Glyph-based Display and Level-of-Detail Particle Rendering for Interactive Scientific Visualization

Mahmoud Zeidan

Glyph-based Display and Level-of-Detail Particle Rendering for Interactive Scientific Visualization

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Mahmoud Zeidan
aus Kairo, Ägypten

Tag der mündlichen Prüfung	:	5. Dezember 2025
Erster Gutachter	:	Prof. Dr.-Ing. Carsten Dachsbacher
Zweiter Gutachter	:	Prof. Dr.-Ing. Martin Eisemann

Acknowledgments

Carsten Dachsbacher: I have many personal reasons to express my sincere gratitude to you. Without your generous support, patience, understanding, and trust, this research would not have been completed. I have learned so much from you, both personally and professionally, and I am deeply thankful for your guidance.

Christoph Peters: Your support and patience guided my early research steps in this thesis. Your advice on writing and debugging GL code was invaluable, even in later stages.

Tobias Rapp: Thank you for your generous support during my early steps into the world of visualization. Your codebase *(i.e., postAtom)* was a great source of inspiration.

Daniel Opitz: Thank you for your consistent and reliable collaboration every time.

Diana Kheil: Thank you for always being more than a colleague—like a supportive sister! I also appreciated the many thought-provoking conversations we had about politics, society, and life in general.

Hisanari Otsu: Thank you for giving me the chance to learn from you. Your engineering style and research experience have always fascinated me.

Jasmin Hoffmann: Thank you for your openness to scientific discussions throughout the past year.

Emanuel Scherade: Thank you for being a wonderful office mate for several years and for your help during my early days in Germany.

Andreas Kratzer: Thank you for the remote setup and solid system support. Your help made my transition to Linux much easier, and I relied heavily on your setup during COVID-19—it's still in use, by the way!

Addis Dittebrandt: Thank you for always being willing to help and for your consistently kind responses.

Pawel Herman: Thank you for the many enjoyable lunch breaks—we had great conversations. I hope America becomes great again!

My IVD colleagues: *Alisa Jung, Benedikt Karl, Boris Neubert, Christoph Schied, Dimitri Klepikov, Florian Simon, Prof. Hartmut Prautzsch, Hauka Refheld, Johannes Hanika, Johannes Meng, Killian Herveau, Lorenzo Tessari, Lukas Alber, Max-Gerd Retzlaff, Max Eifried, Max Piochowiak, Moritz Grauer, Mikhail Dereviannykh, Nathan Lerzer, Reiner Dolp, Stefan Bergmann, Tobias Zirr, Thorsten-Walter Schmidt, and Vincent Schußler*—thank you for every pleasant moment, every bit of help or support, and even just a smile.

My beloved family: *Mama & Papa*—thank you for your unconditional love and support. This work is dedicated to you. *Aya*, my dear wife—thank you for your immense love and unwavering support; this work often took me away from you. *Moustafa*, my beloved son—thank you for your patience and sacrifices during this journey. *Hala*, my only sister, my second mother, and my role model for dedication and patience. *Hesham & Mohamed*, my kind brothers—thank you for your constant love and encouragement. *Dr. Alaa & Dr. Aliaa*, my parents-in-law—thank you for your support. *Prof. Hany AbdelDayem*, my dear brother-in-law—thank you for your continued guidance and help.

My dear friends: *Ahmad Zoheir*—thank you for always being a close friend and mentor, and for your continuous support, especially during difficult times. *Mahmoud Salem*, my dear friend—thank you for your unconditional and generous support. *Ahmed (Ismaiel) Zakaria*—thank you for being both a great colleague and a close friend.

My Egyptian colleagues: *Mohamed Fahmy*, *Dr. Amr El-Desoky*, *Dr. Tawheed Hashem*, *Dr. Ali Abou-Sena*, *Mohamed El-Shenawy*—thank you for your help and support. *Prof. Hassan Ramadan*—thank you for your generous support before and throughout this journey.

Ain Shams University, Faculty of Computer and Information Sciences, and Basic Sciences Department: I would also like to express my sincere appreciation for the support provided by my home university, faculty, and department during my study period in Germany.

I would also like to acknowledge my sources of funding:

From February 2020 onward, I was supported by the *Institute for Visualization and Data Analysis (IVD), KIT, Germany*.

From February 2020 to August 2020, I was supported by the *German Academic Exchange Service (DAAD), KIT, Germany*.

From February 2016 to February 2020, I was supported by the *Ministry of Research and Higher Education, Egypt*.

Abstract

Motivation and Problem Statement

Scientific visualization is a discipline that transforms raw data into meaningful representations, enhancing our understanding of underlying phenomena. Through visualization, domain experts, scientists, engineers, and even novice users can gain valuable insights into processes such as scientific simulations and real-world systems.

Vector field visualization is an active area of research that supports experts across many disciplines—including biology, chemistry, and medicine—in exploring critical phenomena and identifying salient features within flow data. Geometric primitives such as points, lines, and surfaces are fundamental to representing flow structures. However, these visualizations often suffer from visual clutter, which can obscure important features beneath dense or less relevant elements.

To address this challenge, recent advances in scientific visualization have introduced techniques such as streamline selection and opacity optimization [GTG17] to reduce geometric clutter and occlusion. Among these, opacity optimization has emerged as a key strategy to balance the visibility of meaningful internal structures with the suppression of occluding or less informative regions. Although these techniques are highly effective, they typically require significant computational resources, thereby constraining their applicability in real-time and interactive settings.

In parallel, understanding large-scale particle simulations of the universe has become increasingly important for physicists, astrophysicists, and other domain scientists. Cosmological structures, galaxy formation, and the evolution of the universe are explored using dynamic particle simulations—such as smoothed particle hydrodynamics (SPH)—executed on large compute clusters. However, advances in computational power and storage have led to simulation outputs at the tera-, peta-, and even exascale. As a result, interactively visualizing

such massive datasets presents a significant challenge—particularly in maintaining responsiveness during data exploration.

Contributions

This dissertation addresses several key challenges in scientific visualization and proposes efficient solutions for the rendering and understanding of complex datasets. The presented work explores visualization techniques for vector field datasets from various scientific domains, as well as large particle sets from cosmological simulations. All proposed solutions leverage the massively parallel architecture of modern Graphics Processing Units (GPUs) to enable fast and interactive rendering.

First, a novel technique for geometry-based visualization of vector fields is introduced. This approach generalizes and unifies several existing methods into a flexible framework with a scalable, GPU-accelerated implementation [ZPRD22]. Second, a new opacity optimization method is proposed, based on recent moment-based techniques for signal reconstruction [ZRPD20]. Third, an interactive visualization framework for large particle datasets is developed, relying on level-of-detail (LoD) structures and GPU-based out-of-core (OOC) paging. This framework enables the interactive rendering of millions of particles entirely on the GPU.

