on how to improve performance evaluation.

Limitations

Besides the still limited number of camera poses we have tested, there are several limitations to our current approach. We restricted all of our measurements to interactive

algorithms running on a single GPU and no out-of-core handling. Further, we collected measurements for only one dimension—the rendering times—while there are many other dimensions that might be interesting or necessary for a detailed evaluation. For instance, the Direct3D pipeline statistics with the pixel shader invocations proved to be very helpful for interpreting certain effects in the results from the particle test runs.

Given the amount of data, we followed a top-down exploratory approach, but investigating correlations between factors is still at such a high level that potentially interesting outliers are hard to find. Also, we only analyzed whether there is a linear or no correlation, which is sufficient to describe the influence of a factor in principle and to derive some guidelines on how to handle this factor. However, we cannot find out whether only certain expressions of a factor influence others this way. For instance, if an optimization only has an impact on a subset of the data sets or in other specific parameter configurations.

3.1.5 Future Directions

This systematic analysis is a first step towards a better understanding of how the performance of scientific visualization applications behaves at large. Despite being broadly applied techniques, volume and particle rendering is still only a subset of scientific visualization. Further measurements and analysis of other techniques could possibly complement and support the results to provide a more general understanding of performance evaluation. This approach is focused on performance in the form of rendering speed. However, there are multiple other interesting performance metrics such as memory usage, energy consumption, rendering quality, etc. that are worth of a similar investigation. Ultimately, a comprehensive understanding of different performance metrics could lay the foundation to a better understanding of performance metrics trade-offs.

There is an inherent link between performance evaluation and performance modeling. The specific values we obtained with our measurements could potentially be used directly in performance models. For instance, using the information gained from the linear regressions in the volume rendering case study, a simple model for the execution time t can be formulated, consisting of a linear part and a more complex part in form of a distribution:

$$t = w_{is} \cdot w_{os} \cdot w_h \cdot D$$

Here, w denote the weights (i.e. speed-ups) for different parameters, relative to a sample configuration ($w_{is/os}$: sampling in image/object space, w_h: hardware factor). D is a distribution representing the remaining factors, i.e. structural information of the data set and acceleration techniques. Concrete instances of rendering performance models can be found in chapter 4 and chapter 5.

3.2 Visually Comparing Performance Specifics

In this section, a fine-grained approach to performance evaluation is introduced that is centered around the visual representation of camera position and orientation, and puts those into context with performance measurements using multiple different parameters [13]. The approach aims to give the user a better understanding of detailed differences between performance influencing factors and rendering techniques. This can help, for instance, in selecting the best technique for a specific scene, avoiding problematic camera perspectives, informing a fair comparative evaluation, detecting quality mismatches between techniques, and spotting configurations with improvable performance.

With this approach we take into account multiple perspectives: different camera paths, different rendering techniques, and different hardware. Following the movement of the camera through a scene and using this as a basis for visualization provides a natural contextualization of the data and eases interpretation. Connecting the performance data to the specifics of the camera view is key for understanding why certain views show similar or dissimilar performance characteristics. The approach aims to specifically support the following four analysis tasks regarding the detailed investigation of performance benchmark data:

- **T1**: Comparing the performance of rendering techniques for a specific camera configuration in context of the rendered images.
- **T2**: Showing the performance of rendering techniques as part of a camera path and support the comparison of different paths.
- **T3**: Identifying clusters and respective outliers of data points with similar performance characteristics across the techniques.
- **T4**: Studying interactions of performance with other rendering parameters and hardware setups.

In part, the runtime measurement data from our empirical approach (section 3.1) was used for development and demonstration. While our empirical approach focuses on analysis of distributions and correlation to determine general trends and interrelations, it is rather unsuitable for a close inspection of outliers and specific, fine-grained performance characteristics that are addressed with this technique. Although the approach was developed with runtime performance in mind, it is transferable to other performance metrics as well.

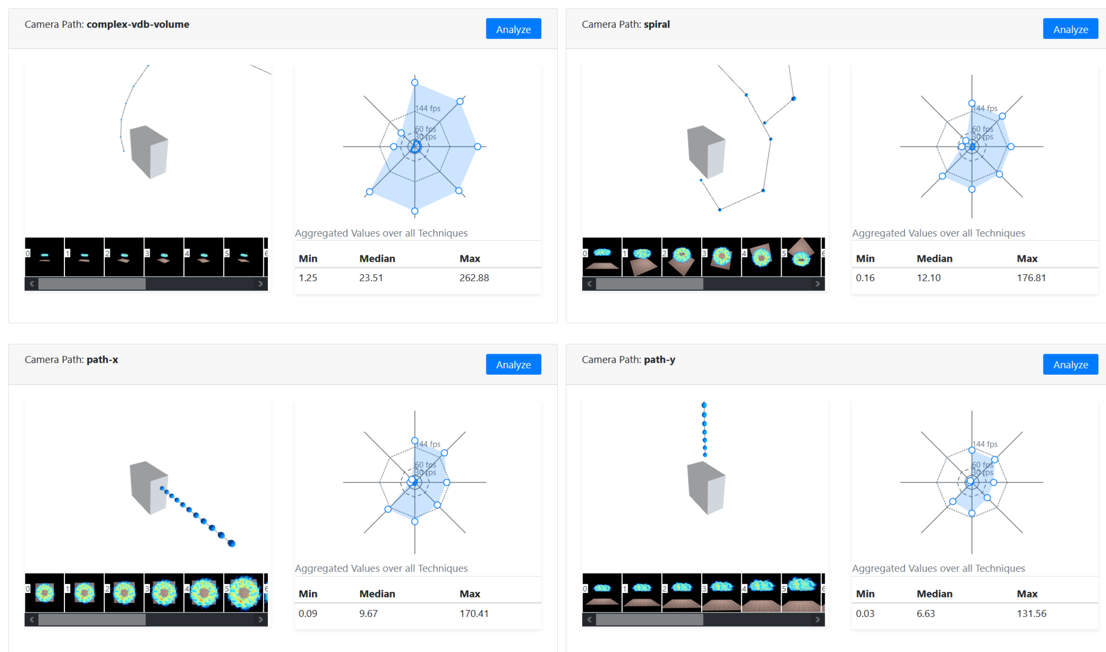


Figure 3.13: The Data Set Explorer provides an overview of camera paths and aggregated performance metrics in radar charts. Shown is a cutout of four camera paths for a volume scene rendered with OSPRay [177], a library for ray tracing based rendering on CPUs.

3.2.1 Multiple Perspectives Analysis System

We propose the Multiple Perspectives system, a visual analysis approach for analyzing rendering performance. The web-based application is divided into two linked screens: the *Data Set Explorer* (Figure 3.13) and the *Camera Path Explorer* (Figure 3.14). The Data Set Explorer uses a small multiples visualization to give an overview of all available camera paths for a selected data set and their performance characteristics. Selecting one of the small multiples opens the Camera Path Explorer that features an interactive custom visualization to explore rendering performance along a single camera path.

Data Set Explorer

The Data Set Explorer shows a grid of small multiples for a selected data set. The small multiples represent all available camera paths as a combination of a 3D thumbnail and a radar chart (see Figure 3.13). This view serves as a summarized preview of the performance data that can be analyzed in more detail for a specific camera path. It addresses tasks **T1** and **T2** from a higher level of abstraction. The 3D thumbnail is an interactive component that visualizes the sampled camera path and therefore provides

the spatial context of the performance measurement with respect to the bounding box of the main scene. Below the 3D representation, a stripe of thumbnails is shown that displays the rendered images along the camera path.

A radar charts next to the 3D thumbnail gives an overview of the recorded performance metric for each camera path. The data displayed in the charts is filtered by the selected data set and camera path, but aggregated across all camera configurations along the path, resolutions, and hardware setups. Each axis of the radar chart represents one technique and shows its measured runtime performance. In other words, the chart summarizes all data points (which are vectors of performance measurements) across all measurements that relate to the camera path. The minimum and maximum values are shown on the respective axis and connected to form a polygon (light blue shape). On top, a polyline visualizes the median. The decision to use radar charts—consistently in the Data Set Explorer as well as in the Camera Path Explorer—is motivated by their compact representation, their ability to form interpretable and memorable visual patterns, and their relative simplicity.

The small multiples of the Data Set Explorer allow for a quick comparison of the characteristics of the different camera paths. The user can select individual camera paths for an in-depth analysis with the Camera Path Explorer.

Camera Path Explorer

The Camera Path Explorer addresses all analysis tasks **T1**–**T4** on a detailed level of abstraction. As shown in Figure 3.14, the Camera Path Explorer consists of five linked views:

(I) Camera Path View—2D abstraction of the sampled camera path enriched with radar charts that summarize the runtime performance of the different rendering techniques next to the respective rendered images (**T1, T2**).
(II) Clustering Panel—interactive scatterplot that shows a 2D projection of all data points (**T3**).
(III) Comparison Panel—interactive scatterplot for comparison of two techniques (**T3**).
(IV) Faceted Browsing Panel—interface to sub-select data points (**T4**).
(V) Data Table—table that shows all currently selected data points (**T1, T4**).

Camera Path View. This second central view of our approach serves as a base layout for our main visualization. Task **T2** is addressed for a single path. Instead of representing the path in 3D, we project it to a fixed plane to create a recognizable representation of the path with a simplified visual appearance. The plane is determined by minimizing the distances to all sampled camera positions of the respective path. This way, the geometric characteristics of the path can be preserved as well as its relative position to the rendered objects inside the scene.

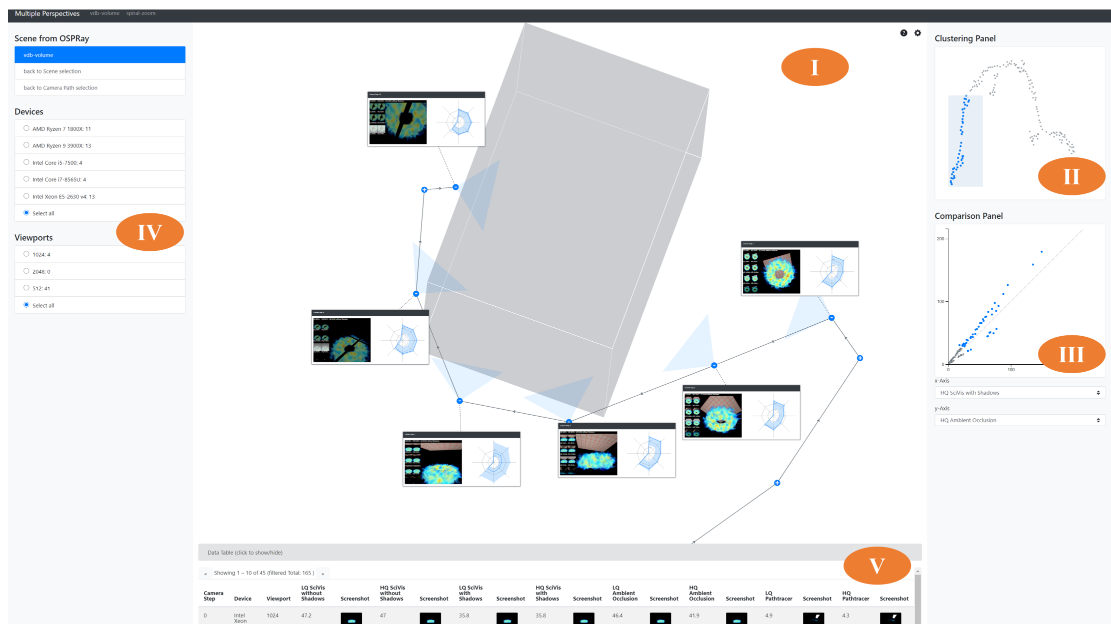


Figure 3.14: The Camera Path Explorer consists of five components for data analysis tasks. A stylized two-dimensional version of the sampled camera path (I), a Clustering Panel (II), a Comparison Panel (III), and a scene or camera path selection as well as radio buttons for faceted browsing (IV). At the bottom is a Data Table (V). Here, the camera zooms into the data along a spiral path around a volume data set rendered with OSPRay. Multiple data points have been selected in the clustering panel (light blue). All visualizations are linked, to display respective sub-selections.

The sampled camera locations, the viewing directions, and the field of view are shown with dots and triangles. By connecting the camera locations, a a polyline is formed that indicates path and direction of the camera. Additionally, a stylized view of the data set's axis-aligned bounding box is rendered by projecting the vertices of the bounding box to the plane that is also used for the projection of the camera positions. All sampling points are connected to an overlay component that shows the performance of all measured techniques in a radar chart. Further, the respective renderings are shown on the left hand side of the overlay component (**T1**). It consists of two parts: a thumbnail-sized image preview of the rendering result, and a magnified version of the selected image. Image selection can be done by hovering a thumbnail, allowing for a comparison of the rendering results. For scenes where the expected visual differences between the rendered images are low, outliers with unexpected high visual deviations are calculated using the structural similarity index measure (SSIM) [182] between each image pair of a data point and highlighted with an indicator.

A force-directed layout is used to calculate the positions of the overlay components.

To further focus the visualization on specific camera configurations, the analyst can hide or show boxes on demand. With zooming and panning, they can enlarge specific regions and boxes to inspect details of the data or images.

Clustering Panel. To address task **T3**, the Clustering Panel contains a scatterplot of data points, plotting samples with similar performance characteristics across all rendering techniques in proximity. Uniform manifold approximation and projection (UMAP) [135] is used to project the multivariate data points, which contain one performance measurement per technique each, to two dimensions. The projection is solely based on the performance measurements, and does not take other factors (e.g., camera configurations of the sampled camera path) into account. This allows for easy identification of data points with similar performance characteristics for all techniques across the remaining parameters.

Comparison Panel. The Comparison Panel is a 2D scatterplot with interactively selectable axes and complements the Clustering Panel with a detailed pairwise comparison, also addressing task **T3**. While points close to the diagonal reflect a balanced performance between the two compared techniques, off-diagonal points indicates better performance for one of the techniques. For example, it is possible to determine if a performance difference of two technique relates to a constant factor for all data points or if it is caused by a larger difference for only a subset of data points. The Clustering and Comparison panels support sub-selection of data points via mouse brushing. Unselected data points are grayed out, all visualizations are linked to display the respective sub-selections accordingly.

Faceted Browsing Panel. A parameter space is sampled for the performance data collection. While the scene (i.e., the data set), and the camera path are selected in the Data Set Explorer, the camera configuration and rendering techniques have dedicated visualizations in the Camera Path View. To also leverage the information about the hardware and the rendered image's output resolution, additional sub-selection options are provided in a sidebar (Figure 3.14 IV). Applying the concept of faceted browsing [193], we interpret hardware and resolution as facets. Selecting one or more facet values will filter the shown data points and update all displayed visualizations. This addresses task **T4** as the selections allow to investigate whether, for instance, different hardware setups influence the performance characteristics along the camera path or relate to certain cluster as shown in the Clustering Panel.

Data Table. The Data Table contains all raw data points of the data set without any aggregation. Each table row contains one data point with images as well as the camera configuration, the used hardware, and the output resolution of the rendered image.

Upon selection of data points in any of the other interactive visualization components, the Data Table hides currently unselected rows. The Data Table contributes to task **T1** and **T4** by contextualizing the performance measurements with the rendered images as well as the rendering parameters and hardware setups, respectively.

3.2.2 Application

We tested our approach and its applicability in two application examples with performance data from the domain of scientific visualization. We investigate performance of rendering techniques integrated in Intel OSPRay [178], a library for ray tracing-based rendering on CPUs. For this, we measured the performance of several CPUs by extending the benchmark capabilities of OSPRay for systematic sampling along camera paths. In a second application example, we analyzed the performance data of our particle visualization techniques (see subsection 3.1.3 on how we generated them).

Using these two application examples, we could demonstrated that our approach helps, for instance, to reveal clusters of camera configurations that have special performance characteristics regarding the compared techniques, performance bottlenecks on specific devices in combination with some techniques, and non-obvious impact of parameter changes.

3.2.3 Future Directions

The current system can be used in post processing only. A promising direction for future work is an extension to capture performance data while the rendering application is running and update the charts and visualizations in situ. Users could then interactively select camera positions with uncharacteristic performance for further analysis.

Further, establishing a direct link between performance metrics, the rendered image, and the source code of the rendering technique might benefit the reasoning over possible deviations in visual output and performance. This would require an integration of the source code into the Camera Path Explorer, for instance. Finally, showing different versions of the code and the performance metrics could potentially enable a closer tracking of the performance impacts caused by the changes in the code.

Chapter 4

Performance Modeling for Runtime Optimizations on GPU Systems

In the previous chapter, approaches for structured performance evaluation of visualization applications were introduced. This assessment is the foundation of modeling and ultimately predicting the performance of applications—the topic of this chapter. The focuses here is on GPU systems, while performance modeling on distributed memory systems is covered in the next chapter.

Today, the parallel processing capabilities of graphics cards are often used to achieve interactive frame rates for scientific visualizations (see subsection 2.3.1). Besides the hardware used for rendering, parameters that can be changed interactively (e.g., transfer function and camera configuration in the case of volume rendering) typically have a substantial impact on performance. In order to accomplish continuous interactivity, those variations in performance need to be accounted for, especially in challenging cases with significant changes between frames, like switching to a different transfer function.

Drops in the frame rate that are caused by a change of parameters may result in unpleasant lags or jerky motions during interaction with the visualization. However, depending on the visualization technique, those drops can be absorbed. For instance, in volume raycasting the sampling density can be adapted in object or image space to speed up the processing time at the cost of rendering quality. For interactive applications, the

basis of such adaptions has to be an assessment of how the performance will evolve in upcoming frames (after potentially big changes) in order to avoid unpleasantly long response times or jerky motions. Modeling the performance on GPUs is a challenging task due to the involved complexity. Several factors have a significant, but not always obvious impact on performance.

In this chapter, a method is introduced to model and predict performance of GPU systems to dynamically adapt quality parameters [2, 3]. Volume rendering is considered as one of the fundamental techniques in scientific visualization. The sampling rate of the raycasting process is dynamically adjusted to reliably meet a user-defined frame rate target (i.e., interactive frame rates). The adjustment is performed in object space and in image space. Finally, the performance model is used for a dynamic, balanced distribution of rendering load among multiple different GPU models. The prediction and tuning approach (section 4.1) is based on the following components:

- Assessing performance-critical numbers of raycasting acceleration techniques, including the impact of the employed acceleration techniques early ray termination (ERT) and empty space skipping (ESS) (section 4.2).
- On the fly prediction of the execution time of upcoming frames using a hybrid performance model (section 4.3).
- Balancing of the computational load among multiple devices in real-time as well as steering rendering quality towards a user-defined frame rate (section 4.4).

This chapter is partly based on these publications

- V. Bruder, S. Frey, and T. Ertl. “Real-time performance prediction and tuning for interactive volume raycasting”. In: *Proceedings of the SIGGRAPH ASIA Symposium on Visualization*. 2016, pp. 1–8 [2]
- V. Bruder, S. Frey, and T. Ertl. “Prediction-based load balancing and resolution tuning for interactive volume raycasting”. In: *Visual Informatics* 1.2 (June 2017), pp. 106–117 [3]

4.1 Load Balancing and Resolution Tuning for Interactive Volume Raycasting

We use a standard front-to-back volume raycasting approach including the acceleration techniques ERT and ESS, as well as local illumination (subsection 2.2.3). Typically, acceleration techniques produce the highest rendering time heterogeneity for volume raycasting (see chapter 3). At the core of our technique is a hybrid model that is able

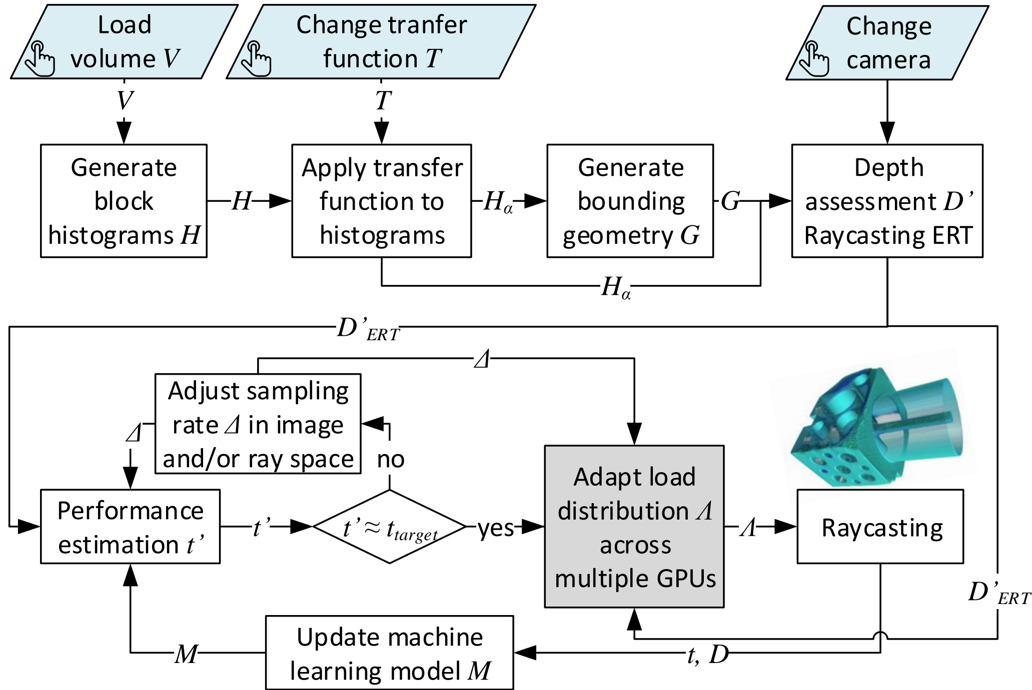


Figure 4.1: Overview of our adaptive volume rendering process. The top row depicts possible user interactions that trigger data generation and assessment methods (second row). The lower part shows our prediction and parameter tuning approach. Adaption of load distribution (gray) is only used for multi GPU setups.

to predict the execution time of the upcoming frame and adjust sampling density in object as well as image space, based on this prediction. Figure 4.1 gives an overview of our approach.

Central to all processing steps are three different user interactions (Figure 4.1 top row): loading a new volume data set, changing the transfer function, and manipulating the camera (i.e., rotation or zoom). When loading a data set, the volume is divided into coarse blocks; we use a resolution of 16^3 voxels per block as it proved fastest for the tested data sets. We calculate a density histogram H for each of those volume blocks, representing the distribution of scalar density values in its respective block. The histograms H have to be updated only if the volume data set changes.

In a next step, we use the user-selected opacity channel T_α of the transfer function to derive opacity histograms H_α from the density distribution histograms H. Again, there is one opacity histogram H_α per block, but in this case representing opacity distributions instead of density distributions. This step has to be performed whenever the user changes the transfer function. By directly evaluating the opacity histograms,

we determine which blocks of the low-resolution volume are empty, and use this information to generate a bounding geometry G that is used for empty space skipping.

For ESS, we use the Open Graphics Library (OpenGL) to rasterize G and determine the depth of the foremost (D_{front}) as well as the backmost (D_{back}) fragment of the bounding geometry G in a single render pass. Those depth values are used as ray entry and exit points. In order to incorporate estimated effects of ERT in our prediction model, we further adjust the depth values D_{back} to D'_{ERT}.

With our approach, we generally target single-node systems. The user may select a target frame rate t_{target} that we aim to constantly hold during user exploration. As parameters, we adjust the sampling rate Δ along each ray and/or the number of rays, to basically trade rendering quality for performance. For this, we follow an iterative optimization approach by looping over the following operations until we approximately predict the target frame rate:

- On the basis of the depth values D'_{ERT} and our prediction model M, we estimate the time t' that would be achieved with the current step size and/or resolution Δ.
- If the prediction t' is close to the selected t_{target}, we stop the adaption process.
- Otherwise, we calculate a new step size and/or resolution candidate for Δ.

In the case of having multiple GPUs available for rendering, we support using our prediction for load balancing between them. We adapt the load distribution Λ between available devices based on the adapted sampling rate (or image resolution) Δ, and the depth estimation D'_{ERT}.

Finally, we raycast the volume by using the obtained value for Δ (ray and/or image space) and the load distribution Λ. After the raycasting, we update our prediction model M by adding the measured values for the execution time t and the actual depth D_{ERT} after ERT, that we assess during the raycast.

4.2 Collection of Performance-Relevant Data

Object-order ESS and ERT are two widely used acceleration techniques for volume raycasting that can have a high impact on rendering times. Therefore, we include them as a central aspect of our performance assessment. In this section, we describe our approach for collecting data that is relevant with respect to those acceleration techniques. We base our assessment, as well as the actual ESS computation on a coarse volume representation. For this, we partition the volume into blocks of 16^3 voxels each and compute a density histogram for each of the blocks (see subsection 4.2.1). The histogram data is used to determine the ray entry and exit points, that define depth D without considering ERT (see subsection 4.2.2). We use those values for the prediction

as well as the actual raycasting acceleration. In subsection 4.2.3, we discuss how we incorporate an estimation of ERT effects in our model, that is based on per block opacity histograms H_α.

4.2.1 Histograms of Volume Blocks (H and H_α)

When loading a volume data set V, we logically partition it into coarse blocks of 16^3 voxels. Using all scalar values contained in a respective block, we generate one density histogram H per block. We chose a size of 64 bins for the histograms since we use data sets with 8 bit and 16 bit precision. A higher bin count could be beneficial for data sets with a higher precision per scalar value.

After applying the transfer function to the density values, parts of the volume typically become transparent in the visualization. Due to the usage of ESS, such transparent regions have a major impact on runtime performance. We compute opacity histograms H_α from every density histogram by applying the opacity transfer function $T_\alpha : \mathbb{R} \rightarrow \mathbb{R}$. Thereby, we distribute the computed values into 16 bins, mainly because it is more efficient during our ERT approximation step, without having much impact on the estimation accuracy (see subsection 4.2.3). Each bin b in the original density histogram H represents a density range $[v_{\min}, v_{\max}]$. We generate H_α from H by basically looping over those bins b. Thereby, we integrate over the range $[v_{\min}, v_{\max}]$ with the user-defined (opacity) transfer function $T_\alpha(b)$, resulting in opacity values b_α:

$$\forall b \in H : b_\alpha = \int_{v_{min}}^{v_{max}} T_\alpha(b).$$

The opacity values b_α are then used to select the respective opacity histogram bin b_α of H_α to which we add the number of corresponding elements from the original bin b of the density histogram H.

We generate one opacity histogram H_α per volume block. Generating the histograms has to be performed whenever the user either loads a new volume data set or changes the transfer function since it depends on the density values as well as the transfer function.

4.2.2 Depth Assessment (D_{front} and D_{back})

The amount of empty space depends on the volume characteristics as well as the selected transfer function. We employ our opacity-mapped histogram H_α (subsection 4.2.1) to implement object-order ESS. In a pre-processing step, we construct a bounding geometry of the volume to determine entry points (D_{front}) and exit points (D_{back}). The bounding geometry is closer to the visible data than a commonly used bounding cuboid.

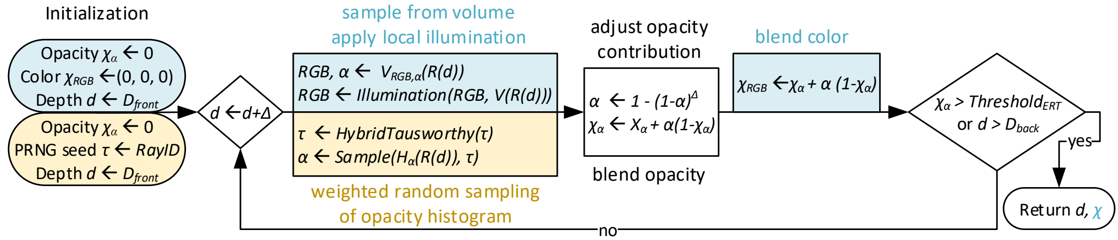


Figure 4.2: Front-to-back raycasting along a ray using sampling distance Δ. Steps in yellow are executed only for ERT estimation (as discussed in Sec. 4.2.3), the blue ones only for the actual rendering.

To decide whether a block of our proxy geometry is fully transparent, we use our previously determined opacity-mapped block histograms H_α, by simply evaluating if there are values in bins for non-transparent voxels. We generate quads for the surfaces of the outermost blocks, thereby creating a polygon mesh of the volume hull. By using a minimum blend equation, we rasterize the bounding geometry. This allows us to write the minimum depth as well as the negated maximum depth values into the frame buffer in a single render pass. Conceptually, this approach does not allow us to skip empty space inside a volume (i.e., our bounding hull). This limitation can be circumvented by using a dual depth peeling approach with multiple rendering passes, at the cost of a higher overhead time. The depth assessment step (i.e., the rasterization) has to be performed whenever the user changes camera parameters, while the generation of the bounding geometry only has to be performed whenever the transfer function or data set changes.

4.2.3 Early Ray Termination (D_{ERT} & D'_{ERT})

ERT is a simple acceleration method for volume raycasting that can possibly result in substantial performance gains (subsection 2.2.3). The actual speedup mostly depends on the data set and the transfer function. However, compared to the simplicity of the approach, the a-priori estimation of the actual performance gain is non-trivial. This stems from the fact that a possible estimation cannot be solved locally (e.g., on a per-block basis), in contrast to the depth assessment. That means we have to consider all accumulated opacities along the rays. To nevertheless achieve an estimation of the ERT impact on a depth segment D in real-time, we implement a modified version of our standard raycasting procedure.

Figure 4.2 outlines the estimation process (yellow) as well as our normal raycasting algorithm (blue). First, we initialize the opacity and the ray starting position. For the regular raycasting, we also initialize the color value, while we use the thread-id to create a seed for our pseudo-random number generator (PRNG) in the ERT estimation case.

After the setup phase, we process the raycasting loop, in which we sample at depth d along the respective ray R in front-to-back order by using step size Δ. The sampling starts at D_{front}, the entry point determined by our depth assessment (subsection 4.2.2), and we sample until we reach D_{back} or the opacity surpasses the ERT threshold value.

For the regular raycasting, we fetch the respective scalar value from the data set and apply the transfer function, resulting in color and opacity values. For the ERT estimation pre-run, we use the opacity block histograms H_α (that we also use for depth assessment), instead of sampling the volume data. For this, we start by generating a pseudo-random number τ, using a hybrid Tausworthe PRNG [89]. Next, we determine the block we are currently sampling at depth d along the ray $R(d)$. Using the opacity histogram $H_\alpha(R(d))$ of this block and τ, we randomly draw an opacity value α. Thereby, we weight each histogram bin according to its size, i.e. the sampling is proportional to the number of elements in each bin.

The core idea behind using opacity histograms H_α is to estimate the ERT behavior in a realistic manner at a fraction of the cost of the actual rendering. The cost savings mainly result from a largely reduced I/O cost, that is particularly high due to the memory bound nature of volume rendering. We use 16 byte histograms with one byte per bin for each block (a block aggregates 16^3 voxels) in our implementation (see subsection 4.2.1). This has the advantage that the whole histogram can be obtained using only a single fetch operation on modern GPUs. This is also comparably fast across multiple rays due to texture caching. By using random sampling of the opacity histogram values, we account for the statistical distribution of the actual opacity values and thereby aim to more closely reproduce the actual raycast. In addition, we sample much more coarsely along the rays, which also contributes substantially to a reduction of the computational cost compared to the regular raycast. In both raycasting passes, we account for changes of the step size Δ by adjusting the opacity accordingly and therefore making the estimation correspondent with the actual rendering, also during dynamic adjustments of the step size.

The raycasting loop terminates if the accumulated opacity χ_α exceeds a threshold (i.e., ERT happens) or the sampling along the ray exits the bounding geometry. In either case, we use the final depth value, as ERT estimation value or training data. Naturally, we present the pixel color value in the case of the actual raycasting pass.

4.3 Hybrid Performance Model

We use a hybrid performance model to perform an online estimation of the execution time of the upcoming frame. Our model may be categorized as a "semi-empirical" performance model [87], since we use empirical measurements of previous execution times as well as known attributes of our volume raycasting algorithm. To learn

hardware-specific characteristics, such as caching or swizzling algorithms, we employ a machine learning model on the basis of execution time measurements. This part of our model effectively learns and estimates the average cost σ per sample during raycasting (subsection 4.3.1). Combining this approximated sample cost with an estimated depth per ray D'_{ERT} (section 4.2), we predict the total cost t' of rendering the upcoming frame (subsection 4.3.2).

4.3.1 Machine Learning: Prediction of Sample Cost σ

We based our selection of the machine learning technique on two requirements. First, the learning algorithm has to be fast enough to work in real-time, i.e. training as well as evaluation has to be significantly faster than a single frame execution. Second, the technique should be able to perform non-linear regression. Based on these requirements and due to its comparably simple design, we chose kernel recursive least squares (KRLS) [60]. The Dlib machine learning library [97] provides an implementation of KRLS that we use in our model. We use a separate machine learning model for each device if multiple GPUs are used for rendering. Due to the nature of the KRLS algorithm, weights cannot be transferred directly between different runs, i.e. we have to build a new model for every data set.

KRLS is a kernel-based regression algorithm that is able to dynamically include measurement samples for training during runtime. This means the model is dynamically trained during runtime and does not need any prior training sequence (see subsection 4.5.2 for a discussion of the approximation accuracy and learning speed). Through the use of recursive least squares (RLS) with the addition of Mercer kernels, non-linear regression is implemented. The core of RLS is an optimization problem (whose solution is maintained every frame) to find weights w by minimization:

$$\min_{w} \left(\sum_{i} \lambda^{n-i} (y_i - x_i^T \times w)^2 \right) \tag{4.1}$$

Here, (x_i, y_i) is a pair of training points, where x_i denotes a feature vector and y_i is a target scalar value. The so-called “forgetting factor” λ may be used to give exponentially less weight to older samples.

We use linear radial basis functions (RBF)s as kernel functions because of their flexibility. The target scalar value we predict is the sample cost σ, while our feature vector consists of five properties that can have a significant impact on σ or are a performance indicator:

- **Viewing angles (ϕ, θ).** We derive them directly from the rotation of our arcball camera. Among others, they impact performance because of different texture respective memory access patterns caused by the perspective.

- **Size of a splatted voxel.** This has a potentially significant impact on texture caching and also varies with viewing distance and resolution. It is one of our tuning parameters.
- **Step size along rays.** This value has similar properties as the size of a splatted voxel, but in ray space. It is also one of our tuning parameters that defines the number of overall samples. Changed caching patterns may impact performance here as well.
- **Execution time.** We use the execution time of our pre-rendering ERT approximation step. It roughly resembles the actual rendering time, when put in relation.
- **Maximum ray depth.** This is a possible indicator for maximum warp/wavefront processing time. All threads (usually 32 or 64) in a single warp/wavefront run in lockstep on current GPUs, meaning that faster threads need to wait until all threads in the warp/wavefront have finished processing.

Overall, those features reflect performance influencing characteristics on a hardware level, such as caching behavior or different texture access patterns [31]. All used features are available with no, or only minimal, computational overhead, therefore being well suited for on-line runtime prediction of our volume raycasting application.

The implementation of KRLS provided by Dlib provides the possibility to change the maximum number of dictionary entries (used to represent the regression function), a tolerance value, and a γ-parameter for the RBFs. We determined the following set of well working parameters by using a grid search auto tuning approach: $\gamma = 0.00025$, a tolerance of 0.006 and a dictionary limit of 10 million entries.

4.3.2 Analytical Model: Prediction of Frame Execution Time

After the previous steps, we can combine the entry and exit values of our proxy geometry, which we also use for ESS (section 4.2), with step size and image resolution to calculate the number of samples we are going to take during raycasting of the upcoming frame. For this, we use the 2D texture that is generated during the rendering pass of our proxy geometry and generate a full mipmap-stack of the texture. The topmost layer of the stack effectively contains the average minimum and maximum depth values $\overline{d}_{\text{front}}$ and $\overline{d}_{\text{back}}$. Combined with our estimated cost per sample σ (see subsection 4.3.1), we can calculate an estimate of the total frame execution time t':

$$t' = \frac{7 \cdot (\overline{d}_{\text{back}} - \overline{d}_{\text{front}})}{\Delta} \cdot \sigma.$$

We compute the average ray length $\overline{l} = \overline{d}_{\text{back}} - \overline{d}_{\text{front}}$ (with d_{front} denoting the ray entry point and d_{back} being the estimated termination depth in ray space). Dividing the value $\overline{l}$ by our step size Δ gives us the average samplings per ray that we multiply by

factor 7 (one RGBA value, plus six that we need for the gradient estimation with central differences). Finally, we multiply the result with our cost per sample estimate σ, to gain the prediction of the total rendering time t' of the frame.

4.4 Prediction-Based Parameter Tuning

Our real-time performance prediction model provides us with the basis for several applications. In this paper, we present two distinct scenarios for our interactive volume rendering application. First, we use our model to dynamically steer the sampling resolution of the volume raycasting application. This is done in ray space as well as in image space, with the goal to achieve constant frame rates and thereby high responsiveness and execution efficiency. Second, we use our online predictions to dynamically distribute and balance computational load among multiple different GPUs.

4.4.1 Adaption of the Sampling Resolution

Central to our approach for dynamic adaption of the sampling resolution, is the definition of a target frame rate t_{target}. By adjusting the sampling rate in ray space and/or image space, our technique tries to consistently achieve the target frame rate during user exploration. We use the tuning parameter Δ, that represents either the step size along rays, or the image resolution in x and y direction in the case of image space adaption. We also support a hybrid approach that adapts both parameters at the same time. We follow an iterative optimization approach, using linear extrapolation and bisection during each iteration:

$$\Delta = \begin{cases} \Delta_{\text{upper}} \cdot \frac{t_{\text{target}}}{t'_{\text{upper}}} & \text{if } \tilde{t}_{\text{upper}} < t_{\text{target}} \\ \Delta_{\text{lower}} \cdot \frac{t_{\text{target}}}{t'_{\text{lower}}} & \text{if } \tilde{t}_{\text{lower}} > t_{\text{target}} \\ \Delta_{\text{lower}} + (\Delta_{\text{target}} - \Delta_{\text{lower}}) \frac{t_{\text{target}} - t'_{\text{lower}}}{t'_{\text{upper}} - t'_{\text{lower}}} & \text{else} \end{cases} \quad (4.2)$$

Here, t'_{upper} and t'_{lower} denote the smallest (respective largest) estimated timing below (respective above) t_{target}. Analogously, Δ_{upper} and Δ_{lower} stand for the respective sampling resolution. We use the same approach for adjustment of the sampling resolution in image and in ray space. Note, that in the case of image space adaption, we normalize relative to the size of a splatted voxel, while also taking into account the quadratic image resolution adaption. For tuning of the sampling distance along the rays, we also factor in the additional samples used to evaluate gradients, which we need for local illumination. Further, we assume (as can be seen in the top two conditions in Equation 4.2) that the sampling resolution has an approximately linear impact on performance. A new candidate resolution Δ in ray and/or image space is generated via a linear interpolation, as denoted in the “else”-branch of Equation 4.2.

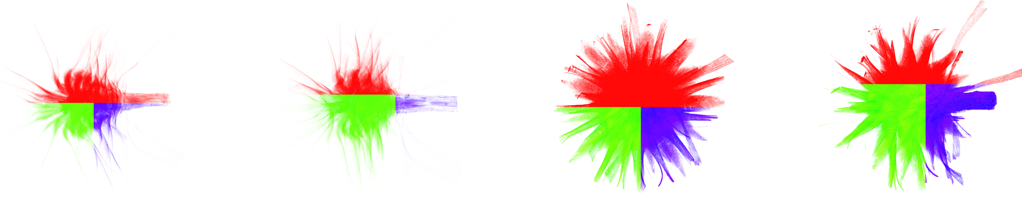


Figure 4.3: Load balancing distribution among three GPUs for different configurations during an interaction sequence of the Flower data set. Color coding by GPU: red for Titan X (Pascal), green for GTX 980 and blue for GTX 960. The load distribution is adapted dynamically.

Finally, we introduce a fixed maximum adaption of $\Delta_{max} = 0.8 \cdot \Delta$ per frame, to avoid overcompensation. This form of damping further helps in avoiding lags that may result from overestimated sampling resolution adjustments. We only use this limit while increasing resolution, since overly decreasing it does not have a negative impact on performance, only on quality. The reasoning behind this is that we aim for the system to be always responsive. However, if the system significantly underestimates the performance impact, it may happen that the application becomes unresponsive and therefore cannot adapt to changes until the kernel run finishes. On the other hand, if the system overestimates the performance, it can quickly re-adapt for better quality.

4.4.2 Load Balancing

As a second use case, we employ our prediction model for load balancing in a multi-GPU setup. Conceptually, we use a separate machine learning model M_i (see subsection 4.3.1), generating a distinct sample cost estimate σ_i for every available GPU i. We calculate the load distribution Λ_i for n GPUs:

$$\Lambda_i = \frac{(1-\zeta)}{n} + \zeta \cdot \frac{\prod_{j=1, j \neq i}^{n} \sigma_j}{\sum_{k=1}^{n} \sigma_k}$$

We multiply the sampling costs σ_i of all devices except the one that is being calculated, and divide the resulting product by the sum of all sampling costs. Here, ζ denotes a damping factor that we use to avoid oscillation effects that are otherwise present during load balancing. By using an empirical grid sampling approach, we determined a damping factor of $\zeta = 0.5$ to give the best results for the tested data sets.

We partition our image space into 2D tiles with a size of 8×8 pixels each, to avoid warp/wavefront divergence (typically, warps on NVIDIA GPUs have a size of 32 threads, wavefronts on AMD GPUs 64 threads). We then use a k-d tree to distribute the tiles

among the available devices, based on the average depth per tile as well as the determined load distribution Λ_i per device. For this, we use the depth values from our rendered proxy geometry texture (see subsection 4.2.2), more precisely the third mipmap-layer that corresponds to our tile size. Figure 4.3 shows four renderings of the Flower data set during a interaction sequence, where the image space partitioning among three distinct GPUs has been encoded via the color channel. Note that the impact of ERT significantly impacts load balancing that is dynamically adjusted during runtime.