Glyph-based Flow Visualization [ZPRD22] Line integral convolution (LIC) is a standard method for texture-based vector field visualization in both 2D and 3D. These methods produce dense representations of vector fields and eliminate the need to select seed points for particle tracing. However, 3D texture-based methods are computationally demanding due to texture convolution and volume rendering.

In contrast, geometry-based techniques generate individual glyphs that convey not only direction but also other attributes, such as rotation (via streamribbons) or divergence (via streamtubes). As a result, multiple glyph types are often needed to capture the diverse phenomena that coexist within a single vector field.

We present a novel geometry-based visualization method that unifies and extends prior approaches within a flexible, GPU-accelerated framework. Characteristic lines are mapped to a variety of glyphs, allowing users to define multiple cross-sectional shapes for extrusion.

These shapes are interpolated based on local field attributes or global user parameters, enabling a single line to use different glyph forms along its path—thus facilitating the visualization of multiple coexisting phenomena.

Additionally, the method approximates the appearance of 3D LIC by simulating volume integrals within the glyphs. Combined with fast, order-independent transparency, our implementation delivers real-time performance at high resolutions with moderate memory usage.

Moment-based Opacity Optimization [ZRPD20] Geometric occlusion is a common challenge in flow visualization, as dense yet uninformative geometry can obscure critical structures. Opacity optimization addresses this by reducing occlusion while enhancing the visibility of relevant features.

We propose a novel opacity optimization approach based on moment-based signal reconstruction. Unlike truncated Fourier series, moment-based reconstructions of feature importance and optical depth along view rays yield highly accurate results in sparse regions while remaining plausible in denser areas. These methods also avoid ringing artifacts and support compact storage and fast per-pixel evaluation—crucial for optimizing large and complex geometry.

Furthermore, we introduce a screen space filtering technique that operates directly on moment buffers to smooth optimized opacities. This method achieves visual quality comparable to object space smoothing while being independent of geometry type, making it broadly applicable and efficient for real-time use.

Out-of-Core Large Particle Visualization We present a visualization method for large particle datasets using LoD structures and GPU-based OOC paging. A GPU-friendly indexing tree is constructed to efficiently manage large-scale particle data, supporting LoD construction, and interactive rendering.

By leveraging this indexing structure and LoD representations, our method enables interactive visualization through parallel tree traversal and GPU rasterization. The system achieves interactive frame rates while rendering millions of particles, thanks to its highly efficient GPU-centric pipeline.

We demonstrate the effectiveness of our approach using the Illustris dataset—a large-scale cosmological simulation of galaxy formation and the evolution of the universe—highlighting its potential for enabling responsive visual exploration of complex scientific data.

Zusammenfassung

Motivation und Problemstellung

Die wissenschaftliche Visualisierung ermöglicht es, Rohdaten in aussagekräftige Darstellungen umzuwandeln, die unser Verständnis der zugrunde liegenden Phänomene verbessern. Mithilfe der Visualisierung können Fachleute, Wissenschaftler, Ingenieure und sogar unerfahrene Benutzer wertvolle Einblicke in Prozesse wie wissenschaftliche Simulationen oder reale Systeme gewinnen.

Die Visualisierung von Vektorfeldern ist ein aktives Forschungsgebiet, das Experten in verschiedenen Bereichen wie Biologie, Chemie, Medizin usw. bei der Untersuchung wichtiger Phänomene in Strömungsdaten unterstützt. Hierbei werden geometrische Strukturen wie Punkte, Linien und Flächen verwendet, um Charakteristika von Strömungen darzustellen. Allerdings sind diese Visualisierungen oft unübersichtlich. Dadurch können wichtige Merkmale durch weniger wichtige Merkmale verdeckt werden. In den letzten Jahren wurde dieses Problem mit verschiedenen Ansätzen angegangen, wie z. B. die Selektion von Stromlinien und Opacity Optimization [GTG17]. Opacity Optimization versucht dabei, wichtige innere Strukturen sichtbar zu machen, indem weniger wichtige Bereiche durchsichtig dargestellt werden. Die meisten dieser Techniken sind jedoch zu rechenaufwendig, insbesondere in Echtzeit und interaktiven Umgebungen.

Darüber hinaus ist das Verständnis großer und komplexer Teilchensimulationen des Universums für Physiker, Astrophysiker und andere Wissenschaftler immer wichtiger geworden. Kosmologische Strukturen, die Bildung von Galaxien und die Entwicklung des Universums können mit dynamischen Teilchensimulationen, wie den smoothed particle hydrodynamics (SPH) untersucht werden, die auf großen Rechenclustern ausgeführt werden. Mit dem kontinuierlichen Wachstum der Rechenleistung und der Speicherkapazität sind

die durch diese Simulationen erzeugten Datenmengen jedoch auf Größenordnungen von Petabytes und mehr angewachsen. Infolgedessen stellt die interaktive Visualisierung großer Partikeldatensätze eine große Herausforderung dar, wenn es darum geht, eine schnelle Rückmeldung für eine effektive Datenexploration zu liefern.

Beiträge

In dieser Dissertation beschäftige ich mich mit verschiedenen Herausforderungen im Bereich der wissenschaftlichen Visualisierung und schlage effiziente Lösungen für die Darstellung und das Verständnis komplexer Datensätze vor. Die vorgestellte Arbeit untersucht Visualisierungstechniken für Vektorfelddatensätze aus verschiedenen wissenschaftlichen Bereichen, sowie für große Partikelsätze aus Universumssimulationen. Die vorgeschlagenen Lösungen nutzen die massiv-parallele Architektur moderner GPUs, um eine schnelle und interaktive Darstellung zu ermöglichen.

Ich schlage zunächst eine neuartige Technik zur geometriebasierten Visualisierung von Vektorfeldern vor. Dieser Ansatz verallgemeinert und kombiniert mehrere bestehende Methoden in einem flexiblen Rahmen mit einer skalierbaren, GPU-beschleunigten Implementierung [ZPRD22]. Darüber hinaus stelle ich eine neue Methode zur Optimierung der Opazität in der Vektorfeldvisualisierung vor, um wichtige Merkmale sichtbar zu machen. Die Methode basiert auf aktuellen momentbasierten Techniken zur Signalrekonstruktion [ZRPD20]. Zuletzt stelle ich einen interaktiven Visualisierungsrahmen für große Partikelsätze vor. Diese Methode stützt sich auf Level-of-Detail (LoD)-Strukturen und GPU-basiertes Out-of-Core (OOC)-Paging. Der vorgeschlagene Ansatz ermöglicht das interaktive Rendering von Millionen von Partikeln auf dem Grafikprozessor unter Verwendung von LoD-Strukturen.

Glyph-based Flow Visualization [ZPRD22] Die line integral convolution (LIC) ist die Standardmethode für die texturbasierte Visualisierung von Vektorfeldern sowohl in 2D als auch in 3D. Diese Methode bietet eine dichte Vektorfelddarstellung, die das Auffinden von Startpunkten für die Partikelverfolgung überflüssig macht. Texturbasierte 3D-Methoden leiden jedoch unter einer hohen Rechenkomplexität aufgrund von Faltungsoperationen und

Volumenrendering.

Geometriebasierte Strömungsvisualisierungstechniken hingegen erzeugen individuelle geometrische Glyphen, die mehr Informationen als nur die Richtung vermitteln, wie z. B. Rotation durch Strömungsbänder oder Divergenz durch Strömungsrohre. Infolgedessen sind verschiedene Glyphen erforderlich, um verschiedene Phänomene zu visualisieren, die oft innerhalb eines einzigen Vektorfeldes nebeneinander bestehen.

Wir präsentieren eine neuartige Visualisierungstechnik für die geometriebasierte Visualisierung von Vektorfeldern. Unser Ansatz verallgemeinert und kombiniert mehrere bestehende Methoden in einem flexiblen Rahmen mit einer skalierbaren, GPU-beschleunig-ten Implementierung. Wir bilden charakteristische Linien auf eine Vielzahl von Glyphen ab, die es dem Benutzer ermöglichen, mehrere Querschnittsformen zu definieren. Unsere Methode interpoliert zwischen diesen Formen auf der Grundlage der Attribute des Vektorfelds und der charakteristischen Linien oder unter Verwendung globaler benutzergesteuerter Parameter. Dadurch kann eine einzige charakteristische Linie in verschiedenen Teilen unterschiedliche Querschnittsformen verwenden, was die Visualisierung verschiedener Phänomene erleichtert.

Darüber hinaus emuliert der vorgeschlagene Ansatz das Aussehen von 3D-LIC durch eine Annäherung des Volumenintegrals innerhalb unserer Glyphen. In Kombination mit order-independent transparency erreicht unsere GPU-Implementierung ein schnelles Rendering bei hohen Auflösungen, während der Speicherbedarf moderat bleibt.

Moment-based Opacity Optimization [ZRPD20] Geometrische Verdeckungen sind ein häufiges Problem bei der Visualisierung von Strömungen, da sie es schwierig machen, wichtige Informationen hinter dichten, aber weniger wichtigen Bereichen zu erkennen. Die opacity optimization ist eine weit verbreitete Technik, die die Durchsichtigkeit unwichtiger Bereiche manipuliert. Hierbei muss zwischen der Auswahl wichtiger Bereiche und der Vermeidung von Verdeckung abgewägt werden.

Es wurde ein neuartiger Ansatz für die opacity optimization entwickelt, der auf momentbasierten Techniken zur Signalrekonstruktion beruht. Im Gegensatz zu abgeschnittenen Fourier-Reihen sind momentbasierte Rekonstruktionen der Wichtigkeit und der optischen Tiefe entlang von Sichtstrahlen in dünn besiedelten Regionen sehr genau und in dicht besiedelten Gebieten noch plausibel. Zusätzlich leiden momentbasierte Methoden, im Gegensatz zu Fourier-Reihen, nicht unter Ringing-Artefakten. Diese Darstellung ermöglicht außerdem eine schnelle Auswertung und kompakte Speicherung, was für die Optimierung pro Pixel, insbesondere bei großen geometrischen Strukturen, unerlässlich ist.

Darüber hinaus beinhaltet die entwickelte Methode einen schnellen Screen Space-Filter--ungsansatz für optimierte Opazitäten, der direkt auf Momentpuffern arbeitet. Dieser Filteransatz ist für Echtzeit-Visualisierungsanwendungen geeignet und bietet eine vergleichbare Qualität wie die Objektraumglättung. Die Implementierung ist unabhängig von der Art der Geometrie, wodurch sie allgemein und leicht zu integrieren ist.

Out-of-Core Large Particles Visualization Es wurde eine Visualisierungsmethode für große Partikelsätze entwickelt, die auf LoD-Strukturen und GPU-basiertem out-of-core (OOC) Paging basiert. Wir verwenden GPU-basiertes OOC Paging, um eine Baumstruktur als Index für große Partikelsätze zu konstruieren. Unsere Baumstruktur kann für level-of-detail (LoD) Konstruktion und LoD Rendering verwendet werden. Mit dieser Methode können Millionen von Partikeln bei interaktiven Bildwiederholraten auf der GPU dargestellt werden. Wir evaluieren unsere Methode anhand der Partikelsimulation des Illustris-Datensatzes.

List of Publications

1. Mahmoud Zeidan, Christoph Peters, Tobias Rapp, and Carsten Dachsbacher. *Versatile Geometric Flow Visualization by Controllable Shape and Volumetric Appearance.* In Proceedings of Smart Tools and Applications in Graphics (STAG), 2022. Paper link.

2. Mahmoud Zeidan, Tobias Rapp, Christoph Peters, and Carsten Dachsbacher. *Moment-based Opacity Optimization.* In Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), 2020. Paper link.

Contents

Acknowledgments **iii**

Abstract **vii**

Zusammenfassung **xi**

List of Figures **xxi**

List of Tables **xxvii**

1 Introduction **1**
1.1 Outline and Thesis Organization 2

2 Introduction to Scientific Visualization and GPU Parallel Processing **5**
2.1 The Visualization Pipeline 5
2.2 OpenGL Programmable Shader Pipeline 7
2.3 The CUDA Programming Model 10
2.4 General-Purpose Computing on Graphics Processing Units (GPGPU) 14
2.4.1 Thread Management on GPU 14
2.4.2 Parallel Primitive Algorithms 16

3 Parallel Generation of Flow Geometry **25**
3.1 Introduction 25
3.2 Streamline Generation 27
3.2.1 Particle Tracing 27
3.3 Streamline Rasterization 32
3.4 Streamline Extrusion 36
3.5 Limitations and Perspectives 39

4 Interactive Glyph-based Flow Visualization **41**
4.1 Introduction 41
4.2 Flow Visualization 42
4.3 Geometry Generation 43
4.3.1 Integration of Characteristic Lines 44
4.3.2 Design and Placement of Cross-Sectional Glyphs 45
4.3.3 Morphing Cross-Sectional Glyphs 47
4.3.4 Controlling the Longitudinal Kernel 47
4.3.5 Generation of Triangle Meshes 48
4.4 Rendering 49
4.4.1 Shading 49
4.4.2 Opacity Computation along Volumetric Cross Sections 49
4.4.3 Order-Independent Transparency 51
4.5 Results and Evaluation 52
4.5.1 Applications 53
4.6 Conclusions and Outlook 56
4.7 Acknowledgments 57