4.5 Results

We evaluate our approach using seven volume data sets (Table 2.2) and compare the results against an implementation without any parameter adjustments as well as two other adaption approaches. Overall, this results in four methods:

1. **No adapt.** A fixed step size of 0.75 times the length of a voxel in z-direction as well as one ray per pixel are used for sampling. We predict the execution time of each frame using our method, but do not adjust any parameters.

2. **Our adapt.** We use our method to predict execution times of upcoming frames and steer the step size and/or image resolution accordingly.

3. **Last frame.** Here, we adjust the sampling of the volume based on the execution time of the last rendered frame.

4. **Two pass.** Two rendering passes are conducted, as a very simple form of progressive rendering. In the first pass, a quarter of the sampling parameters from the last frame is used for rendering. In case of the execution time being lower than half of the target frame time, we linearly extrapolate the sampling parameters according to the leftover rendering budget, and render a second time.

We evaluate our approach in two scenarios with different hardware setups:

(A) **Single-GPU system** that is used to test general performance characteristics at the example of single data sets. Evaluation includes the analysis of a frame time diagram, the overall accuracy of our approximations and prediction, and the computational overhead of our prediction model.

(B) **Multi-GPU system** that features three distinct GPUs and is used to evaluate our approach including load balancing. We compare our technique against three others, analyze our load balancing in detail, and compare adaption in image space against adaption in ray space and against a hybrid adaption in both spaces.

For all adaption modes, the frame target was set to 30 fps (A) or 40 fps (B). Both values are generally considered interactive in this context. We recorded a 30 s long sequence of user interactions using the four modes mentioned above for comparison. The sequences contain changes of the transfer function as well as rotation and zooming of the camera in an arcball-style. Example renderings for different configuration in such a sequence are shown in Figure 4.4c–f. The parameters of the machine learning model used for prediction are reset after the execution of each sequence.

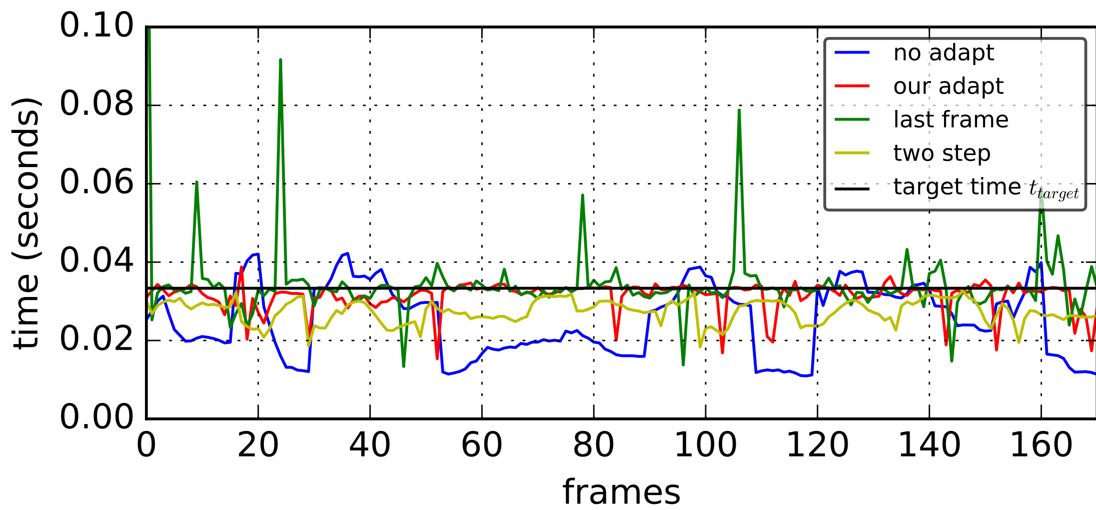
We conducted all measurements on a workstation with an Intel Core i7-6700 CPU, 16 GB of RAM and either one (A) or three graphics cards (B). Table 2.1 lists the four GPUs used for rendering. The GTX 680 was used in the single GPU system (A), while the others were used to evaluate load balance (B).

4.5.1 Analysis and Comparison of a Sequence with a Single GPU

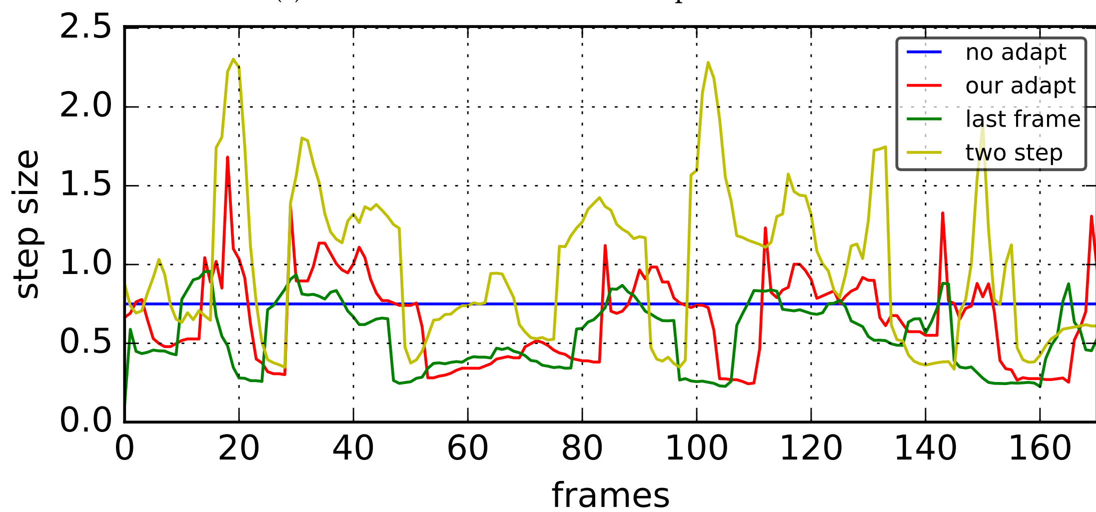
Figure 4.4a shows a frame time diagram for a sequence of rendering the Parakeet data set in four different modes on a single GPU (scenario A). The black marker line depicts the frame target of 30 fps. Figure 4.4b shows the corresponding step size factors for the respective frames. The step size factor is relative to the voxel length, i.e. a smaller step size factor indicates a higher sampling rate (and typically a higher rendering quality). We use a fixed step size of 0.75 times the length of a voxel in z-direction as step size for the mode without adaption (blue line Figure 4.4). As can be seen in the graph, using no adaption leads to significant deviations from the target frame rate. This is especially the case for changes to the transfer function, e.g. frames 20 and 95 (renderings Figure 4.4c-Figure 4.4f). Even small changes to the transfer function can have substantial performance impacts if large portions of the volume become transparent or opaque. Smaller deviations are usually caused by changes to the camera configuration that are comparably smooth during typical exploration of the data set.

In contrast, when using the adaption based on our model, the frame time stays around the selected target, even in the case of changes to the transfer function. At the same time, an overall higher sampling rate is achieved. However, the frame times for our approach also show a few outliers with shorter execution times. Those are mainly caused by our conservative adaption for higher sampling resolution (see subsection 4.4.1). Some of the outliers can also be traced back to underestimation or overestimation of the performance impact cased by ERT (see subsection 4.5.2). We assume that smaller deviations are caused by our machine learning-based sample cost estimation σ'. Overall, there are no outliers with longer frame times of note. This means that during the sequence interactivity was kept at all times, guaranteeing a high responsiveness for the user.

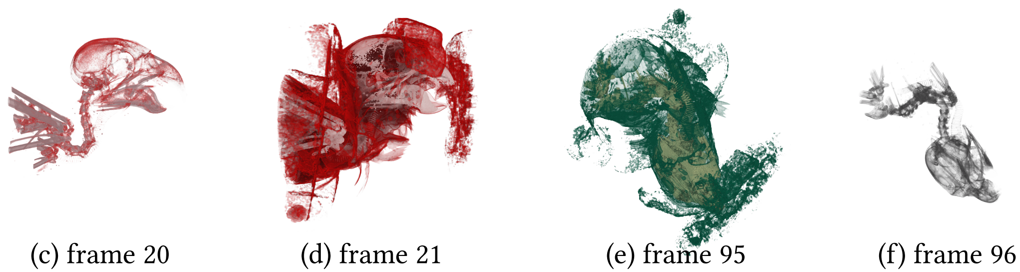
In comparison, the adaption mode based on the last frame (green curve) shows huge frame time spikes that may cause poor responsiveness and jerky motions during user exploration. Those outliers are mainly the result of changes to the transfer function



(a) Frame times with different adaption modes.

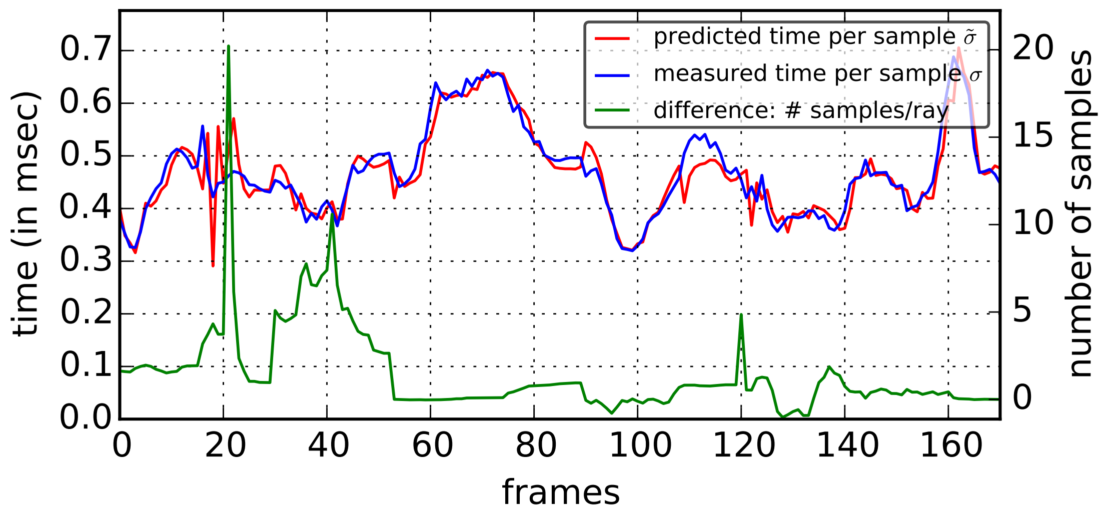


(b) Step sizes with different adaption modes.

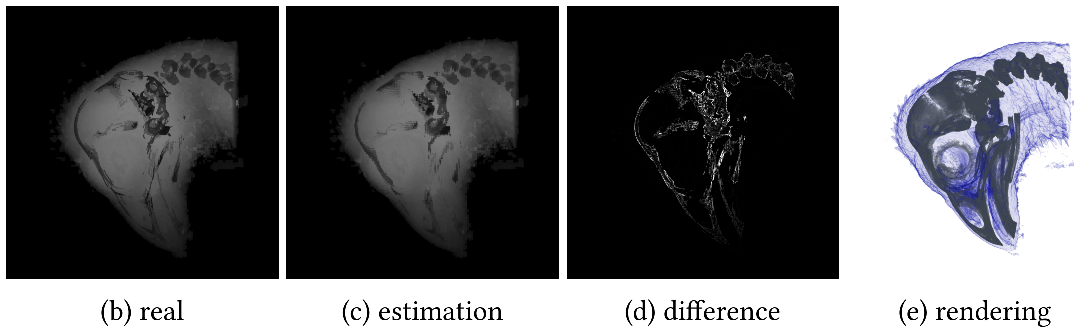


(c) frame 20 (d) frame 21 (e) frame 95 (f) frame 96

Figure 4.4: Plot (a) shows a frame time sequence at the example of the Parakeet data set. Frame times of our approach (red) are shown in comparison to the three other methods. The corresponding step sizes can be found in (b), a lower step size indicates a higher quality. (c)-(f) depict example pairs of consecutive renderings from the sequence.



(a) Approximation efficiency of sample count and cost.



(b) real (c) estimation (d) difference (e) rendering

Figure 4.5: (a) shows the estimation accuracy for average samples per ray (green) and sample cost σ (blue/red) at the example of the Parakeet sequence (see Figure 4.4). (d) shows the difference of measured ERT depths (b) and our approximation (c) for the Hoatzin data set (e). Values in (d) are enhanced by factor 4 for better visibility.

since the last frame approach cannot handle those by design (basically, it has one frame delay). For these cases, the sampling density is naturally higher for the last frame mode, while otherwise being on a similar level compared to our prediction technique.

We also compare our approach against a two pass rendering mode (yellow curve). This mode has the advantage, that the frame target is hardly ever exceeded. The major drawback of this technique is the much lower sampling rate that eventually leads to a lower overall rendering quality. This is caused by the lower rendering budget for each frame, because of the additional time required for the pre-rendering pass.

4.5.2 Approximation and Prediction Accuracy

Figure 4.5a shows the difference of our prediction (red curve) compared to the measurement (blue curve) of the cost per sample σ for the same rendering sequence as in the previous section. As can be seen, our machine learning model is able to make fairly accurate predictions of the cost per sample σ after learning only a few samples (30 frames). The differences are also reflected in the overall prediction (see Figure 4.4a).

Further, Figure 4.5a shows the difference between the estimated number of samples per ray and the measured number (green curve). Here, the discrepancies are caused by underestimation or overestimation of the impact of ERT on the number of samples. We assume that this is caused by inaccurate results from our probabilistic estimation for some difficult cases.

To investigate the efficiency of our depth estimation including ERT, Figure 4.5 shows the measurement (b), our estimation (c) and the difference (d) of a rendering of the Hoatzin data set (e). The intensity of (d) has been scaled by factor 4 to emphasize the differences. Overall, the depth estimation is fairly accurate although there are some discrepancies, mainly at the edges. Those are caused by difference between the proxy geometry (used for our pre-rendering step) and the original high resolution data. Other differences can be caused by our stochastic methods (see subsection 4.2.3).

4.5.3 Prediction Overhead

One important aspect of our prediction approach is real-time capability. That means, the computational overhead needs to be significantly lower than actually rendering the volume data. We explicitly designed our pipeline to provide interactive exploration capabilities, e.g. training, which is done on the CPU, and pre-processing run interleaved. Furthermore, it is conceptually possible to do the pre-processing steps (i.e., depth assessment and ERT approximation) on a separate device, freeing up resources for the actual frame computation. For instance, one could use an integrated GPU for the pre-processing and a dedicated graphics cards for rendering. Table 4.1 lists upper bounds for the processing times that the various steps in our pipeline needed. All times were measured using the Chameleon data set on the NVIDIA GTX 960. Refer to Figure 4.1 for an overview of our pipeline and the listed steps.

As can be seen, the computational overhead of our prediction model is comparably low. For the tested Chameleon data set, generating the bounding geometry (only needed when changing the transfer function) and performing the depth assessment has a combined execution time of about 6 ms, a fraction of an interactive frame time. Training and prediction are substantially faster yet. Overall, the measurements show an acceptable computational overhead, that implies a smooth online usage of our prediction method in average workstation environments.

Table 4.1: Maximum Execution Times for the Chameleon Data Set

Action	Required if	Max. time [ms]
Generate histograms	Volume data set changes	1000
Generate bounding geometry	Transfer function changes	3.551
Generate opacity histograms	Transfer function changes	0.053
Depth assessment	Camera changes	2.680
ERT approximation	Camera changes	0.028
Training	Frame was processed	0.015
Single prediction	Step size Δ is adjusted	0.001

4.5.4 Interaction Sequences

We conducted detailed measurements of interaction sequences for seven different volume data sets listed in Table 2.2. For this, we created individual $30\,\mathrm{s}$ sequences for each data set to capture different usage scenarios. The measurements discussed in this section were done using system setup (B) with three GPUs (Table 2.1) and our load balancing technique enabled. In case of the mode without adaption, the step size along the rays was set to 0.25 times the voxel length in z-direction. The target frame rate was 40 fps ($0.025\,\mathrm{ms}$ frame time). We chose a higher frame rate target and smaller step size to increase rendering complexity to stress the three GPUs sufficiently. The viewport was kept at 1024^2 pixels, the ray sampling size was adjusted for the three adaption modes.

Figure 4.6 shows the results of our measurement series. As an indicator of the rendering quality, we use the average step size along the rays (smaller is better). To judge the prediction accuracy, we use the root-mean-square error (RMSE), as a measure for the difference of predictions $\hat{y}_t$ and measurements y_t across a sequence of n frames:

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{t=1}^{n} (\hat{y}_t - y_t)^2}{n}}. \tag{4.3}$$

In the third column of Figure 4.6, the maximum absolute error above the frame target is shown. This represents the biggest absolute difference between the target execution time and the measured time. It is an indicator of the worst case performance, since a high error usually results in lags or jerky motions during user exploration. One of the main goals of our technique is to avoid such high frame times.

Figure 4.6 shows that our approach (red) has a comparably low RMSE (i.e., a low deviation from the target frame rate). Only the two pass mode (yellow) performs better in some of the sequences. At the same time, the step size (i.e., rendering quality) of our approach is substantially better than the two pass mode in all cases and only slightly worse than the last frame mode (green) in some cases. The mode without any step size adaptations

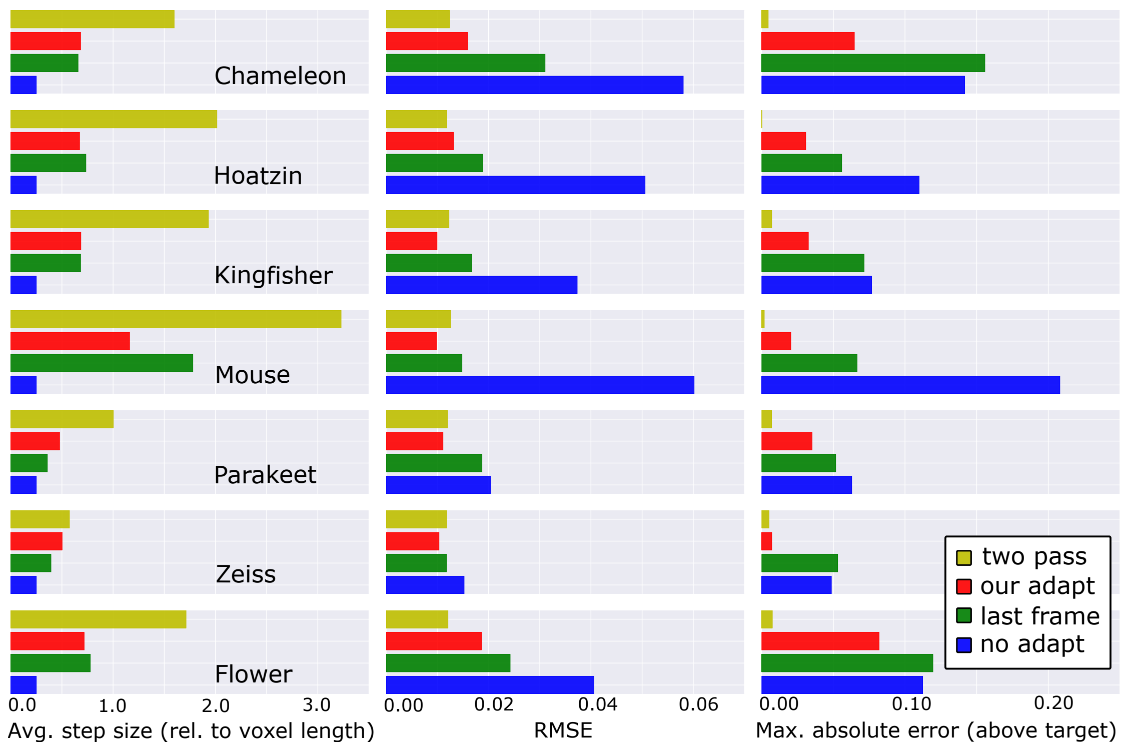


Figure 4.6: Comparison of our approach against the other modes using individual interaction sequences for seven data sets. A smaller average step size factor indicates a better quality, while a lower RMSE indicates a better performance (i.e., the frame rate stays closer to the target). Maximum absolute errors indicate worst case performance and possible lags.

(blue) performs best in terms of rendering quality, but has a significant deviation from the frame target in all sequences. The maximum absolute error indicates a bad worst case performance of the two pass mode, while our approach performs fairly well in this regard. Naturally, the two step mode is better than ours in this regard. The results reflect the ones observed and discussed above with the detailed analysis of the sequence, the considerations are basically the same. Overall, our measurements show that our technique keeps a balance between rendering quality and speed while guaranteeing responsiveness during interactive exploration of the volume data set.

4.5.5 Load Balancing

We compare our load balancing approach against a static distribution based on all our empirical measurements, i.e. the average sample cost over all measured data sets respective sequences. Figure 4.7 shows the frame diagrams of the Chameleon and the Flower data sets for the two modes. The solid lines depict the frame times for the three

Table 4.2: Quality Impact of Adaption

Adaption type	MSE	SSIM	PSNR
Image space	34.256	0.981	37.215
Ray space	20.563	0.988	40.237
Hybrid	20.170	0.992	40.747

All values are averaged over the whole sequence and relative to a reference rendering without adaption (Zeiss data set).

tested GPUs. The stacked semi-transparent plots show the relative load distribution among the GPUs that is dynamic in case of our approach and static otherwise.

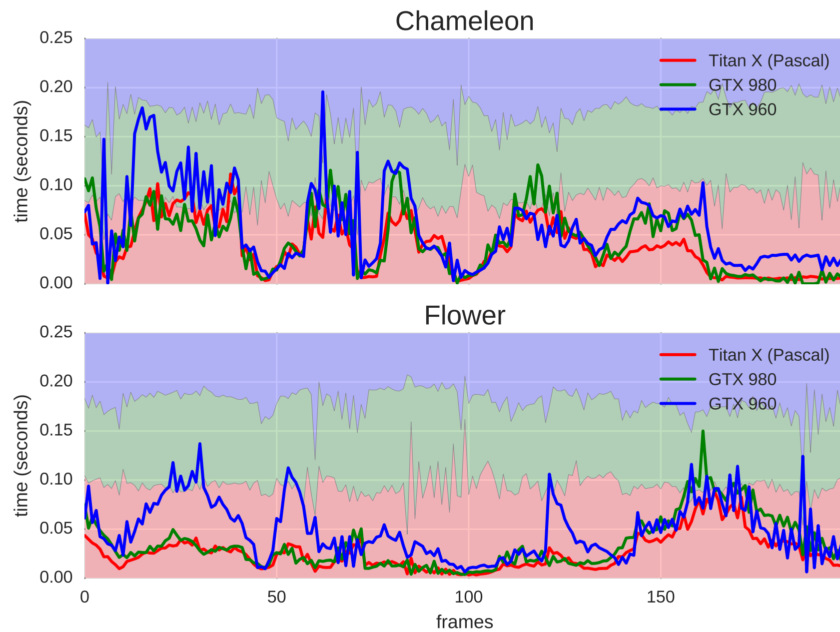
Although the overall trend is similar, our load balancing generally outperforms the static approach. However, the dynamic balancing is not without flaws: An oscillation pattern is noticeable, which is common to load balancing methods in general. We counter this effect by using a damping factor of $\zeta = 0.5$, an empirically determined value that worked best across the tested volumes. A higher damping value resulted in converging of the load balancing and static modes, while lower values worsen the oscillation effect. In addition, the general problems of our model, which are discussed above, also show in the load balancing. Throughout all tested volumes, the achieved load distribution efficiency (i.e., how well the timings of all GPUs match on average) is about 18% better for our load balancing approach compared to the static distribution, proving our approach to be well suited for this task. Improvements with respect to the damping factor and prediction accuracy could possibly further improve the results.

4.5.6 Image Versus Ray Space Adaption

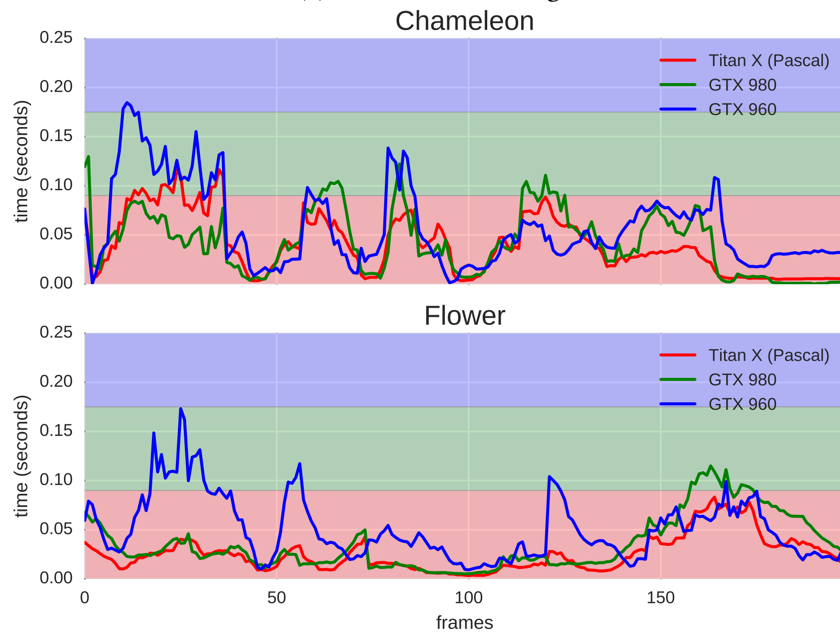
With our prediction approach we can dynamically steer the rendering quality by adapting the sampling resolution in ray space and/or in image space. In this section, we present our evaluation of the image quality when adapting the sampling factor Δ only in ray space, only in image space, or both at the same time (hybrid). We use three measures to evaluate the image rendering quality:

- Mean-squared error (MSE) (the same as the RMSE (Equation 4.5.4) but without applying the square root), lower is better.
- SSIM, closer to 1 means a higher similarity to the reference.
- Peak signal-to-noise ratio (PSNR), higher is better.

Table 4.2 lists the averages of the image quality measures for the interaction sequence of the Zeiss data set, while Figure 4.8 shows a direct comparison of a single frame for



(a) Our load balancing



(b) Static balancing (empirical)

Figure 4.7: Comparison of frame times (solid lines) and distribution (stacked areas) among three different GPUs: one shows measurements with dynamic load balancing based our prediction (a); the other one depicts a static load distribution (b), based on empirically determined average sample costs $\bar{\sigma}_i$ per device.

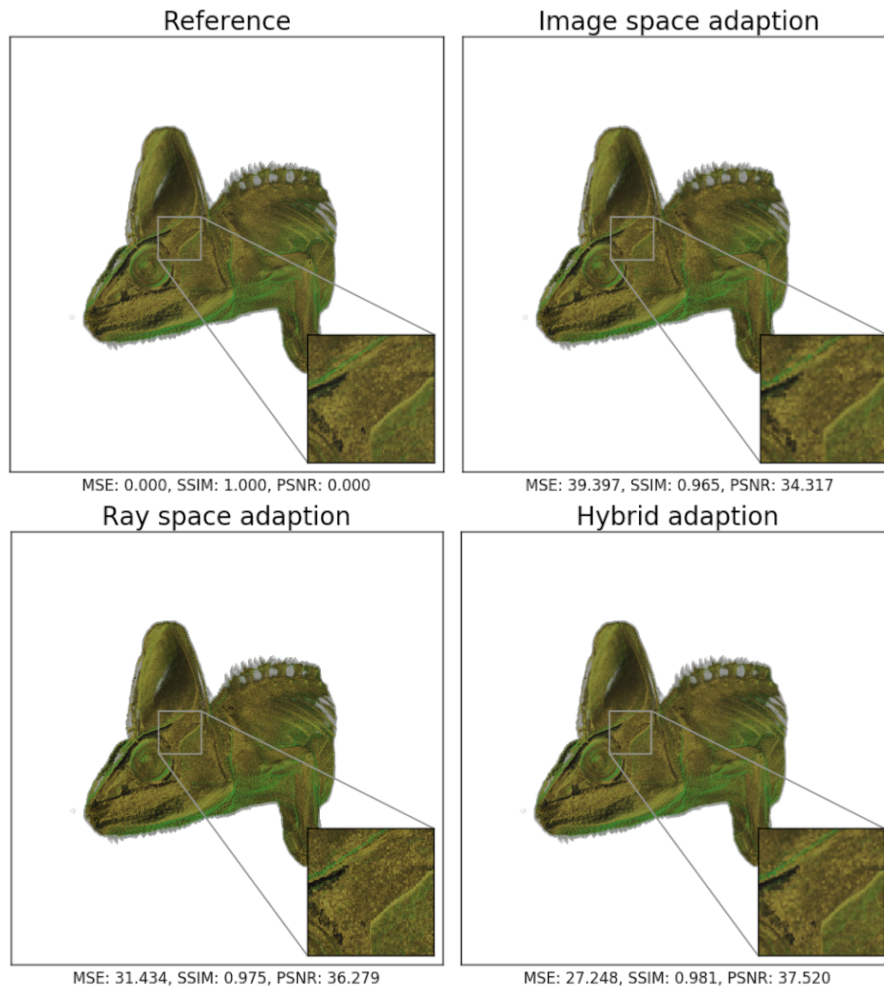


Figure 4.8: Comparison of adaption in image space, ray space, and hybrid for a frame of the Chameleon data set.

the three methods against a reference rendering. The reference is rendered with an image resolution of 1200^2 pixels and an integration step size of 0.1 times the voxel length in z-direction, the adapted sampling resolution is at most 1024^2 rays respective a step size of 0.25 times the voxel length. As can be seen in Figure 4.8 and Table 4.2, the ray space adaption generally outperforms the image space adaption in the calculated metrics. The hybrid approach yields similar results or slightly outperforms ray space adaption. The exemplary renderings in Figure 4.8 show that a low integration step size in ray space may result in ringing artifacts that are less prominent when using the hybrid adaption, resulting in a higher image quality. In particular, this is the case for data sets containing sharp edges and thin surfaces where under-sampling may occur more easily (e.g., the Zeiss data set).

4.6 Future Directions

A possible direction for future work is the improvement of the prediction accuracy, since the ERT approximation in particular may lead to inaccurate results in some cases. A possible approach to counter this inaccuracy could be the introduction of an uncertainty quantifier that incorporates a measure of the distance of the current configuration to all previously learned ones into the model. For instance, a high distance could then trigger a more conservative estimation. Other means to increase the prediction accuracy could be to perform a short learning run after loading a data set that covers multiple important configurations. However, those configurations with a characteristic performance are highly dependent on the data set, which is why it is often difficult to make a clear assessment in advance (see section 3.1). Finally, transferring learned features between data sets might be an option to improve prediction accuracy, in particular in the beginning of an exploration session. However, this is not possible with the currently used KRLS machine learning method.

The load balancing would also improve from such refinements to the prediction approach, possibly allowing for a lower damping factor and therefore higher cost savings overall. Currently, the whole data set needs to be resident on all devices to avoid expensive host-to-device copies. The partitioning of the rendering work is done in image space. A possible extension could use a partitioning in object space with an additional compositing step. This would allow only parts of the data to reside on the respective GPU. However, some overlap of the data partitions would be needed on the different devices to allow for actual load balancing. Lowering this overlap would lead to less flexibility regarding the load distribution while increasing it would lead to higher memory requirements.

CHAPTER 5

Performance Modeling for Runtime Optimization and Cost Savings on Distributed Memory Systems

The previous chapter discussed performance modeling and prediction approaches on workstation systems that use one or multiple GPUs for parallel computation of visualizations. This chapter complements this by introducing techniques for performance modeling and prediction of visualizations on distributed memory systems such as supercomputers.

Computational simulations on supercomputers produce massive data sets, with meshes containing billions or even trillions of cells per time step. To complete the visualization on interactive time scales, the process is typically parallelized across hundreds of supercomputer nodes. Further, the visualization often occurs on the same supercomputer that performs the simulation, obviating the need to relocate simulation data. Performance modeling and prediction in such environments plays an important role since it can be used to reduce compute cycles that are particularly expensive on supercomputers. Further usages include infrastructure planning and optimized remote streaming of visualization results.

This chapter covers three contributed techniques that model and predict performance on distributed memory systems with different objectives and metrics:

- An approach for in situ generation of image databases to achieve cost savings on supercomputers is introduced in section 5.1 [6]. It is a hybrid between traditional inline and in transit techniques that dynamically distributes visualization tasks between simulation nodes and visualization nodes, using probing as a basis to estimate rendering cost.
- A data-driven, neural network-based approach for prediction of the runtime performance of a distributed volume renderer is presented in section 5.2 [14, 9]. It aims to support cluster hardware acquisitions.
- A technique that dynamically adapts encoder settings for image tiles to yield the best possible quality for a given bandwidth in remote rendering scenarios is introduced in section 5.3 [11].

This chapter is partly based on these publications

- V. Bruder, M. Larsen, T. Ertl, H. Childs, and S. Frey. "A Hybrid In Situ Approach for Cost Efficient Image Database Generation [in preparation]" [6]
- G. Tkachev, S. Frey, C. Müller, V. Bruder, and T. Ertl. "Prediction of Distributed Volume Visualization Performance to Support Render Hardware Acquisition". In: *Proceeding of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. 2017, pp. 11–20 [14]
- S. Frey, V. Bruder, F. Frieß, P. Gralka, T. Rau, T. Ertl, and G. Reina. "Trade-offs and Parameter Adaptation in In Situ Visualization [to appear]". In: *In Situ Visualization for Computational Science*. Ed. by H. Childs, J. C. Bennett, and C. Garth. Springer, 2022 [9]
- F. Frieß, M. Landwehr, V. Bruder, S. Frey, and T. Ertl. "Adaptive Encoder Settings for Interactive Remote Visualisation on High-Resolution Displays". In: *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. Oct. 2018, pp. 87–91 [11]

5.1 A Hybrid In Situ Approach for Cost Efficient Image Database Generation

Generally, leading-edge supercomputers are expensive, meriting significant investigation into optimizing their usage. Several new supercomputers are built annually with hardware procurement costs in the range of hundreds of millions of euros, and their true costs rising higher over time, including energy, staffing, and upkeep. Each job running on a supercomputer shares these costs. An important way to optimize a supercomputer's usage is to optimize individual jobs, i.e., having a job complete using

fewer node hours. If a speedup can be aggregated over all jobs, then the result can be profound, potentially creating millions of euros of extra node-hours for additional computations.

With our approach, we optimize jobs on supercomputers in the context of in situ visualization, i.e., visualizing data as it is generated (see section 2.3). Thereby, we specifically consider an important technique for in situ visualization: the generation of image databases of volume renderings in the style of the Cinema project [16]. However, our technique is applicable to any in situ visualization setting that can be split into many small tasks.

Despite the growing popularity of in situ visualization, there is still much room for improvements in terms of cost and efficiency. Regarding cost, in situ visualization will, despite saving on I/O, still require significant computational resources —in situ routines sometimes use 10% or more of the simulation's resources. The exact proportion of time between visualization and simulation varies based on the nature of the simulation and the data it produces, the visualization algorithm, the visualization frequency and other factors. The second observation, that in situ visualization is inefficient, is discussed further in the next section.

5.1.1 In Situ Visualization

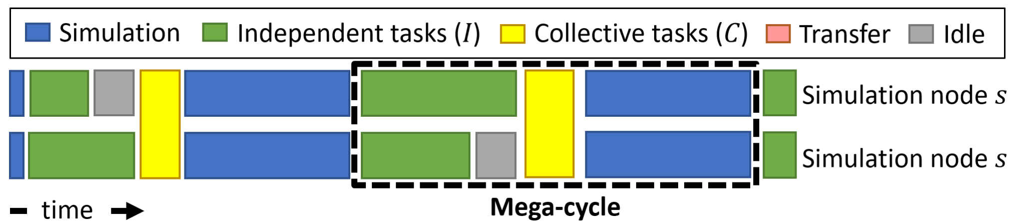
In situ visualization generally occurs within "mega-cycles," which perform both simulation and visualization. Specifically, a mega-cycle consists of advancing a simulation from some cycle n to another cycle $n + m$ as well as visualizing the data from the previous mega-cycle (i.e., cycle $n - 1$). Further, within a mega-cycle, there are two different types of visualization tasks to perform: (1) tasks executing independently from each other (e.g., rendering images of a data partition), and (2) collective tasks executing on all pieces of data at once (e.g., compositing partial result into one final image). Figure 5.1 shows how these tasks are scheduled within a mega-cycle for the three in situ approaches using notional Gantt charts.

The remainder of this section is organized as follows: First, the four types of inefficiency are described. Second, the question how the traditional in situ processing types (inline and in transit) suffer from different kinds of inefficiency is addressed. Finally, opportunities for a hybrid approach to reduce inefficiency are discussed.

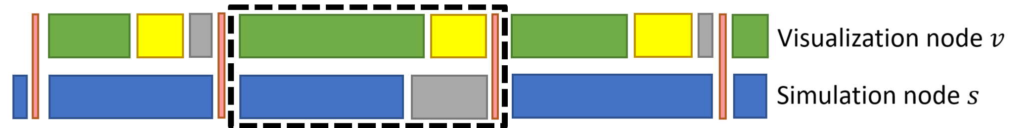
In Situ Inefficiencies

Inefficiencies with in situ processing stem from two main categories: running in parallel and running on separate resources. Further, each of these two categories has two distinct types of inefficiency.

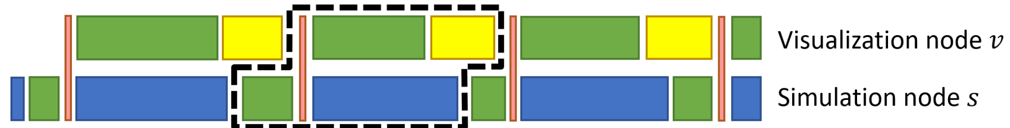
The two inefficiencies from running in parallel are:



(a) Inline: No dedicated visualization resources, alternate between simulation, independent tasks, and collective tasks. Idle times are caused by variability.



(b) In transit: Independent and collective tasks are done on dedicated resources, idle times caused by rightsizing.



(c) Hybrid: Dynamic distribution of independent tasks between visualization and simulation nodes. Collective tasks are processed on visualization nodes only.

Figure 5.1: Gantt diagrams of the two conventional in situ processing schemes (a, b) and our hybrid approach (c).

(i) **Variability**: certain operations execute for variable amounts of time, and the nodes that run for longer periods of time create a bottleneck that leads other nodes to sit idle. In particular, the cost of rendering images varies significantly across nodes depending on the input data—which often changes with the progression of the simulation—and rendering parameters (e.g., transfer function for volume rendering), especially when using acceleration techniques like empty space skipping.

(ii) **Scalability**: certain operations exhibit limited scalability, and running them at scale causes all nodes to run inefficiently. In particular, the compositing of partial images (sub-images) into one final image frequently exhibits poor scalability [142].

These inefficiencies are related, but distinct. In particular, an algorithm can suffer from delays due to scalability even if every compute node has the same amount of work to perform. Further, an algorithm with no parallel coordination is very scalable, but it can suffer from delays due to variability if some compute nodes have much more work to perform than others.

The two inefficiencies from running on separate resources (i.e., dedicated visualization nodes) are:

(iii) **Overhead**: transferring data from the simulation nodes to the visualization nodes causes delays in multiple ways: the simulation must take time to send its data (which often involves encoding into a string of bytes), the visualization routines must receive the data (which often involves decoding a string of bytes), and the network has extra traffic.
(iv) **Rightsizing**: visualization tasks rarely exactly align with the number of visualization nodes. If there are too many nodes for the desired tasks, then the visualization nodes sit idle. If there are too few visualization nodes for the desired tasks, then either the simulation nodes will need to block and wait for them to complete or tasks need to be dropped.

Traditional In Situ: Inline and In Transit

Traditionally, in situ visualization can be roughly categorized into two types: inline and in transit. In the following formulas, timings and costs are denoted with Greek letters, while tasks and nodes are denoted by Latin letters.

For ***inline*** (Figure 5.1a), each (simulation) node $s \in S$ both conducts the simulation (consuming time $\hat{\sigma}$) and carries out independent and collective visualization tasks. Independent tasks $(s, i) \in I$ are specified via the simulation node s whose data is visualized and i depicting the visualization configuration (e.g., camera position, transfer function, etc.); corresponding timings are denoted via $\iota_{s,i}$. The timings of collective tasks $c \in C$ appear as ζ_c:

$$\hat{\tau}_{\text{inline}} = \hat{\sigma} + \max_{s \in S} \left(\sum_{i \in I_n} \iota_{s,i} + \sum_{c \in C_n} \zeta_c \right), \text{with} \bigcup_{s \in S} I_s = I.$$

This indicates that the biggest issues with inline are **(i) variability** and **(ii) scalability**. For variability, depending on the data generated by simulation node s, the total cost $\sum_i \iota_{s,i}$ can vary significantly, forcing some nodes to wait and sit idle. Regarding scalability, running collective tasks at high scale—on all simulation nodes S—is inefficient, yielding comparably large ζ_s.