5 Moment-based Opacity Optimization for Geometric Structures **59**
5.1 Introduction 59
5.2 Opacity Optimization 60
5.2.1 Decoupled Opacity Optimization 61
5.2.2 Fourier Opacity Optimization 62
5.3 Moment-Based Opacity Optimization 63
5.3.1 Accumulation of Importance 64
5.3.2 Opacity Optimization 66
5.3.3 Screen Space Filtering for Smoothed Opacity 67
5.4 Implementation 68
5.4.1 Generating Visualization Primitives 68
5.4.2 Moment-Based OIT 69
5.4.3 A-Buffers 70
5.4.4 Fourier Opacity Optimization 71
5.5 Results and Evaluation 71
5.5.1 Quality of Opacity Optimization 71
5.5.2 Validation Against Ground Truth 74
5.5.3 Screen Space Filtering 75

5.5.4 Performance Evaluation 75
5.6 Conclusions and Outlook 78

6 Efficient Visualization of Large Particle Datasets on GPU 79
6.1 Introduction 79
6.2 Related Work 80
6.3 Out-of-Core Paging 83
6.3.1 Homogeneous Data Paging 84
6.3.2 Two-Level Hierarchical Data Paging 86
6.4 Out-of-Core Chunk-Based Merging for Large Array Sorting 87
6.4.1 Dynamic Scheduling for Out-of-Core Chunk Merging 88
6.5 Two-Level Indexing Structure Construction for Level-of-Detail Particle Sets . 90
6.5.1 Dynamic Scheduling for Node Processing 91
6.5.2 Two-Level Hierarchy Construction 92
6.6 Level-of-Detail Construction 94
6.7 Results and Discussion 95
6.8 Conclusions and Outlook 105

7 Summary and Future Directions 107
7.1 Conclusions, Limitations, and Future Directions 107

Appendices 111

A Implementation Details of Homogeneous Data Paging 113

B Implementation Details of Out-of-Core Chunk-Based Merging for Large Array Sorting 121
B.1 Merging Two Sorted Chunks 122

C Implementation Details of Level-of-Detail Selection and Frustum Culling 131

Bibliography 133

List of Figures

2.1 The four common stages of the visualization pipeline. 5
2.2 OpenGL shader pipeline stages. 8
2.3 Blocks and grids hierarchy of 512 threads. Each thread block is organized into $3D$ $4 \times 4 \times 4$ threads, and the grid is organized into $2D$ 4×2 blocks. . 11
2.4 General NVIDIA device architecture with memory specifications as in the GeForce GTX 1080 Ti. 12
2.5 On the left, a scan example illustrates both inclusive and exclusive variants. On the right, a segmented exclusive scan is shown where each partition, marked by a head flag of 1, is scanned independently. The first partition is implicitly defined by the start of the array and does not require an explicit head flag. 17
2.6 A reduction operation over N elements runs in parallel in $\log(N)$ recursive steps. In each step, half of the elements are accumulated into the other half. 18
2.7 List compaction is performed by applying a scan operation over selection flags to determine the new locations of selected elements. On the right, the unselected elements are marked with flag values of 0 and are excluded from the compacted output. 18
2.8 Key-value sorting. 19
2.9 Sprite rendering of presampled sets using rejection sampling on the CPU from snapshot number 100 of the Illustris-3 dataset: top rendering using semi-transparent opacity and bottom rendering using fully opaque opacity. For each column, the number of particles in the corresponding presampled set is indicated below. 21
2.10 Clusters of particles with random color patterns generated on the Graphics Processing Unit (GPU), with the total number of particles indicated below the image. 22

3.1 Illustration of vector field orientation using two parallel particles. The unit tangent vector u follows the flow direction. The vector v is orthogonal to u, computed by projecting the separation vector between particles onto the direction perpendicular to u. 31

3.2 Turning a line segment into a quad (left) can result in gaps and overlaps (right) when applied to each segment independently. 32

3.3 Miter joint between successive line segments using a shared edge. 33

3.4 Streamline triangulation. Each pair of consecutive vertices along a streamline is expanded into a screen-aligned quad, which is then triangulated into two triangles for rendering. 33

3.5 Streamline extrusion. Left: a cross-sectional glyph is extruded along the streamline vertices. Middle: each consecutive pair of vertices is expanded into a closed surface by extending corresponding parallel edges into triangulated quads. Right: the start and end of the streamline are closed with triangulated caps derived from the cross-sectional shape. 37

3.6 Extrusion modes; left, a shared cross-sectional shape vertex results in a shared vertex between neighbor quads, right, two vertices are emitted for each quad. 38

4.1 An overview of our pipeline for vector field visualization. Data in green boxes is controlled by the user interactively. Data in blue boxes is updated at run-time. All operations (black rectangles) are GPU-accelerated. 44

4.2 Extrusion of cross-sectional glyphs using different shapes. We use flat normals for triangle and quad glyphs, and smooth normals for circle and ellipse glyphs. We use a constant radius as shown in the kernel in the bottom right of top row. 45

4.3 Using circle and quad glyphs, we allow several modes of interpolations along extruded geometry. In row A, we extrude all interpolation instants between quad and circle (left), and between circle and quad (right). In row B, we display four separate example instants of interpolation, where we selectively extrude an intermediate shape between quad and circle over the whole streamline. In row C, we use a user-specified threshold t along a streamline to selectively extrude the first glyph shape to all attribute values below t, and extrude the other glyph shape to the rest of the input range. 46

4.4 Extruding an ellipse cross-section shape using a constant kernel for extrusion (left), and ramp kernel (right). 48

4.5 Precomputed traveled distance (right) over cross-sectional disk shape (left). 50

4.6 The tornado dataset with a circular cross-sectional glyph and several ramps as a longitudinal kernel. 52

4.7 The tornado dataset with circular cross-section glyph and a ramp longitudinal kernel. Left: Geometry is rendered with a constant opacity value of 0.7 with all geometry. Right top: Approximate cross-sectional transmittance along geometry using our method. Right bottom: Approximate traveled distance using our cross-sectional transmittance approximation. 53

4.8 A ramp kernel and a circular cross-sectional shape are used to emulate texture sparsity with the trefoil knot dataset. From left to right we increase the number of streamlines. 54

4.9 Encoding vector field attributes using different cross-section glyphs and interpolation instants. High magnitude velocities are marked with streamribbons and low values are marked with interpolation instants between a disk shape and a streamribbon. 55

4.10 Encoding the rotational component of the vector field using different cross-sectional glyphs. We use a vertex curvature threshold ($t = 0.015$) so that high curvature values are marked with streamribbons and low values are extruded with circular cross-sectional glyphs. 55

5.1 Overview of our proposed moment-based opacity optimization technique. . 64

5.2 Visualization of the tornado vector field using 100 k points. The colors and importance are based on the velocity magnitude, and the rendering utilizes moment-based OIT. Notice how opacity optimization reveals the vortex at the center while preserving the less significant structures. 72

5.3 Comparison of three opacity optimization techniques using four challenging streamline datasets: Tornado, Borromean rings, Rayleigh-Bénard convection and trefoil knot. Note the overall improvement in clarity through opacity optimization and the differences in filtering between DOO and FOO or MBOO. Our technique achieves comparable quality to DOO at a substantially lower cost. 73

5.4 A comparison of opacity optimization techniques on the trefoil knot with both object space filtering and screen space filtering disabled. OIT uses A-buffers. In this setting, all differences are due to the approximation of the truncated Fourier series or the moment-based reconstruction. The bottom row shows L1-difference images with the Viridis color map. 74

5.5 The tornado dataset with two different line counts and different methods of filtering for opacity optimization. OIT uses A-buffers. Note how screen space filtering removes clutter in a fixed radius around important structures. 76
5.6 Two complicated triangle meshes with different importance. Opacity optimization removes foreground clutter that would otherwise hide the bunny in the bush. OIT uses an A-buffer. 77

6.1 The page table structure includes the I bit, which indicates whether a page resides in GPU memory; the R bit, which represents a page request; and the W bit, which is used for page write-back. The resident block index array is used during page evacuation to invalidate the corresponding I flag for a previously loaded page. 85
6.2 Dynamic page table structure. Given a set of nodes, each containing a different number of elements, we compute splits at the nodes so that each consecutive set of node elements can fit into a single page. In this illustrative example, we assume that each page can hold up to 8 elements. 86
6.3 Merging two sorted arrays using sampling merge sort. The sorted splitters in the middle provide guidance for identifying block boundaries in the source arrays. 89
6.4 Main building blocks of our tree construction algorithm: (a) generating Morton codes for the input particles; (b) sorting the particles using a chunk-based merge sort; (c) constructing the top-level hierarchy; (d) scheduling large leaf nodes from the top-level hierarchy to construct bottom-level hierarchies and merging particle attributes by adaptively loading subsets of nodes and their particles into the GPU in multiple rounds; (e) propagating particle attributes in the bottom-level treelets, with attribute merging performed at each level; and (f) propagating particle attributes from the bottom hierarchies back to the top-level hierarchy, with attribute merging performed at each level. . . . 91
6.5 We use binary search in the prefix sum of the number of particles in each node to find the partitions of nodes that fit within the GPU memory budget. In this illustrative example, we assume that the memory budget can accommodate up to 10 geometry elements, and each node does not contain more than 8 geometry elements. 92

6.6 For any red node to be split, we search in the Morton code splitters array to find the relevant Morton code block that contains the exact node split. Once the relevant Morton code block is loaded into the GPU, we locate the exact split point to divide the parent red node into two child blue nodes. 93
6.7 Bottom-up LoD construction. Left: Initial treelet where particles reside at leaf nodes, and each internal node references two child nodes. Right: After LoD construction, starting from the leaf nodes, we merge particle attributes and bounding boxes upwards until we reach the root node. 95
6.8 Frustum culling and LoD selection. The selected nodes forming the tree cut are shown in red. These nodes have passed the frustum check and represent the most appropriate LoD relative to the viewer's position. Blue nodes indicate already visited tree nodes, while grey nodes are either outside the viewing frustum or have a projected size below a defined threshold. 97
6.9 LoD rendering of snapshot 100 from the Illustris-3 dataset. The first column shows the full reference rendering, while the next three columns correspond to screen space thresholds of 2×2, 4×4, and 8×8 pixels. The cyan curve overlaid on the colormap represents the particle distribution within the dataset, while the yellow curve depicts the transfer function profile that controls opacity modulation. The bottom row shows the corresponding L1-difference images using the Viridis colormap. All images are rendered at a resolution of 1024^2 pixels. 99
6.10 LoD rendering of snapshot 100 from the Illustris-3 dataset using different screen space projection thresholds, indicated at the top of each column. From top to bottom, the viewing distance increases. The numbers below each image show the total number of displayed particles (or particle proxies). The first percentage indicates the ratio relative to the full reference rendering of 88,683,022 particles, while the second percentage indicates the ratio relative to the number of particles in the first column (i.e., with 1×1 pixel threshold) of the same row. All images are rendered at a resolution of 1920×1080 pixels. 100
6.11 Rendering of snapshot 100 from the Illustris-3 dataset (right), shown alongside 50% (middle) and 20% (left) uniformly sampled particles, using a 1×1 pixel threshold for the LoD cut. Below each image, we list the number of selected particles relative to the total number of particles in the dataset. All images are rendered at a resolution of 1024^2 pixels. 102

6.12 Rendering of near and far views of snapshot 100 from the Illustris-3 dataset (right), shown together with subsampled versions containing 50% (middle) and 20% (left) of the particles. A 4×4 pixel resolution is used for the LoD cut threshold, and all images are rendered at 1024×1024 pixels. The values reported below each image indicate the number of selected particles relative to the total particle count in the dataset, with the corresponding percentages given in parentheses. 104

List of Tables

2.1 NVIDIA GeForce GTX 1080 Ti technical specifications. 14

4.1 Frame time and geometry construction performance for various datasets and figure examples. Geometry is only regenerated if the dataset changes. . . . 56

5.1 Total frame times in milliseconds and primitive counts for the results in the figures above. Our techniques are marked with an asterisk. The fastest technique in each comparison is marked bold. Note that our technique clearly outperforms decoupled opacity optimization and performs similar to Fourier opacity optimization. 77

6.1 OOC sorting is performed on 88 million Morton codes, each 64 bits in length. Since the keys are divided into three chunks, two merge rounds are required to sort the entire key sequence. Please note that for the first row, the reported in-core time is included in the OOC time. In this table, each chunk C_i^r is marked by the merge round r and chunk index i in the respective merge round. 101

6.2 Indexing structure details and build time for the main construction stages. Numbers marked with * indicate the inclusion of OOC paging on the GPU and are expanded with further details in Table 6.3. 102

6.3 Out-of-core paging statistics of particle data during different stages of tree construction. In the second column, Morton codes are generated by chunking particle positions directly to GPU memory based on the allocated memory block size (i.e., page size). 103

6.4 Time measurements of LoD rendering using near and far views of several datasets with a 4×4 pixel threshold. Numbers marked with * indicate the inclusion of OOC paging on the GPU. 104

A.1 Unified bit flags and masks used for out-of-core page control, including loading, eviction, and write-back. 113
A.2 Host and device variables used in the out-of-core paging system. 114

1 Introduction

Scientific visualization is a branch of science that transforms raw data into meaningful representations, enabling a better understanding of the underlying phenomena. By utilizing visualization techniques, domain experts, scientists, engineers, and even novice users can gain valuable insights into processes such as scientific simulations or real-world experiments. In recent decades, scientific visualization has emerged as a standalone research field, building upon other scientific disciplines such as mathematics, statistics, cognitive science and perception, computer graphics, and rendering.

Scientific visualization is a well-established research field that conveys important information from input data in visual form. Users employ visualization to investigate, evaluate, explore, and gain insights into raw numerical data through visual and graphical elements. This includes, but is not limited to, drawings, plots, graphs, and animations extracted or derived from the input data.