For ***in transit*** (Figure 5.1b), simulation nodes S are supplemented with dedicated visualization nodes V that only perform visualization tasks. This means that all collective tasks only occupy the visualization nodes, and all independent tasks are transferred from simulation nodes $s \in S$ to visualization nodes $v \in V$. Simulation nodes conduct the simulation and copy the data, inducing communication overhead for both types of

nodes. Overhead times are denoted as χ_s and χ_v:

$$\begin{aligned}
\hat{\tau}_{\text{sim}} &= \hat{\sigma} + \max_{s \in S} \left(\chi_s\right), \\
\hat{\tau}_{\text{vis}} &= \max_{v \in V} \left(\chi_v + \sum_{(s,i) \in I_v} \iota_{s,i} + \sum_{c \in C_v} \zeta_c\right), \\
\hat{\tau}_{\text{in transit}} &= \max\left(\hat{\tau}_{\text{sim}}, \hat{\tau}_{\text{vis}}\right), \text{with} \bigcup_{v \in V} I_v = I.
\end{aligned}$$

The distribution of independent tasks $(s, i) \in I$ from simulation nodes $s \in S$ to dedicated visualization nodes becomes an opportunity to address **(i) variability**. Running the collective tasks C on the generally significantly lower number of visualization nodes means concurrency will be lower, and so inefficiency due to lack of **(ii) scalability** will be reduced. However, in transit suffers from issues due to **(iii) overhead** and **(iv) rightsizing**. Regarding overhead, χ_s and χ_v needed to be introduced to account for communication costs. For rightsizing, if $\hat{\tau}_{\text{vis}} > \hat{\tau}_{\text{sim}}$, then the visualization nodes will block the simulation nodes (or else tasks must be dropped); if $\hat{\tau}_{\text{sim}} > \hat{\tau}_{\text{vis}}$, then visualization nodes will sit idle.

Hybrid In Situ

Hybrid in situ (Figure 5.1c) refers to using a mixture of inline and in transit techniques. With this work, we consider a specific form of hybrid in situ where simulation nodes perform all simulation work as well as some visualization work, while visualization nodes only do visualization. At the beginning of a mega-cycle, both simulation nodes and visualization nodes tackle visualization tasks. At some point, simulation nodes stop performing visualization tasks and resume the simulation, while visualization nodes concurrently process their visualization tasks.

$$\begin{aligned}
\hat{\tau}_{\text{sim}} &= \hat{\sigma} + \max_{s \in S} \left(\chi_s + \sum_{i \in I_s} \iota_{s,i}\right), \\
\hat{\tau}_{\text{vis}} &= \max_{v \in V} \left(\chi_v + \sum_{(s,i) \in I_v} \iota_{s,i} + \sum_{c \in C_v} \zeta_c\right), \\
\hat{\tau}_{\text{hybrid}} &= \max(\hat{\tau}_{\text{sim}}, \hat{\tau}_{\text{vis}}), \text{with} \bigcup_{v \in V} I_v \cup \bigcup_{s \in S} I_s = I.
\end{aligned}$$

An independent task $(s, i) \in I$ is only processed once (i.e., $I_a \cap I_b = \emptyset$ for any nodes a, b), but this can be done in different locations: either on the simulation node s that generated the respective data (including it in I_s), or on a visualization node v, transferring it to the independent task list I_v. This form of hybrid in situ creates opportunities for addressing all four inefficiencies:

Table 5.1: The Four Types of Inefficiency for In Situ Processing

	Inefficiency Type	Inline	In transit	Hybrid
(i)	Variability	–	✓	✓
(ii)	Scalability	–	✓	✓
(iii)	Overhead	✓	–	✓
(iv)	Rightsizing	✓	–	✓

The "–" signs indicate that the inefficiency afflicts a technique, while "✓" signs indicate the technique is able to address the inefficiency.

(i) **Variability.** The visualization nodes can take work from the most overloaded simulation nodes, reducing delays due to bottlenecking.
(ii) **Scalability.** Non-scalable tasks can be run on the dedicated visualization nodes, saving time.
(iii) **Overhead.** Transfer times can be overlapped with doing visualization work on the simulation nodes.
(iv) **Rightsizing**. The work distribution between simulation and visualization nodes is dynamically adapted.

Regarding **(iv) rightsizing**, if there are few visualization nodes, then they will receive only the work they can perform in their allotted time and the remainder can be performed on the simulation nodes. If there are more visualization nodes, then they can assume more work, and simulation nodes resume the simulation more quickly. In Figure 5.2, this is depicted by the broad valley in-between the two main failure modes "underwhelmed" and "overwhelmed." With "underwhelmed" visualization nodes, there is not enough rendering work. At the extreme, the "underwhelmed" category effectively becomes an in transit scenario where visualization nodes will sit idle at the end of each mega-cycle. With "overwhelmed" visualization nodes, there is too much rendering work. At the extreme, the "overwhelmed" category effectively becomes an inline scenario, with the visualization nodes unable to take on enough work to prevent variability on the simulation nodes. We hypothesize rightsizing viability falls in distinct categories based on the amount of resources to carry out this work (i.e., number of visualization nodes) and the amount of work to perform (e.g., number of images to render).

Table 5.1 summarizes the three in situ approaches with respect to the four possible inefficiencies. While inline and in transit can only address two of the inefficiencies each, our hybrid approach can address all of the inefficiencies.

Finally, this analysis makes two key assumptions: (1) that the visualization work consists of a series of atomic tasks that can be scheduled on either simulation or visualization nodes and (2) that the time to execute each task is known a priori in order to facilitate scheduling. With respect to image database generation, these assumptions hold, with

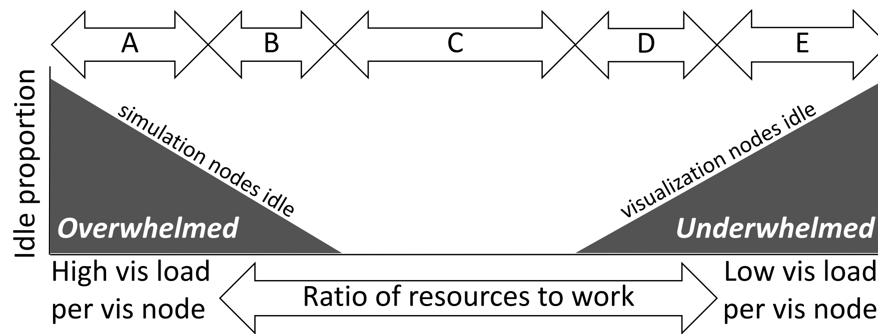


Figure 5.2: Notional organization of the viability of rightsizing into five categories (A, B, C, D, and E) based on the ratio of resources to work. Hybrid in situ should be more efficient than inline and in transit for categories B, C, and D, with C being the most ideal since it can eliminate idle time altogether.

the second assumption possible due to our performance probing approach.

In all, the main challenge for hybrid in situ is the distribution of independent tasks I and collective tasks C to simulation nodes S and visualization nodes V in a way that minimizes inefficiencies. Addressing this challenge is a main focal point of our proposed method.

5.1.2 Hybrid In Situ Method for Image Database Generation

This section describes our hybrid in situ method for generating Cinema-style image databases. An image database consists of a collection of n renderings, each corresponding to a different camera position in our case. Creating each image involves two types of operations: (1) rendering sub-images across nodes $s \in S$ (independent tasks I_s), and (2) compositing sub-images to the final result (a collective task C). This means that there are $|S| \cdot n$ independent tasks and n collective tasks in each mega-cycle.

Our hybrid in situ system presented in the next subsection addresses the involved inefficiencies. Compositing suffers from (ii) scalability inefficiency, which we reduce by performing it only with the generally significantly lower number of visualization nodes. To address (iii) overhead, the system overlaps data transfer and visualization work. Rendering suffers from (i) variability inefficiency, as the rendering cost heavily depends on the data generated by simulation node $s \in S$ and the camera configuration associated with task $i \in I$. This is addressed together with (iv) rightsizing by distributing rendering tasks I such that idle time across all nodes is minimized. For this, we first estimate how long rendering and compositing will take, and then schedule visualization work accordingly.

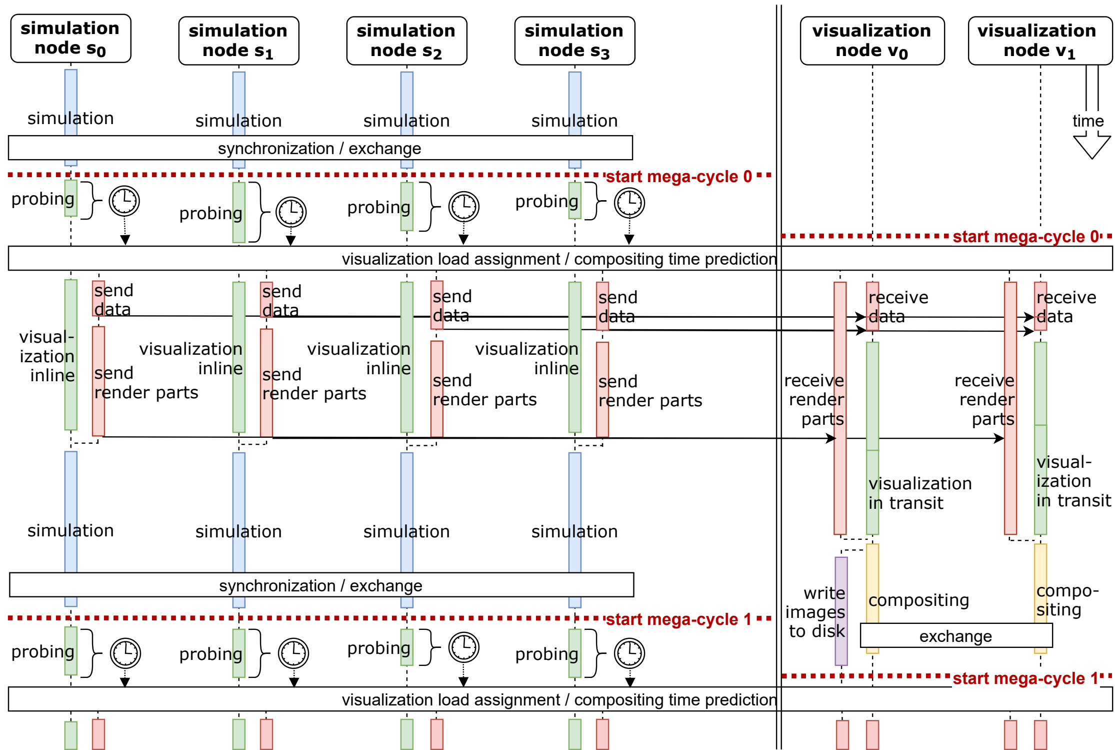


Figure 5.3: Sequence diagram of our system. Probing times are used as a basis to distribute the rendering load. Our main objective is to minimize the inline visualization time and the upper bound for time spent rendering on the visualization nodes is constrained by the combined time of rendering inline (including probing) and the simulation time before the next visualization step. Compositing is done on visualization nodes only, sub-images created inline or during probing are sent there.

System Overview

Figure 5.3 gives a sequential overview of our system. Although our approach is not limited to a specific rendering or compositing technique, we use volume raycasting (see subsection 2.2.3) and direct-send compositing in our system. A mega-cycle starts with visualizing the simulation results from the previous iteration. First, we create an estimation of induced cost on the simulation nodes $s \in S$. Probing is carried out and measures a subset of the render tasks $I'_s \subset I_s$ to estimate their cost. There is global synchronization between all nodes to exchange probing timings (this is the only instance of global synchronization in our approach). Compositing time is predicted using a simple performance model.

This provides the basis for visualization load assignment, which consists of two phases: First, each simulation node is assigned to one visualization node ($N : S \to V$), and

then the remaining render tasks $I_s^* = I_s \setminus I_s'$ are distributed between a simulation node s and its visualization node $N(s)$. Simulation data is accordingly distributed to the visualization node that took over respective rendering tasks. Likewise, all images produced on s are moved to its assigned $N(s)$. For faster transfer and smaller footprints we apply run-length encoding to the renderings being sent. The lossless compression is applied to pixel color values as well as the depth values. All image and data sending and receiving operations are asynchronous both on simulation and visualization nodes. This allows the system to effectively hide induced latency, i.e., simulation nodes can render while they are sending, and also visualization nodes can process tasks while they are receiving.

After rendering, simulation nodes immediately continue with the simulation. Visualization nodes v perform classic direct send compositing with images rendered by them as well as associated simulation nodes s (i.e, with $v = N(s)$). Since the images are composited sequentially, it allows us to compress and write them to disk concurrently.

Render and Compositing Time Predictions

Render Probing. At the beginning of each mega-cycle, all simulation nodes carry out probing rendering. For this, we randomly sample from all render tasks I_s of the respective nodes s in each mega-cycle to select probing samples $I_s' \subset I_s$. We then render and measure timings for these on s as the basis to predict the costs of the remaining tasks I_s^*. We use the arithmetic mean of the probing render times as a runtime estimate (per render) of the respective data partition. Typically, a random sampling of positions of an arcball-style camera provides a good coverage of the overall performance distribution (see subsection 3.1.4). During probing, we also detect if rendering can be skipped by checking whether the scalar value range in the data partition of s always yields opacity values below a threshold of 0.001 for the provided transfer function.

Compositing Time. For direct send compositing, cost can be estimated as a function of nodes participating [142, 119]. In our case, we specifically consider whether nodes produce images that contribute to the final results (i.e., which were not skipped via opacity checking):

$$1.05 \cdot (\alpha + \beta \cdot |V^*| \cdot \frac{|S^*|}{|S|} + \gamma \cdot |V^*|(1 - \frac{|S^*|}{|S|})). \tag{5.1}$$

Here, $|V^*|$ (with $V^* \subset V$) is the number of visualization nodes actively participating in compositing and α, β, γ are empirically determined constants on a target system. $|S|$ is the number of simulation nodes, while $|S^*|$ divided by the total number of simulation nodes represents the normalized amount of data set partitions that need to be visualized (i.e., the ratio between skipped and non-skipped partitions).

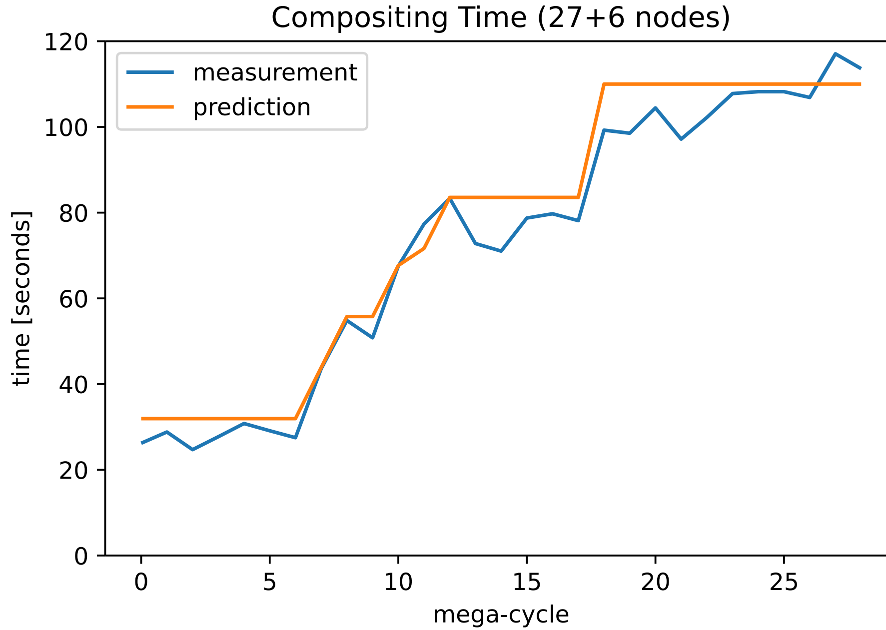


Figure 5.4: Compositing times compared to our estimate using 27 simulation nodes and 6 visualization nodes. Our prediction model depends on the number of nodes participating in visualization and uses initial measurements on the target systems.

Since we determine α, β, γ empirically, a few measurements of compositing times are needed when running our system on hardware using a different interconnect. We determine the constants using the measurements and a non-linear least squares function fit that resulted in a normalized RMSE of 8.83% for our compositing time estimation. We conservatively overestimate the time by 5% to avoid blocking of the simulation nodes. Figure 5.4 shows a comparison between the measured compositing time and our estimate using Equation 5.1 with $|S| = 27$ and $|V| = 6$.

Visualization Load Assignment

Node assignment $N : S \to V$**.** The estimated rendering time from probing for the remaining images $I^* = I_s \setminus I'_s$ provide the basis for assigning each simulation node $s \in S$ to one visualization node $v \in V$. For this, we iteratively assign the simulation node with the highest cost to the visualization node with the lowest accumulated cost until all nodes are distributed.

Rendering task assignment $A_s : I^*_s \to (s, N(s))$**.** Next, rendering tasks I^*_s remaining after probing on a simulation node s are scheduled to be either tackled by s or its assigned visualization node $N(s)$. Our main objective is to minimize the inline visualization time to maximize the time the simulation node can use for simulation.

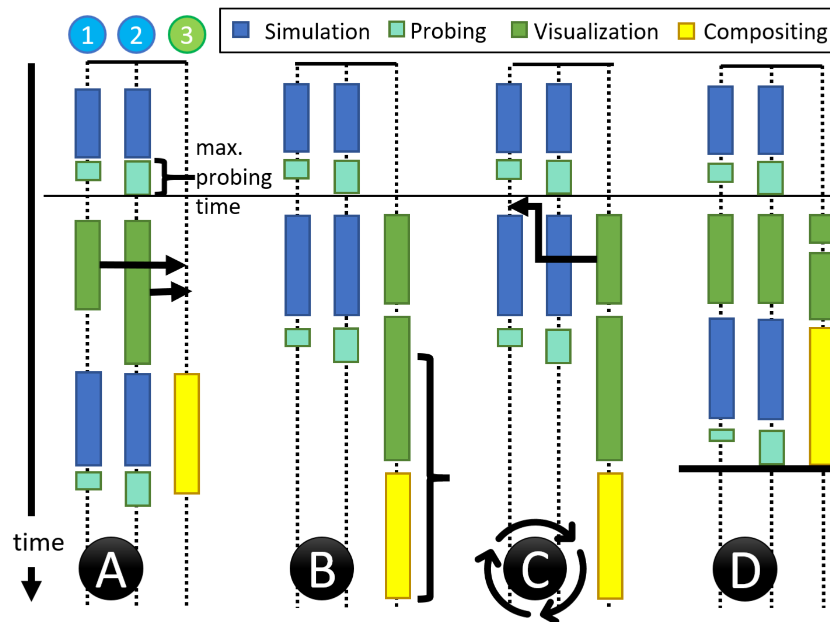


Figure 5.5: Our load balancing using two simulation nodes (1, 2) and one visualization node (3). Initially, all visualization load is assigned to the visualization resources (A). If the estimated runtime there exceeds the full cycle time on the simulation nodes (B), we gradually shift render load back to be processed by the (then) fastest simulation node (C) until runtimes are balanced out across all nodes (D).

Initially, we expect all rendering to be done on the visualization nodes (Figure 5.5A). However, when aiming to avoid idle times, there is an upper bound for time spent rendering and compositing on the visualization nodes (Figure 5.5B). It is constrained by the combined time of three parts: (inline) rendering time on simulation nodes (influenced by A_s), the simulation steps before the next visualization run on the simulation nodes, and the maximum probing time of the next visualization cycle. The probing time is approximated from the current mega-cycle. If the render time prediction combined with the compositing time prediction on the visualization nodes exceeds the time of those three parts, we gradually shift rendering load back to be processed inline on the simulation nodes, until the estimated time on the visualization load is below the bound. Thereby, we iteratively move a random rendering of the fastest simulation node from being processed on the visualization node back to be rendered inline, updating the respective time estimates in the process (Figure 5.5C). This balances the render load across simulation and visualization nodes (Figure 5.5D).

5.1.3 Implementation Details

As a basis for our hybrid in situ approach, we use the *Ascent* [116] framework that supports a wide range of data filters and several parallel scientific rendering algorithms using the *vtk-m* [143] toolkit. Furthermore, Ascent supports distributed memory and comes integrated with several simulation codes.

We use only CPUs for both simulation and visualization in our test. A future extension to GPUs in thinkable since they are supported by vtk-m. MPI is used to distribute tasks among nodes, running multiple MPI tasks per node is possible. All data sending and receiving operations are asynchronous both on simulation and visualization nodes. We use Open Multi-Processing (OpenMP) for shared memory parallelization of the simulation as well as the rendering tasks on the nodes. Ascent supports the generation of *Cinema* [16] image databases that we use in our approach for visualization. In our experiments, we generate volume renderings of the 3D scalar fields generated by the simulations. We perform volume rendering, this is done using an arcball-style camera, i.e., we choose camera positions on a sphere around the data set using pre-defined zoom levels and facing at the center of the dataset. For acceleration, we employ block-based ESS and ERT (subsection 2.2.3).

For our tests with volume rendering on regular grids, we use the *Cloverleaf3D* [82] simulation, a 3D Lagrangian-Eulerian hydrodynamics benchmark. Ascent comes with a Cloverleaf3D version featuring an integration of the framework. As a second real world example we use *Nyx* [18], a massively parallel code for cosmological hydrodynamics simulations. Nyx uses *AMReX* [197], a software framework for block structured adaptive mesh refinement (AMR). All frameworks as well as our modifications are open source.

Integration into the Simulation Codes

The integration of our system into the simulation codes is minimally invasive. Besides the integration of the Ascent in situ framework (i.e., mainly coupling of the simulation data), we split the MPI communicator into a simulation and a visualization group based on a user-defined split factor. The MPI ranks belonging to the simulation group proceed regularly with the simulation and start visualization once the in situ condition triggers, i.e., a user-defined time interval has passed or a number of simulation steps have been processed. This is done by calling Ascent with the simulation data of the respective rank, the simulation time that we use as an estimate of the time for the next simulation block, and a set of user-defined actions. The latter include the visualization setup: filter pipelines, the scenes to be rendered (e.g., volume rendering including transfer function, Cinema configuration, etc.) and the probing setup for our hybrid approach such as the split factor between simulation and visualization ranks and the amount of probing renders.

The visualization ranks wait for incoming data and perform visualization immediately after receiving data from the simulation ranks. In contrast to the simulation ranks, Ascent is called without a data set but the same configuration. A synchronization between the simulation and the visualization ranks only happens inside Ascent during the exchange of probing times.

Integration into Ascent

We extend the Ascent in situ framework by adding flexible in transit capabilities including a modified compositing that can work solely on visualization resources. Further, we extend Ascent to handle our hybrid approach. This primarily includes a probing step in which we distribute the rendering load across the MPI ranks. When starting our runtime, we first split the MPI communicator that is passed on by the simulation and includes all ranks into a simulation and a visualization communicator. For this, we use the same split factor as in the simulation code to exactly reproduce the split. A pseudo-random sequence is generated to determine the images used for probing, the same sequence is generated on all ranks.

All simulation ranks then proceed with the probing run, i.e. by rendering the random image subset. The resulting render times are then gathered on all MPI ranks to determine the load distribution and the assignment of the simulation ranks to the visualization ranks. Then, the simulation ranks asynchronously send their simulation data to their assigned visualization ranks, where the respective subset of images is rendered in transit on arrival of the data. Meanwhile, inline rendering is performed on the simulation ranks. This happens in batches to facilitate asynchronous sending of the render parts to the respective visualization ranks where compositing happens. After rendering, we apply run-length encoding to the pixel RGBA and depth values. After sending the last encoded render parts, the simulation ranks return to the simulation.

The visualization ranks receive the image parts of their assigned simulation ranks, decode them and use them with the ones rendered themselves for compositing. We use direct send compositing that is integrated in Ascent using the *DIY2* [144] library. Ascent uses direct send since it has to support the worst case scenario, i.e., unstructured meshes that fit together like puzzle pieces. Once the final image is composited, we compress it as a portable network graphics (PNG) and write it to disk on one of the visualization ranks. To process the writing in parallel, we use a producer-consumer approach. After the last image is written to disc, the visualization ranks return to the simulation process and proceed with the next visualization cycle.

5.1.4 Overview of Experiments

Our experiments are organized into three phases:

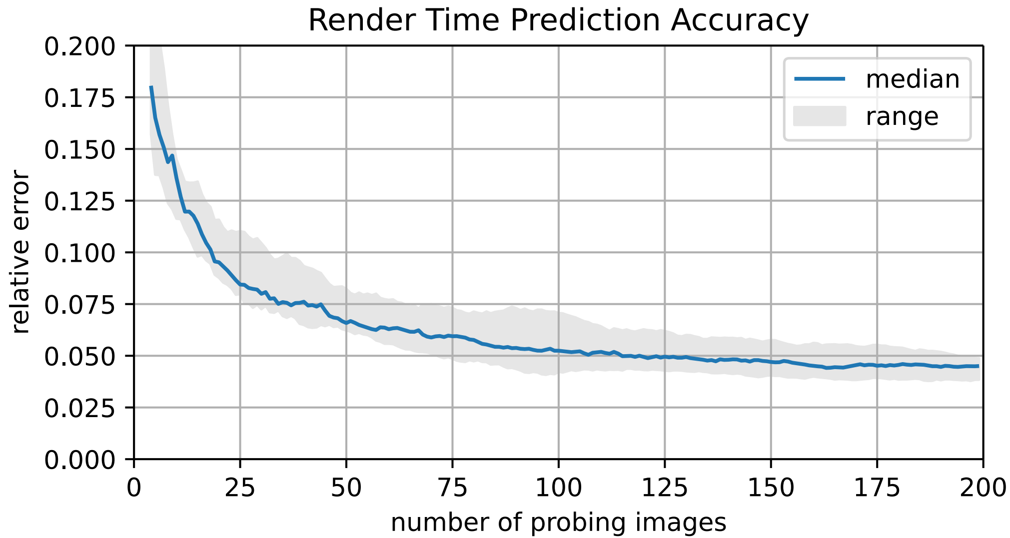


Figure 5.6: Render time estimation error depending on the number of probing images (total: 400). The data is based on twelve random sequences.

- **Baseline experiments**, which compare our hybrid method with inline and in transit approaches for a fixed configuration.
- **Parametric study**, which extends the comparisons from the baseline experiments to consider configurations with varying amounts of work (number of images in the image database and different image resolutions) and resources (number of visualization nodes).
- **Weak scaling study**, which evaluates how changes in concurrency affect cost.

The remainder of this section describes the details behind our experiments: probing, hardware, and workloads.

Probing: One important aspect to our technique is how much probing to perform. Prior to our experiments, we conducted an analysis to identify a good trade-off between performance and accuracy. This analysis used an example case with 400 renderings. To calculate the accuracy, we generated twelve random sequences of camera positions, and generated estimations using the sequence with the first $n \in [4, 200]$ probings (Figure 5.6). With this, we identified $15\,\%$ of the renderings to be a good compromise of scheduling flexibility versus accuracy (yielding an error of $6.3\% \pm 1.3\%$), and use this ratio in all experiments that generate 400 renderings.

Hardware: We ran all our experiments on the *Stampede2* supercomputer of the Texas Advanced Computing Center (TACC). We used Skylake nodes that feature Intel Xeon Platinum 8160 CPUs with 48 cores on two sockets (24 cores/socket). The CPUs support

two hardware threads per core, adding up to a total of 96 threads per node. Typically, we ran six MPI tasks per node with 16 OpenMP threads per task.

When comparing across inline, in transit, and hybrid, we fixed the number of simulation nodes and considered different numbers of visualization nodes. This is crucial for comparability as the number of simulation nodes impacts the domain decomposition of the simulation, which not only influences the simulation itself but also the rendering tasks.

Workloads: A workload consists of running a simulation code for some number of mega-cycles, as well as generating an image database for each mega-cycle. We determine the number of simulation steps in all mega-cycles by running the simulation in intervals of 120 seconds of wall clock time before invoking the in situ visualization. For the image databases, we generate a Cinema database of 400 volume renderings per visualization cycle using an orbital camera with regular spacing of the angles and a single zoom level. All images are rendered with a resolution of 800×800 pixels. Finally, we employed supersampling during the weak scaling phase.

We used two simulation codes throughout our study. The first two phases used *Cloverleaf3D* [82], a 3D Lagrangian-Eulerian hydrodynamics benchmark. The third phase used *Nyx* [18], a massively parallel code for cosmological hydrodynamics simulations as a real world example. We visualize the energy field for Cloverleaf3D (Figure 5.7) and the density field for Nyx (Figure 5.10). While Cloverleaf3D uses regular grids, Nyx uses block structured adaptive mesh refinement with *AMReX* [197].

The simulations also differ in how they evolve over time. The Cloverleaf3D simulation starts with two initial energy fields in opposite corners of its domain, and these two energy fields extend over the course of the simulation until they visually fill the whole domain. The volume rendering's transfer function treats low energy regions as fully transparent, resulting in imbalanced work from empty space skipping. As the simulation continues this effect fades, but rendering imbalances emerge due to early ray termination. With Nyx, the simulation starts with an initial random seed of dark matter particles distributed across the whole domain. Over the course of the simulation, the particles attract each other to form clusters, creating empty spaces as a side effect. As a result, data blocks become less active over time.

5.1.5 Results

Our results are organized into baseline experiments, parametric study, and weak scaling experiments.

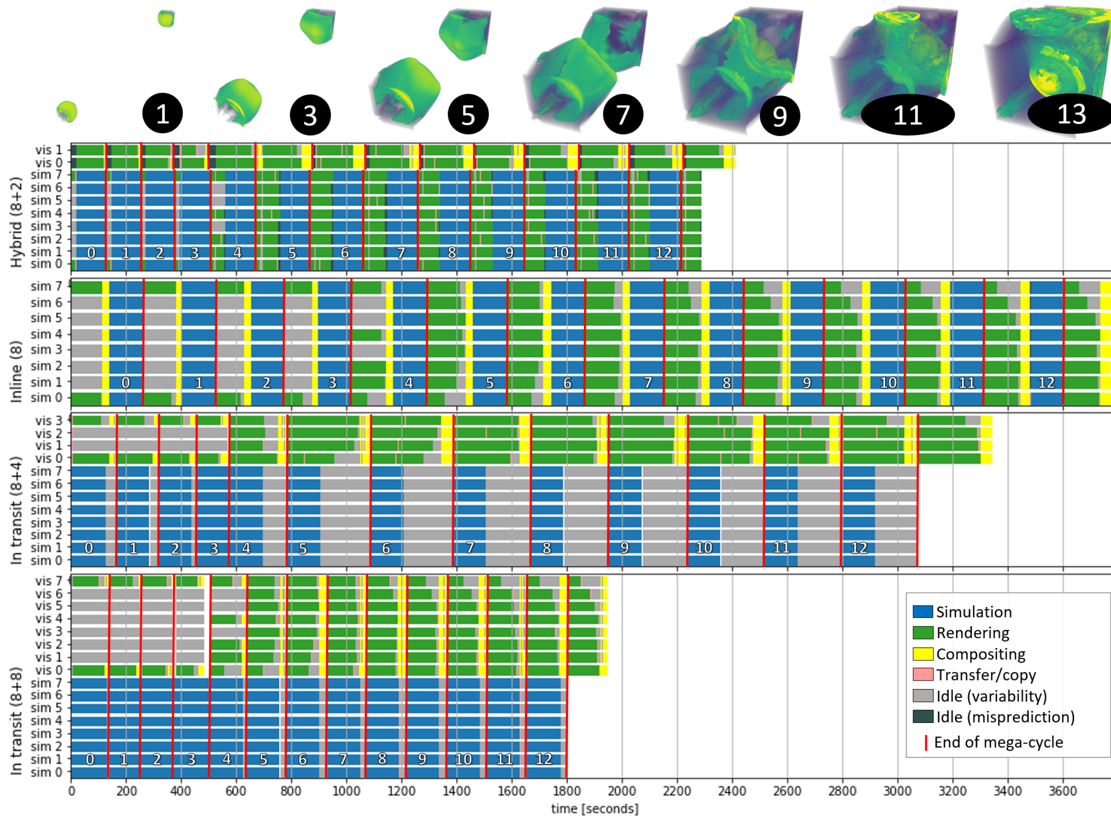


Figure 5.7: Baseline experiment of 14 mega-cycles of Cloverleaf3D simulation and Cinema database generation. Images at the top show volume renderings from odd mega-cycles. The remaining rows show Gantt charts for inline, hybrid, and in transit configurations. The charts are colored to indicate the activity being performed on a given node, including two types of idle. Light gray indicates idle caused by variability while dark gray shows idle times caused by prediction errors. Although in transit (8+8) has a shorter completion time, hybrid is more efficient using less node seconds overall.

Baseline Experiments

Our baseline experiment considered a single workload. For simulation, we ran Cloverleaf3D for 14 mega-cycles on eight simulation nodes, with a grid resolution of 384^3 (192^3 per node). For image database, we generated 400 images each mega-cycle. We compared four in situ approaches: inline, our hybrid configuration receiving two additional visualization ranks (ten total) and two different in transit configurations receiving four and eight additional visualization ranks (i.e., twelve and 16 total). Figure 5.7 shows volume renderings from these experiments, as well as Gantt charts for each in situ approach.

Our hybrid technique was the most efficient approach since it took the least node seconds to complete the 14 mega-cycles. It completed the simulation and visualization tasks in $2408\,\mathrm{s}$ using ten nodes (24.1K node-seconds), while the inline configuration took $3804\,\mathrm{s}$ using 8 nodes (30.4K node-seconds — $26.2\,\%$ more), and the in transit configuration took $3342\,\mathrm{s}$ with 12 nodes (40.1K node-seconds — $66.5\,\%$ more) and $1940\,\mathrm{s}$ with 16 nodes (31.0K node-seconds — $28.9\,\%$ more). The flexibility of hybrid enabled it to do better in terms of the four types of in situ inefficiency (see subsection 5.1.1), despite introducing **(iii) overhead** for transfer (pink) in comparison to inline, and also exhibiting some **(i) variability** issues due to sub-optimal work assignments (black).

For inline, the effects from **(i) variability** can be seen in the high proportion of idle time (gray) in simulation ranks 1 through 6 while simulation ranks 0 and 7 are rendering (green). These effects are significant from mega-cycles 0 through 4, with only two corners of the volume actually contributing to the volume rendering. As the simulation evolves, this improves when all nodes engage in rendering work in mega-cycles 5 through 8, but inefficiencies re-emerge in mega-cycles 9 through 13 due to ERT (node 7 in particular). Our hybrid technique addresses variability by adapting the assignments to visualization ranks accordingly. Inline also demonstrates **(ii) scalability** issues during the compositing phase (yellow). Compositing costs were 160 node-seconds per mega-cycle with 8 simulation ranks participating for about $20\,\mathrm{s}$, while hybrid only required 100 node-seconds with just 2 nodes being involved for $50\,\mathrm{s}$.

Both in transit configurations perform quite poorly. Hybrid reduces **(iii) overhead** inefficiencies in comparison to in transit both by overlapping data transfer with visualization work and only triggering it when simulation nodes cannot process all visualization work themselves. However, for both in transit cases, the major issue is **(iv) rightsizing** — one configuration has too few visualization nodes and the other has too many. With too few visualization nodes (four visualization nodes, twelve total), the rendering tasks cannot be completed in time, blocking the simulation nodes. This results in significant idle time (gray) on the simulation nodes, especially after mega-cycle 4. With too many visualization nodes (eight visualization nodes, 16 total), there are not enough rendering tasks to occupy the visualization nodes. This again results in signifi-

cant idle time, although this time on the visualization nodes and before mega-cycle 4. Together, these two configurations demonstrate the difficulty in rightsizing in transit resources: whether too few or too many visualization resources, the result is idle time. In contrast, hybrid achieves rightsizing over a variety of visualization workloads by dynamically assigning render tasks. In early mega-cycles, the visualization ranks can almost exclusively handle the rendering tasks, allowing simulation ranks to focus on the simulation. When the cost of rendering tasks increase (around mega-cycle 5), work is shared between visualization and simulation ranks such that they complete their respective tasks right as the mega-cycle ends.

Although our hybrid approach was able to improve efficiency overall, this experiment shows fundamental limitations that it cannot improve on. Specifically, the variability is extreme in mega-cycle 0: simulation nodes #0 and #7 have all the inline visualization work and the others have none. Between this level of imbalance, the ratio between visualization nodes and simulations, and the amount of rendering work compared to the length of the mega-cycle, there is no way to schedule tasks that will fully prevent idle on simulation nodes #1–#6.

Parametric Study

The next phase is split into two parts and extends the baseline experiment to compare different combinations of workload (i.e., varying image count and varying image resolution) and resources (i.e., number of visualization nodes). The first part of this phase consisted of 36 experiments, as a cross product of four image database sizes (81, 144, 256, and 400 images) and nine in situ configurations. Eight of the in situ configurations came from varying the number of visualization nodes (1, 2, 4, and 8) for both hybrid and in transit. The ninth configuration was running inline. As in the baseline experiment, each configuration ran Cloverleaf3D with eight simulation nodes (four of the 36 experiments in this phase appeared in the baseline experiments). The second part consisted of 27 similar experiments, but this time the image resolution was changed instead of the image count (which remained at 400 images). Three different resolutions were used (800^2px, 1100^2px, and 1400^2px), where the latter two roughly correspond to doubling, respective tripling, the pixel count of the 800^2 default configuration.

Regarding the first part, Figure 5.8 compares the efficiency of our hybrid approach with inline and in transit in a 4×4 matrix. The lower left of this matrix has the least work per visualization node (81 images and 8 visualization nodes). We refer to this as “underwhelmed” (see Figure 5.2). The upper right of this matrix is where there is the most work per visualization node (400 images and one visualization node). We refer to this as “overwhelmed.” In terms of results, our approach was the most efficient approach (i.e., fewest node-seconds) in eight of 16 configurations, and each of these eight tended towards “overwhelmed.” The other eight tended towards “underwhelmed.”

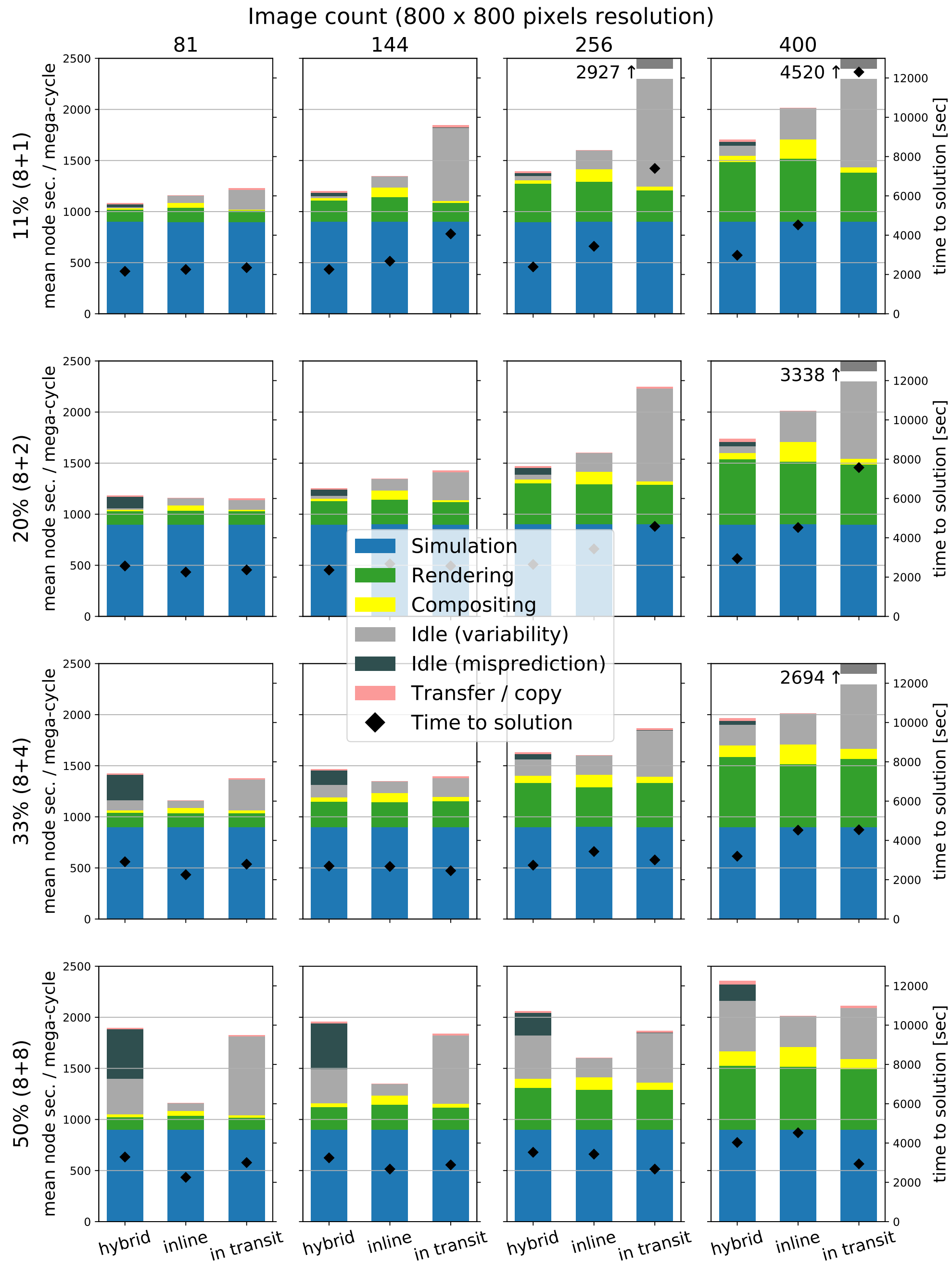


Figure 5.8: Columns in the 4×4 matrix correspond to the number of images rendered, while the rows correspond to the number of visualization nodes. Each of the 16 stacked bar charts compares our hybrid method with inline and in transit. The colors correspond to different activities, the heights for each color indicates how much time was spent (on average) per mega-cycle (left y-axis). Broken bars indicate higher y-values. Black diamonds indicate the total time-to-solution (right y-axis). The inline configuration does not use visualization nodes, a single inline run is repeated along each column.

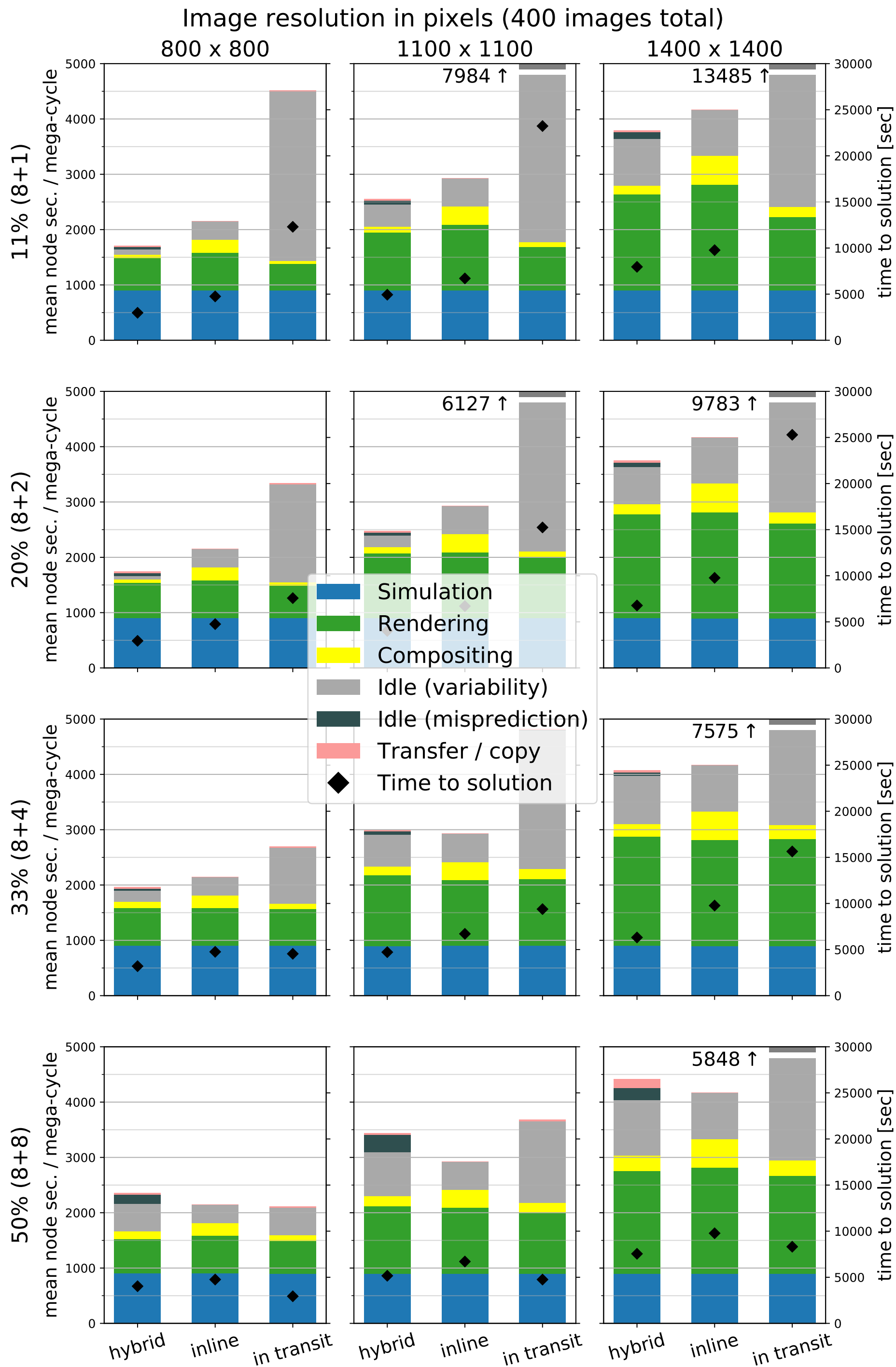


Figure 5.9: Columns in the 4×4 matrix correspond to the images resolution (400 images total per run), while the rows correspond to the number of visualization nodes. Otherwise, the properties are similar to Figure 5.8

Of our eight “underwhelmed” (losing) configurations, the best possible schedule would assign rendering work to the visualization nodes. That said, if there is not enough work per node, then visualization nodes sitting idle is inevitable—**(iv) rightsizing** is not possible. Worse, our probing step always occurs on the simulation nodes, meaning that our assignments (including the rendering done with probing) is even worse than a pure in transit approach; this is why in transit is faster for “81 images and 8 visualization nodes” (among others).

Conversely, being “overwhelmed” did not affect our performance. At the extreme, being overwhelmed would become like a pure inline situation, and thus be subject to **(i) variability**. While this does occur in practice (see discussion in the previous subsection on unavoidable idle from variability at mega-cycle 0), the effect is small enough compared to savings that our method is still the most efficient. Finally, some of these configurations show our technique’s flexibility in rightsizing. For example, our hybrid technique took approximately the same amount of time to render 256 images whether it was assigned one visualization node or two visualization nodes. This is because our algorithm was able to adapt the assignments to do more work on the simulation nodes when there was one visualization node and more work on the visualization nodes where there were two. The cost is the same across the two experiments because both have the same savings on scalability and variability, overheads do not increase, and (critically) rightsizing is maintained for both.

Our second experiment series demonstrates the impact of higher render loads when when using a larger image resolutions that might be desirable in some visualization scenarios, e.g., when fine detail are important. The results are shown in Figure 5.9, the columns represent the different resolutions while the rows again correspond to the number of visualization nodes. As in our first set of experiments examining a varying number of images, the inline case was run only once per configuration since it does not use dedicated visualization resources. When compared, both experiment series show similar results, while the distance of our hybrid technique to inline even increases for higher resolutions in the “overwhelmed” cases (upper half).

In all, both series demonstrate that our method yields rightsizing in fairly wide ranges of configurations, while in transit is only able to achieve this in the rare case when all conditions align.

Finally, we consider the time-to-solution in Figure 5.8 and Figure 5.9 (diamond shapes), i.e., the wall-clock time it takes to finish all 14 mega-cycles using the respective configuration. Generally, our hybrid approach finishes first in cases in the “overwhelmed” category (upper right triangle in Figure 5.8 respective upper half in Figure 5.9), while in transit typically finishes faster in the clearly “underwhelmed” cases. Inline is often the slowest to finish, except for cases with high render load and low visualization resources where in transit performs worse. This can partly be attributed to the fact that these are

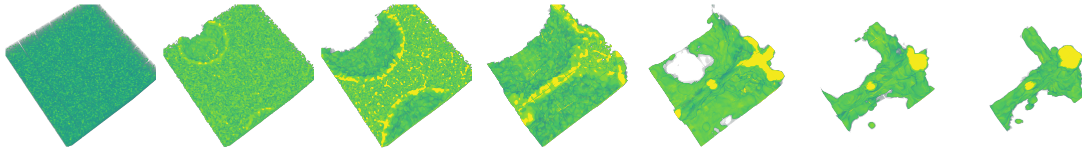


Figure 5.10: Renderings of the Nyx simulation data.

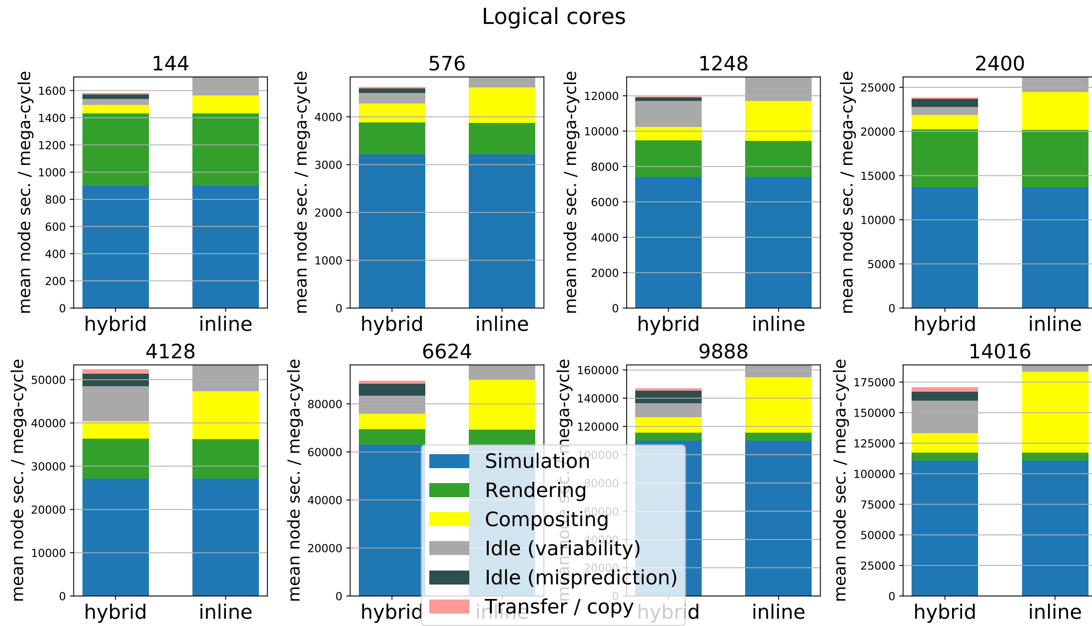


Figure 5.11: Nyx weak-scaling results. For inline, compositing times grow with higher core count, while rendering gets relatively faster. For hybrid, overhead and render times grow, as well as idle times caused by prediction errors. In transit runs are not considered since their results are substantially worse than hybrid or inline due to the rightsizing inefficiency.

absolute numbers and we use the least resources for inline (no additional visualization nodes).

Weak Scaling

As our last experiment, we consider a weak scaling study with the Nyx simulation code, comparing our hybrid approach against pure inline. We aimed for a visualization to simulation resource ratio of $f = \frac{|V|}{|S|} = 0.2$ and used a data partition size of 32^3 voxels per simulation rank. The randomly seeded dark matter particle count was adapted accordingly. We ran Nyx with eight different node configurations on up to $14\,016$ logical cores. Configuration details are listed in Table 5.2. The numbers result from

Table 5.2: Node Configurations for the Nyx Simulation on Stampede2

Nodes	Logical cores	Sim. ranks	Vis. ranks	Factor	Grid size	Supersampling
2	144	8	1	0.125	64^3	1×1
6	576	27	9	0.333	96^3	1×1
13	1248	64	14	0.219	128^3	2×2
25	2400	125	25	0.200	160^3	3×3
43	4128	216	42	0.194	192^3	4×4
69	6624	343	71	0.207	224^3	4×4
103	9888	512	106	0.207	256^3	4×4
146	14016	729	147	0.202	288^3	4×4

the constraints that the MPI task count is divisible by six (to fully load the nodes) and a simulation task count whose cube root is an integer for even partitioning. We increased supersampling in the volume raycaster for higher node counts to balance out the render load decrease at higher concurrency that is caused by our constant image resolution. The run was interrupted after 60 minutes of execution (hybrid) respective 80 minutes (inline). As discussed above, we allowed the inline case more processing time to get a similar number of full cycles as in the hybrid case.

The performance summaries plotted in Figure 5.11 show that the four types of inefficiency change as concurrency increases. The most obvious effect is with scalability. The inline approach devotes more and more time to compositing (yellow color) due to poor scalability, while our hybrid approach is able to reduce compositing time significantly. That said, as can be seen in Figure 5.11, the compositing time increases with concurrency for our hybrid technique as well, as the number of visualization nodes increases proportionally and begin to exhibit their own scalability inefficiency. Another notable effect is with overhead, which can be seen in the increasing transfer/copy times (pink) for our hybrid approach. However, overlapping copy and transfer with rendering, combined with run-length encoding counters much of the visible cost, keeping overhead times fairly low even at larger scales.

With respect to rightsizing, our algorithm was able to make “rightsized” assignments for visualization and simulation work (i.e., all tasks should finish a mega-cycle at the same time), but these assignments did sometimes lead to idle resources because of mispredictions (black color). That said, the amount of misprediction does not appear to change significantly as concurrency increases.

Finally, variability effects (gray color) again do not appear to change significantly as concurrency increases for our hybrid method (with the exception of the largest run, which we attribute to the simulation output), although they improve for inline visualization. This change is because rendering times are getting smaller, since each

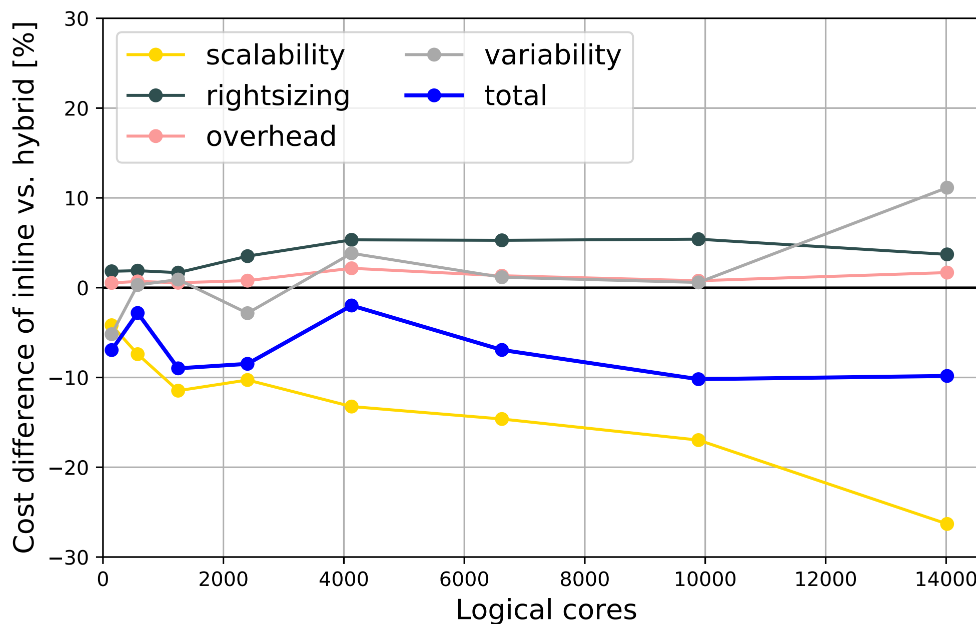


Figure 5.12: Differences in cost between hybrid and inline as a function of concurrency. The differences are calculated relative to the inline run. For example, the total cost for the 2400 core inline experiment was 26077 seconds, of which 4317 s were spent compositing. The corresponding hybrid run took 1636 s for compositing, representing a savings of 2681 s, which is 10.3% of the total inline run. In turn, the yellow (scalability) curve has a point at (2400, -10.3%). The overhead curve (pink) considers both transfer costs and reduced performance in rendering due to the overlap with transfers.

block contributes fewer fragments. In actuality, variability is increasing relative to render time, but render time drops sharply enough that this increase is insignificant.

In terms of actual savings, our hybrid approach had lower cost for all concurrencies, although these savings varied (144 cores: 6.9%, 576: 2.8%, 1248, 9.0%, 2400: 8.5%, 4128: 2.0%, 6624: 6.9%, 9888: 10.2%, 14016: 9.8%). While the savings are fairly consistent, the factors behind them are changing. These changes can be seen in Figure 5.12. As concurrency increases, scalability savings are growing fast enough to offset additional overhead costs and reduced savings in variability.

5.1.6 Future Directions

Although the actual savings in cost may appear to be modest (on the order of 7%), this research has a chance to be very impactful—small speedups for ubiquitous operations on expensive devices add up to large impact overall. With the success of the Cinema project, image database generation has a high chance of becoming a ubiquitous task. In

all, this approach should be strongly considered for in situ image database generation going forward. That said, the design is somewhat complex, and will need to be hidden behind production software, such as done in this work using Ascent.

In terms of future work, the hybrid approach should generalize to other analysis tasks that can be split into fine granular sub-tasks for dynamic distribution. Further refinements could possibly improving the accuracy of cost predictions via importance sampling for probing and through advanced compositing and simulation time predictions.

5.2 Performance Prediction to Support Render Hardware Acquisition

For HPC environments in general and small sized clusters in particular, a performance estimation prior to hardware acquisition may be crucial to meet performance requirements and at the same time use a limited budget. We propose a neural network-based approach to predict the runtime performance of a distributed GPU-based volume renderer [14]. Using timing measurements of a single cluster and individual GPUs, the performance gain of upgrading or extending a cluster's graphics hardware can be predicted. Using the model, a performance prediction of upgrading the whole cluster but keeping the network configuration is also possible.

Formally, the main objective of the approach is to predict the total render time of a frame T_{cluster} based on cluster size C, data set D, the node's hardware H, image resolution I, and view parameters V:

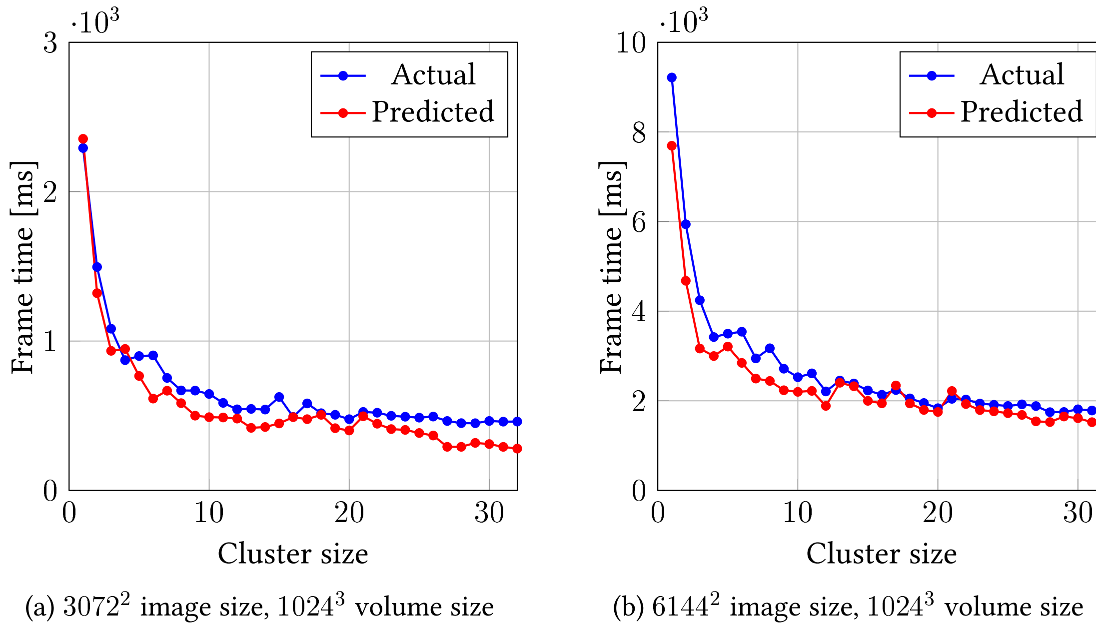
$$(C, D, H, I, V) \rightarrow T_{\text{cluster}}. \tag{5.2}$$

Typically, prediction accuracy of neural networks improves with a larger amount of training data, which keeps the model general and avoids a possible bias. However, different GPU clusters are rare and exchanging all GPUs of an operating cluster unfeasible. Therefore, the initial model has to be split into two levels aligned with the two phases of an object-space distributed volume renderer: local rendering (resulting in local render time T_{local}) and compositing. Equation 5.2 can be split accordingly:

$$(D, H, I, V) \rightarrow T_{\text{local}}, \tag{5.3}$$

$$(I, C, T_{\text{local}}) \rightarrow T_{\text{cluster}}. \tag{5.4}$$

In Equation 5.3, data set D, hardware H, image resolution I, and view parameters V define a local render time T_{local}. Equation 5.4 models the compositing phase, mapping cluster size C, image resolution I, and local render time T_{local} to the final cluster frame time T_{cluster}. Reformulating Equation 5.2 this way has the advantage that Equation 5.4 does not contain information about the employed rendering hardware. This allows to



(a) 3072^2 image size, 1024^3 volume size

(b) 6144^2 image size, 1024^3 volume size

Figure 5.13: Evaluation of the performance prediction model in the cluster upgrade scenario: training is done using data in single GPU mode, predicted is performance of a dual GPU cluster.

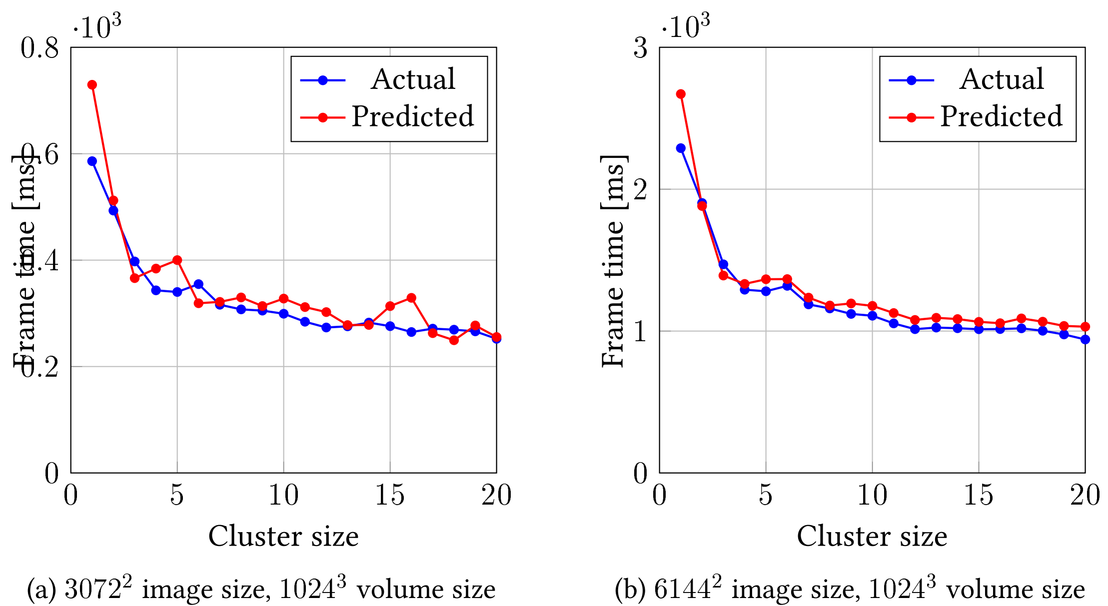
emulate different rendering times on single nodes by stalling local execution time T_{local}, effectively generating more measurement data on a single cluster. This data is used to train the neural network that predicts Equation 5.4. The model eventually captures performance characteristics of hardware for compositing, network, and topology. This means that by using this model meaningful predictions for a cluster on the basis of local render time measurements from one node equipped with the target hardware can be made.

The neural network contains an input layer for the input features I, C, T_{local} and T_{cluster}. Further, the experimentally determined inner structure for the network contains two hidden layers, consisting of 16 and eight neurons respectively. Finally, the rectified linear unit (ReLU) is chosen as activation function for faster training.

The model is evaluated using two different scenarios:

- Predicting the impact of a GPU upgrade in a cluster on volume rendering performance.
- Investigating how accurately the model can predict rendering performance across multiple clusters.

For the first scenario, a 33 node cluster consisting of nodes equipped with two Intel



(a) 3072^2 image size, 1024^3 volume size

(b) 6144^2 image size, 1024^3 volume size

Figure 5.14: Evaluation of the performance prediction model to predict the performance of a different cluster. Training is done using data from one cluster, predicted is the performance of a different cluster with a similar network configuration.

Xeon E5620 CPUs, 24 GB RAM, two NVIDIA GeForce GTX 480 GPUs (see Table 2.1), and a double data rate (DDR) InfiniBand are used. The upgrade is emulated by deactivating one of the GPUs per node and only using data from single GPU mode for training. Testing the renderer in dual GPU mode enables a comparison between prediction and actual performance. Figure 5.13 shows the results for two different configurations. As can be seen, the model is able to accurately predict performance for different cluster sizes (R^2 score of 0.95).

For the second scenario, training is performed on the 33 node cluster with both single and dual GPU modes. For testing, a second cluster consisting of 20 nodes, each equipped with two Intel Xeon E5-2640 v3 CPUs, 128 GB RAM, an NVIDIA Quadro M6000 (see Table 2.1), and a fourteen data rate (FDR) InfiniBand was used. The results for different configurations are depicted in Figure 5.14. The model is able to perform well in this scenario (R^2 score of 0.93), too. However, in this case the graphs show a small bias (prediction of slower performance) that can be attributed to the different network interconnects used.

Overall, by using our model, accurate performance predictions can be made for the upgrade of GPUs in small clusters as well as upgrading to a completely new cluster with a comparable network configuration.

5.3 Adaptive Encoder Settings for Interactive Remote Visualization on High-Resolution Displays

Pixel streaming is a convenient way to display large amounts of data on remote locations, e.g. to monitor simulations on supercomputers. A typical approach to pixel streaming for large scale simulation data is to render the visualization directly on the cluster where the simulation is running and send encoded images to a client for display since this is typically less data than sending the whole raw data. Another possible usage of pixel streaming is collaborative visualization where domain experts analyze the data in multiple locations at the same time. However, maintaining a low latency for interactive explorability is still a major challenge for remote rendering in general and high-resolution setups in particular. A typical approach to keep the latency low is to send less data, e.g. by using a higher compression rate on the sent images which usually comes at the cost of a lower quality. We propose an approach to dynamically adapt encoder settings on a per image-tile-basis to optimize the quality for a given bandwidth.

Our adaptive remote visualization algorithm operates in four stages:

1. Capturing the last rendered frame and splitting it into equally sized tiles.
2. Predicting the compressed tile sizes and qualities.
3. Optimization of the encoding settings.
4. Encoding and sending of the tiles.

We start by converting the last rendered frame to grayscale since our predictions works on a single channel only. Our specific setup, which we used for development and testing, features a powerwall that uses ten display nodes with a resolution of 4096×1200 pixels each, to display the visualization. We split the image per node into 40 tiles, resulting in a total of 400 tiles.

We use three different encoder settings that differ in quality parameter and average bitrate: LOW (0.65 Mbit), MEDIUM (7.74 Mbit), and HIGH (17.42 Mbit). Our objective is to find an optimal encoding setting for each tile based on the quality and size. We use a convolutional neural network (CNN) to predict the size and quality of an image tile for each encoder setting. The output of our network is a vector containing six values, one for the expected quality and one for the expected size after encoding with each of the three settings. We combine the those predictions with an optimizer to decide on the encoder setting for each tile. We use the SSIM to assess the image quality of the encoding compared to the original.

The architecture of our CNN is shown in Figure 5.15. We use two consecutive combinations of convolution, ReLU activation, and max-pooling each and end the network with a dropout and dense layer to get our desired resulting vector of six values. For training of the CNN, we used more than 22 000 images of volume renderings that we encoded with our three quality setups. Evaluation was done using 5632 images.

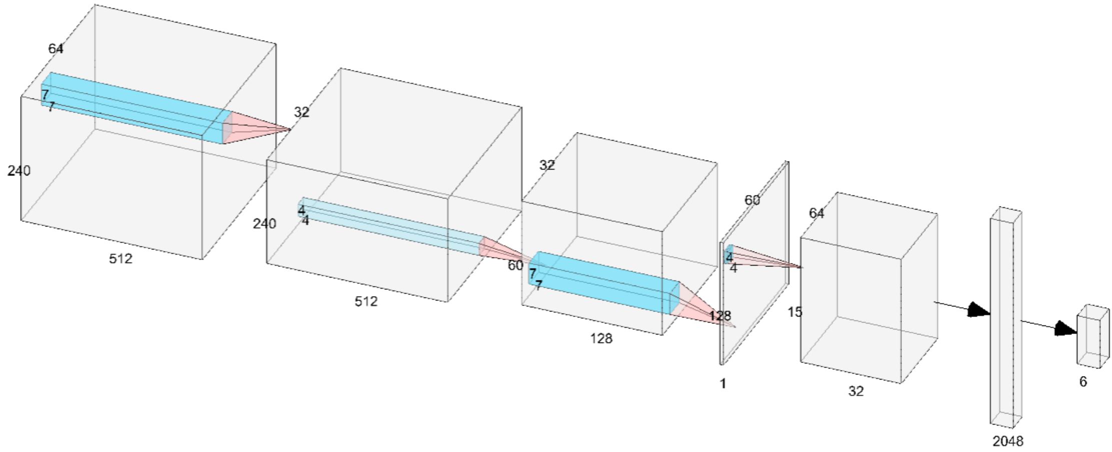


Figure 5.15: CNN architecture used for prediction of compressed tile size and quality.

Our goal is to optimize the quality of all tiles given a size threshold T. In other words, we want the highest possible encoding setting per tile while keeping the overall size of the tiles below the threshold. Since this is equivalent to the multiple-choice knapsack problem and can be optimally solved by minimizing the objective function:

$$\min \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_{ij} \cdot (1 - \text{SSIM}_{ij})^2 \tag{5.5}$$

Here, N is the number of tiles and M the number of encoding seconds. The first constrained arises from the fact that we can take exactly one encoder setting for each image tile: $\forall j \in N : \sum_{i=0}^{M-1} x_{ij} = 1, x \in \{0, 1\}$. The second constraint restricts the overall size of all tiles to be less or equal to the given threshold: $\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_{ij} \cdot \text{size}_{ij} \leq T$. For faster processing, we use a greedy approximation.

We tested our approach using by transferring two scientific visualization streams between two workstations equipped with an Intel Xeon E5-2640 v3, 256 GB of RAM and an NVIDIA Quadro M6000 GPU. The resolution was set to 4096×1200 pixels and the maximum bandwidth target to 100 Mbit/s. One of the streams is a 40 seconds long particle simulation of a laser ablation, the other a 30 seconds long interaction with the Chameleon volume data set (see Table 2.2). Table 5.3 shows an overview of average and maximum throughput achieved using our technique in comparison to encoding the all frame with the three encoder settings. On average, capturing, subdivision, converting, and predicting the encoder setting takes about 50 milliseconds. For the particle data sequence, our algorithm produced on average an encoding quality distribution for all tiles of 29% LOW, 69% MEDIUM, and 2% HIGH. Regarding accuracy of our prediction, the error rate is about 5% for all six values in our result vector (quality and size for

Table 5.3: Throughput of our Adaptive Encoding in Comparison to Flat Encodings.

Test	LOW	MEDIUM	HIGH	Ours
Particle (avg)	29.19	86.63	310.92	26.35
Particle (max)	46.13	133.98	544.55	42.21
Volume (avg)	4.00	13.90	143.23	10.83
Volume (max)	6.49	22.18	255.94	21.72

each encoder setting). The accuracy is acceptable with respect to the required real-time capability.

With our technique, we can achieve a good overall quality while preserving details in areas of interest in both tested scenarios. For the particle rendering sequence, our algorithm uses the lowest average and maximum throughput while delivering a slightly better overall quality than the MEDIUM setting. In contrast, using only the MEDIUM setting produces three times the throughput of our technique on average, while producing a similar output. The difference in throughput stems from the fact that our algorithm chooses the LOW encoding setting for some tiles with less visual complexity in order to save bandwidth. This allows it to use the HIGH setting for tiles with fine structures to improve the overall quality of the final image. Finally, only the LOW setting and our algorithm stay below our target of 100 Mbits/s during the tests.

Chapter 6

Foveated Rendering to Improve Application Performance

Modern output devices are steadily increasing in pixel density and refresh rate. While this trend improves the experience for users of large projection screens and head-mounted devices, the rendering performance of image order approaches such as volume raycasting (see subsection 2.2.3) is heavily impacted by the associated performance requirements (section 3.1). For example, a higher screen resolution typically requires a denser sampling of the image plane that can only partly be accounted for by advances in graphics hardware. In this chapter, two approaches are presented that use foveated rendering to improve performance of scientific visualization applications. For this, the user's gaze needs to be tracked by the compute device and perceptual characteristics of the HVS used to improve performance.

The first approach (section 6.1) is concerned with the specifics of volume raycasting, where the widely used acceleration techniques ESS and ERT are complemented by adapting the sampling pattern in different areas of an image depending on the observer's fixation [8]. The user's gaze is measured using an eye-tracking system to determine the areas of the image that are in foveal vision where humans perceive sharp, colorful details; and areas in peripheral vision, where users perceive less details and colors. The second approach (section 6.2) deals with foveated encoding on large high-resolution displays in remote rendering scenarios, with the goal of keeping the throughput continuously below a bandwidth limit [10]. The gaze of multiple users is determined by means of head tracking and the encoding quality adapted depending on the users' fixations: higher quality for foveal regions on the display and lower quality for peripheral ones.

This chapter is partly based on these publications

- V. Bruder, C. Schulz, R. Bauer, S. Frey, D. Weiskopf, and T. Ertl. "Voronoi-Based Foveated Volume Rendering". In: *Proceedings of EuroVis (Short Papers)*. The Eurographics Association, 2019, pp. 67–71 [8]
- F. Frieß, M. Braun, V. Bruder, S. Frey, G. Reina, and T. Ertl. "Foveated Encoding for Large High-Resolution Displays". In: *IEEE Transactions on Visualization and Computer Graphics* 27.2 (Feb. 2021), pp. 1850–1859 [10]

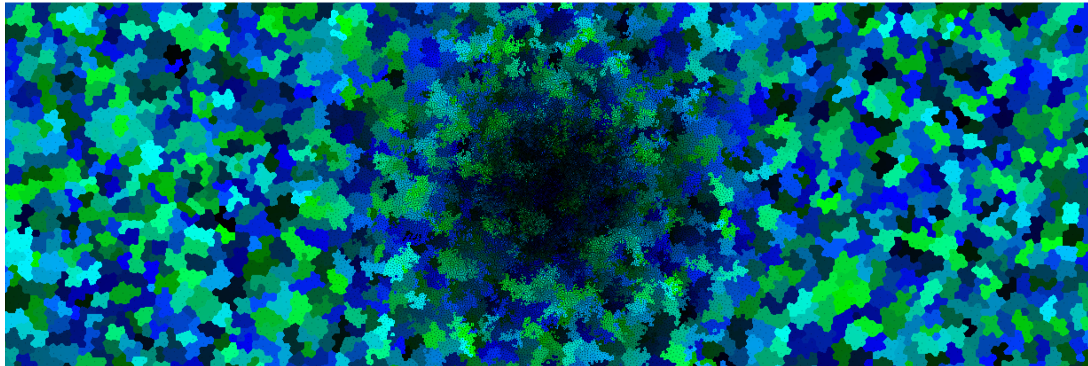
6.1 Voronoi-Based Foveated Volume Rendering

In this work, we focus on the specifics of foveated volume raycasting. The contribution can be broken down into the modeling, realization, and evaluation of a foveated volume rendering system: We pre-compute a sampling mask based on visual acuity fall-off using the Linde-Buzo-Gray (LBG) algorithm, shift this mask according to user gaze, reconstruct the image based on Voronoi cells using natural neighbor interpolation, and apply temporal smoothing to make undersampling artifacts less disturbing.

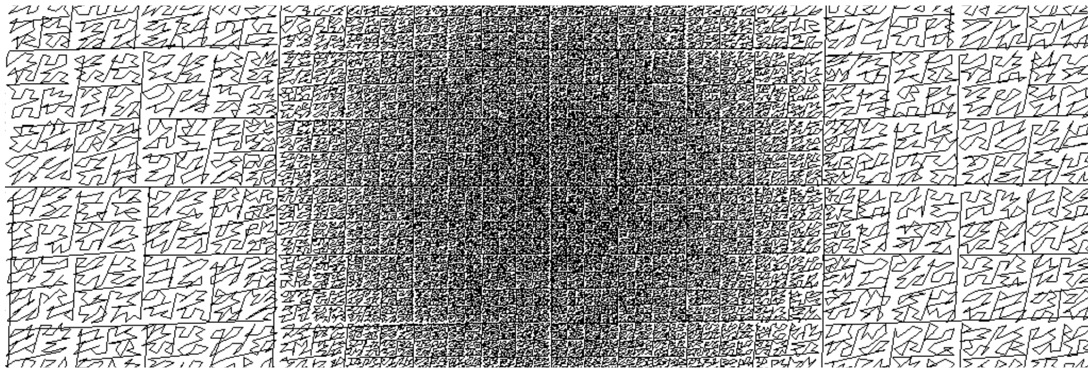
The acuity fall-off of the HVS can be modeled using a hyperbolic function (see subsection 2.1.3). We approximate this function using a 2D Gaussian that depends on screen resolution, size, viewing distance, and estimated photo-receptor distribution in the human eye. Our photo-receptor topology estimation is based on an average foveal acuity for healthy adults below the age of 50 [59]. In addition, the smooth properties of the Gaussian result in a visually more pleasant sampling mask (Figure 6.1a). Based on our acuity fall-off function around the gaze, we can now infer the number and spread of cast rays in screen space as well as the sampling density along the rays in object space.

6.1.1 Method

Our technique carries out several computationally intensive pre-processing steps, minimizing the induced cost at runtime. In the first pre-processing step, we compute a sampling mask to determine the starting points of the rays for raycasting using the LBG stippling algorithm. The algorithm has several advantages over other straight forward sampling strategies: It arranges the rays' starting positions in a way that little or no visible patterns manifest, which could irritate the viewer [50]. According to our observations, this property translates well to foveated volume rendering. Also during pre-processing, we optimize the spatial locality of sampling rays by sorting them according to Morton order, also called Z-curve (see Figure 6.1b for an illustration). This improves ray locality and therefore caching behavior when using the generated sampling masks for volume raycasting at runtime. The sampling mask has at least



(a) Color-coded Voronoi cells and corresponding ray origins in black.



(b) Ray origins connected by lines to illustrate the Morton ordering.

Figure 6.1: Illustration of the sampling mask for volume raycasting. © 2019 The Eurographics Association [8].

twice the size of the screen resolution since we translate it during volume rendering according to the gaze, effectively keeping the high-density part in the middle of the texture at the foveal region of the user.

Our sampling method in image space is based on Voronoi cells, as is natural neighbor interpolation, which provides a mathematical symmetry between sampling and reconstruction strategy. Accordingly, we modified the LBG algorithm to compute neighbors and weights for natural neighbor interpolation of the sparsely sampled rays during pre-processing. These can then be used directly during runtime for reconstruction. Finally, we apply a temporal smoothing filter to attenuate undersampling artifacts in peripheral vision.

Weighted Linde-Buzo-Gray Algorithm

The objective of the LBG algorithm [50, 68] is to arrange stipples depending on a function. In this work, we want to arrange starting positions of representative rays, instead of

stipples, according to a density function over the sampling mask. Let $\Phi : \mathbb{R}^2 \rightarrow [0, 1]$ be a function that maps pixel coordinates to sampling density. The algorithm is initialized with a random distribution of ray positions. During each iteration, the neighborhood of each ray position r is inferred from the Voronoi diagram to assess how well each ray represents its proximity in Φ by integrating over the corresponding Voronoi cell V_r. Formally, the target density for a ray position representing the cell T_r is defined as:

$$T_r = \iint\limits_{V_r} \Phi(x, y)\, dA \,. \tag{6.1}$$

In our case, this can be approximated by accumulating the density of all pixels of the sampling mask that are part of to the respective Voronoi cell. Then, the algorithm compares the cost function T_r with the area occupied by the ray position A_r to measure the error ε. The ray positions are then adjusted according to one of three cases:

(1) If $(T_r \in [A_r - \varepsilon, A_r + \varepsilon])$: move and relax the ray position.
(2) If $(T_r > A_r + \varepsilon)$: split the ray.
(3) If $(T_r < A_r - \varepsilon)$: delete the ray.

This process is repeated until the error is below a given threshold, i.e., the amount of rays remains unchanged. To stabilize iteration, we use a hysteresis function for the lower and upper bound of the cases above:

$$h(i) = A_r \cdot (1 \pm (\varepsilon_0 + i \cdot \varepsilon_\Delta) \,/\, 2), \tag{6.2}$$

with $\varepsilon_0 = 0.5$ and $\varepsilon_\Delta = 0.1$ in the ith iteration.

Volume Raycasting

Our GPU-based volume raycaster samples 3D textures using perspective projection, performing front-to-back compositing using a post-classification model with linear transfer functions. Density values that are sampled along casted rays are determined using trilinear interpolation. The renderer features local Phong illumination, based on gradients (central differences), ERT, and ESS for acceleration (subsection 2.2.3). We adjust the distance between samples as given by our sampling mask proportional to our acuity fall-off function, not only in image but also in object space. Akin to a decreased sampling density in image space that is a direct result of the visual model (subsection 2.1.3), a coarser sampling along rays produces a lower resolution approximation with respect to a reference image (e.g., [30]).

Natural-Neighbor-Based Image Reconstruction

To reconstruct an image of the volume, we have to perform an interpolation of the sparsely sampled screen space. We chose a natural neighbor interpolation scheme [163]

because it provides a smooth approximation, requires only local neighbors, and is generally C_1 continuous. Moreover, it fits our sampling strategy well since it is also Voronoi-based.

Despite being computationally expensive, the method has been used in different application domains such as engineering, mechanics, and also in scientific visualization [150]. To compute the interpolated value at a given point, a new Voronoi cell is inserted into the existing tessellation at the position (x, y) of the point. The estimate G of the new point is then calculated by using the areas A of the intersections with the neighboring cells in relation to the total area of the new cell as weighting factors for the interpolation:

$$G(x, y) = \sum_{i=0}^{k} \frac{A(S_i \cap N)}{A(N)} \cdot f(x_i, y_i) \, , \tag{6.3}$$

with $A(N)$ being the area of the new cell N and $f(x_i, y_i)$ being the known values at the k neighboring cells S_i.

The computation of the weights is offloaded into a pre-processing step that results in two textures. Both have twice the screen's resolution to accommodate for gaze-dependent translation of the mask. One texture stores the indices of the neighboring cells, the other one stores the weights that are used for interpolation. This design makes it simple and efficient to compute the interpolations on a per-pixel basis after raycasting.