Visualization is a combination of science and art. A visualization technique uses scientific tools to generate a meaningful representation of data, while artistic principles are applied to enhance the perception of the displayed concepts. In practice, there is no universal visualization technique that works for every problem. A good visualization method should match the objectives of the studied problem and provide deeper insights into the input data. Therefore, visualization techniques vary depending on the input data, its dimensionality, and associated attributes.

Today, visualization is an essential tool in many fields such as astrophysics, chemistry, physics, medicine, and beyond. With increasing computational power and processing capabilities, data sources have become more complex and significantly larger in size. This introduces new challenges for visualization techniques and opens promising directions for research and scientific investigation.

In this dissertation, we address several challenges in scientific visualization and propose efficient solutions for the rendering of complex datasets. Our work focuses on visualization techniques for vector-field data from diverse scientific domains, as well as large particle sets originating from cosmological simulations. The proposed methods exploit the massively parallel architecture of modern Graphics Processing Units (GPUs) to enable fast and interactive visualization.

In writing this thesis, I chose to provide as much technical detail as possible in order to give the reader a clearer understanding of the presented concepts and implementation choices. I also include and explain several code snippets to support reproducibility and to make the proposed techniques more accessible to interested readers and less experienced developers. This decision was inspired in part by Ingo Wald's PhD thesis [Wal04], which remains a valuable reference for many low-level aspects of ray tracing. I was also influenced by the level of abstraction and detailed explanation in Tobias Günther's PhD thesis [Gü16], which presents a robust and well-structured framework for flow visualization techniques.

Throughout this thesis, to aid readability, all code snippets focus on the essential algorithmic structure rather than implementation completeness. The code snippets are therefore provided for illustrative purposes to highlight the key algorithmic steps; non-essential implementation details are omitted, and the code is not intended to be a full or directly compilable program.

1.1 Outline and Thesis Organization

This thesis is organized into three main parts: an introduction to scientific visualization, efficient techniques for flow visualization, and methods for visualizing large particle datasets.

Firstly, we provide a brief introduction to scientific visualization and parallel processing in Chapter 2. We begin by describing the visualization pipeline in Section 2.1, followed by the OpenGL rasterizer in Section 2.2, the CUDA programming model in Section 2.2, and finally, the General-Purpose Computing on Graphics Processing Units (GPGPU) concepts in Section 2.4.

Secondly, we continue with an overview of flow visualization techniques and parallel methods for generating geometric flow structures on the GPU in Chapter 3. Chapter 4 introduces a novel geometric flow visualization technique for streamlines using controllable cross-

sectional shapes and arbitrary longitudinal kernels. Chapter 5 presents a novel streamline selection algorithm based on moment theory, leveraging recent advancements in moment-based techniques.

Thirdly, we investigate efficient techniques for visualizing large particle datasets. Chapter 6 presents a comprehensive visualization framework for large-scale particle sets, leveraging level-of-details (LoDs), GPU-driven out-of-core (OOC) paging, and OOC merge sorting. This system achieves interactive frame rates while rendering millions of particles using GPU-efficient strategies. Finally, in Chapter 7, we summarize our findings and outline promising directions for future research.

2 Introduction to Scientific Visualization and GPU Parallel Processing

This chapter provides a basic introduction to scientific visualization, rasterization, and parallel processing on the Graphics Processing Unit (GPU). We begin in Section 2.1 with an overview of the visualization pipeline, which outlines the main processing steps common to most visualization applications. Since we extensively use OpenGL [Khr24a] as the primary rendering platform throughout this thesis, we describe the OpenGL rendering pipeline in detail in Section 2.2. In Section 2.3, we introduce NVIDIA CUDA [NVI20] as a programming model for massively parallel processing on the GPU. Finally, in Section 2.4, we present several fundamental parallel algorithms that serve as core building blocks for processing and rendering in the subsequent chapters.

2.1 The Visualization Pipeline

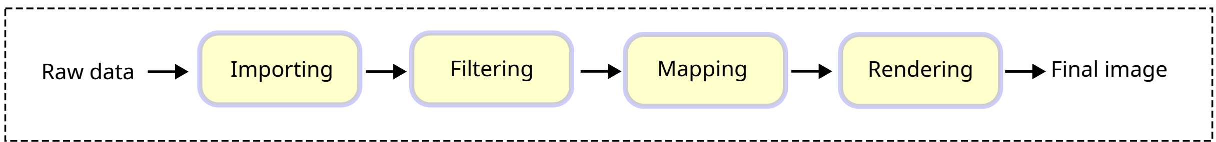


Figure 2.1: The four common stages of the visualization pipeline.

The visualization pipeline [HM90] represents a conceptual structure of visualization applications. It consists of four main stages: data importing, data filtering, data mapping, and data rendering (see Figure 2.1). The goal of the visualization pipeline is to process input data through a sequence of steps and deliver the desired images. Once the desired output is produced, users and experts should be able to examine and investigate the studied experiment,

providing interactive feedback that can be used to modify any of the previous visualization steps. In the following, we briefly explain each of the basic visualization stages. For more details on the visualization pipeline, we refer interested readers to the books [Wei06, JH04], the survey by Moreland [Mor13], or the recent survey by Ltifi et al. [LKB20].

Data Importing As mentioned earlier, input data usually comes from a simulation process or a physical experiment. In this stage, we select the features to be studied and determine the appropriate data representation. Additionally, we convert the input data into a suitable format for easier storage, access, and transfer.

Data Filtering In practice, not all input features can be directly mapped to aspects of the visualization process, making feature extraction a vital processing step. In this stage, we select the important features and relevant attributes for our study to be processed and analyzed in the following steps.

Data Mapping In this stage, input features are transformed into visual elements for exploration. This step precedes the rendering stage and helps identify the most suitable visual representations of the input data.

Data Rendering In the final stage, output images, plots, graphs, and animations are produced to give users an insightful understanding of the input data. Rendering and mapping are closely related in that both transform input features into visual representations. However, mapping is best viewed as an exploratory process, where design choices are made in preparation for producing high-quality renderings.

User Feedback At different stages of the visualization process, user feedback is used to interactively modify or enhance the pipeline. This includes, for example, selecting different input features, transforming data formats, or altering the rendering technique.

2.2 OpenGL Programmable Shader Pipeline

Rasterization is a vital technique for interactive and real-time rendering on the Graphics Processing Unit (GPU), making it a suitable choice for many visualization techniques. Over the years, two main development streams for rasterization have emerged: OpenGL [KSS16], an open standard developed by the Khronos Group [Khr24a] and implemented by various GPU vendors [Int68, Adv69, NVI93]; and DirectX [Mic23], a proprietary API developed and maintained solely by Microsoft, in close collaboration with GPU vendors to ensure driver support.