Temporal Smoothing Filter

While natural neighbor interpolation provides a precise and smooth reconstruction, sparse sampling introduces aliasing artifacts, especially at hard transitions in the volume data. In addition, undersampling artifacts may occur especially near fine structures due to the low sampling density in peripheral vision. However, peripheral vision is particularly sensitive to contrast changes and movement. Therefore, we attenuate those artifacts using a temporal smoothing filter by averaging between n previous frames. We found $n = 8$ to be a good compromise between sufficient smoothing, perceived fade, and perceived frame rate [57]. Effectively, the filter also provides a form of anti-aliasing through implicit temporal supersampling. Moreover, it hides the transition from blurry to sharp during rapid eye movements that can be noticeable if the screen update lags behind the eye movement. This approach also helps with eye-tracking devices with low sampling rate. To avoid introducing a motion blur effect, we do not apply the temporal smoothing during image-altering changes such as camera manipulation and transfer function modifications.

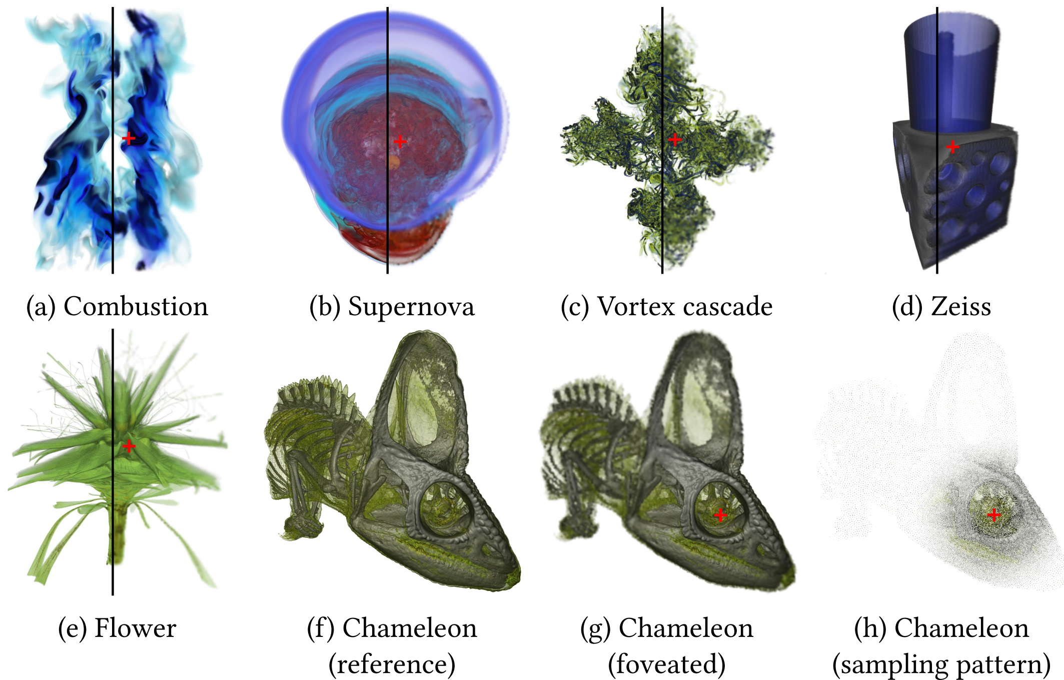
Table 6.1: Foveated Volume Rendering Performance

Data set	Mean fps		Relative speedup
	Regular	Foveated	0 1 2 3 4 5
Combustion	73.85	154.20	
Supernova	37.60	105.28	
Vortex cascade	33.27	104.87	
Zeiss	98.50	177.09	
Flower	22.79	74.08	
Chameleon	35.99	99.78	

6.1.2 Results

We tested our approach using a stationary Tobii Pro Spectrum eye-tracker with a sampling rate of 1200 Hz. For runtime performance evaluation, we simulated deterministic and randomly scattered gaze positions for comparability and reproducibility. All images were rendered with a resolution of 1024×1024 px on a 1080 p screen with a 24" diagonal, using a workstation equipped with an Intel Core i7-7700K, 32 GB RAM, an NVIDIA GeForce GTX 1070, and running Windows 10. Our single-pass implementation of volume raycasting uses OpenCL 1.2 as the parallel compute backend for reasons of device and platform portability. The evaluated pixel colors are written to a texture and shown with a screen-filling quad using OpenGL. The sampling rate along the rays was set to be twice the data resolution (i.e., a sampling distance of half a voxel) as base value. We used several CT-scan and simulation data sets for testing (see Table 2.2). For each data set, we designed a specific transfer function that we used across our measurements, the resulting renderings are shown in Figure 6.2. To give an impression of the sampling pattern, Figure 6.2h shows the same configuration as Figure 6.2g but without natural neighbor interpolation applied, i.e., only pixels containing ray starting positions are colored. For comparison, Figure 6.2f shows the reference without foveated rendering applied.

Based on our results from section 3.1, we measure 256 random camera configurations, rotating around the volume and changing the distance to the volume, while the target of the camera is the center of the data set. The maximum distance is set to the minimum distance required to fit the whole projected volume onto the screen for any camera configuration. For each camera configuration, we test 256 random gaze positions, resulting in a total of 65 536 measurement configurations. We measure five frames for each configuration and keep the median frame time as representative value. We also calculate the mean frame rate and relative speedup based on those representative



(a) Combustion (b) Supernova (c) Vortex cascade (d) Zeiss

(e) Flower (f) Chameleon (reference) (g) Chameleon (foveated) (h) Chameleon (sampling pattern)

Figure 6.2: Example renderings of each data set with transfer functions from the benchmark. The left sides show the reference renderings, while the right ones show our foveated renderings with a red cross indicating the respective fixation point. For the Chameleon (g), we also provide sampling results before reconstruction (h). © 2019 The Eurographics Association [8].

measurements. Table 6.1 shows frame times with and without foveated rendering as well as the relative speedup for each data set. On average, we achieved speedups between factor 1.8 and 3.2 using our foveated rendering, depending on the data set and transfer function. Generally, the speedup is higher for volumes with less empty space. We could barely experience perceptible changes with respect to visual quality.

The reduced number of rays when using our approach (less than 8%) may indicate an even bigger performance gain. However, the sparse regions in the periphery often correspond to those rays accelerated by empty space skipping. Furthermore, we cannot take advantage of caching as efficiently compared to using the regular volume rendering due to the sparser voxel-access pattern (see Figure 6.2h). Finally, there is the small overhead of natural neighbor interpolation, which amounts to 1.5 ± 0.157 ms for all data sets, as well as a general overhead (kernel invocation, etc.).

6.1.3 Discussion and Future Directions

We approximate the acuity fall-off by using a 2D Gaussian function. While this is sufficiently accurate for people with normal vision, individual differences exist regarding the distribution of rod and cone cells in the retina. Thus, calibrating our system according to the physiology of individuals could potentially improve the quality and performance of our foveated rendering approach.

Further, the sparse sampling in the peripheral regions can lead to artifacts, especially near sharp edges or fine structures in the data. While our temporal smoothing filter helps to avoid such artifacts, it may potentially induce undesired motion blur effects. Therefore, we have refrained from using the filter for interaction and dynamic data. We found that explicit and expected movement tends to make rendering artifacts less prominent. We also explored a 3D mipmap stack and linear interpolation between the levels based on the sampling density. However, the performance hit caused by additional 3D texture fetches was too high compared to the improvement in image quality. In general, we suppose that our implementation can be further optimized to yield even higher speedups based on the substantial reduction in emitted rays when using our technique.

High frame rates are crucial in virtual reality to prevent dizziness and motion sickness when using head-mounted displays. The presented foveated volume rendering technique can be employed for head-mounted displays with integrated eye-tracking devices in the future to help achieving the required frame rates.

6.2 Foveated Encoding for High-Resolution Displays

Streaming and conferencing technology are becoming increasingly ubiquitous in recent time. However, many of the existing systems do not support resolutions beyond 4K, as are common for large display walls that are used in visualization contexts. Streaming solutions with a particular focus on such high-resolution display typically make compromises between quality and bandwidth (e.g., [33, 132]). They either deliver a high quality image and therefore require bandwidths that may not be met by the network infrastructure, or they uniformly decrease the quality to maintain adequate frame rates.

We address this issue by employing foveated rendering (see subsection 2.1.3) for encoding settings. That means we track the users' gazes and use that to dynamically adapt the compression in parts of the image to meet the capabilities of the HVS (see Figure 6.3). This leads to overall lower bandwidth requirements while keeping the perceived image quality at a high level. To the users, the result is typically indistinguishable to a high quality encoding of the whole screen. To meet the latency requirements, our approach

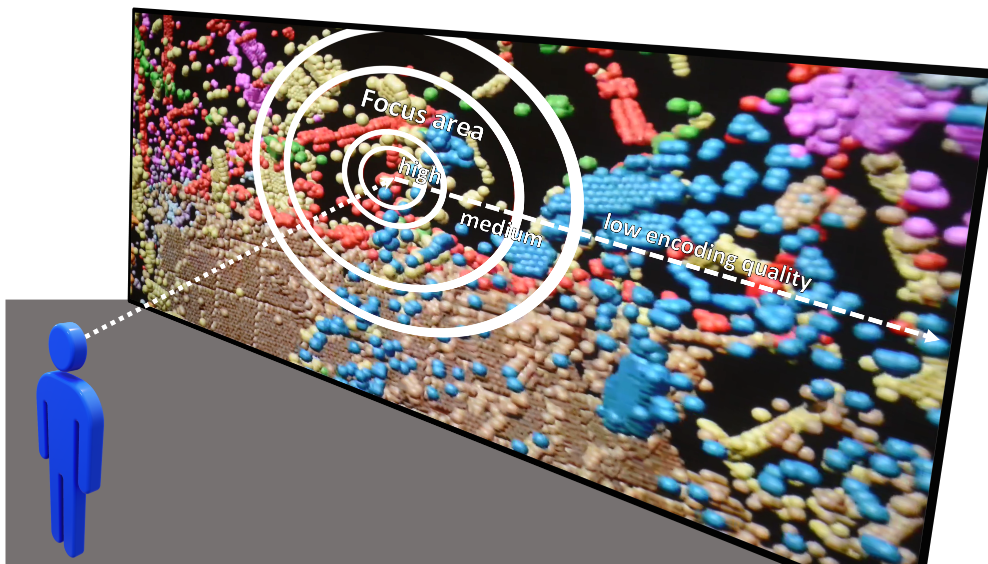


Figure 6.3: Illustration of our foveated encoding approach for large high-resolution displays. Macroblocks near the user's focal point are encoded with high quality, while the encoding quality decreases towards the periphery according to the acuity fall-off of the human visual system (HVS).

is based on frame buffer streaming and dedicated hardware video encoders and decoders found on modern GPUs.

6.2.1 Method

Parts of the system run on the server, others on the client. Onside communication is implemented using MPI, while we use internet communication between the sides. The server side renders and captures the visualization and carries out foveated encoding of the captured frames. For this, the quality of the macroblocks (16×16 pixel blocks) used by the H.264 video compression standard are changed based on their distance to the gaze points of the observers. By tracking the gaze of the users, the client provides the respective foveated regions and displays the decoded frames.

The following steps are performed on the **server side** to produce the foveated encoding: The computation is done in parallel across all nodes that render parts of the potentially distributed frame buffer. First, the last rendered frame is acquired as a texture. Initially, all macroblocks are considered to be in the peripheral region and therefore use the lowest possible quality parameter. In the second step, the intersection between the last received foveated regions and the macroblocks is computed. In case of an intersection, the new quality parameter for the respective macroblock is computed based on the distance to the region's center using a hyperbolic fall-off function. In order to adapt

the quality, we use a map that contains an offset for the quality parameter of each macroblock. Encoding is done asynchronously on the GPU. After encoding, all frames are sliced to fit the User Datagram Protocol (UDP) packages and forwarded to the client.

The **client side** also processes the potentially distributed frame buffer in parallel. The received UDP packages are reordered based on the timestamp and a sequence number. The potentially sliced frames are reassembled and then queued for asynchronous decoding. Incomplete frames are dropped after a user specified time. We render the decoded frames using NVIDIA Quadro Sync for synchronization, with an MPI alternative as portable fallback. One of the client's nodes computes the foveated regions based on the tracking data and forwards size and location to the server. As a basis for this calculation, we use average acuity values (see subsection 2.1.3), so the size of the foveated region spans 50° horizontally and 50° vertically from the user's position.

6.2.2 Results

We evaluated of our system quantitatively by measuring the latency and throughput required to share the content of a tiled display with 10800×4096 pixels resolution in a local area network, with and without using our foveated encoding. The resolution is split horizontally into nine regions, each powered by a single display node featuring an NVIDIA Quadro M6000 GPU. Two dedicated nodes are used for streaming that are equipped with a GeForce GTX 1060 GPU each. We used a recorded exploration session of a molecular dynamics simulation (Figure 6.3). The additional latency of our system was hardly noticeable in a side-to-side comparison for multiple scenarios. Typically, it stayed below 30 ms from capturing to displaying the frame.

Figure 6.4 shows the throughput for different encoding setups while streaming the molecular dynamics simulation. On the left, three fixed, non-foveated encoding setups are compared: low (51), medium (31), and high (11). Using the high encoding quality for the whole frame yielded a maximum throughput of almost 2000 Mb/s, while medium peaks at around 540 Mb/s and low at 60 Mb/s. In comparison, the throughput stayed consistently below 200 Mb/s when using our foveated encoding, with the highest measured value at 150 Mb/s (Figure 6.4 right). Tracking a second user did not increase this value substantially. Overall, the foveated encoding required on average between 10 and 20 Mb/s more bandwidth than the low setting, except for situations when users want to get a complete overview of the visualization by walking farther away from the screen (see the peaks in Figure 6.4). All in all, our foveated encoding approach used on average 14% of the measured throughput required for the non-foveated medium encoder settings and 4% compared to high encoder settings. At the same time, the changes in quality were hardly perceptible to the observers.

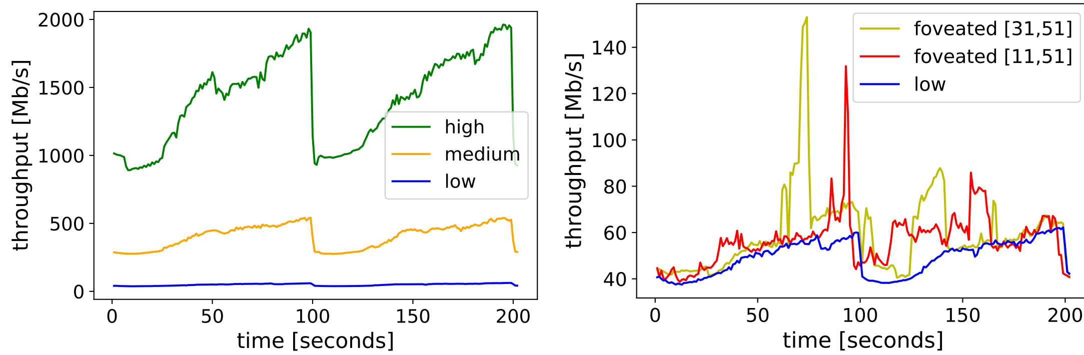


Figure 6.4: Aggregated throughput of all nodes for three non-foveated tests (left), and two foveated tests and the low setting for comparison (right). Both foveated tests use a static region. The peaks in the foveated curves can be attributed to the user moving far away from the display, which leads to a bigger foveated region with more macroblocks using a higher quality.

6.2.3 Future Directions

A natural step to extend the presented foveated encoding approach would be with a more accurate, user validated model that also limits the size of the foveated region based on the users distance and the dimensions of a pixel. Further, a progressive encoding in case of static frames would be a useful extension. Another interesting path to move forward would be the combination of the two approaches presented in this chapter to enable real time streaming of detailed volume visualizations on high resolution displays. For instance, the foveated sampling could be directly streamed, thereby saving throughput, while the reconstruction could be done at the client site.

CHAPTER 7

Performance-Optimized Volume Rendering Applications

This chapter introduces two applications to show how performance optimization can open up new application areas for cost intensive visualization algorithms. Both application examples use volume rendering at the core of their respective technique but aim at different domains and data to visualize, which is not the classical volume data from simulations and CT-scans. Although volume rendering is a comparably computational expensive technique, specific performance optimizations allow for interactive rendering of data sets containing over a billion of scalar values.

The first approach (section 7.1) is concerned with the visualization and interactive analysis of dynamic graphs that contain a large number of time steps [4, 1]. A special focus is set on the support of analyzing the temporal aspects in the data. For this, a central component of the technique is an interactive volumetric representation of the graph based on the concept of a space-time cube (STC) that is generated by stacking the adjacency matrices of all time steps. In an integrated application, this central representation is complemented with different views, adjustment options, and evolution provenance to enable interactive exploration of dynamic graphs.

The second approach (section 7.2) combines the visualization of eye-tracking data, video data, and optical flow [5]. As in the other technique, the concept of a STC is adapted to create an interactive volumetric representation of the data. This allows for a spatio-temporal analysis of gaze data from multiple participants in the context of a video stimulus. With specifically designed transfer functions, different data aspects can

be emphasized, making the visualization suitable for explorative analysis of the data as well as illustrative support of statistical findings.

This chapter is partly based on these publications

- V. Bruder, M. Hlawatsch, S. Frey, M. Burch, D. Weiskopf, and T. Ertl. "Volume-based large dynamic graph analytics". In: *Proceedings of the International Conference Information Visualisation (IV)*. Dec. 2018, pp. 210–219 [4]
- V. Bruder, H. Ben Lahmar, M. Hlawatsch, S. Frey, M. Burch, D. Weiskopf, M. Herschel, and T. Ertl. "Volume-based large dynamic graph analysis supported by evolution provenance". In: *Multimedia Tools and Applications* 78.23 (2019), pp. 32939–32965 [1]
- V. Bruder, K. Kurzhals, S. Frey, D. Weiskopf, and T. Ertl. "Space-time volume visualization of gaze and stimulus". In: *Proceedings of the ACM Symposium on Eye Tracking Research and Applications (ETRA)*. 2019, pp. 1–9 [5]

7.1 Volume-Based Large Dynamic Graph Analysis

Graphs can be used to analyze and visualize relational data in many application fields, such as network traffic, biological processes or social relationships. When also considering the evolution of the data over time, the graph becomes a dynamic graph. Visualizing dynamic graphs is a challenging task, especially if they contain hundreds of nodes and thousand of links.

We introduce a visual analytics approach that allows for interactive analysis of graphs in this category. With our technique and application, we support typical analysis tasks such as detecting temporal patterns, e.g., clusters forming and disintegrating or repeating occurrences of similar node-link structures. We present and implement four classes of analytics methods that we believe to be central:

1. Data views
2. Aggregation and filtering
3. Comparison
4. Evolution provenance

At the core of our techniques is a volumetric representation of the dynamic graph that is similar to a space time cube [24]. In this representation, every time step is represented by an adjacency matrix of the node link structure. The adjacency matrices are arranged according to their temporal order in a third dimension, generating a cuboid structure that is a concise and static representation of the whole dynamic graph that preserves the mental map [137, 21].

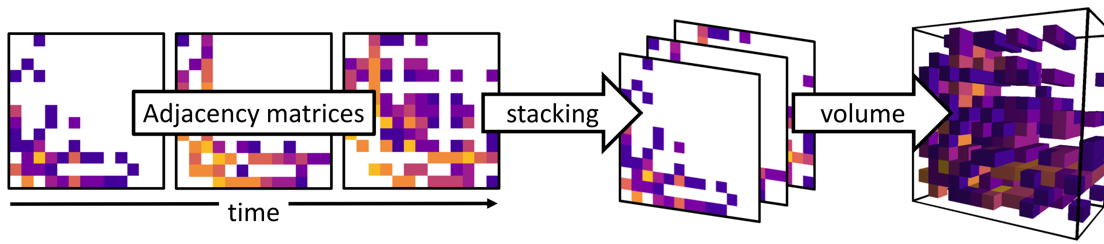


Figure 7.1: For each time step of the dynamic graph, 2D adjacency matrices are stacked to create a 3D volume, forming our static graph representation.

7.1.1 Static Volumetric Graph Representation

A focus of our approach for visual analysis of dynamic graphs are temporal aspects. In this context, a static representation of the data has several advantages compared to animations [175]. In our approach, (directed) graphs are represented as adjacency matrices, i.e., the rows and columns denote the nodes of the graph. Therefore, an entry (i, j) at the i-th row and j-th column of the matrix denotes a (weighted) edge from the node with index i to the node with index j. One matrix is generated for each time step of the dynamic graph. An adjacency matrix representation has several advantages [27]:

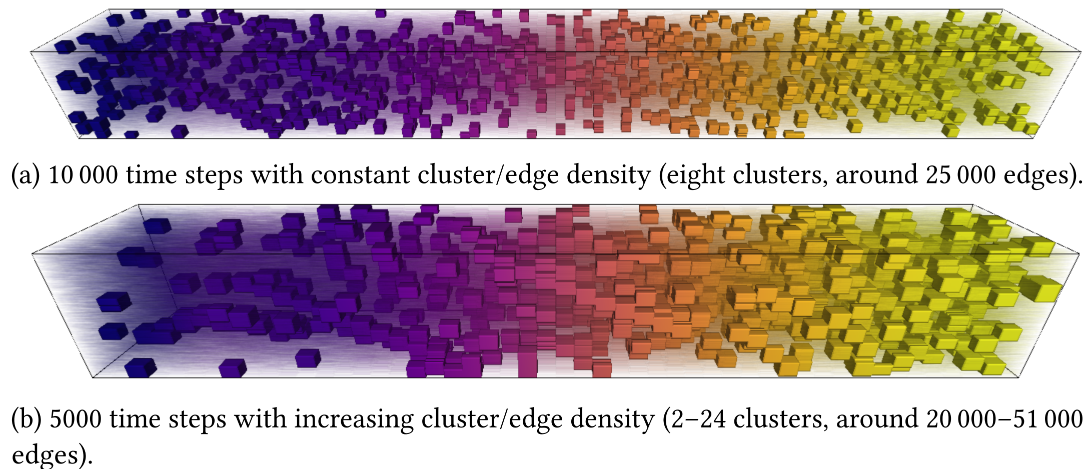
- Large graphs can be depicted without crossing or intersecting graphical elements.
- A volumetric representation can be generated without layout algorithms and is consistent for all time steps.
- Adjacency matrices are well suited for revealing cluster structures.

As illustrated in Figure 7.1, adjacency matrices are 2D structures that are stacked to incorporate the temporal evolution of the graph. In the resulting 3D structure, the x-axis and y-axis represent nodes, while entries in the plane defined by those two axes represent edges (including their weights). The z-axis represents time.

The visual appearance of adjacency matrices strongly correlates with the ordering of the nodes. Therefore, we support reordering of the nodes based on three different sparse matrix ordering algorithms: Cuthill-McKee [47], King [98], and Sloan [166]. As basis for ordering, an arbitrary amount of time steps (i.e., matrices) is selected that are aggregated and used in the respective algorithm.

Scalability

The technical scalability of the dynamic graph size is limited by the available VRAM on the GPU. In our implementation, every edge including its corresponding weight is represented by an 8 bit scalar value. This results in a memory requirement of $n \times n \times t$



(a) 10 000 time steps with constant cluster/edge density (eight clusters, around 25 000 edges).

(b) 5000 time steps with increasing cluster/edge density (2–24 clusters, around 20 000–51 000 edges).

Figure 7.2: Scalability of the volumetric graph representation. The graph contains 512 nodes in both cases.

bytes, with n being the number of nodes and t the number of time steps. Additionally, we need approximately $\frac{1}{7}$ of this size to store a full 3D mipmap-stack of the aggregated representations. For instance, we can load a dynamic graph with 1000 nodes and 10 000 time steps on an NVIDIA Titan X (Pascal) GPU with 12GB VRAM. The number of nodes has a quadratic effect on the data size, while the number of time steps only has a linear impact. The number of edges has no influence on the memory requirements, i.e., a sparse and a dense graph with the same number of nodes and time steps requires the same size. The size limitations could be mitigated in part by using compression, a different data structure, or out-of-core techniques to overcome GPU memory limitations.

The visual scalability strongly depends on the 3D representation, the main issue being the occlusion of edges. Figure 7.2a demonstrates the scalability with respect to the temporal dimension of the graph. It shows a graph with 512 nodes, 10 000 time steps, a constant edge density of around 25 000 edges (around 10% of the fully connected graph). All time steps include some random edges and eight clusters that are randomly redistributed every 50 time steps. Figure 7.2b shows the scalability with respect to graph density. The depicted graph features 512 nodes, 5 000 time steps and the edge density increases from around 20 000 edges (8% of a fully connected graph) in the beginning to 51 000 edges (20% of a fully connected graph) in the last time steps. The number of clusters increases from two to 24, and the graph also contains random edges.

Comparing the renderings, we can see that the visual scalability with respect to the temporal dimension is good when using our approach. Doubling the number of time steps only slightly decreases visibility of the data since the same image resolution is used for a larger volume. Furthermore, problems with a reduced resolution can be often alleviated with camera navigation, e.g., zooming and panning. However, the example

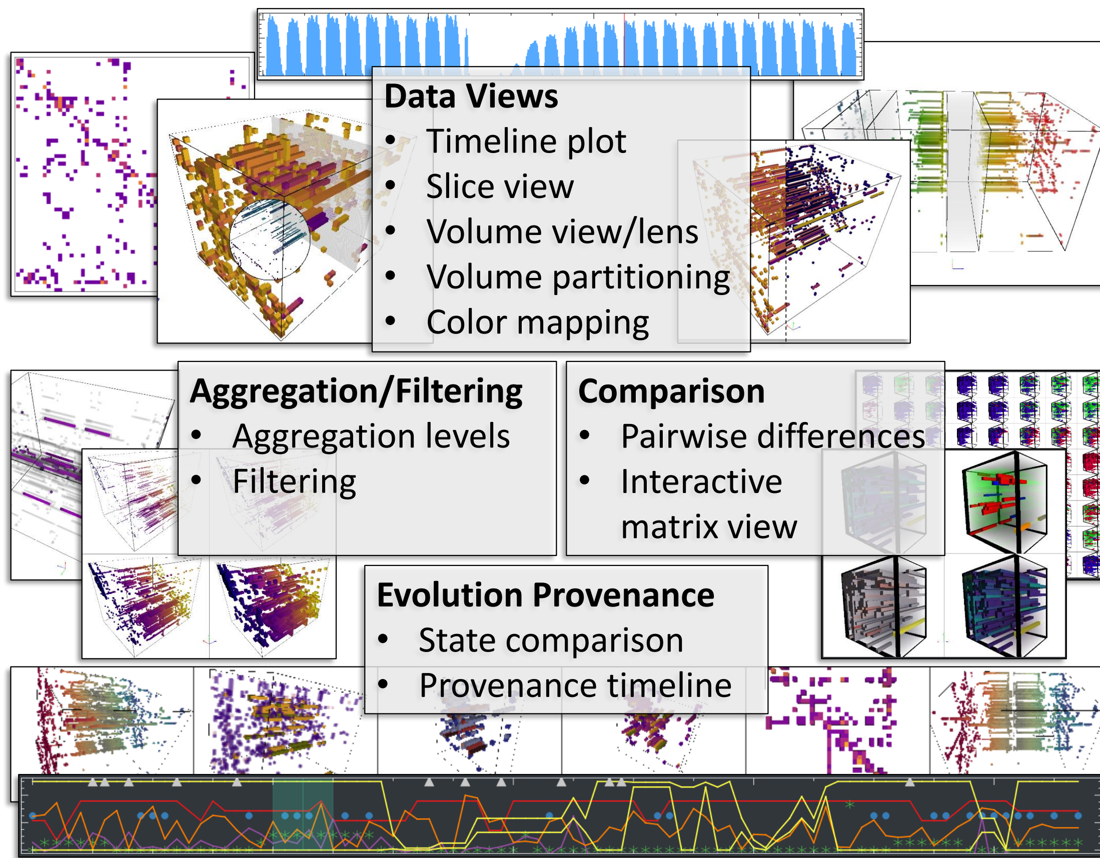


Figure 7.3: Four classes of analytics methods for large dynamic graphs. We implement and discuss respective techniques, and show their utility.

with increasing edge density shows the limitations of the visual scalability. Doubling the number of edges can introduce substantial occlusion, making analysis more difficult. To address this issue, we support various filtering and aggregation modalities, and complement our volumetric view with additional data views.

7.1.2 Classes of Analytics Methods

The efficient analysis of large dynamic graphs in general and their temporal features in particular can be supported by combining different classes of analytics methods. We identified four central classes (see Figure 7.3):

- **Data Views.** Large dynamic graphs typically exhibit a lot of information in many interesting aspects. A single visualization is often not capable to convey them all.

Therefore, it is important to offer distinct perspectives on the data using multiple linked views with suitable visualizations.

- **Aggregation and Filtering.** Showing all edges of a large dynamic graph, especially a dense one, quickly leads to visual clutter, overload, and occlusion using our volumetric approach. This makes filtering and aggregation of adjacent edges an essential part of the analysis process to reduce the visualization to relevant information.
- **Comparison.** Comparing different sections within a temporal graph or even several distinct dynamic graphs becomes challenging using the methods discussed above. Including a dedicated visualization for this specific task is therefore important for the analysis process.
- **Evolution provenance.** Oftentimes during an analysis process, a parameter configuration is selected that shows interesting features of the data. Going back to such a analysis step can be very useful during data exploration. Furthermore, tracking and navigating the provenance evolution also helps to understand and reconstruct the analysis process. Our implementation of this class in described in subsection 7.1.3.

We implement methods for all classes in our integrated graph analytics system. In the following, we describe these methods and their implementation in detail and discuss how they support the analysis of large dynamic graphs.

Data Views

A key to providing an analysis system that supports many different data sets and tasks is to offer multiple data views with different types of visualizations, because some are typically better suited for a specific task than others. Figure 7.4 shows the graphical user interface (GUI) of our system offering multiple views that can reveal different aspects of the data.

Timeline Plot. The timeline shows different graph metrics on a 2D plot over time (Figure 7.4a). Those metrics include, for instance, the number of edges in each time step, or the linear arrangement of the matrices. This visualization is intended to give an overview of the temporal evolution of the graph and thereby reveal recurring patterns. The timeline plot is also an interface for interaction tasks such as selecting and highlighting a single time step, browsing through the time slices or setting split marks (see Volume Partitioning below).

Slice View. While the timeline plot often provides a good overview on the data, it is highly aggregated and lacks details. Therefore, we also provide information of

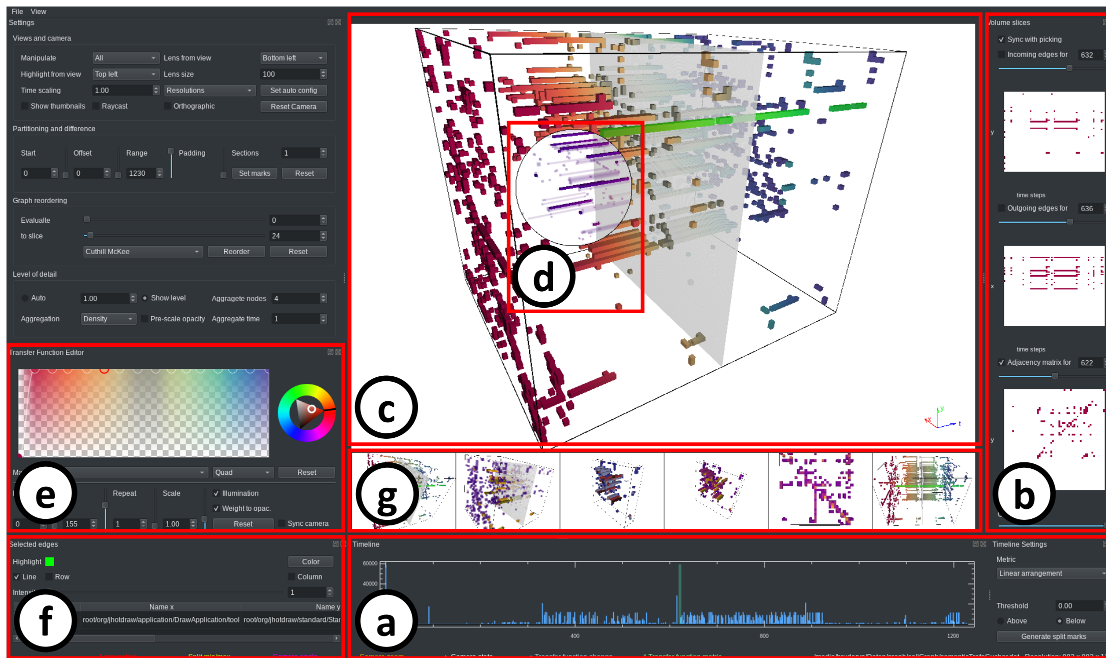


Figure 7.4: The GUI of our dynamic graph analysis tool, consisting of: (a) timeline/provenance plot, (b) slice views, (c) volume view, (d) lens, (e) color map editor. The list view (f) shows information of selected edges and nodes, selected provenance states are shown as thumbnails (g).

individual, selected time steps with 2D slice views (Figure 7.4b). One view for each dimension is displayed, i.e., the slices show all incoming edges of a selected node (x-axis), all outgoing edges of a selected node (y-axis), or one adjacency matrix (time axis). The slices are aggregated and colored according to the current selection and may be linked to picking of elements and the section on the timeline plot. The respective slice planes can be visualized inside the volumetric view to give context in the spacial structure.

Volume View. While the slice views provide a convenient method to analyze single nodes or time steps, it is hard to analyze the temporal evolution of structures on a global level. For this task, we provide a direct visualization of the full graph volume (Figure 7.4c). Here, the analyst can interactively rotate, zoom, pan, and tilt the camera and thereby change the view on the data. Multiple tiled view modes showing a different configuration per tile are supported. One of the view setups features a main view and a thumbnail band containing six consecutive provenance states as shown in Figure 7.4g. Which states to display can be selected using the evolution plot (see subsection 7.1.3).

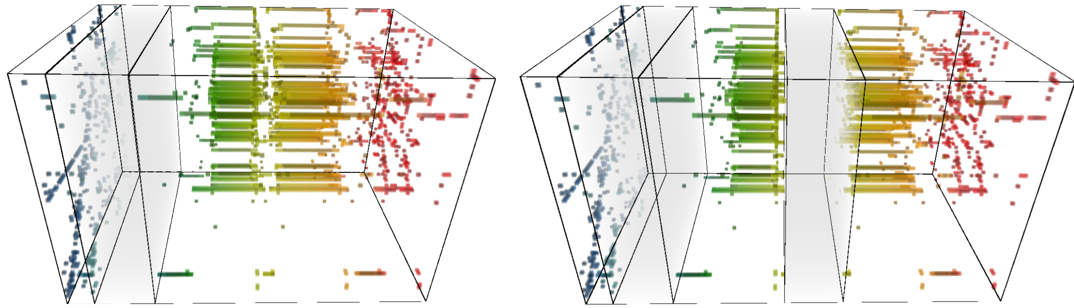
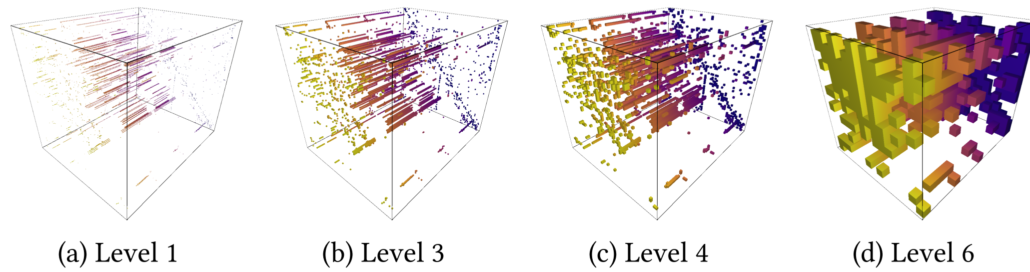


Figure 7.5: Splitting the graph volume into sub-volumes along the time dimension enables dedicated analysis of specific sections.

Detail Views. To inspect details, we provide a lens feature that is controlled via mouse (Figure 7.4d). It shows a different visualization parametrization inside the volume view, such as color mapping and/or aggregation level. Further, we support interactive selection and highlighting of edges (single elements or aggregated blocks) inside the volume view. The selected edges are shown in a list view (Figure 7.4f) in which corresponding nodes and weights are displayed as well.

Volume Partitioning. To improve the interaction possibilities with regard to temporal analysis of the graph, we support partitioning and splitting of the volume into sub-volumes along the temporal axis (Figure 7.5). The partitioning is helpful in reducing visual clutter and potentially occlusion. Split marks can dynamically be added and removed along the timeline plot, a visual gap between the sections in the volume indicates the locations. The gap size can be adjusted and single blocks selected for individual inspection. Finally, we offer the possibility to automatically generate split marks based on different graph metrics.

Color Mapping. Properties of the dynamic graph may be mapped to color by using a color map editor (Figure 7.4e). That means, the volumetric representation as well as the slice views are colored according to a selected metric and a user defined color map. Examples of possible color mappings are shown in Figure 7.7. They include the weights of the edges and the order of the nodes in the adjacency matrix. Additional metrics support the analysis of the temporal evolution in the graph by mapping graph metrics, such as density or linear arrangement, as colors directly onto the volume. Further possible mappings include the temporal dimension and the lifetime of edges. We implement the color mapping for our static volumetric representation via the concept of transfer functions from traditional volume rendering as it is used in scientific visualization. It bears some similarity to our filtering discussed in the next section: color mapping defines the color, while filtering steers the opacity.



(a) Level 1 (b) Level 3 (c) Level 4 (d) Level 6

Figure 7.6: Aggregation levels (aggregated by edge density) of a software call graph.

Aggregation and Filtering

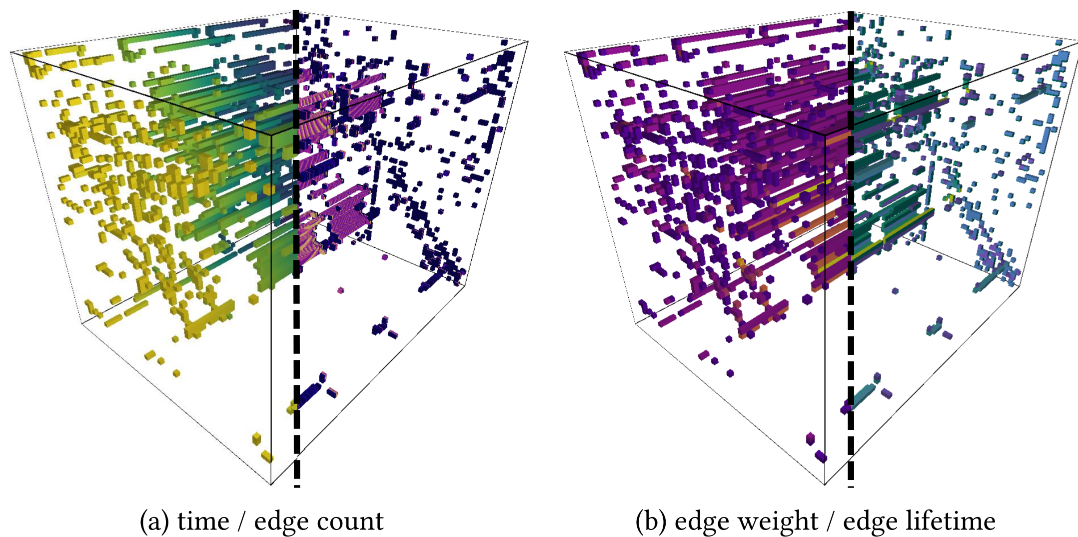
A direct visualization of all edges of a large graph is usually not feasible, especially for dense graphs, because of visual clutter, occlusion, and space constraints, depending on the visualization technique. In the case of our volumetric representation, the former two apply. To mitigate those issues, we offer several techniques for aggregation and filtering.

Aggregation. In the volumetric and the slice views, individual edges (rendered as small cuboids/rectangles) can be very small, especially for larger graphs. Those small elements may be hardly visible or not at all, due to the projection and rasterization, which is limited by the screen resolution. Irritating visual artifacts, such as Moiré patterns may occur. Therefore, we support multiple data aggregation methods that can be applied to the original data and generate a stack of representations with smaller resolutions. We support aggregation by minimum, maximum, and average weight, as well as density of edges. Aggregation may be applied to the spacial domains only. When applying the aggregation, a single voxel in the volumetric view may represent multiple edges (Figure 7.6). Typically, coarse patterns can be seen using a high aggregation level, while details may be better analyzed with a low level and closeup view.

Filtering. Filtering is a method essential to reducing visual clutter and occlusion, or highlight parts of the graph with specific properties. We support it by applying transparency to edges with selected properties, for instance those with low weights. For this, we use a transfer function mapping that can be dynamically adjusted and is applied to our volumetric representation.

Comparison

With our system, we focus on supporting the analysis of temporal aspects in particular. Therefore, we support another modality: the comparison of arbitrary time sequences (see Figure 7.12 for an example). The analyst can select starting points and a range



(a) time / edge count

(b) edge weight / edge lifetime

Figure 7.7: Different color mappings of dynamic graph properties.

of time steps to compare them against one another. A matrix view is presented that shows the different time sequences on the main diagonal. We use the upper and lower triangles of the matrix to show different aspects: The upper one shows a comparison of the existence of edges, i.e. edges that appear, disappear, or exist in both sequences are highlighted by color and/or filtered by transparency. The lower triangle shows the difference in weights between the sections. The number of time sequences that are shown can be dynamically adjusted.

7.1.3 Evolution Provenance

With the variety of analytics methods presented so far, our approach offers users the possibility to perform different analysis operations and thereby obtain various results and observations. The collection of "history" of manipulations was shown to be effective for reproducibility as well as in remembering intermediate steps conducted to reach insights [86]. To this end, we have extended our approach to support the collection of evolution provenance. Traditionally, users are engaged in a visual analysis session where they perform various *visual analysis steps* iteratively. In this case, evolution provenance keeps track of the set of visual analysis steps performed and thereby comprises the "full story" of a visual analysis session.

Model and Definitions

Given an initial dynamic graph G that contains a set of time steps $\{t_1..t_k\}$, we define a *visual analysis step* $S = \{G_S, V_S\}$, where a sub-graph G_s contains time steps

Table 7.1: Permitted Analytics Operations

Operation	**Parameters**	**Output**
Selection	$\{t_{i'}, t_{j'}\}$; selected range of time steps	$G'_s = \{t_{i'}..t_{j'}\}$, V_s
Partition	$\{m_1..m_y\}$; the set of split marks	G_s, $V'_s = \{v'_1..v'_p\}$
Aggregation	l, where l is a level	G_s, $V'_s = \{v'_1..v'_p\}$
Filtering	$d \times \alpha$, α is opacity and d a graph property	G_s, $V'_s = \{v'_1..v'_p\}$
Color mapping	$d \times rgb$, rgb is color and d a graph property	G_s, $V'_s = \{v'_1..v'_p\}$
Camera config.	angles ϕ, θ, zoom ζ	G_s, $V'_s = \{v'_1..v'_p\}$

$\{t_i..t_j\}, 1 \leq i \leq j \leq k$ and is visualized using a visual analytics configurations V_s defined by the parameters $\{v_1..v_p\}$.

Our evolution provenance collector tracks visual analysis steps encompassing the volumetric representation. Table 7.1 summarizes supported analytics operations enabling the transition from a visual analysis step $S = \{G_S, V_S\}$ to another $S' = \{G'_S, V'_S\}$, and lists the set of parameters needed for each operation. We distinguish six operation types that allow the application of different analytics methods presented in subsection 7.1.2. These operations are captured with our evolution provenance model.

The *selection* operation is associated with the timeline plot, whose x-axis depicts a set of time steps $\{t_i..t_j\}$. The user can select a single or a range of time steps $\{t_{i'}..t_{j'}\}$ with $t_i \leq t_{i'} \leq t_{j'} \leq t_j$ to analyze them visually. The *partition* operation corresponds to the volume partitioning feature, where the user interactively specifies split marks $\{m_1..m_y\}$ to split the time steps $\{t_i..t_j\}$ of a sub-graph G_s into sub-ranges $[\{t_i..t_{m_1}\}, \{t_{m_1}..t_{m_2}\}, \ldots, \{t_{m_y}..t_j\}]$. Our evolution provenance model also encompasses the *aggregation* and the *filtering* operations. The former aggregates the data according to a level l to alleviate the complexity of the visualization, while the latter operation defines the visibility of parts of the data based on a factor α. Further, our provenance model contains the *color mapping* operation where a user maps a graph property d (e.g., weights of edges) to color rgb. Finally, we record selected *camera configurations*, where the user selected a certain zoom level ζ and rotation angles ϕ, θ.

Overall, the evolution provenance is modeled by an analysis session graph that gathers all visual analysis steps made by the analyst. Figure 7.8 shows an exemplary analysis session graph, augmented with screenshots of the respective analytics step.

An *analysis session graph* summarizes the analyst's manipulations when exploring a dynamic graph data set. The analysis session graph is a labeled directed acyclic graph consisting of $n \in N$ nodes and $e \in E$ labeled edges. Each node n corresponds to a visual analytics step S. An edge $e = (n, n', L)$ represents the transition from one visual analytics step $S = \{G_s, V_s\}$ to the next visual analytics step $S' = \{G'_s, V'_s\}$. L is a pair

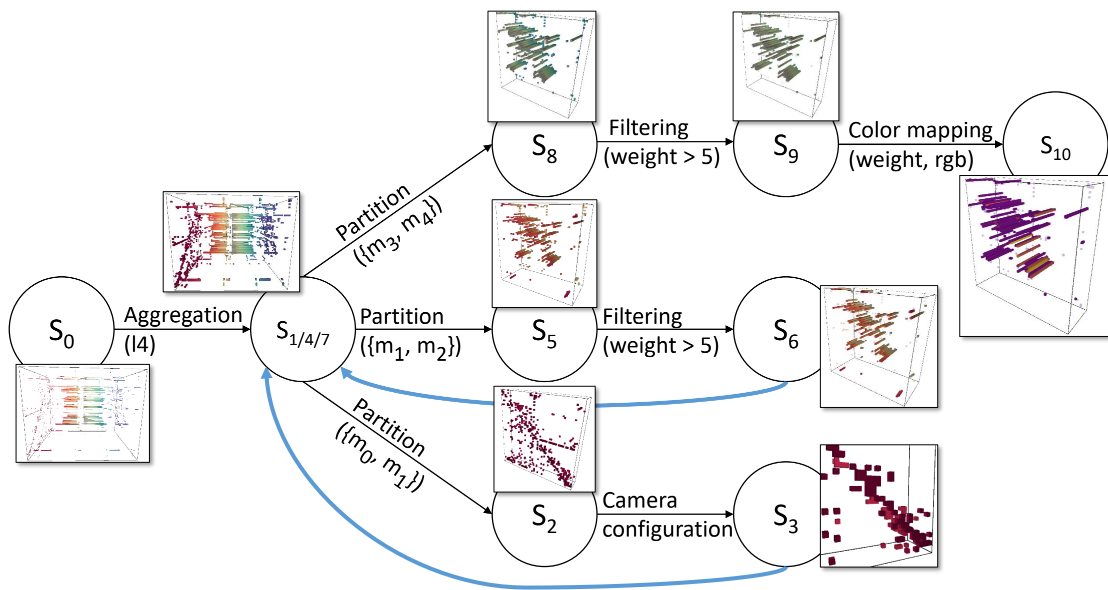


Figure 7.8: Example of an analysis session graph, augmented with images of the respective analytics steps. Blue arrows indicate going back to a former state. Indices indicate the temporal order of the states S.

$\langle op, param \rangle$ where op is an identifier of the analytical operation type (see Table 7.1) and $param$ is the set of parameters used to navigate from S to S'.

Visualization and Navigation

We implement the provenance model described in the previous section. Important analysis steps are saved automatically or on demand by the user: selection, aggregation, partitioning, camera configuration (on demand), color mapping and filtering via opacity. The evolution of the changes can be tracked and navigated in a visualization that shows important parameter changes in a combination of lines and glyphs.

Since the rendering of the full analysis session graph can get spacious for longer sessions, we chose a simplified representation of the provenance graph as a 2D plot (Figure 7.9). The tracked analysis steps are plotted consecutively along the x-axis (ordered by time of configuration), while the value of the respective properties is plotted either binary (using a glyph) or by using a normalized y-value. In this example, the red line indicates changes to the aggregation level. The two yellow lines represent the sub-volume size (partitioning and selection) of the respective state by plotting the normalized minimum and maximum value of the visible time steps. For instance, values one and zero indicate that the analyst investigated the whole volume. The magenta and orange curves show the camera configuration at the respective state, while a triangle glyph indicates that

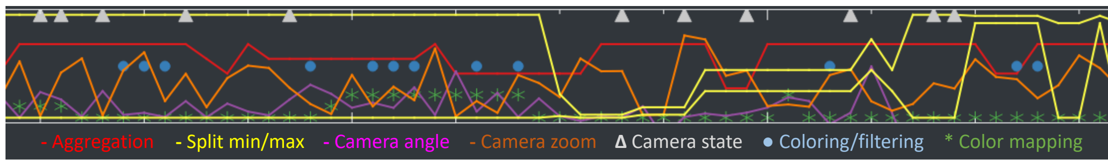


Figure 7.9: Section of an example of our evolution provenance visualization, indicating different analysis steps during a session with plots and glyphs.

the user saved this camera setup explicitly. A blue circle shows a change of the color mapping or filtering, a green star the selected graph property that is used for the color mapping in a categorical order.

The user may interact with this plot to browse through the history of analysis steps. We show six consecutive steps as thumbnails below the volumetric view (see Figure 7.4g). Those thumbnails can optionally be synchronized to the main camera configuration (i.e., they interact similar to the main view) to ease comparison tasks. All tracked states may be selected to apply their configuration to the main view, i.e., "going back" to this state.

7.1.4 Implementation

To enable interactive exploration of the volumetric data structure, even of large dynamic graphs with thousands of time steps and hundreds of nodes, we accelerate the compute heavy calculations by using parallel processing on GPUs. This includes the rendering of the volume and slice views, as well as the hierarchical generation of different aggregation levels. Some of the computations are also performed in parallel on the CPU such as reordering operations on the matrices. By using OpenCL, we can accelerate data generation and processing across heterogeneous platforms such as GPUs, CPUs or parallel accelerators. On GPUs, we take advantage of their integrated texture units and interpolation capabilities making them the preferred devices. In this case, the stacked adjacency matrices are saved and processed as a 3D texture.

We support two different front-to-back raycasting algorithms that have been adapted for the needs of visualizing dynamic graphs (subsection 2.2.3). The first is a parallel 3D digital differential analyzer (DDA), featuring local illumination of voxel surfaces. Although having slightly less runtime performance than our second algorithm, it has several advantages: single and isolated voxels can be easily distinguished because of the lighting approach. The same applies to opaque structures, making this technique well suited for analyzing sparse graphs. The second rendering algorithm is based on a standard raycasting with equidistant sampling points along viewing rays. In contrast to the first method, local illumination is performed based on the gradient that is calculated

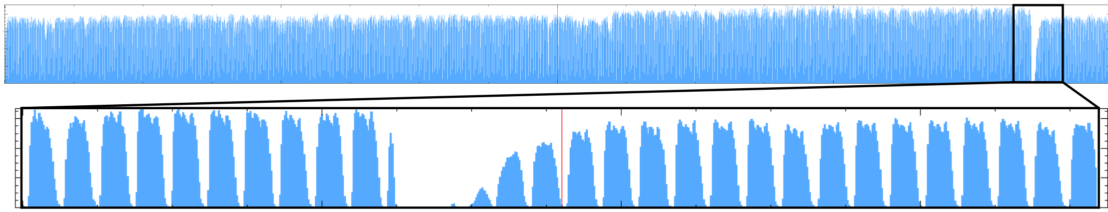


Figure 7.10: Timeline plot of the flight data graph showing the number of edges per time step. The complete data set is shown on the top, while the bottom figure depicts data of September 2001 only.

using central differences. This approach makes the technique better suited for the usage on dense graph data when making use of transparency for filtering.

For both techniques, we map sampled density values (i.e., edge weights) to opacity and color with our filtering and color mapping features. The values are then composed into the final pixel color and opacity. Using raycasting instead of rendering geometry with rasterization has several advantages. Foremost, we do not need to order elements and can therefore easily apply transparency for filtering or highlighting purposes. This improves performance and enables interactive explorations with high frame rates even for large graphs with millions of edges. Furthermore, raycasting enables us to easily display multiple 3D views with synchronized camera configurations without a significant overhead.

We tested our application on a workstation equipped with an NVIDIA Titan X (Pascal) GPU, an Intel Core i7-6700 and 16GB of RAM. Using a data set with over 1000 nodes and 8000 time steps, we were able to achieve interactive frame rates with a minimum of 20 fps during exploration (i.e., camera interactions such as rotation and zoom). Due to the use of early ray termination (ERT), dense graphs can even be rendered faster depending on the opacity mapping.

7.1.5 Application Examples

We demonstrate our techniques with two application scenarios, focusing on the different classes of analytics methods we describe in subsection 7.1.2. In the first scenario, a graph data set containing flight connections over time is analyzed with a focus on the comparison features of our application. In the second scenario, a temporal software call graph is analyzed, the main focus lies on using the evolution provenance tracking features.

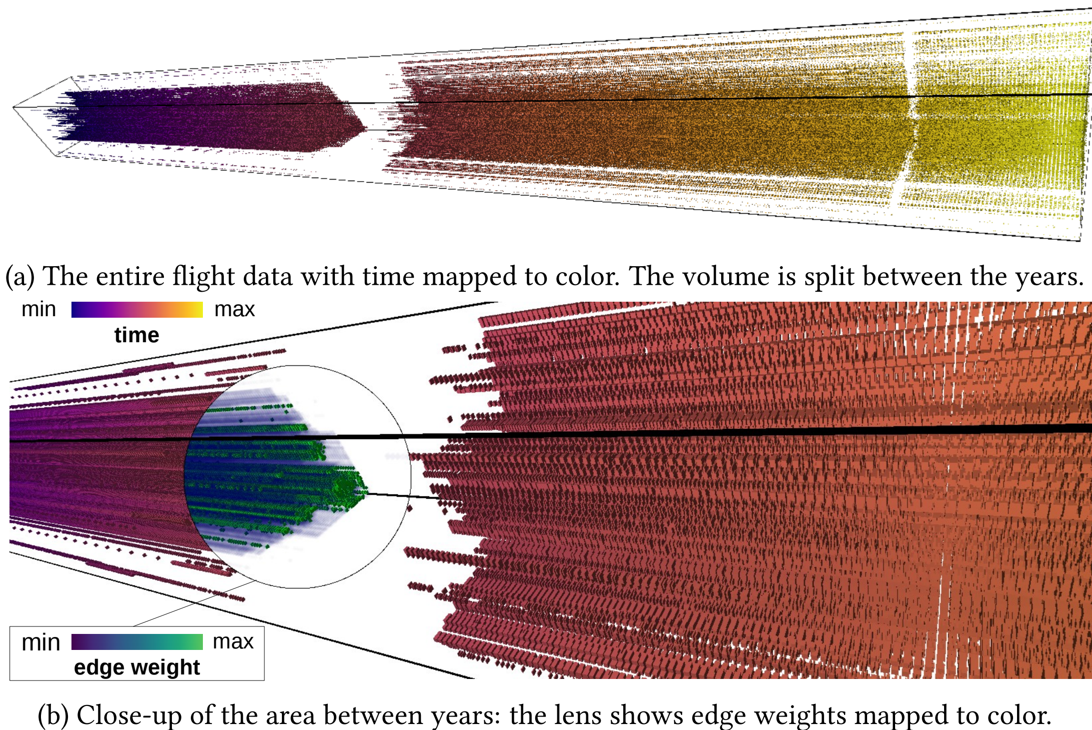
Flight Connection Analysis Using Difference Views

Our first application example is a dynamic graph representing the domestic flights in the United States during the years 2000 and 2001. The graph contains 234 nodes (airports) and more than 16 000 discrete time steps (one per hour). An edge represents a connection between two airports, the weight indicates the frequency. The aggregated amount of edges in every time step is depicted in the timeline plot (Figure 7.10). Here, a gap in September 2001 is immediately visible, which was caused by the 9/11 terrorist attacks and a shutdown of all air traffic in the hours following this incident. Furthermore, a regular pattern is noticeable that repeats on a daily basis, visualizing the day-night changes.

Figure 7.11a shows the complete graph in the volumetric representation, time is mapped to color. We applied the Cuthill-McKee graph ordering algorithm to change the node order based on the first 24 hours. To better see the temporal pattern and reduce visual artifacts, we selected aggregation level 2, i.e., 2^3 neighboring voxels are aggregated using the average weights. To investigate the changes between the year 2000 and 2001, the volume is split accordingly, revealing an increase of the edge count in the second year. The structure of the adjacency matrices shows that several new edges appear. Closer investigation reveals that some airports do not have any flights in the year 2000, leading to the conclusion that they either opened, or were only added to the data base in 2001, which seems more likely.

Figure 7.11b shows our lens: We overlay the visualization with another visual representation showing a different color mapping that highlights edges with a higher weight (frequency of flights on the same connection). We can infer from this visualization, that higher frequencies occur at larger airports in particular. Those get clustered in the bottom right corner by our reordering algorithm, because of the many connections to other airports.

A direct comparison of time sequences is of special interest in this application example because of recurring patterns. Using our difference view (Figure 7.12), we can visualize differences between several consecutive days of the dynamic graph. We select eight days around the 9/11 attack to further investigate the disruption of air traffic. The difference matrix (8×8 grid view) gives an overview of the coarse patterns: canceled flights (blue) and resumed ones (red) are immediately visible. To investigate the differences of two consecutive days more closely (e.g., Sunday and Monday), we can enlarge/reduce the view to only show those two in a 2×2 tiled view. We can infer from the detailed view that many of the connections are stable over the two days (transparent green). However, there are more flights on Sunday morning compared to Monday morning. In contrast, there are more flights scheduled Monday evening than on Sunday evening.



(a) The entire flight data with time mapped to color. The volume is split between the years.

(b) Close-up of the area between years: the lens shows edge weights mapped to color.

Figure 7.11: Volumetric representation of the U.S. flight data of the years 2000 and 2001.

Software Call Graph and Evolution Provenance Tracking

In the second example, we analyze a dynamic graph representing software calls of a drawing application written in Java. We use this example to demonstrate the utility of provenance information that is captured during an analysis session and their navigation as supported by our application.

The data set has been previously used by Beck et al. [26]. It contains 982 nodes representing functions of the application on the code level, 32 259 weighted and directed edges representing calls of these functions, and 1231 time steps that were recorded during the execution of the drawing application. Using a Java profiling tool, the following phases were captured: program start, creation of a new document, drawing a rectangle and a circle, and finally writing text into the circle. Weights of the edges represent execution times of the functions. Nodes of the graph are ordered according to the hierarchy of functions in the software project. Therefore, we did not apply matrix reordering to keep the semantics of this hierarchy.

We capture analysis steps performed by the user and store them as provenance information (see subsection 7.1.3). The evolution provenance is visualized in our tool as timeline plot to enable browsing through the history. Figure 7.13 depicts provenance

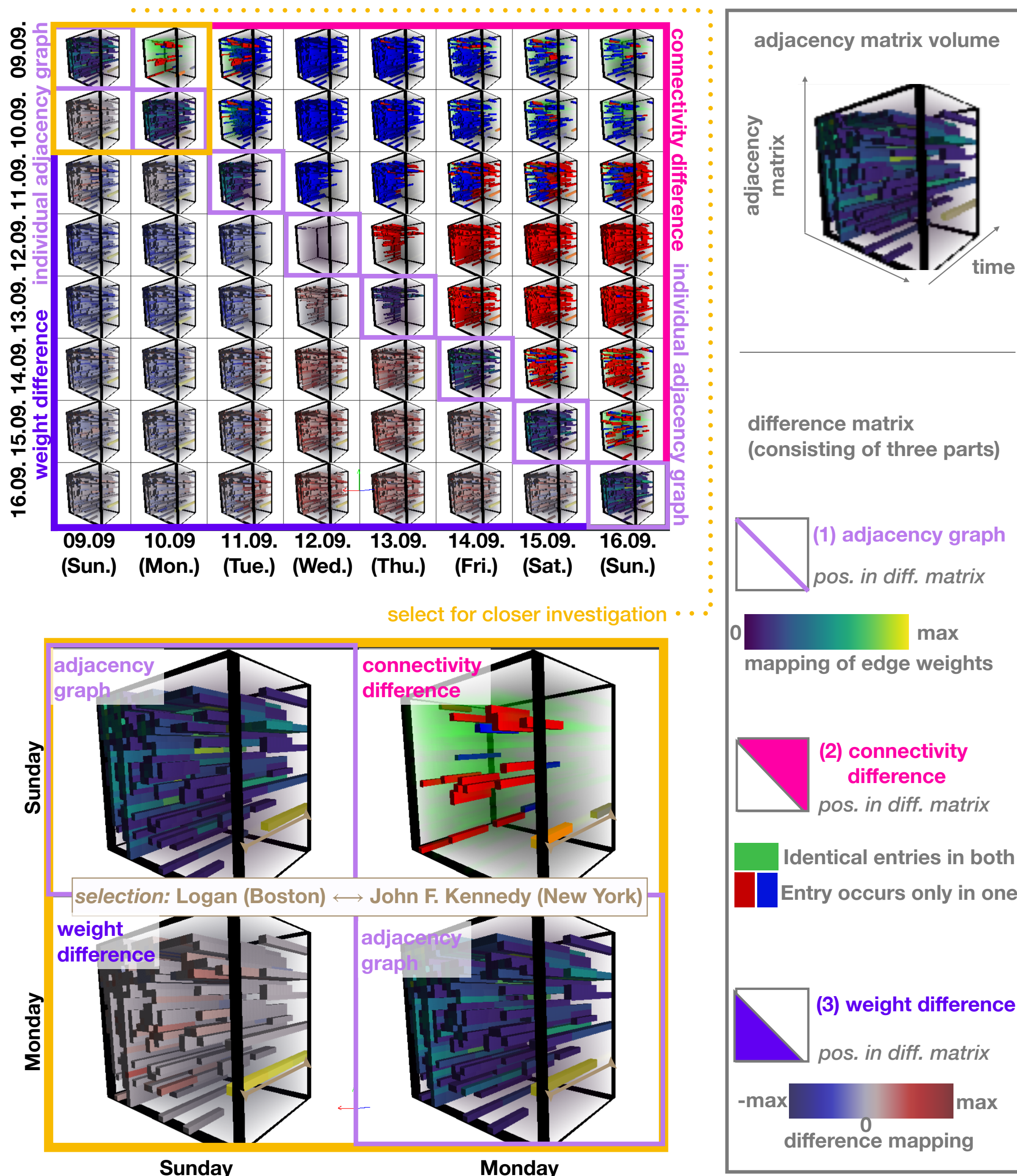


Figure 7.12: Top: Difference visualization matrix to investigate changes in temporal sequences of dynamic graphs. This instance depicts connections between U.S. airports in September 2001 (diagonal), with differences between days in the upper and lower triangles. Bottom: Changes from Sunday to the following Monday (enlarged difference view). Connections are largely similar (indicated in transparent green), but there are less flights in the morning on Sunday and more in the afternoon; e.g., flights between Boston Logan and New York JFK (highlighted in yellow/orange).

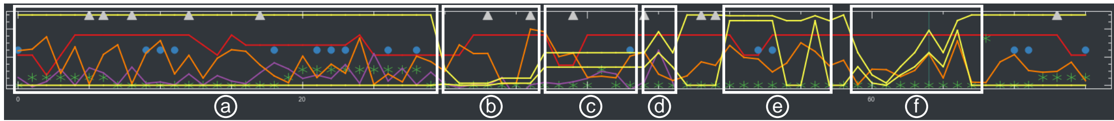


Figure 7.13: Evolution provenance plot for an analysis session of the call graph data set. Different time spans of the analysis sessions are annotated.

information captured during this analysis session. Three examples for sequences of analysis steps and the related thumbnails are shown in Figure 7.14.

In the provenance plot, we can see that the camera parameters are permanently adapted during the whole duration of the analysis session. This is not surprising since a 3D representation of data usually requires a lot of camera changes to circumvent occlusions. Further, the aggregation stays above a certain level during the whole analysis session. One reason for this might be the large size of the dynamic graph with around 1000 nodes and time steps leading to very small visual representations of the edges if no aggregation is used.

We partitioned the provenance plot based on the split parameter (yellow lines) that defines the sub-volume that is shown in the 3D visualization (see Figure 7.5). From the beginning to the middle of the sequence, the complete graph volume is analyzed (Figure 7.13a). The user initially adapted the aggregation level of the graph volume to his requirements, which is visible through changes in the red line. After they found a suitable setting, the user modified the transfer function, probably to highlight certain aspects of the data or filter out parts that are not of interest. To better understand these changes, an inspection of the thumbnails for these parameter changes provide a preview for the respective states. Figure 7.14a shows those thumbnails. The opacity for some of the edges is reduced to filter or emphasize edges with high weights (visible in orange).

In the next phase (Figure 7.13b), the user applied the volume partitioning feature to explore a short time span at the beginning of the data set. The respective thumbnails and one example state are shown in Figure 7.14b. In the plot, we can recognize that the aggregation level is increased in the beginning. After that, primarily the camera zoom is adapted. Two analysis steps were explicitly saved by the user indicating that they might have found something interesting. The thumbnail sequence shows how the sub-volume is selected and that the camera is adapted to a orthogonal view onto the volume similar to a single adjacency matrix. This is reasonable since the graph volume captures only a few time steps and the interesting aspects of the data might lie in the connectivity information.

Next, a larger sub-volume in the middle of the time range was selected for analysis. The provenance plot (Figure 7.13c) shows a similar user behavior as in the previous

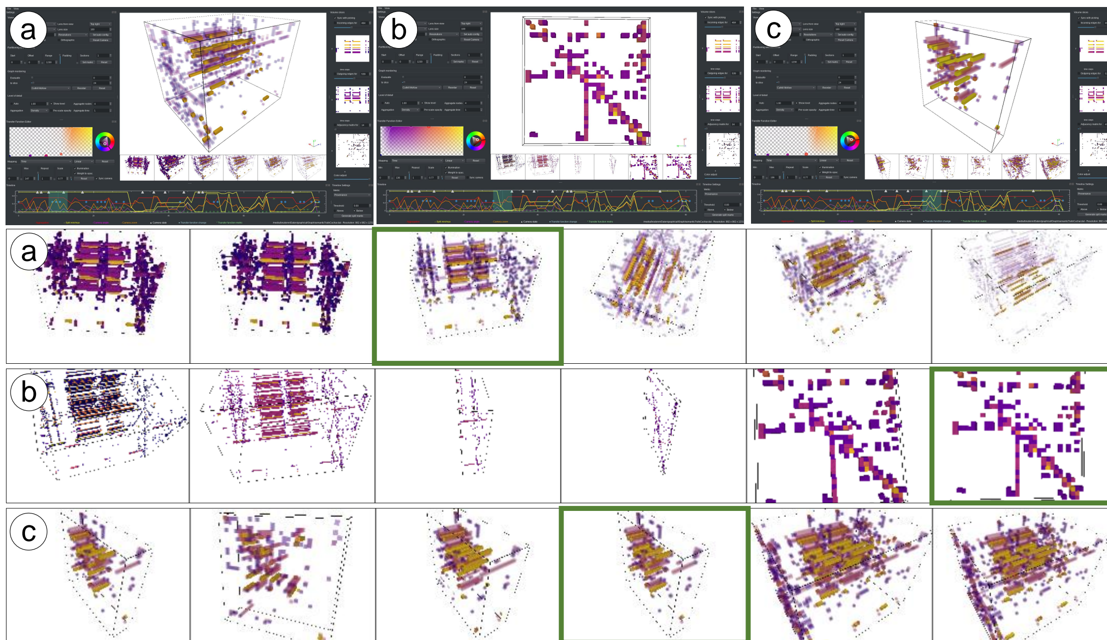


Figure 7.14: Selected analysis steps (top row) and the respective thumbnail list (bottom three rows) for the exploration of the call graph data set. The respective analysis steps shown in the top row are highlighted with a green border in the thumbnail lists.

phase: The aggregation level is changed first, followed by adaption of the camera configuration. The next short phase marked in the plot (Figure 7.13d) is interesting because the sub-volume is changed to a later time span and then the previous time span is restored (indicated by the peak in the yellow line). This indicates that the user visually compared these two time spans, since other parameters besides the camera settings were not changed. The thumbnails confirm this (Figure 7.14c), the two different sub-volumes are shown with the same visualization parameters. In the second to last of the marked phases (7.13e), a shorter time range at the end of the data set is analyzed. We can trace changes in almost all parameters (color mapping, filtering, aggregation level, camera configuration) as well as how the user switches between selecting the short time range and the remaining time range of the data set.

In the last annotated phase (Figure 7.13f), the user seems to switch between the different time spans they identified in the data set. An explanation for this might be that the user wants to recap what they found in the data set in different time spans.

Based on the provenance information, we summarize an exemplary analysis process of the call graph data. The illustration in Figure 7.15 shows the steps during investigation in a flow chart. The procedure is a typical top-down approach: starting at the overview, then splitting the graph into sections along the temporal domain, and finally investi-

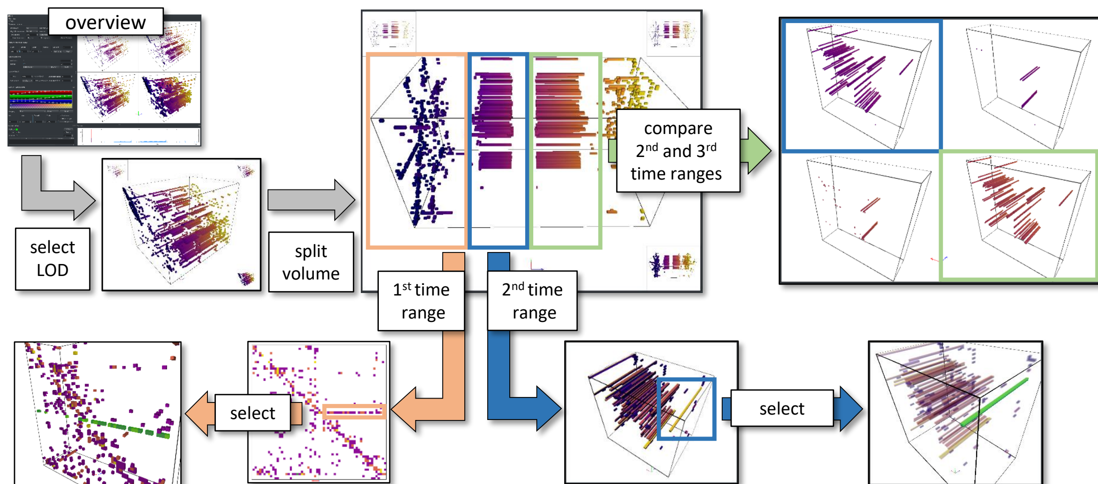


Figure 7.15: Summary of the different steps during the analysis of the call graph data set with our visual analytics approach. First, an adequate level of detail (LOD) is chosen for the temporal analysis of the graph. After that, the graph volume is split into different time ranges with prominent visual patterns and analyzed in detail. Besides selecting edges, the graph difference feature is used to compare two time ranges.

gating details and differences. Typically, a similar procedure should be applicable for different dynamic graphs using our system.

We can identify four different phases in the graph using the volumetric representation: a initial startup phase, two similar looking phases in the middle of the graph, and finally a shutdown phase. The initial and the shutdown phase are different from the other two in that they show many scattered edges (i.e., method calls). A closer investigation shows many edges along the diagonal, implying the call of neighboring functions in the code hierarchy. A direct comparison of two similar looking sections in the middle shows that they have almost the same edges but a different overall time span. a direct selection of the differing edges reveals that both phases represent a drawing operation but one is a rectangle figure, the other one an ellipse.

7.1.6 Expert Evaluation

Our two use cases show how trained users can successfully use our techniques and system to analyze large dynamic graphs. To gather qualitative feedback on how untrained visualization researchers can work with our techniques and system, we conducted a think-aloud user study. We asked five visualization experts from our institute (three with an information visualization and two with a scientific visualization background) to solve several data analysis tasks. While all of them are familiar with the general

concepts of graphs and their visualization, most of them have other research topics.

We chose this type of evaluation because a fair comparison of our system to other techniques for large dynamic graph visualization is difficult. This is because we propose an integrated approach that features different data views, comparison, and aggregation and filtering, besides the volumetric view. However, since our components were designed to complement the 3D volumetric representation, they are not directly portable to other techniques or less useful in combination with them. We have refrained from performing a quantitative study with time and error measurement because of two reasons. First, statistical meaningful results need a larger amount of participants. Second, it is difficult to find specific measurable tasks that can meaningfully represent the whole analytics process with our system.

Study Design

At the beginning, we gave a structured demonstration of the functionality and how to use the features of the system, followed by a short learning period. We used the flight data set for this purpose, the software call graph was used as the data set for the analysis tasks. The participants were given a short introduction to the data set, including the program type, the meaning of the weights and a high-level overview on the sequence (program start, user interaction, program close). No further details about the program execution and user interactions were provided. The study took about one hour on average, including demonstration (about ten minutes) and the learning period (about five minutes). While the experts were working on the tasks, we recorded the screen and spoken comments of the participants. Further, we tracked the mouse position to determine which view or widget was being used, and saved the provenance traces that were automatically generated by our system. Finally, we asked the participants to briefly summarize their results in a questionnaire. We asked the experts to solve six tasks:

1. Select a suitable aggregation level to gain an overview of the entire graph.
2. Identify time spans with different behavior.
3. Partition the graph into those time spans and investigate them.
4. Identify two equally looking time ranges and validate whether they are different.
5. Describe a given provenance trace.
6. Compare their own workflow against the one shown by the provenance trace.

The participants were free to use any of the available views and features of the system. We chose specific tasks because they had a higher chance of providing comparable results with our small group of participants. Furthermore, we wanted to test if our proposed workflow (Figure 7.15) is suitable for visualization experts that are not familiar with the data set to get insights into the dynamic graph.

Table 7.2: Usefulness of Components

Component	Mean
Timeline view	4.4 ± 0.8
Volume view	4.8 ± 0.4
Slice views	3.0 ± 1.1
Partitioning	4.2 ± 0.8
Difference view	4.8 ± 0.4
Provenance graph and views	3.6 ± 1.4
Overall system	4.4 ± 0.5

Scale from 1 (not helpful) to 5 (very helpful).

Results

We asked the experts to rate the visualization components of our system on a Likert scale from 1 (not helpful) to 5 (very helpful) with the option to give no rating. The participants could also write free-text comments and suggestions for improving the system and visualizations. Average ratings of our system's components are listed in Table 7.2. The overall system was rated very helpful, with the volume view and the difference view being rated the most helpful components for the tasks.

Table 7.3 lists the results of the mouse tracking during the user study. We captured the mouse positions in half-second intervals and increased a counter for the respective widget below. The relative usage was calculated by normalizing those counters at the end of the session. As can be seen, the volume view was interacted with the most. The timeline view was the second most used widget that was also rated (very) helpful by most participants. In terms of usage, adjusting the settings such as the aggregation level follows next, while the slice views and the transfer function editor were used less or not at all by the experts. Only one of the participants used the transfer function editor (for filtering low weights). To improving this widget, one of the experts suggested easily accessible, pre-defined transfer functions and showing the editor only on demand. Both, the user rating and the widget usage are influenced by the fact that the volumetric graph representation is the central and most prominent component of our system, whereas the additional components and features were designed to support and complement this technique. Therefore, the high relative usage of the volume view (almost 60% on average) as well as the positive rating of the component (4.8 ± 0.4) could be expected.

Suggestions for improving visual representations an usability included tool-tips for different terminology and sharp borders between the slice views for better separation. Further, one expert suggested improving the slice views with a separate aggregation levels and a zoom functionality to improve their helpfulness.

Table 7.3: Relative Usage of the Widgets

Widgets	Average usage (%)
Timeline view	16.6 ± 5.1
Volume view	58.5 ± 3.6
Slice views	7.4 ± 6.1
Transfer function editor	2.1 ± 3.6
Settings	15.0 ± 6.7

Regarding task 1, the participants selected aggregation levels between 2 to 4 because they yield desirable properties: "visual appeal, not too much clutter, still visible details". Several experts adjusted the aggregation level during the completion of the other tasks, as exemplified in the provenance graph (Figure 7.16). This confirms the utility of our aggregation feature and its real-time adjustability. In tasks 2 and 3, the experts' selections generally match our own partitioning of the graph (Figure 7.15). All experts could identify and interpret the coarse structure of the graph using our system. The main differences to our own analysis were that two experts created a finer partitioning and one a coarser by combining the two sections in the middle. The participants could identify several details and assumptions were made about the characteristics of the program flow, although the participants were only given a high level description of the data set. For instance, several experts pointed out the initialization and de-initialization phases and they were able to identify user interactions.

Regarding task 4, which required the experts to compare two equally looking sections, three of the five participants compared parts similar to the 2nd and 3rd time range in Figure 7.15. One validated the periodicity within the 2nd time range, while another one compared the sequences right before and after those two blocks. According to their feedback, the difference feature proved very helpful for this task. In the last two tasks, all experts were able to comprehend the example workflow we had provided them and spot differences to their own provenance trace. This included differences in temporal partitioning of the graph data, aggregation levels, and the use of the transfer function for filtering.

Figure 7.16 shows a sample provenance graph that was recorded for one of the participants. Identifiable is the change of the aggregation level (red curve) in the beginning, as required by the first task. Afterwards follows a detailed investigation of seven identified sections of the graph as indicated by the yellow curves (task 2 and 3), concluding with the investigation regarding the differences (task 4). Notably, this expert did not change the transfer function or the mapping metric during the whole time and adjusted the aggregation level several times during the analysis task. Compared to our example analysis of the same data set in the previous section, the participant made a more fine-

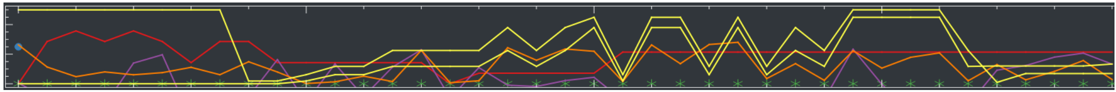


Figure 7.16: Provenance graph of one of the experts, generated during the user study.

grained separation of sections in the temporal domain. However, the overall separation matches ours.

Overall, the visualization experts typically not working with graph data were able to successfully analyze an unknown dynamic graph data set using our system. The experts came to similar results in their analysis and were able to detect major structures in the data. While we could not directly quantify the effectiveness of our approach, the expert feedback indicates its utility for the analysis of large dynamic graphs.

7.1.7 Future Directions

Besides implementing the suggested improvements for our visualization components and interactions, there are two promising directions to continue this work. The first one is addressing the technical scalability limitation, i.e., using compression or out-of-core techniques to circumvent the current data set size limit that is bound to the available VRAM. The second direction is to improve the integrated support for evolution provenance, which turned out to be very helpful during data exploration. Possible extensions and refinements include the addition of user comments or bookmarks, using a heuristic for automatic recording of interesting camera states, and suggesting new parameter setups based on the evolution provenance.

7.2 Space-Time Visualization of Gaze and Stimulus

Gaze data recorded from multiple participants watching dynamic stimuli, such as videos, poses a challenge for eye tracking researchers. Complex spatio-temporal patterns that might appear in the data are hard to capture with statistical methods alone and often require visual support for (1) explorative data analysis, (2) displaying statistical results, and (3) the illustration of the results.

Established visualization techniques such as gaze plots and heat maps are limited for these purposes because they require animation to represent changing gaze patterns over time. In contrast, a static overview of gaze data from videos that conveys important contextual information allows for an efficient navigation in the data. A space-time cube (STC) representation of the data is presented that uses GPU volume raycasting at its core (subsection 2.2.3). By applying multiple transfer functions, we can combine data aspects for filtering and emphasizing important regions and time spans in the data.

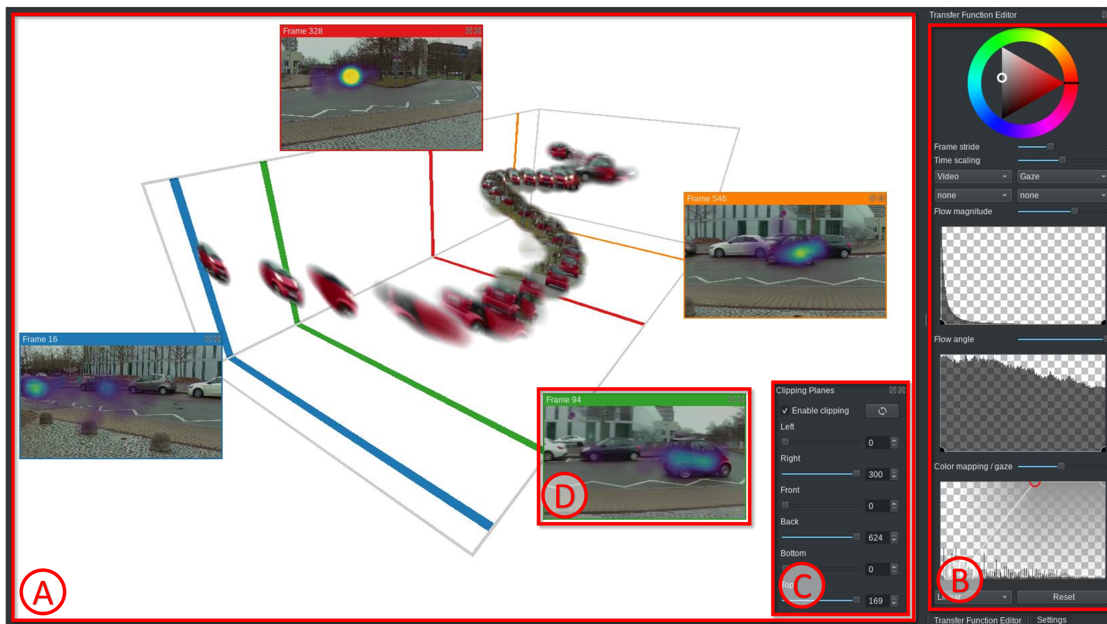


Figure 7.17: Our application combines: (A) the main view with the space-time cube visualization, (B) a transfer function editor for filtering and coloring that also shows data histograms, (C) an editor to adjust clipping planes, and (D) the visualization of selected frames as annotation that also shows the gaze heat map.

With this visualization approach, multiple space-time volumes (video, optical flow, gaze) are combined into a spatio-temporal overview that conveys gaze patterns as well as information on what caused these patterns. To this end, specifically designed transfer functions are presented that reveal different aspects in the data. The applicability of the approach is demonstrated on various videos with gaze data from multiple participants, using a performance optimized implementation.

7.2.1 Method

To explain our approach for combined analysis of gaze and stimulus data, we first discuss the visual design, followed by three core aspects (Figure 7.18): (1) data pre-processing, (2) volume rendering, and (3) interactive data exploration.