Rasterization processes a stream of geometric elements—such as triangle lists—through several parallel pipeline stages and produces screen pixels at the final step. Over time, rasterization has evolved to offer more flexibility and programmability to graphics developers [Wol18], enabling more complex applications and advanced techniques. Additionally, recent versions of DirectX have introduced hardware support for ray tracing to enable real-time, physically based, and realistic rendering [HAM19, AMHM21]. Moreover, Vulkan (a.k.a., next-generation OpenGL) [Khr24b] has been developed by the Khronos Group and hardware vendors to address the limitations of OpenGL. Vulkan provides developers with more control and flexibility for application development on modern GPUs [KLM25].

This thesis focuses on interactive vector field and particle-based visualization techniques. To support real-time and interactive rendering, frequent updates, and dynamic interaction required by these methods, we adopt OpenGL as the rendering platform. This choice reflects a deliberate emphasis on interactivity, flexibility, and performance rather than photorealistic image synthesis. In this section, we describe the OpenGL rendering pipeline and discuss its relevance to the visualization techniques presented in this work.

OpenGL defines six shader stages: vertex, geometry, tessellation control, tessellation evaluation, fragment, and compute. The compute shader is a standalone stage used for processing arbitrary data to leverage the GPU's high parallel throughput. The remaining shader stages form a fixed pipeline order, as shown in Figure 2.2. All stages except the vertex shader are optional and can be selectively used depending on application requirements. An OpenGL shader is a small program written in a C-style language known as GLSL (OpenGL Shading Language). Each shader is executed in parallel by the GPU for its respective input elements.

Recent graphics APIs have introduced alternative rendering pipelines that depart from the traditional stage-based rasterization pipeline described above. In particular, modern versions

of DirectX and Vulkan support mesh shaders, which replace the conventional vertex, tessellation, and geometry shader stages with a unified, fully programmable geometry processing stage. Mesh shaders enable GPU-driven rendering by allowing applications to generate, cull, and assemble geometry entirely on the GPU, thereby reducing CPU overhead and improving scalability for large and dynamic scenes [Khr22].

While mesh-shader-based pipelines are well suited for data-driven rendering and massive geometry workloads, they are not currently supported in OpenGL. Consequently, the visualization techniques presented in this thesis are implemented using the traditional OpenGL rasterization pipeline. Nevertheless, many of the concepts discussed—such as particle expansion, view-dependent filtering, and level-of-detail control—are largely pipeline-agnostic and could be adapted to mesh-shader-based frameworks in future work.

In the following, we briefly explain each stage of the OpenGL rendering pipeline.

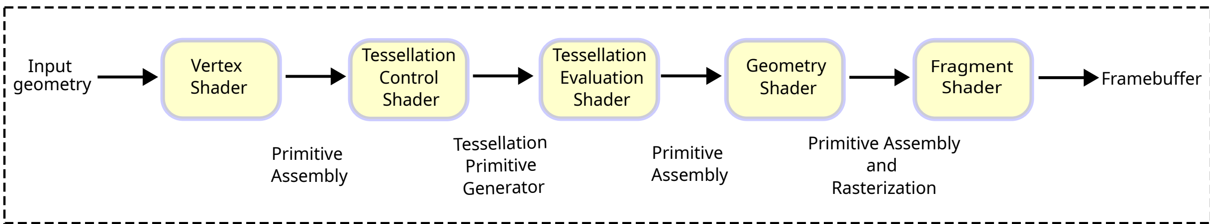


Figure 2.2: OpenGL shader pipeline stages.

Vertex Shader In this stage, input geometry is passed from the CPU via input buffers. For each input vertex, a corresponding output vertex is produced. The vertex shader typically applies transformations (e.g., camera mapping and screen projection), and unless additional transformations are applied in later stages, vertices are passed to the fragment shader in normalized device coordinates (NDC). NDC represents a 2D virtual viewport, with coordinates ranging from -1 to $+1$ along both the x and y axes. During this stage, vertices are assembled according to the specified configuration—for example, three vertices are grouped into a triangle.

Tessellation Shader This optional stage allows geometric patches to represent object surfaces based on viewing parameters. Applications such as level-of-detail (LoD) rendering use tessellation to adaptively generate surface subdivisions depending on the viewing distance.

Geometry Shader Although optional, the geometry shader is useful in certain applications. It receives the complete input primitive (e.g., triangle or point) and can modify or augment it. The output geometry type may differ from the input type—for instance, a single point can be expanded into a triangle.

Because geometry shader accesses full primitives, the vertex shader output is passed to the geometry shader as an array. The geometry shader may emit zero or more primitives, but all must be of the same output type.

Primitive Assembly The previous stages operate on vertices. Before reaching the fragment shader, the pipeline assembles vertices into complete geometric shapes. The fragment shader expects input in normalized device coordinates, which corresponds to the spatial extent of the framebuffer object. This is where output values (e.g., color, depth, opacity) are accumulated. Per-vertex attributes are interpolated across the primitive and passed to the next stage.

Clipping Geometry that lies outside the NDC extent is clipped. New geometry may be generated if a primitive intersects the boundary. For example, a triangle partially outside the NDC bounds may become a convex polygon after clipping.

Rasterization In this stage, updated geometry is mapped to screen pixels. The GPU determines which pixels are covered by which geometric elements.

Fragment Shader The fragment shader generates values for each output buffer (i.e., the framebuffer or G-buffer [Lau10]). For example, the color buffer receives color values, while the depth buffer receives depth values. Fragment shader outputs are optional and depend on application needs. Many applications write to multiple buffers to enable realistic rendering and post-processing effects [Lau10, FLB*09, LD12].

2.3 The CUDA Programming Model

Flynn [Fly72, Fly66] classified computer architectures based on the number of instruction streams and data streams into four categories: 1) Single Instruction Stream, Single Data Stream (SISD), 2) Single Instruction Stream, Multiple Data Streams (SIMD), 3) Multiple Instruction Streams, Single Data Stream (MISD), and 4) Multiple Instruction Streams, Multiple Data Streams (MIMD) [Roo99, Par99]. Originally, this classification assumed simple instructions and data elements, with parallelism achieved by concurrently processing multiple data items. As computational demands increased, parallel execution evolved to support larger and more complex instruction sets, enabling full programs to be executed in parallel. This gave rise to models such as Single Program, Multiple Data Streams (SPMD) [Dar01, MRG18, DGNP88, DRPS87, ABDVC87, AL83] and Multiple Programs, Multiple Data Streams (MPMD).

Modern Graphics Processing Unit (GPU) architectures have operationalized the concept of Single Instruction, Multiple Threads (SIMT) [LKA13]. SIMT enables hardware-level scheduling and the parallel execution of thousands to millions of threads. A key difference between SIMT and SPMD is that SIMT enforces a lock-step execution model within a group of threads (i.e., warps in NVIDIA GPUs) [AL09].