Visual Design

Figure 7.17 shows an overview of our system. It features the main view with the STC visualization (A), controls for filtering and highlighting parts of the data (B, C), and the possibility to annotate important frames (D). The two data sources for our visualization are video frames and gaze positions from multiple participants. Each data

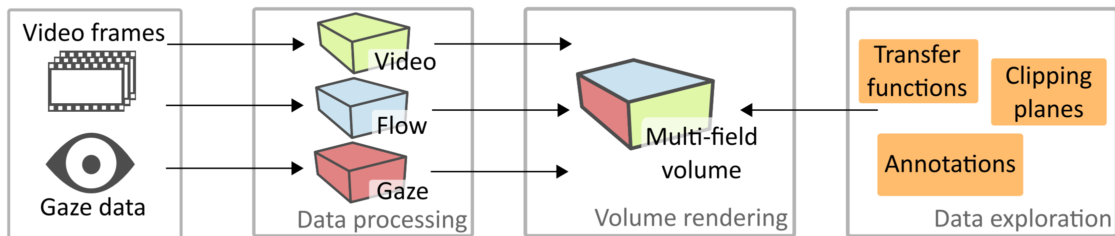


Figure 7.18: The stimulus video and the recorded gaze data from multiple participants is processed to derive three volumes containing spatio-temporal information of the data. Rendering is performed with a multi-field approach, combining the volumes. Interactive data exploration is supported via transfer functions, clipping planes, and annotations for the temporal dimension.

source can be investigated separately in the STC as depicted in Figure 7.19. The raw video volume provides an overview of motion at frame borders. This corresponds to slit-scan visualizations that are used to summarize a video [101]. However, content inside the volume is occluded. Figure 7.19 also shows the aggregated gaze data rendered as a STC. While this visualization provides an overview of the gaze distribution, there is no direct link to the video content. To make this connection visible, a combination of the two data sources is necessary. Such a combination can be represented as a dense volume or slices, to reduce occlusion and reveal more details (Figure 7.19).

To support effective analysis of the combined data sources, pre-processing is required to transfer the data into a unified multi-field volume. To render the volume interactively, we use GPU-accelerated raycasting (subsection 2.2.3). For appropriate representation of important gaze patterns, we support interactive data exploration. Thereby, a key aspect is the manipulation of transfer functions to change the visualization based on different aspects in the data.

Data Pre-Processing

To yield real-time rendering performance for interactive analysis, we convert the data into dense volumes in a pre-processing step.

Video Volumes. A video volume contains all visual information of the stimulus, and interaction methods from volume visualization can be employed. The spatial plane depicts the video frames, clipping the volume along the time axis emulates a video replay. The side planes of the volume represent slit-scans of the video. Adjusting the clipping for these planes provides the corresponding slit-scan that can show important regions and events of the video.

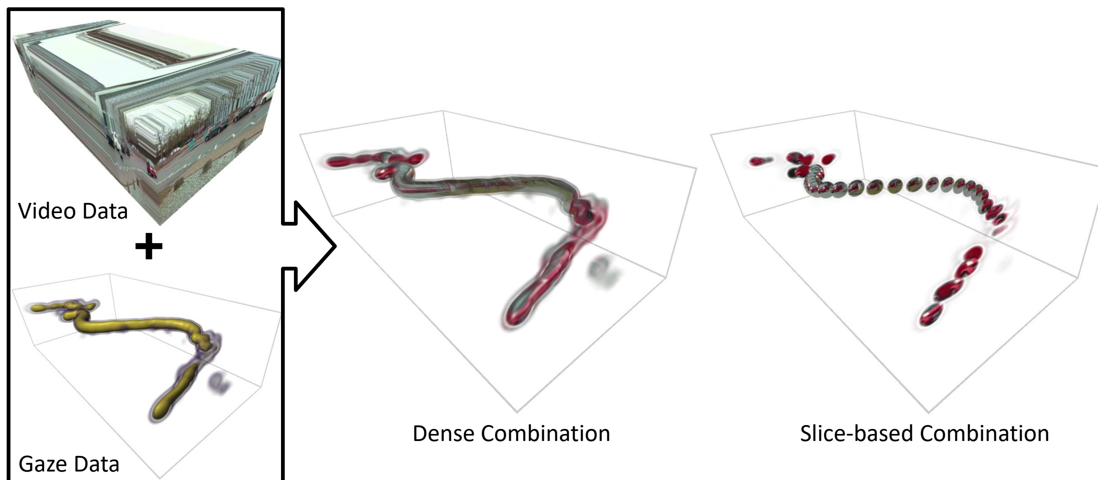


Figure 7.19: Example renderings of our data input (gaze density and video frames) as volumes and a combination thereof.

The investigated stimuli consist of n frames, often in typical multimedia resolutions, e.g., 1920×1080 pixels. Including the temporal dimension in the data significantly increases the amount of memory necessary to hold such data. Hence, a reduction of the visualized data becomes necessary. For the representation of the data as a volume, the temporal resolution is more important because it allows depicting longer sequences, while the spatial resolution can be reduced without drastically changing the overview of the data set. However, with a low resolution the image content becomes blurry and details can be missed. We found that processing the videos for all frames with a width between 200–400 pixels and a height adjusted with respect to the aspect ratio of the video provides smaller volumes with enough details to interpret the stimulus. Furthermore, the ratio between the spatial and the temporal resolution increases with lower spatial resolutions, leading to elongated, less compact STCs, which might require additional scaling of the temporal axis for a better overview. The data is stored as RGBA unsigned characters in raw data files. This format is compatible with most applications for volume rendering. Note that the alpha channel in this format could be used to also store the gaze volume at the cost of precision.

Optical Flow. The optical flow for a video sequence describes how individual pixels move between two consecutive frames. We apply a variational method [37] that provides a dense vector field of absolute displacement for image pairs in the video sequence. For a more intuitive interpretation, the values are converted to angle and magnitude of the vectors. For the filtering of motion regions, pixel-precise accuracy of the flow is less important. Hence, a trade-off between flow precision and computational performance can be made by reducing the number of iterations for the applied approach. We store

Table 7.4: Data Formats

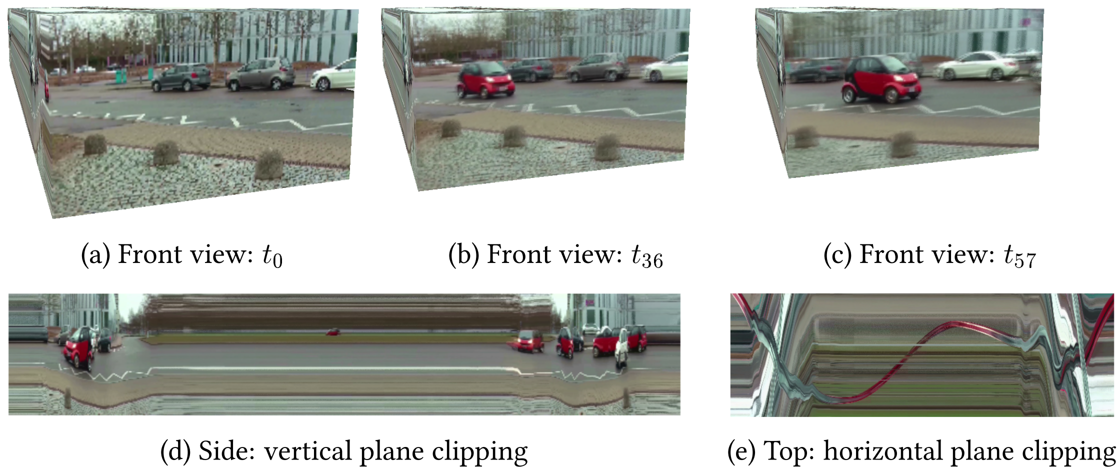
Volume	Format	Channels	Content
Video	UCHAR	4	RGBA values of the video frames
Optical Flow	FLOAT	2	Angle & magnitude of the displacement vector field
Gaze	FLOAT	1	Gaze density based on kernel density estimation

the angle and magnitude as single precision floats in two separate channels. The normalization factor of the magnitude can be dynamically adjusted during runtime according to the needs of the analyst and specifics of the data.

Gaze Volumes. Heat maps are a common visualization to represent aggregated gaze data. The aggregation can be calculated over time and/or for multiple participants. For dynamic stimuli, it is necessary to provide a dynamic heat map that conveys the changes of gaze patterns over time. To achieve this, we apply a sliding window approach that respects temporal coherence by summarizing gaze points from the current frame and $m \in \mathbb{N}$ previous frames. For the heat map calculation, we apply an Epanechnikov kernel [164] for an efficient approximation of a Gaussian kernel. The kernel covers 10% of the frame height, which roughly corresponds to the foveal area that was covered at a distance of 65 cm showing the videos with a resolution of 1920×1080 pixels on a 24" screen. As with the optical flow, the applied techniques are interchangeable according to the requirements for precision and performance. The data is stored as single-precision floats without normalization. Again, the normalization of the data can be adjusted in the rendering process, allowing the analyst to change the heat map dynamically, depending on the task. Table 7.4 summarizes the data volumes and how they are stored for volume rendering.

Volume Rendering

We apply multi-field volume rendering to depict the three spatio-temporal volumes. To enable interactive exploration of the data, even when rendering large sequences with several thousand frames, we accelerate the compute-heavy calculations by using parallel processing on GPUs. For this, we use OpenCL that allows for cross-platform execution and device portability. We also take advantage of texture units integrated in GPUs for their interpolation capabilities. For this, we process the video volume, optical flow, and gaze data as a 3D textures. We implement direct volume rendering by using front-to-back raycasting, including early ray termination (ERT) and empty space skipping (ESS) (see subsection 2.2.3).



(a) Front view: t_0 (b) Front view: t_{36} (c) Front view: t_{57}

(d) Side: vertical plane clipping (e) Top: horizontal plane clipping

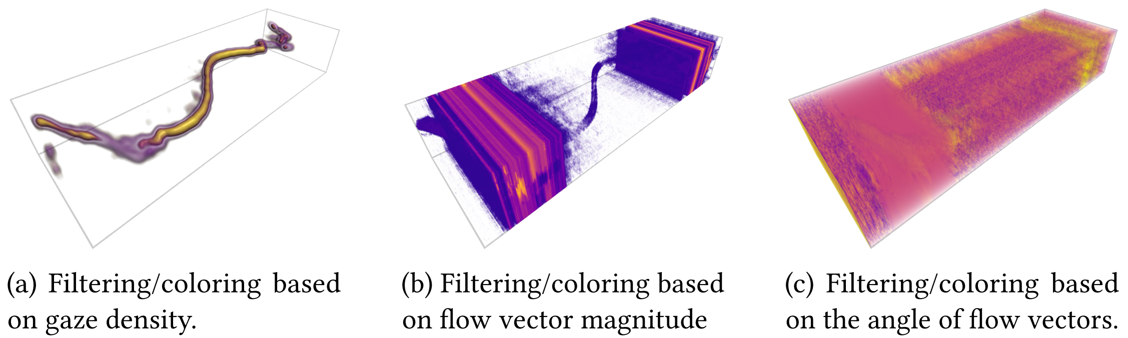
Figure 7.20: Volume Clipping along the three dimensions. (a)–(c) Temporal clipping emulates a playback of the video. (d) Clipping the volume with a vertical plane results in slit-scans that depict objects whenever they moved through the scene. (e) Similarly, clipping with a horizontal plane reveals motion patterns.

Multi-Field Rendering. In case of the video data, the sampled RGB colors are directly used to determine the color of the pixel. Density and optical flow data (angle and magnitude) are interpreted as three distinct scalar fields, their samples are evaluated using transfer functions. We support three transfer functions, one for color and opacity values and two for transparency only. If multiple transfer functions are employed at the same time, the opacity values are composited into one final opacity value. The sequence in which the functions are applied to the data (and if at all) can be dynamically adjusted by the analyst.

Slice-Based Video Context. The dense nature of the data makes it hard to make out the content of single frames, especially if little transparency is used in the mappings. To counter this, we support rendering only a subset of frames at regular intervals, where the density can be dynamically adjusted. In Figure 7.17 for instance, a stride of 15 is used to show only the data of 42 of the 624 frames, making it possible to see most parts of the content of the rendered frames.

Interactive Data Exploration

The presented multi-field approach allows us to filter specific parts of the data and to emphasize regions in the STC that are relevant for the research question at hand. For example, one could only be interested in the parts where participants looked at moving objects. Combining our three data properties, such a query can be modeled by combining transfer functions. Further, the application of clipping planes and timeline



(a) Filtering/coloring based on gaze density.

(b) Filtering/coloring based on flow vector magnitude

(c) Filtering/coloring based on the angle of flow vectors.

Figure 7.21: Different transfer functions for optical flow and gaze density applied to the same data set. Two camera pans and the moving car are clearly visible in the flow data, the gaze data shows that participants follow the car's movement accurately.

annotations provides means to explore the data and create supportive illustrations. Finally, basic interactions along the temporal domain are supported, such as scaling the data along the axis and showing only every n-th frame.

Clipping Planes. Clipping of the volume can be performed individually for each dimension. If clipped along the z-axis, the volume depicts how the video content changes over time, i.e., this corresponds to a playback of the video (Figure 7.20a–Figure 7.20c). If clipped along the x- or y-axis, the volume depicts individual slit-scans [112, 101] that summarize all motion over time at the clipping border, acting as a scanline. This helps to identify when an object appeared in the video (Figure 7.20d) or how it moved (Figure 7.20e).

Transfer Functions. Transfer functions determine the visual mapping of voxels to values such as color and opacity. A common approach to transfer function design is to select value ranges and their corresponding opacity based on a 2D histogram. For example, values that correspond to the hue of the sky in a video volume can be set to appear fully transparent to remove one important area that often occludes interesting details. This corresponds to chroma keying techniques known from visual effects in video production. The flow data also contains information that helps filter the data further. The histogram of the flow vector magnitude is helpful to remove areas without motion, analog to the previous example, regions such as the static sky can be masked out this way. Furthermore, camera panning motion can be removed by appropriate filter settings (Figure 7.21b). Filtering the gaze data by its density allows us to highlight hotspots of attentional synchrony where the gaze density is high (Figure 7.21a). Regions with lower density values can also be emphasized, which is usually of interest if multiple regions attracted attention, or if the gaze data is dispersed.

Table 7.5: Example Videos with Gaze Data

Video title	Duration	Figure
Car Pursuit	0:25 min	7.17, 7.19, 7.20, 7.21
Kite	1:37 min	7.22
Thimblerig	0:30 min	7.23
UNO game	2:01 min	7.24

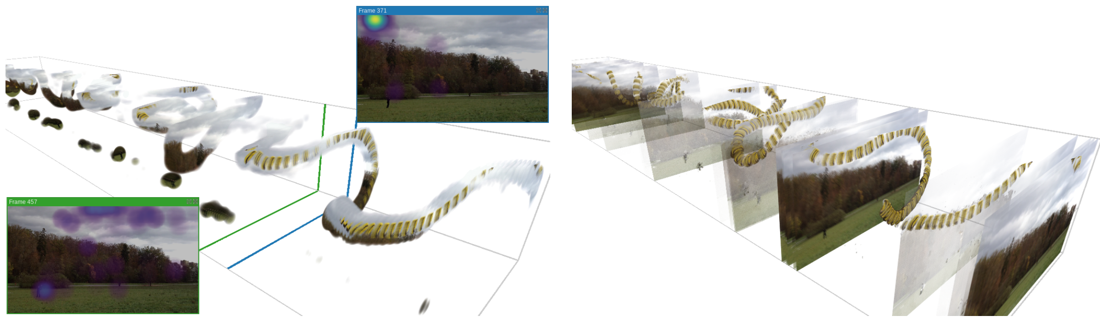
Annotations. Filtering the volume with appropriate transfer functions provides a better overview of the data set because of the local stimulus context, compared to visualizations without stimulus information. However, for the illustration of results, the global context, i.e., the whole video frame and the corresponding heat map are beneficial. Hence, we adapt the idea of annotations for narratives of historical events in the STC from [103], who annotate events in a geo-spatial context by pictorial labels to summarize important events. We support such an interactive labeling of individual time steps (i.e., video frames). For this, the analyst can simply click on a frame to select in the STC. This triggers the generation of a hovering window containing the frame with a colored border that matches an also generated marker on the sides of the volumetric view that highlights annotated frames (Figure 7.17D).

7.2.2 Examples

We apply our technique to different videos from a publicly available data set [110]. Table 7.5 summarizes the video examples. All data was recorded in a user study with 25 participants using a Tobii Pro T60 XL with a stimulus resolution of 1920×1080 pixels. The *Car Pursuit* video is shown in the previous sections to illustrate concepts of our technique. It depicts a red car driving from the left side of the screen to the right and back. The video contains two horizontal panning motions at the beginning and the end to adjust the field of view. Participants were asked to follow the car, leading to smooth pursuits and attentional synchrony [138].

Kite

In the *Kite* video sequence (Figure 7.22), a person steers a yellow kite that the participants were asked to follow. The kite leaves the recorded field of view several times during the sequence. Filtering out low gaze density from the video data reveals patterns and outliers, thereby providing an overview (Figure 7.22a). Mostly, participants follow the path of the kite smoothly, when it is visible. However, if it leaves the field of view, some participants try to estimate the path outside the view and predict the spot where the kite reenters into the video. Other participants focus on the person steering the kite on



(a) Frames with high gaze density. Selections before and after the kite leaves the field of view.

(b) Frames with high motion. Small camera pans are visible (not filtered out).

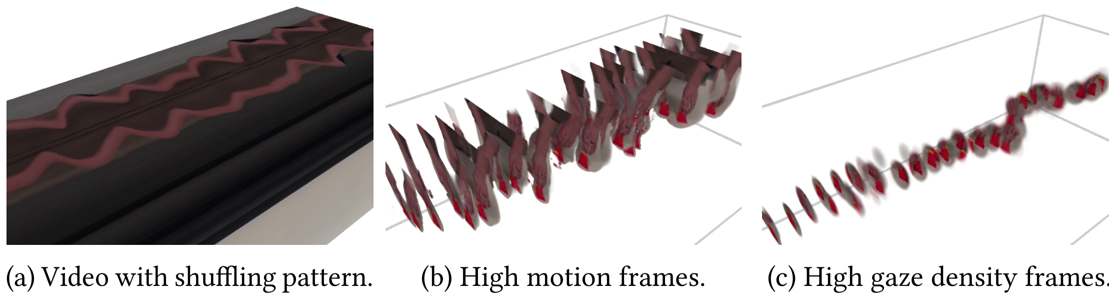
Figure 7.22: Video of a person steering a yellow kite. Filterings based on gaze as well as flow are applied to highlight different aspects.

the meadow. Figure 7.22b shows the video while using our technique to filter based on the magnitude of the flow vectors (i.e., removing low magnitudes). This rendering also reveals the path of the kite but also highlights the sections where the kite leaves the field of view. Slight camera pans in the data appear as fully visible frame slices but could be filtered out using another pre-processing step. This setting also shows that the person on the ground is hardly moving (only slight indications), making it hard to identify the person as a potential area of interest (AOI) solely based on motion information. The example shows that using both flow and gaze for filtering is advantageous over visually identifying AOIs.

Thimblerig

In our second example, we apply our technique to a video showing a hat game (thimblerig). The participants were asked to follow one of three hats that hides a marble underneath while they are being shuffled. Figure 7.23 shows renderings of the video data, which contains 749 frames, with different configurations. Looking only at the video data without any filtering applied (Figure 7.23a), the shuffling pattern is roughly visible.

Filtering out regions with low flow magnitude yields a concise overview of the shuffling patterns. They can be enhanced further by using our slice-based video context view that regularly skips several frames (Figure 7.23b). Alternatively, we can filter out regions with low gaze density (Figure 7.23c). This reveals that most participants followed a single hat—the one hiding the marble. Investigating the frames before the hat with the marble is lifted reveals that most participants were successful in following the hidden object. Comparing the two filterings (gaze and motion) shows that gaze is directed by motion.



(a) Video with shuffling pattern. (b) High motion frames. (c) High gaze density frames.

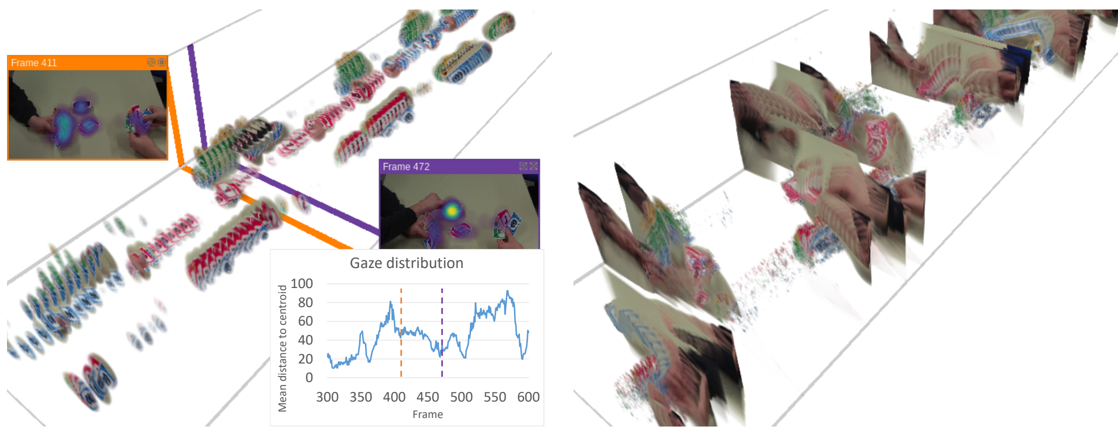
Figure 7.23: Video showing a hat game with and without filtering applied for different properties. Participants were tasked to follow the hat hiding a marble.

UNO Card Game

In this example, we apply our technique to a video of two people playing the UNO card game. During each player's turn, participants were asked to focus on playable cards in the hands of the players. Filtering the video data by omitting regions with low gaze provides a good overview of the major patterns (Figure 7.24a). For instance, it is clearly visible when the majority of the participants looked either at the cards of the right or the left player. Typically, attention on one of the players is followed by gaze on the discard pile, then on the opposing player. This seems to be the main pattern in the gaze data. Participants follow the card put on the discard pile and look at the other player's hand next to anticipate the upcoming move. Visualizing the gaze data in the STC also reveals four time spans where the participants focus on the draw stack. Further investigation shows that all of these are related to events where players have to draw new cards. This can be visualized by filtering out regions with little motion (Figure 7.24b). By using this filtering, one can reconstruct which player had to draw cards and even how many.

A close comparison of flow magnitude filtering that shows only parts with large motion and filtering with respect to high gaze density reveals aspects of interest. For instance, after the player on the right hand side plays the red card 1, the opponent cannot play a valid card and is forced to draw from the stack. Figure 7.24a shows that participants who watch the video react to this event differently. Some anticipate the draw as the next action, which is visible in the gaze visualization in that it shows attention on the draw stack before the motion of the hand begins, as can be seen in the flow visualization. However, most of the participants seem to follow the motion of the hand to the draw stack. This indicates that a few participants followed the game attentively and are able to anticipate the next move correctly before it happens. The attention of the other participants who follow the hand movement are probably drawn by the motion because they did not follow the game carefully or are not fully aware of the rules.

Our approach is also well suited to support statistical measures with illustrations. As an



(a) High gaze density, frames before and of the first draw action are selected.

(b) High motion data, the plays of the two payers are clearly visible.

Figure 7.24: Two persons playing the UNO card game visualized with our technique showing gaze and flow magnitude. The plot shows the mean distance of the gaze to the centroid for a selected time range. Selected frames are marked as vertical lines.

example, one can calculate the mean distance of gaze positions relative to the centroid over time, which is an indicator for attentional synchrony if values are low [111]. The stimulus context is not directly available if the mean distance alone is used. However, such a measure can be applied to segment respective time spans. With our STC approach, we complement the measure to directly depict what happened in the stimulus that caused the changes in the values. As an example, Figure 7.24 shows such a gaze distribution plot for a range around the selected frames.

7.2.3 Discussion

The examples in subsection 7.2.2 demonstrate the usefulness of our approach especially for gaining a combined overview of video, gaze, and flow data. In the first example, investigating the Kite video, our volume STC rendering applied with a gaze filter yields a concise overview of the participants' gaze distribution across the whole video. For example, the analyst can directly see how participants follow the kite, and that their focus shifts to the person steering the kite when it leaves the field of view. Further, it is possible to make out details in our visualization that would be obscured by a traditional heat map approach. Starting at the overview, the analyst can easily pick single frames for further investigation or compare the gaze to the optical flow vectors, e.g., movement patters as also demonstrated in the second example (Thimblerig). The flow data also contains camera pans that clutter the visualizations of some of the videos (Car Pursuit and Kite). However, they could be automatically detected and removed with further

pre-processing. The UNO card game example in particular demonstrates how much detail can be shown in our STC visualization. A combination of filtering with gaze and flow data reveals major game moves as well as gaze patterns.

Filtering out specific angle ranges of the optical flow vector field did not reveal prominent patterns in the examples. However, we anticipate that this could be useful for analyzing other data sets, especially in combination with the filtering based on the magnitude of flow vectors. For instance, it could be used to filter for objects moving only in a certain direction such as a person walking from left to right while others move from right to left.

7.2.4 Future Directions

The adjustment of transfer function could be further improved. For instance, by adding presets for common tasks such as the identification of attentional synchrony, smooth pursuit of objects, or areas with high dispersion. Another direction for future work would be the addition of pixel-precise AOI labels into the volumetric representation. This would allow analysts to filter the volume for regions where a specific AOI is visible.

Chapter 8

Conclusion

This thesis addresses challenges of performance quantification in the context of visualization systems. A visualization system describes a visualization algorithm running on a compute architecture or device. By taking a holistic approach, a better understanding of performance in the visualization domain and novel techniques to improve performance in selected scenarios were achieved. The focus was on three aspects:

Performance evaluation. The proposed evaluation techniques and best practices provide insights and well-founded suggestions on how to improve performance evaluation for interactive visualizations.

Performance modeling. Contributions to performance modeling and prediction in the context of scientific visualization allow for real-time parameter tuning to keep a frame rate or limit data throughput. Techniques to balance rendering load between devices in shared and distributed memory environments are proposed.

Performance optimization. The technical contributions include approaches for performance optimizations based on foveated rendering and performance optimized, non-conventional volume rendering techniques that support interactive exploration of large data sets of abstract data.

In the following, the thesis is summarized chapter by chapter. An overarching discussion provides an evaluation of the presented work in the context of the central research questions posed in chapter 1. Finally, overarching directions for future research are sketched in an outlook.

8.1 Summary

This thesis contributes work on performance quantification of visualization systems. The topic is addressed from various aspects: performance evaluation, modeling, and optimization of visualization applications in the context of foveated rendering and volume rendering.

Runtime Performance Evaluation In chapter 3, the current approach on performance quantification in scientific visualization was investigated by reviewing respective efforts in related work and comparing these methods to an extensive measurement approach. For this, volume rendering and particle rendering were considered as two established fields in scientific visualization. A large systematic series of performance measurements was conducted for the two algorithms, followed by an in-depth statistical analysis of the measurement data that showed several characteristics and commonalities between the two rendering techniques. Based on those, a list of best practices was compiled that can act as a guideline for empirical performance evaluation of interactive scientific visualization techniques. Another approach was presented for the performance evaluation and comparison of rendering algorithms, focusing on supporting the visual analysis of runtime performance differences, in particular in the context of distinct camera configurations.

Performance Modeling for Runtime Optimizations on GPU Systems In chapter 4, an integrated approach for on-the-fly prediction of rendering performance of a volume raycaster was presented and used for load balancing as well as dynamic tuning of the sampling resolution. Computational load can be distributed among different devices to substantially reduce lags and jerky motions during interactive exploration. To overcome such unpleasant effects, methods were proposed to explicitly assess the impact of acceleration techniques on the raycasting performance, including a novel technique to estimate the effect of early ray termination. Based on those methods, a hybrid performance prediction model was presented that is capable of predicting accurate frame execution times on-the-fly. The model consists of two parts. One used the assessed acceleration data together with general information on the data set and sampling density for an analytical depth estimation. This was combined with a second part, an estimate of the cost per sample using a machine learning technique. The usability of the model was demonstrated by means of two use cases. The first one adjusted the sampling density in ray and image space to reliably meet user defined performance requirements. In the second one, a prediction-based load balancing among multiple GPUs was conducted with the goal to consistently maximize hardware usage and thereby improve rendering speed and image quality.

Performance Modeling for Runtime Optimization and Cost Savings on Distributed Memory Systems In chapter 5, different performance models for distributed memory systems were introduced. The central one was a cost efficient hybrid in situ visualization approach—dynamically combining inline and in transit visualization—for creating Cinema-style image databases. Substantial speedups compared to pure inline and in transit approaches were achieved by addressing four types of inefficiencies in in situ visualization. The rendering work was distributed dynamically between nodes to tackle load imbalances caused by heterogeneous rendering costs. Compositing of partial images, a task with scalability issues, was only executed on the comparably small subset of visualization nodes. Transfer overhead aws hidden behind task processing, i.e., rendering overlaps the distribution of raw data and partial images. The rightsizing of visualization resources was greatly facilitated as simulation nodes can also process visualization tasks. With the presented hybrid in situ visualization, cost estimations—e.g., via runtime probing for predicting rendering costs—provided the basis for the dynamic optimization of task distribution in each mega-cycle. Further approaches presented in chapter 5 include a model to predict the runtime performance of a distributed volume renderer to support hardware acquisition. Finally, a technique to dynamically adapt encoder settings for image tiles in remote rendering setups was introduced that increases the quality while at the same time lowering the bandwidth requirements.

Foveated Rendering to Improve Application Performance In chapter 6, it is discussed how foveated rendering techniques can increase application performance. A novel approach was presented that utilizes the acuity fall-off in the human visual system (HVS) to accelerate volume rendering. To this end, a typical volume raycaster was modified to adapt sampling parameters in image and object space to the gaze of the observer. For the approximation of sample density and reconstruction during rendering, a novel technique based on the Linde-Buzo-Gray (LBG) algorithm and natural neighbor interpolation was proposed. Overall, average speedups between 1.8 and 3.2 could be achieved for different data sets, with hardly perceptible changes in image quality in peripheral areas. Furthermore, a system that reduces the bandwidth requirements for remote visualization on large high-resolution displays was proposed. The approach determines foveated regions based on the gaze of one or more users and encodes them with a higher quality than the rest of the image.

Performance-Optimized Volume Rendering Applications In chapter 7, unconventional volume rendering applications are presented that enable exploration of abstract and spacial data from different domains through performance optimization. An approach for the interactive visual analysis of large dynamic graphs with several thousand time steps was presented. The approach comprises four analysis classes that

were integrated into an interactive visual analytics system: data views, aggregation and filtering, comparison, and evolution provenance. Central was a GPU-accelerated, volumetric view of the dynamic graph based on the concept of space-time cubes. The scalability of the approach was demonstrated by analyzing dynamic graphs with several thousand time steps. In this chapter, a second approach was introduced to visualize gaze data from participants watching video. The data was rendered as a space-time volume with multiple fields, providing an overview of the stimulus context and how it relates to occurring gaze patterns. Analysts were supported in identifying important time spans without having to replay the whole video stimulus. The technique was optimized to be explored flexibly and interactively via different transfer functions.

8.2 Discussion

In chapter 1, three major research questions are introduced that have been addressed in this thesis. In the following, those questions are discussed in the context of the presented work.

8.2.1 Research Question 1

> *How can we improve the current practice in runtime performance evaluation of scientific visualizations?*

Research Question 1 is approached by a literature review on the current practice in performance evaluation for scientific visualization, in particular recent works on new or improved rendering techniques (chapter 3). The survey shows heterogeneity and no common methodological foundation for evaluating runtime performance. Typically, only a very limited set of parameters such as different data sets, camera paths, viewport sizes, and GPUs are investigated. This makes a comparison with other techniques or generalization to other parameter ranges at least questionable.

The systematic runtime performance evaluation of millions of different parameter configurations, combined with a statistical analysis focusing on correlations, has shown to be a promising approach to derive common guidelines for future performance evaluations. However, the parameter selection remains a challenge since the high dimensionality of the space can quickly lead to infeasibility of a systematic benchmark. An extension to other visualization algorithms besides volume and particle rendering is possible since the proposed framework supports a plugin system. To what extent the findings are transferable to other algorithms remains a question to be addressed in future research. Finally, due to the many configurations that need to be measured on different devices, the process is time and resource consuming. It would have to be

repeated to investigate other performance metrics besides runtime, such as memory consumption or energy usage.

Systematic measurements and statistical analysis help to gain a deeper understanding of qualitative runtime behavior and quantitative parameter dependencies from a high-level perspective. However, a more fine-grained approach proved useful for more detailed performance analysis, which was particularly promising for comparing rendering techniques or variants thereof.

8.2.2 Research Question 2

> *How can we use performance modeling and prediction in the context of scientific visualization systems to improve performance?*

The work presented in this thesis shows applications for performance modeling and prediction in several different scenarios in the domain of scientific visualization. This includes load balancing for (1) cost savings on supercomputers and (2) increased performance through load distribution among multiple GPUs in a workstation environment. Further, runtime performance prediction was used to keep fluent interactivity for an improved user experience, while increasing the rendering quality if surplus resources are available. Quality was also optimized for encoding in remote rendering, while keeping below a bandwidth limit. Finally, performance modeling was applied to save costs in hardware procurement.

Looking at the different approaches, it becomes clear that they also require different models depending on the available compute time, information, and also the data type. This ranges from comparably simple but fast performance probing and linear regression, over a combination of analytical modeling and real-time capable kernel recursive least squares (KRLS), to neural networks that can be evaluated offline, and convolutional neural network (CNN)s for image data. Some of the performance influencing characteristics, such as early ray termination (ERT) in volume rendering are hard to estimate and require hand-tailored probabilistic approaches. Even then, some uncertainty remains that needs to be handled and remains an interesting direction for future work. Finally, the differences in requirements and the results from the practical implementation presented in this thesis show that there is probably no universal model for performance prediction in the field of scientific visualization.

8.2.3 Research Question 3

> *How can we leverage performance evaluation to develop optimized visualization applications?*

Two of the techniques presented in this thesis show that foveated rendering is a promising addition to complement classical acceleration techniques to improve runtime performance. This is particularly relevant in light of the fact that gaze-tracking or eye-tracking devices are becoming more common due to cheaper prices and integration into virtual reality devices. For this, a clear understanding of how perceived rendering quality and performance characteristics are related is vital to further optimize foveated techniques.

The work in this thesis also shows that a better understanding of performance characteristics is a first step to improving it. The optimization of compute heavy rendering algorithms in particular, such as volume raycasting, opens up the possibility for new application domains. Presented were two techniques that both use volume raycasting at their core to visualize hundreds of thousand of data points interactively. One visualizes large dynamic graphs, a data type that is typically visualized in two dimensions. The other one combines gaze data with video stimulus and visualizes them in an integrated space-time view. In both techniques, the typical advantages of volume rendering can be used, such as transfer functions for flexible manipulation of the view to gain new insights.

8.3 Outlook

The methods presented in this thesis are part of a holistic approach to performance quantification of visualization systems. Thereby, most of the presented techniques focus on runtime performance (i.e., frame execution time) for evaluation, modeling, and acceleration. However, there are several other performance metrics worth investigating, depending on the usage scenario and application: memory usage, energy consumption, data throughput, etc. Two of the approaches presented in this thesis already focus on throughput as a performance metric. Typically, the different metrics are linked. That means for example, a slower execution time might lead to energy savings [12]. Incorporating several of these metrics into the models presented in this thesis to form a combined model seems to be a promising direction for future research. This could be a step towards better understanding of their trade-offs and ultimately lead to a multi-objective optimization approach.

Energy usage in particular could play an important role in performance analysis in the future. For instance, embedded devices often have strict energy consumption requirements to save battery life or limit heat emissions. Also, in the domain of high performance computing (HPC) power consumption is increasingly becoming a challenge due to ever growing systems and rising costs for energy and cooling. On the other hand, there is still a huge potential in saving costs for (in situ) visualization, in the form of energy and resources, by avoiding rendering irrelevant or unperceivable parts and details of the data.

Several of the presented approaches show that there is also an inherent link between quality and performance. Rendering quality itself can be considered a performance metric. As can be seen at the example of the presented foveated approaches, perceived quality can be subjective and dependent on the visual system of the observer. Therefore, a direct quantification of quality is not always trivially possible—despite the existence of several general image quality metrics, such as structural similarity index measure (SSIM) or peak signal-to-noise ratio (PSNR). Hence, an interesting direction for future research would be the creation of an expressive quality metric for the context of major scientific visualization algorithms, such as volume rendering. With the help of this metric, it would be possible to fine-tune the presented performance models and ultimately save rendering cost or storage space.

Another possible path forward would be the application of the performance modeling and analysis algorithms presented in this thesis to other domains such as software engineering or human computer interaction. Regarding software engineering, it would be thinkable that possible performance regressions can already be detected during software development or that it can be predicted how a code change will affect performance. For human computer interfaces, performance prediction during runtime could be used to change the interaction, for instance by warning the user of long wait times for certain operations or suggesting performance optimized visual mappings.

Finally, the work on performance evaluation showed the substantial impact of data sets on runtime performance, in particular if acceleration techniques are used. The creation of a standard collection of data sets that cover many performance characteristics would be a major step towards a better comparability of performance measurements and benchmarks. Ultimately, this would greatly simplify future research in the area of performance quantification.

Author's Work

[1] V. Bruder, H. Ben Lahmar, M. Hlawatsch, S. Frey, M. Burch, D. Weiskopf, M. Herschel, and T. Ertl. "Volume-based large dynamic graph analysis supported by evolution provenance". In: *Multimedia Tools and Applications* 78.23 (2019), pp. 32939–32965 (cited on pages 7, 8, 127, 128).

[2] V. Bruder, S. Frey, and T. Ertl. "Real-time performance prediction and tuning for interactive volume raycasting". In: *Proceedings of the SIGGRAPH ASIA Symposium on Visualization*. 2016, pp. 1–8 (cited on pages 6–8, 33, 34, 62).

[3] V. Bruder, S. Frey, and T. Ertl. "Prediction-based load balancing and resolution tuning for interactive volume raycasting". In: *Visual Informatics* 1.2 (June 2017), pp. 106–117 (cited on pages 6–8, 62).

[4] V. Bruder, M. Hlawatsch, S. Frey, M. Burch, D. Weiskopf, and T. Ertl. "Volume-based large dynamic graph analytics". In: *Proceedings of the International Conference Information Visualisation (IV)*. Dec. 2018, pp. 210–219 (cited on pages 7–9, 127, 128).

[5] V. Bruder, K. Kurzhals, S. Frey, D. Weiskopf, and T. Ertl. "Space-time volume visualization of gaze and stimulus". In: *Proceedings of the ACM Symposium on Eye Tracking Research and Applications (ETRA)*. 2019, pp. 1–9 (cited on pages 7, 8, 127, 128).

[6] V. Bruder, M. Larsen, T. Ertl, H. Childs, and S. Frey. "A Hybrid In Situ Approach for Cost Efficient Image Database Generation [in preparation]" (cited on pages 6, 8, 84).

[7] V. Bruder, C. Müller, S. Frey, and T. Ertl. "On Evaluating Runtime Performance of Interactive Visualizations". In: *IEEE Transactions on Visualization and Computer Graphics* 26.9 (2020), pp. 2848–2862 (cited on pages 5, 7, 8, 29, 30, 32, 46).

[8] V. Bruder, C. Schulz, R. Bauer, S. Frey, D. Weiskopf, and T. Ertl. "Voronoi-Based Foveated Volume Rendering". In: *Proceedings of EuroVis (Short Papers)*. The Eurographics Association, 2019, pp. 67–71 (cited on pages 7–9, 115–117, 121).

[9] S. Frey, V. Bruder, F. Frieß, P. Gralka, T. Rau, T. Ertl, and G. Reina. "Trade-offs and Parameter Adaptation in In Situ Visualization [to appear]". In: *In Situ Visualization for Computational Science*. Ed. by H. Childs, J. C. Bennett, and C. Garth. Springer, 2022 (cited on page 84).

[10] F. Frieß, M. Braun, V. Bruder, S. Frey, G. Reina, and T. Ertl. "Foveated Encoding for Large High-Resolution Displays". In: *IEEE Transactions on Visualization and Computer Graphics* 27.2 (Feb. 2021), pp. 1850–1859 (cited on pages 7–9, 115, 116).

[11] F. Frieß, M. Landwehr, V. Bruder, S. Frey, and T. Ertl. "Adaptive Encoder Settings for Interactive Remote Visualisation on High-Resolution Displays". In: *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. Oct. 2018, pp. 87–91 (cited on pages 6–8, 84).

[12] M. Heinemann, V. Bruder, S. Frey, and T. Ertl. "Power Efficiency of Volume Raycasting on Mobile Devices". In: *Proceedings of EuroVis (Posters)*. The Eurographics Association, 2017, pp. 49–51 (cited on page 168).

[13] H. Tarner, V. Bruder, T. Ertl, S. Frey, and F. Beck. "Visually Comparing Rendering Performance from Multiple Perspectives [in preparation]" (cited on pages 6, 8, 30, 55).

[14] G. Tkachev, S. Frey, C. Müller, V. Bruder, and T. Ertl. "Prediction of Distributed Volume Visualization Performance to Support Render Hardware Acquisition". In: *Proceeding of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. 2017, pp. 11–20 (cited on pages 6–8, 84, 108).

Bibliography

[15] Advanced Micro Devices. *RDNA Architecture*. Tech. rep. 2019, pp. 1–25. URL: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf (visited on 10/29/2021) (cited on page 20).

[16] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. "An Image-Based Approach to Extreme Scale In Situ Visualization and Analysis". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Jan. 2014, pp. 424–434 (cited on pages 85, 95).

[17] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. "The Graphics Rendering Pipeline". In: *Real-Time Rendering*. 4th ed. New York: A K Peters/CRC Press, Aug. 2018. Chap. 2, pp. 11–27 (cited on page 13).

[18] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. V. Andel. "Nyx: A Massively Parallel AMR Code for Computational Cosmology". In: *The Astrophysical Journal* 765.1 (Feb. 2013), pp. 39–53 (cited on pages 95, 98).

[19] J. Amanatides and A. Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *Proceedings of Eurographics*. 1987, pp. 3–10 (cited on page 18).

[20] M. Ament and C. Dachsbacher. "Anisotropic Ambient Volume Shading". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 1015–1024 (cited on page 34).

[21] D. Archambault, H. Purchase, and B. Pinaud. "Animation, small multiples, and the effect of mental map preservation in dynamic graphs". In: *IEEE Transactions on Visualization and Computer Graphics* 17.4 (2011), pp. 539–552 (cited on page 128).

[22] U. Ayachit, A. Bauer, B. Geveci, P. O'leary, K. Moreland, N. Fabian, and J. Mauldin. "ParaView Catalyst: Enabling In Situ Data Analysis and Visualization". In: *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. 2015, pp. 25–29 (cited on page 22).

[23] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel. "The SENSEI Generic In Situ Interface". In: *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. Feb. 2016, pp. 40–44 (cited on page 22).

[24] B. Bach, P. Dragicevic, D. Archambault, C. Hurter, and S. Carpendale. "A Descriptive Framework for Temporal Data Visualizations Based on Generalized Space-Time Cubes". In: *Computer Graphics Forum* 36.6 (Sept. 2017), pp. 36–61 (cited on page 128).

[25] L. Bavoil and K. Myers. *Order Independent Transparency with Dual Depth Peeling*. Tech. rep. NVIDIA Corp., 2008, pp. 1–9. URL: https://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf (visited on 10/29/2021) (cited on page 19).

[26] F. Beck, M. Burch, C. Vehlow, S. Diehl, and D. Weiskopf. "Rapid Serial Visual Presentation in Dynamic Graph Visualization". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2012, pp. 185–192 (cited on page 142).

[27] M. Behrisch, B. Bach, N. H. Riche, T. Schreck, and J.-D. Fekete. "Matrix Reordering Methods for Table and Network Visualization". In: *Computer Graphics Forum* 35.3 (June 2016), pp. 693–716 (cited on page 129).

[28] J. C. Bennett et al. "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2012, pp. 1–9 (cited on page 22).

[29] C. Bentes, B. B. Labronici, L. Drummond, and R. Farias. "Towards an efficient parallel raycasting of unstructured volumetric data on distributed environments". In: *Cluster Computing* 17 (June 2014), pp. 423–439 (cited on page 24).

[30] S. Bergner, T. Möller, D. Weiskopf, and D. J. Muraki. "A spectral analysis of function composition and its implications for sampling in direct volume visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 1353–1360 (cited on page 118).

[31] E. W. Bethel and M. Howison. "Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning". In: *International Journal of High Performance Computing Applications* 26.4 (Nov. 2012), pp. 399–412 (cited on pages 24, 69).

[32] J. Beyer, M. Hadwiger, and H. Pfister. "State-of-the-Art in GPU-Based Large-Scale Volume Visualization". In: *Computer Graphics Forum* 34.8 (2015), pp. 13–37 (cited on pages 21, 23).

[33] T. Biedert, P. Messmer, T. Fogal, and C. Garth. “Hardware-Accelerated Multi-Tile Streaming for Realtime Remote Visualization”. In: *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. 2018, pp. 33–43 (cited on page 122).

[34] R. Binyahib, T. Peterka, M. Larsen, K. L. Ma, and H. Childs. “A Scalable Hybrid Scheme for Ray-Casting of Unstructured Volume Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.7 (July 2019), pp. 2349–2361 (cited on page 21).

[35] I. Bowman, J. Shalf, K.-L. Ma, and W. Bethel. *Performance Modeling for 3D Visualization in a Heterogeneous Computing Environment*. Tech. rep. 2004, pp. 1–9. URL: https://escholarship.org/uc/item/1hp5w4gg (visited on 10/29/2021) (cited on page 24).

[36] A. Bringmann, S. Syrbe, K. Görner, J. Kacza, M. Francke, P. Wiedemann, and A. Reichenbach. “The primate fovea: structure, function and development”. In: *Progress in Retinal and Eye Research* 66 (Sept. 2018), pp. 49–84 (cited on page 14).

[37] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert. “High Accuracy Optical Flow Estimation Based on a Theory for Warping”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Vol. 3024. 2004, pp. 25–36 (cited on page 153).

[38] S. Bruckner and E. Gröller. “VolumeShop: An Interactive System for Direct Volume Illustration”. In: *Proceedings of the IEEE Visualization Conference*. Oct. 2006, pp. 671–678 (cited on pages 33, 34, 36).

[39] S. Bruckner and E. Gröller. “Style transfer functions for illustrative volume rendering”. In: *Computer Graphics Forum* 26.3 (Sept. 2007), pp. 715–724 (cited on page 19).

[40] M. Chavent, A. Vanel, A. Tek, B. Levy, S. Robert, B. Raffin, and M. Baaden. “GPU-accelerated atom and dynamic bond visualization using hyperballs: A unified algorithm for balls, sticks, and hyperboloids”. In: *Journal of Computational Chemistry* 32.13 (Oct. 2011), pp. 2924–2935 (cited on pages 44, 45).

[41] Z. Chen and C. Guillemot. “Perceptually-friendly H.264/AVC video coding based on foveated just-noticeable-distortion model”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 20.6 (June 2010), pp. 806–819 (cited on page 15).

[42] H. Childs, J. Bennett, C. Garth, and B. Hentschel. “In Situ Visualization for Computational Science”. In: *IEEE Computer Graphics and Applications* 39.6 (Nov. 2019), pp. 76–85 (cited on page 21).

[43] H. Childs et al. “A terminology for in situ visualization and analysis systems:” in: *The International Journal of High Performance Computing Applications* 34.6 (Aug. 2020), pp. 676–691 (cited on pages 2, 22).

[44] D. Cohen and Z. Sheffer. “Proximity clouds - an acceleration technique for 3D grid traversal”. In: *The Visual Computer* 11.1 (1994), pp. 27–38 (cited on page 19).

[45] B. Csébfalvi. “Beyond trilinear interpolation: Higher quality for free”. In: *ACM Transactions on Graphics (TOG)* 38.4 (July 2019), pp. 1–8 (cited on page 22).

[46] B. Csébfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller. “Fast visualization of object contours by non-photorealistic volume rendering”. In: *Computer Graphics Forum* 20.3 (Sept. 2001), pp. 452–460 (cited on page 19).

[47] E. Cuthill and J. McKee. “Reducing the bandwidth of sparse symmetric matrices”. In: *Proceedings of the 24th ACM National Conference*. Aug. 1969, pp. 157–172 (cited on page 129).

[48] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki. “Flexpath: Type-based publish/subscribe system for large-scale science analytics”. In: *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*. 2014, pp. 246–255 (cited on page 22).

[49] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. “GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models”. In: *Proceedings of the International Conference on High Performance Computing*. Oct. 2016, pp. 489–507 (cited on page 24).

[50] O. Deussen, M. Spicker, and Q. Zheng. “Weighted linde-buzo-gray stippling”. In: *ACM Transactions on Graphics (TOG)* 36.6 (Nov. 2017), pp. 1–12 (cited on pages 116, 117).

[51] Z.-Y. Ding, J.-G. Tan, X.-Y. Wu, W.-F. Chen, F.-R. Wu, X. Li, and W. Chen. “A Near Lossless Compression Domain Volume Rendering Algorithm for Floating-Point Time-Varying Volume Data”. In: *Journal of Visualization* 18.2 (2015), pp. 147–157 (cited on page 34).

[52] E. Dirand, L. Colombet, and B. Raffin. “TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics”. In: *Proceedings of the Asian Conference on Supercomputing Frontiers*. 2018, pp. 159–178 (cited on page 23).

[53] T. M. A. Do, L. Pottier, S. Thomas, R. F. da Silva, M. A. Cuendet, H. Weinstein, T. Estrada, M. Taufer, and E. Deelman. “A Novel Metric to Evaluate In Situ Workflows”. In: *Proceedings of the International Conference on Computational Science*. June 2020, pp. 538–553 (cited on page 23).

[54] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf. “Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations”. In: *ACM Transactions on Parallel Computing (TOPC)* 3.3 (Oct. 2016), pp. 1–43 (cited on page 22).

[55] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro. “Damaris/Viz: A nonintrusive, adaptable and user-friendly in situ visualization framework”. In: *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV).* 2013, pp. 67–75 (cited on page 22).

[56] M. Dorier, O. Yildiz, T. Peterka, and R. Ross. “The Challenges of Elastic In Situ Analysis and Visualization”. In: *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV).* 2019, pp. 23–28 (cited on page 23).

[57] E. DoVale. “High frame rate psychophysics: Experimentation to determine a JND for frame rate”. In: *SMPTE Motion Imaging Journal* 126.9 (Nov. 2017), pp. 41–47 (cited on page 119).

[58] S. Dutta, C. M. Chen, G. Heinlein, H. W. Shen, and J. P. Chen. “In Situ Distribution Guided Analysis and Visualization of Transonic Jet Engine Simulations”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 811–820 (cited on page 21).

[59] D. B. Elliott, K. C. Yang, and D. Whitaker. “Visual acuity changes throughout adulthood in normal, healthy eyes: seeing beyond 6/6.” In: *Optometry and Vision Science* 72.3 (Mar. 1995), pp. 186–191 (cited on page 116).

[60] Y. Engel, S. Mannor, and R. Meir. “The kernel recursive least-squares algorithm”. In: *IEEE Transactions on Signal Processing* 52.8 (Aug. 2004), pp. 2275–2285 (cited on page 68).

[61] M. Falk, M. Klann, M. Reuss, and T. Ertl. “Visualization of signal transduction processes in the crowded environment of the cell”. In: *Proceedings of the IEEE Pacific Visualization Symposium.* 2009, pp. 169–176 (cited on pages 43–45).

[62] D. G. Feitelson. *Experimental Computer Science: The Need for a Cultural Change.* Tech. rep. The Hebrew University of Jerusalem, Dec. 2006, pp. 1–37. URL: https://cs.uwaterloo.ca/~brecht/courses/854-Experimental-Performance-Evaluation-2018/readings/feitelson-exp05.pdf (visited on 10/29/2021) (cited on page 23).

[63] J.-D. Fekete, J. J. van Wijk, J. T. Stasko, and C. North. “The Value of Information Visualization”. In: *Information Visualization: Human-Centered Issues and Perspectives.* Ed. by A. Kerren, J. T. Stasko, J.-D. Fekete, and C. North. Springer, Berlin, Heidelberg, 2008, pp. 1–18 (cited on page 12).

[64] S. Frey, F. Sadlo, K. L. Ma, and T. Ertl. “Interactive Progressive Visualization with Space-Time Error Control”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014), pp. 2397–2406 (cited on pages 19, 33, 34).

[65] M. Friendly. "Visions and Re-Visions of Charles Joseph Minard". In: *Journal of Educational and Behavioral Statistics* 27.1 (Nov. 2002), pp. 31–51 (cited on page 11).

[66] B. Friesen, A. Almgren, Z. Lukić, G. Weber, D. Morozov, V. Beckner, and M. Day. "In situ and in-transit analysis of cosmological simulations". In: *Computational Astrophysics and Cosmology* 3.1 (Aug. 2016), pp. 1–18 (cited on page 23).

[67] W. F. Godoy et al. "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management". In: *SoftwareX* 12 (July 2020), pp. 1–9 (cited on page 22).

[68] J. Görtler, M. Spicker, C. Schulz, D. Weiskopf, and O. Deussen. "Stippling of 2D Scalar Fields". In: *IEEE Transactions on Visualization and Computer Graphics* 25.6 (June 2019), pp. 2193–2204 (cited on page 117).

[69] A. Goswami, Y. Tian, K. Schwan, F. Zheng, J. Young, M. Wolf, G. Eisenhauer, and S. Klasky. "Landrush: Rethinking In-Situ Analysis for GPGPU Workflows". In: *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid).* July 2016, pp. 32–41 (cited on page 23).

[70] P. Gralka, I. Wald, S. Geringer, G. Reina, and T. Ertl. "Spatial Partitioning Strategies for Memory-Efficient Ray Tracing of Particles". In: *Proceedings of the Symposium on Large Data Analysis and Visualization (LDAV).* 2020, pp. 42–52 (cited on pages 43–45).

[71] C. P. Gribble, T. Lze, A. Kensler, I. Wald, and S. G. Parker. "A coherent grid traversal approach to visualizing particle-based simulation data". In: *IEEE Transactions on Visualization and Computer Graphics* 13.4 (July 2007), pp. 758–768 (cited on pages 43–45).

[72] S. Grottel, M. Krone, C. Müller, G. Reina, and T. Ertl. "MegaMol - A prototyping framework for particle-based visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 21.2 (Feb. 2015), pp. 201–214 (cited on pages 44, 45).

[73] S. Grottel, M. Krone, K. Scharnowski, and T. Ertl. "Object-space ambient occlusion for molecular dynamics". In: *Proceedings of the IEEE Pacific Visualization Symposium.* 2012, pp. 209–216 (cited on pages 43–45).

[74] S. Grottel, G. Reina, C. Dachsbacher, and T. Ertl. "Coherent culling and shading for large molecular dynamics visualization". In: *Computer Graphics Forum* 29.3 (June 2010), pp. 953–962 (cited on pages 43–45).

[75] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder. "Foveated 3D graphics". In: *ACM Transactions on Graphics (TOG)* 31.6 (Nov. 2012), pp. 1–10 (cited on page 14).

[76] S. Gumhold. “Splatting Illuminated Ellipsoids with Depth Correction”. In: *Proceedings of the Symposium on Vision, Modeling and Visualization (VMV)*. 2003, pp. 245–252 (cited on pages 43, 44).

[77] D. Guo, J. Nie, M. Liang, Y. Wang, Y. Wang, and Z. Hu. “View-dependent level-of-detail abstraction for interactive atomistic visualization of biological structures”. In: *Computers and Graphics* 52 (Aug. 2015), pp. 62–71 (cited on pages 44, 45).

[78] R. B. Haber and D. A. McNabb. “Visualization ldioms: A Conceptual Model for Scientilic Visualization Systems”. In: *Visualization in Scientific Computing* (1990), pp. 74–93 (cited on page 12).

[79] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. “SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Jan. 2018), pp. 974–983 (cited on pages 19, 33, 34).

[80] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski. “Advanced Illumination Techniques for GPU-Based Volume Raycasting”. In: *Proceedings of ACM SIGGRAPH ASIA Courses*. 2008, pp. 1–166 (cited on pages 21, 33).

[81] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. “Real-time ray-casting and advanced shading of discrete isosurfaces”. In: *Computer Graphics Forum* 24.3 (2005), pp. 303–312 (cited on page 21).

[82] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. “Accelerating hydrocodes with openACC, opeCL and CUDA”. In: *Proceedings of the International Conference for High Performance Computing, Networking Storage and Analysis (SC)*. 2012, pp. 465–471 (cited on pages 95, 98).

[83] P. Hermosilla, M. Krone, V. Guallar, P. P. Vázquez, À. Vinacua, and T. Ropinski. “Interactive GPU-based generation of solvent-excluded surfaces”. In: *Visual Computer* 33.6-8 (June 2017), pp. 869–881 (cited on pages 43, 44).

[84] F. Hernell, P. Ljung, and A. Ynnerman. “Local ambient occlusion in direct volume rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.4 (2010), pp. 548–559 (cited on pages 33, 34).

[85] R. Hero, C. Ho, and K. L. Ma. “Volume rendering of curvilinear-grid data using low-dimensional deformation textures”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.9 (2014), pp. 1330–1343 (cited on page 34).

[86] M. Herschel, R. Diestelkämper, and H. B. Lahmar. “A survey on provenance: What for? What form? What from?” In: *The VLDB Journal* 26.6 (Oct. 2017), pp. 881–906 (cited on page 136).

[87] T. Hoefler, W. Gropp, M. Snir, and W. Kramer. “Performance modeling for systematic performance tuning”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2011, pp. 1–12 (cited on page 67).

[88] S. Hong and H. Kim. “An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness”. In: *Proceedings of the International Symposium on Computer Architecture*. 2009, pp. 152–163 (cited on page 24).

[89] L. Howes and D. Thomas. “Efficient Random Number Generation and Application Using CUDA”. In: *GPU Gems 3*. Addison-Wesley Professional, 2007, pp. 805–830 (cited on page 67).

[90] M. Ibrahim, P. Rautek, G. Reina, M. Agus, and M. Hadwiger. “Probabilistic Occlusion Culling using Confidence Maps for High-Quality Rendering of Large Particle Data”. In: *IEEE Transactions on Visualization and Computer Graphics* (Sept. 2021), pp. 1–10 (cited on pages 44, 45).

[91] M. Ibrahim, P. Wickenhäuser, P. Rautek, G. Reina, and M. Hadwiger. “Screen-Space Normal Distribution Function Caching for Consistent Multi-Resolution Rendering of Large Particle Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Jan. 2018), pp. 944–953 (cited on page 44).

[92] G. Illahi, M. Siekkinen, and E. Masala. “Foveated video streaming for cloud gaming”. In: *Proceeding of the IEEE International Workshop on Multimedia Signal Processing (MMSP)*. Nov. 2017, pp. 1–6 (cited on page 15).

[93] B. Jin, I. Ihm, B. Chang, C. Park, W. Lee, and S. Jung. “Selective and adaptive supersampling for real-time ray tracing”. In: *Proceedings of the Conference on High-Performance Graphics (HPG)*. 2009, pp. 117–126 (cited on page 14).

[94] D. Jönsson, J. Kronander, T. Ropinski, and A. Ynnerman. “Historygrams: Enabling interactive global illumination in direct volume rendering using photon mapping”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2364–2371 (cited on page 34).

[95] D. Jönsson and A. Ynnerman. “Correlated Photon Mapping for Interactive Global Illumination of Time-Varying Volumetric Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 901–910 (cited on pages 33, 34).

[96] A. Jurčík, J. Parulek, J. Sochor, and B. Kozlíková. “Accelerated visualization of transparent molecular surfaces in molecular dynamics”. In: *Proceedings of the IEEE Pacific Visualization Symposium*. May 2016, pp. 112–119 (cited on pages 43–45).

[97] D. E. King. “Dlib-ml: A Machine Learning Toolkit”. In: *Journal of Machine Learning Research* 10 (2009), pp. 1755–1758 (cited on page 68).

[98] I. P. King. “An automatic reordering scheme for simultaneous equations derived from network systems”. In: *International Journal for Numerical Methods in Engineering* 2.4 (Oct. 1970), pp. 523–533 (cited on page 129).

[99] T. Klein and T. Ertl. “Illustrating Magnetic Field Lines using a Discrete Particle Model”. In: *Proceedings of the Symposium on Vision, Modeling and Visualization (VMV).* 2004, pp. 387–394 (cited on page 44).

[100] A. Knoll, I. Wald, P. Navratil, A. Bowen, K. Reda, M. E. Papka, and K. Gaither. “RBF volume ray casting on multicore and manycore CPUs”. In: *Computer Graphics Forum* 33.3 (June 2014), pp. 71–80 (cited on pages 43–45).

[101] M. Koch, K. Kurzhals, and D. Weiskopf. “Image-based scanpath comparison with slit-scan visualization”. In: *Proceedings of the ACM Symposium on Eye Tracking Research and Applications (ETRA).* June 2018, pp. 1–5 (cited on pages 152, 156).

[102] E. Konstantinidis and Y. Cotronis. “A practical performance model for compute and memory bound GPU kernels”. In: *Proceedings of the Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP).* 2015, pp. 651–658 (cited on page 24).

[103] M.-J. Kraak and I. Kveladze. “Narrative of the annotated Space–Time Cube – revisiting a historical event”. In: *Journal of Maps* 13.1 (Jan. 2017), pp. 56–61 (cited on page 157).

[104] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire. “Comparing the Efficiency of In Situ Visualization Paradigms at Scale”. In: *Proceedings of the International Conference on High Performance Computing.* June 2019, pp. 99–117 (cited on page 23).

[105] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire. “Opportunities for Cost Savings with In-Transit Visualization”. In: *Proceedings of the International Conference on High Performance Computing.* June 2020, pp. 146–165 (cited on page 23).

[106] J. Kronander, D. Jönsson, J. Löw, P. Ljung, A. Ynnerman, and J. Unger. “Efficient visibility encoding for dynamic illumination in direct volume rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.3 (2012), pp. 447–462 (cited on page 34).

[107] M. Krone, K. Bidmon, and T. Ertl. “Interactive visualization of molecular surface dynamics”. In: *IEEE Transactions on Visualization and Computer Graphics.* Vol. 15. 6. Nov. 2009, pp. 1391–1398 (cited on pages 43–45).

[108] J. Krüger and R. Westermann. “Acceleration Techniques for GPU-based Volume Rendering”. In: *Proceedings of the IEEE Visualization Conference.* 2003, pp. 287–292 (cited on page 21).

[109] J. Krüger and R. Westermann. "Acceleration Techniques for GPU-based Volume Rendering". In: *Proceedings of the IEEE Visualization Conference*. 2003, pp. 287–292 (cited on page 34).

[110] K. Kurzhals, C. F. Bopp, J. Bässler, F. Ebinger, and D. Weiskopf. "Benchmark data for evaluating visualization and analysis techniques for eye tracking for video stimuli". In: *Proceedings of the Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization (BELIV)*. Nov. 2014, pp. 54–60 (cited on page 157).

[111] K. Kurzhals and D. Weiskopf. "Space-time visual analytics of eye-tracking data for dynamic stimuli". In: *IEEE Transactions on Visualization and Computer Graphics* 19.12 (2013), pp. 2129–2138 (cited on page 160).

[112] K. Kurzhals and D. Weiskopf. "Visualizing eye tracking data with gaze-guided slit-scans". In: *Proceedings of the Workshop on Eye Tracking and Visualization (ETVIS)*. Feb. 2017, pp. 45–49 (cited on page 156).

[113] P. Lacroute and M. Levoy. "Fast volume rendering using a shear-warp factorization of the viewing transformation". In: *Proceedings of the Conference on Computer Graphics and Interactive Techniques,* July 1994, pp. 451–458 (cited on page 16).

[114] O. D. Lampe, I. Viola, N. Reuter, and H. Hauser. "Two-level approach to efficient visualization of protein dynamics". In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), pp. 1616–1623 (cited on pages 44, 45).

[115] R. S. Laramee. "How to write a visualization research paper: A starting point". In: *Computer Graphics Forum* 29.8 (Dec. 2010), pp. 2363–2371 (cited on page 23).

[116] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison. "The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman". In: *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. 2017, pp. 42–46 (cited on pages 22, 95).

[117] M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. S. Meredith, and H. Childs. "Performance Modeling of in Situ Rendering". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2017, pp. 276–287 (cited on page 24).

[118] M. Larsen, S. Labasan, P. Navrátil, J. Meredith, and H. Childs. "Volume Rendering Via Data-Parallel Primitives". In: *Proceeding of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. 2015, pp. 53–62 (cited on page 24).

[119] M. Larsen, K. Moreland, C. R. Johnson, and H. Childs. "Optimizing multi-image sort-last parallel rendering". In: *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. 2016, pp. 37–46 (cited on page 92).

[120] M. Le Muzic, J. Parulek, A. K. Stavrum, and I. Viola. "Illustrative visualization of molecular reactions using omniscient intelligence and passive agents". In: *Computer Graphics Forum* 33.3 (June 2014), pp. 141–150 (cited on pages 43, 44).

[121] B. Lee and Y. G. Shin. "Advanced interactive preintegrated volume rendering with a power series". In: *IEEE Transactions on Visualization and Computer Graphics* 19.8 (2013), pp. 1264–1273 (cited on pages 33, 34, 36).

[122] S. Lee and A. C. Bovik. "Fast algorithms for foveated video processing". In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.2 (Feb. 2003), pp. 149–162 (cited on page 15).

[123] M. Levoy. "Efficient Ray Tracing of Volume Data". In: *ACM Transactions on Graphics (TOG)* 9.3 (Jan. 1990), pp. 245–261 (cited on pages 17, 19).

[124] M. Levoy and R. Whitake. "Gaze-Directed Volume Rendering". In: *Proceedings of the Symposium on Interactive 3D Graphics (SI3D)*. 1990, pp. 217–223 (cited on page 15).

[125] N. Lindow, D. Baum, and H.-C. Hege. "Interactive rendering of materials and biological structures on atomic and nanoscopic scale". In: *Computer Graphics Forum* 31.3 (June 2012), pp. 1325–1334 (cited on pages 43–45).

[126] N. Lindow, D. Baum, S. Prohaska, and H. C. Hege. "Accelerated visualization of dynamic molecular surfaces". In: *Computer Graphics Forum* 29.3 (June 2010), pp. 943–952 (cited on pages 43–45).

[127] B. Liu, G. J. Clapworthy, F. Dong, and E. C. Prakash. "Octree rasterization: Accelerating high-quality out-of-core GPU volume rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 19.10 (2013), pp. 1732–1745 (cited on pages 33, 34).

[128] P. Ljung, C. Winskog, A. Persson, C. Lundström, and A. Ynnerman. "Full body virtual autopsies using a state-of-the-art volume rendering pipeline". In: *IEEE Transactions on Visualization and Computer Graphics*. Vol. 12. 5. Sept. 2006, pp. 869–876 (cited on page 34).

[129] W. E. Lorensen and H. E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM SIGGRAPH Computer Graphics* 21.4 (Aug. 1987), pp. 163–169 (cited on page 16).

[130] C. Lundström, P. Ljung, A. Persson, and A. Ynnerman. "Uncertainty visualization in medical volume rendering using probabilistic animation". In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), pp. 1648–1655 (cited on page 34).

[131] J. G. Magnus and S. Bruckner. "Interactive Dynamic Volume Illumination with Refraction and Caustics". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Jan. 2018), pp. 984–993 (cited on pages 33, 34).

[132] T. Marrinan, S. Rizzi, J. A. Insley, L. Long, L. Renambot, and M. E. Papka. "PxStream: Remote Visualization for Distributed Rendering Frameworks". In: *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. Oct. 2019, pp. 37–41 (cited on page 122).

[133] N. Max. "Optical Models for Direct Volume Rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (June 1995), pp. 99–108 (cited on page 16).

[134] B. H. McCormick, T. A. DeFanti, and M. D. Brown. "Visualization in Scientific Computing". In: *Computer Graphics* 21.6 (Nov. 1987), pp. 1–14 (cited on page 12).

[135] L. McInnes, J. Healy, and J. Melville. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction". Sept. 2020. URL: http://arxiv.org/abs/1802.03426 (cited on page 59).

[136] K. Mehta et al. "A codesign framework for online data analysis and reduction". In: *Proceedings of the Workshop on Workflows in Support of Large-Scale Science (WORKS)*. Nov. 2019, pp. 11–20 (cited on page 23).

[137] K. Misue, P. Eades, W. Lai, and K. Sugiyama. "Layout Adjustment and the Mental Map". In: *Journal of Visual Languages & Computing* 6.2 (June 1995), pp. 183–210 (cited on page 128).

[138] P. K. Mital, T. J. Smith, R. L. Hill, and J. M. Henderson. "Clustering of Gaze During Dynamic Scene Viewing is Predicted by Motion". In: *Cognitive Computation* 3.1 (Oct. 2010), pp. 5–24 (cited on page 157).

[139] D. P. Mitchell. "Generating antialiased images at low sampling densities". In: *Proceedings of the Conference on Computer Graphics and Interactive Techniques*. Vol. 21. 4. Aug. 1987, pp. 65–72 (cited on page 14).

[140] S. Molnar, D. Ellsworth, H. Fuchs, and M. Cox. "A Sorting Classification of Parallel Rendering". In: *IEEE Computer Graphics and Applications* 14.4 (July 1994), pp. 23–32 (cited on page 21).

[141] K. Moreland. "Comparing Binary-Swap Algorithms for Odd Factors of Processes". In: *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. Institute of Electrical and Electronics Engineers Inc., Oct. 2018, pp. 56–60 (cited on page 21).

[142] K. Moreland, W. Kendall, T. Peterka, and J. Huang. "An Image Compositing Solution at Scale". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2011, pp. 1–10 (cited on pages 86, 92).

[143] K. Moreland et al. "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures". In: *IEEE Computer Graphics and Applications* 36.3 (May 2016), pp. 48–58 (cited on page 95).

[144] D. Morozov and T. Peterka. “Block-parallel data analysis with DIY2”. In: *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. Mar. 2017, pp. 29–36 (cited on page 96).

[145] N. Morrical, W. Usher, I. Wald, and V. Pascucci. “Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing”. In: *Proceeding of IEEE Visualization Conference (Short Papers)*. 2019, pp. 256–260 (cited on pages 33, 34).

[146] C. Mueller. “The Sort-First Rendering Architecture for High-Performance Graphics”. In: *Proceedings of the Symposium on Interactive 3D Graphics (SI3D)*. Apr. 1995, pp. 75–84 (cited on page 21).

[147] C. Müller, M. Krone, M. Huber, V. Biener, D. Herr, S. Koch, G. Reina, D. Weiskopf, and T. Ertl. “Interactive Molecular Graphics for Augmented Reality Using HoloLens”. In: *Journal of integrative bioinformatics* 15.2 (June 2018), pp. 1–13 (cited on pages 43–45).

[148] NVIDIA. *Ampere GA102 GPU Architecture*. Tech. rep. 2021, pp. 1–53. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (visited on 10/29/2021) (cited on page 20).

[149] P. O’Leary, J. Ahrens, S. Jourdain, S. Wittenburg, D. H. Rogers, and M. Petersen. “Cinema image-based in situ analysis and visualization of MPAS-ocean simulations”. In: *Parallel Computing* 55 (July 2016), pp. 43–48 (cited on page 21).

[150] S. W. Park, L. Linsen, O. Kreylos, J. D. Owens, and B. Hamann. “Discrete sibson interpolation”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.2 (Mar. 2006), pp. 243–252 (cited on page 119).

[151] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. P. Sloan. “Interactive ray tracing for isosurface rendering”. In: *Proceedings of the IEEE Visualization Conference*. 1998, pp. 233–238 (cited on page 16).

[152] T. Peterka, D. Bard, J. C. Bennett, E. W. Bethel, R. A. Oldfield, L. Pouchard, C. Sweeney, and M. Wolf. “Priority research directions for in situ data management: Enabling scientific discovery from diverse data sources”. In: *International Journal of High Performance Computing Applications* 34.4 (July 2020), pp. 409–427 (cited on page 21).

[153] B. T. Phong. “Illumination for Computer Generated Pictures”. In: *Communications of the ACM* 18.6 (June 1975), pp. 311–317 (cited on page 19).

[154] H. Qu, M. Wan, J. Qin, and A. Kaufman. “Image based rendering with stable frame rates”. In: *Proceedings of the IEEE Visualization Conference*. 2000, pp. 251–258+564 (cited on page 19).

[155] G. Reina and T. Ertl. "Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization". In: *Proceeding of the Eurographics / IEEE VGTC Symposium on Visualization*. 2005, pp. 177–182 (cited on pages 43, 44).

[156] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. D. Uram, and V. Vishwanath. "Performance Modeling of vl3 Volume Rendering on GPU-Based Clusters". In: *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. 2014, pp. 65–72 (cited on page 24).

[157] S. Röttger, M. Kraus, and T. Ertl. "Hardware-accelerated volume and isosurface rendering based on cell-projection". In: *Proceedings of the IEEE Visualization Conference*. 2000, pp. 109–116 (cited on page 16).

[158] F. Sans and R. Carmona. "Volume ray casting using different GPU based parallel APIs". In: *Proceedings of the Latin American Computing Conference (CLEI)*. Jan. 2017, pp. 1–11 (cited on page 34).

[159] P. Schlegel, M. Makhinya, and R. Pajarola. "Extinction-based shading and illumination in gpu volume ray-casting". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 1795–1802 (cited on page 34).

[160] C. Schulz et al. "Generative data models for validation and evaluation of visualization techniques". In: *Proceedings of the Workshop on Beyond Time and Errors on Novel Evaluation Methods for Visualization (BELIV)*. Vol. 24-October. Oct. 2016, pp. 112–124 (cited on page 52).

[161] H. Schumann and W. Müller. *Visualisierung: Grundlagen und allgemeine methoden*. Springer Berlin Heidelberg, 2000, pp. 1–370 (cited on page 12).

[162] H. W. Shen and C. R. Johnson. "Differential volume rendering: a fast volume visualization technique for flow animation". In: *Proceedings Visualization*. IEEE, 1994, pp. 180–187 (cited on page 19).

[163] R. Sibson. "A vector identity for the Dirichlet tessellation". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 87.1 (1980), pp. 151–155 (cited on page 118).

[164] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. 1st. London: Chapman and Hall, 1986, pp. 1–175 (cited on page 154).

[165] R. Skånberg, P. P. Vázquez, V. Guallar, and T. Ropinski. "Real-Time Molecular Visualization Supporting Diffuse Interreflections and Ambient Occlusion". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 718–727 (cited on pages 44, 45).

[166] S. W. Sloan. "An algorithm for profile and wavefront reduction of sparse matrices". In: *International Journal for Numerical Methods in Engineering* 23.2 (Feb. 1986), pp. 239–251 (cited on page 129).

[167] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. “A simple and flexible volume rendering framework for graphics-hardware - Based raycasting”. In: *Proceedings of the Eurographics/IEEE VGTC Workshop on Volume Graphics.* 2005, pp. 187–195 (cited on pages 21, 33, 34, 36).

[168] M. Stengel, S. Grogorick, M. Eisemann, and M. Magnor. “Adaptive Image-Space Sampling for Gaze-Contingent Real-time Rendering”. In: *Computer Graphics Forum* 35.4 (July 2016), pp. 129–139 (cited on page 14).

[169] H. Strasburger, I. Rentschler, and M. Jüttner. “Peripheral vision and pattern recognition: A review”. In: *Journal of Vision* 11.5 (May 2011), pp. 1–82 (cited on page 14).

[170] Y. Sugimoto, F. Ino, and K. Hagihara. “Improving cache locality for GPU-based volume rendering”. In: *Parallel Computing* 40.5-6 (May 2014), pp. 59–69 (cited on pages 33, 34, 36, 42).

[171] N. Tack, F. Morán, G. Lafruit, and R. Lauwereins. “3D graphics rendering time modeling and control for mobile terminals”. In: *Proceedings of the Web3D Symposium.* 2004, pp. 109–117 (cited on page 24).

[172] M. Tarini, P. Cignoni, and C. Montani. “Ambient occlusion and edge cueing to enhance real time molecular visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 1237–1244 (cited on page 44).

[173] T. Terraz, A. Ribes, Y. Fournier, B. Iooss, and B. Raffin. “Melissa: Large scale in transit sensitivity analysis avoiding intermediate files”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* Vol. 14. Nov. 2017, pp. 1–14 (cited on page 23).

[174] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. “Experimental evaluation in computer science: A quantitative study”. In: *The Journal of Systems and Software* 28.1 (1995), pp. 9–18 (cited on page 23).

[175] B. Tversky, J. B. Morrison, and M. Betrancourt. “Animation: can it facilitate?” In: *International Journal of Human-Computer Studies* 57.4 (Oct. 2002), pp. 247–262 (cited on page 129).

[176] K. Vaidyanathan et al. “Coarse pixel shading”. In: *Proceedings of High-Performance Graphics (HPG).* 2014, pp. 9–18 (cited on page 14).

[177] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil. “OSPRay - A CPU Ray Tracing Framework for Scientific Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 931–940 (cited on page 56).

[178] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka. "CPU ray tracing large particle data with balanced P-k-d trees". In: *Proceedings of the IEEE Scientific Visualization Conference (SciVis)*. Mar. 2015, pp. 57–64 (cited on pages 43–45, 60).

[179] I. Wald, S. Zellmann, and N. Morrical. "Faster RTX-Accelerated Empty Space Skipping using Triangulated Active Region Boundary Geometry". In: *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*. 2021, pp. 37–44 (cited on page 34).

[180] J. Wang, F. Yang, and Y. Cao. "A cache-friendly sampling strategy for texture-based volume rendering on GPU". In: *Visual Informatics* 1.2 (June 2017), pp. 92–105 (cited on pages 33, 34, 36, 42).

[181] Z. Wang, P. Subedi, M. Dorier, P. E. Davis, and M. Parashar. "Staging Based Task Execution for Data-driven, In-Situ Scientific Workflows". In: *Proceedings of the IEEE International Conference on Cluster Computing (ICCC)*. Sept. 2020, pp. 209–220 (cited on page 23).

[182] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image quality assessment: From error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (Apr. 2004), pp. 600–612 (cited on page 58).

[183] A. Waschk and J. Kruger. "FAVR - Accelerating Direct Volume Rendering for Virtual Reality Systems". In: *Proceeding of the IEEE Visualization Conference (Short Papers)*. 2020, pp. 106–110 (cited on pages 33, 34).

[184] M. Weier et al. "Perception-driven Accelerated Rendering". In: *Computer Graphics Forum* 36.2 (May 2017), pp. 611–643 (cited on page 14).

[185] D. Weiskopf. *GPU-based interactive visualization techniques*. 1st ed. Springer, Berlin, Heidelberg, 2007, pp. 1–312 (cited on page 12).

[186] L. A. Westover. "Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm". PhD thesis. Chapel Hill: The University of North Carolina, July 1991, pp. 1–101. URL: http://www.cs.unc.edu/techreports/91-029.pdf (cited on page 16).

[187] B. Whitlock, J. M. Favre, and J. S. Meredith. "Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System". In: *Proceeding of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. 2011, pp. 101–109 (cited on page 22).

[188] M. Wimmer and P. Wonka. "Rendering Time Estimation for Real-Time Rendering". In: *Proceedings of the Eurographics Symposium on Rendering*. 2003, pp. 118–129 (cited on page 24).

[189] J. Woodring, M. Petersen, A. Schmeißer, J. Patchett, J. Ahrens, and H. Hagen. "In Situ Eddy Analysis in a High-Resolution Ocean Climate Model". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 857–866 (cited on page 21).

[190] K. Wu, A. Knoll, B. J. Isaac, H. Carr, and V. Pascucci. "Direct Multifield Volume Ray Casting of Fiber Surfaces". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 941–949 (cited on pages 33, 34).

[191] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. "Scalable rendering on PC clusters". In: *IEEE Computer Graphics and Applications* 21.4 (July 2001), pp. 62–70 (cited on page 21).

[192] F. Yang, Q. Li, D. Xiang, Y. Cao, and J. Tian. "A versatile optical model for hybrid rendering of volume data". In: *IEEE Transactions on Visualization and Computer Graphics* 18.6 (2012), pp. 925–937 (cited on pages 33, 34).

[193] K.-P. Yee, K. Swearingen, K. Li, and M. Hearst. "Faceted Metadata for Image Search and Browsing". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2003, pp. 401–408 (cited on page 59).

[194] A. Zare, A. Aminlou, M. M. Hannuksela, and M. Gabbouj. "HEVC-compliant tile-based streaming of panoramic video for virtual reality applications". In: *Proceedings of the ACM Multimedia Conference (MM)*. Oct. 2016, pp. 601–605 (cited on page 15).

[195] S. Zellmann. *Comparing Hierarchical Data Structures for Sparse Volume Rendering with Empty Space Skipping*. Tech. rep. Dec. 2019, pp. 1–9. URL: http://arxiv.org/abs/1912.09596 (visited on 10/29/2021) (cited on page 19).

[196] T. Zhang, Z. Yi, J. Zheng, D. C. Liu, W. M. Pang, Q. Wang, and J. Qin. "A Clustering-Based Automatic Transfer Function Design for Volume Visualization". In: *Mathematical Problems in Engineering* (2016), pp. 1024–1037 (cited on page 34).

[197] W. Zhang et al. "AMReX: a framework for block-structured adaptive mesh refinement Software". In: *The Journal of Open Source Software* 4.37 (2019), pp. 1370–1373 (cited on pages 95, 98).

[198] Y. Zhang and K. L. Ma. "Decoupled Shading for Real-time Heterogeneous Volume Illumination". In: *Computer Graphics Forum* 35.3 (June 2016), pp. 401–410 (cited on pages 33, 34).

[199] Y. Zhang and J. D. Owens. "A quantitative performance analysis model for GPU architectures". In: *Proceedings of the International Symposium on High-Performance Computer Architecture*. 2011, pp. 382–393 (cited on page 24).

[200] F. Zheng, H. Abbasi, J. Cao, J. Dayal, K. Schwan, M. Wolf, S. Klasky, and N. Podhorszki. “In-situ I/O processing: A case for location flexibility”. In: *Proceedings of the Parallel Data Storage Workshop (PDSW)*. 2011, pp. 37–42 (cited on page 23).

[201] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. “GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2013, pp. 1–12 (cited on page 23).

[202] F. Zheng et al. “PreDatA - Preparatory data analytics on peta-scale machines”. In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 2010, pp. 1–12 (cited on page 22).

[203] J. Zhou and M. Takatsuka. “Automatic transfer function generation using contour tree controlled residue flow model and color harmonics”. In: *IEEE Transactions on Visualization and Computer Graphics*. Vol. 15. 6. Nov. 2009, pp. 1481–1488 (cited on page 34).

[204] K. J. Zuiderveld, A. H. J. Koning, and M. A. Viergever. “Acceleration of ray-casting using 3-D distance transforms”. In: *Visualization in Biomedical Computing* 1808 (Sept. 1992), pp. 324–335 (cited on page 19).