With the evolution of GPU hardware, manufacturers introduced software APIs like OpenCL [SGS10] and CUDA [NVI20] to support parallel programming and general-purpose applications. This led to the emergence of General-Purpose Computing on Graphics Processing Units (GPGPU), extending the use of GPUs beyond graphics and rendering. GPGPU allows a wide range of scientific and computational applications to leverage the high parallel throughput of modern GPUs, achieving substantial performance gains in fields such as physics, chemistry, scientific computing, cybersecurity, and machine learning.

CUDA Programming Model CUDA (Compute Unified Device Architecture) was introduced by NVIDIA in 2007 as a C++ extension for parallel processing on GPUs. It provides fine-grained control over memory management and thread scheduling on the GPU device.

Thread Hierarchy In CUDA, threads are organized into thread blocks. Each block can be configured in up to three dimensions, and these blocks are arranged into a grid (1D, 2D, or 3D) (see Figure 2.3). The grid and block dimensions are specified using the kernel launch

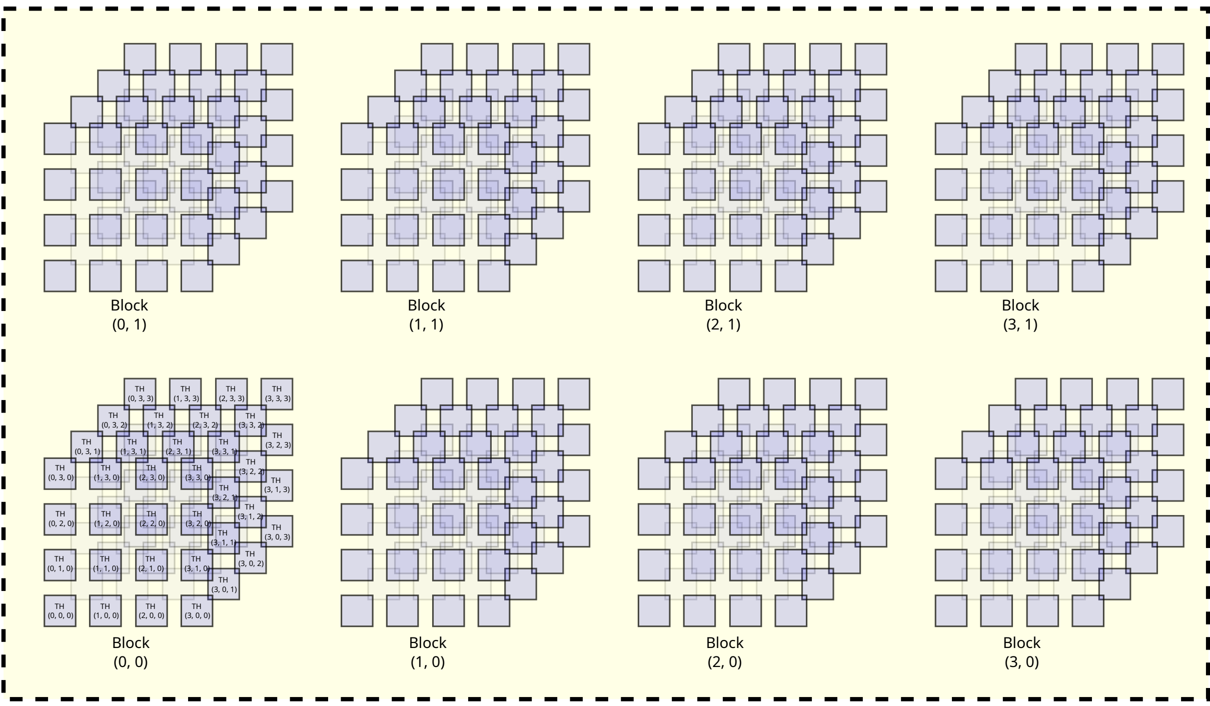


Figure 2.3: Blocks and grids hierarchy of 512 threads. Each thread block is organized into $3D$ $4\times4\times4$ threads, and the grid is organized into $2D$ 4×2 blocks.

syntax <<< grid_dim, block_dim >>>. Within a thread, *threadIdx* gives the thread's index in its block, *blockIdx* gives the block's index in the grid, and *blockDim* provides the block size [SK10]. Flattening these indices allows for the computation of a globally unique thread ID across the entire kernel launch.

Memory Hierarchy GPU threads can access data from several memory spaces, each with different size, scope, and latency (see Figure 2.4). Global memory is the largest memory space and is visible to all threads. Shared memory is faster than global memory and is accessible to all threads within a block. Local memory is private to each thread and is mapped to global memory, typically used for temporary variables. Constant and texture memory are read-only, cached memory spaces designed for fast access. Constant memory holds shared constant values, while texture memory enables fast lookups with optional filtering support.

Processing Barriers and Thread Synchronization CUDA provides intra-block synchronization via the *__syncThreads()* function, which ensures that all threads in a block reach a specific execution point before continuing. This is essential for scatter/gather operations and

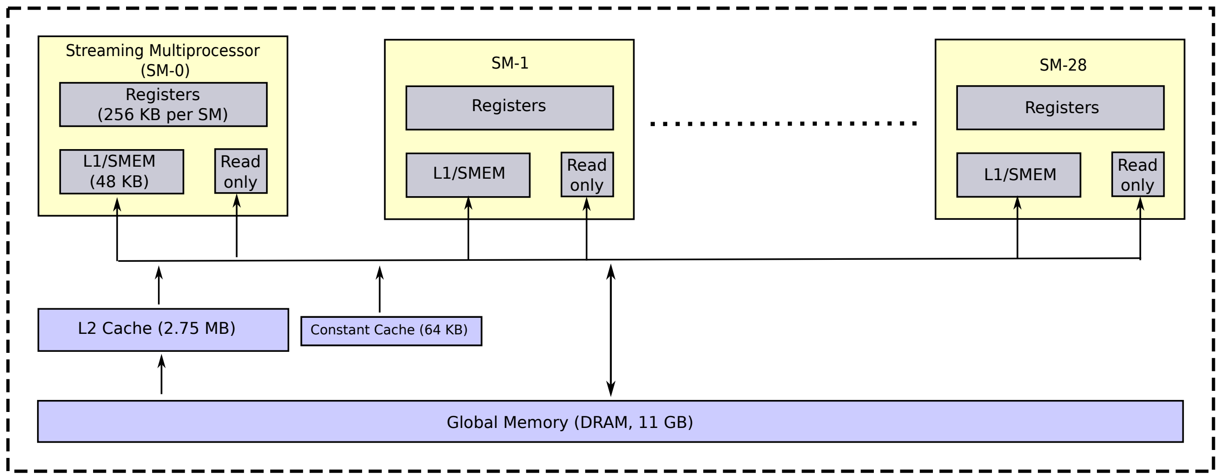


Figure 2.4: General NVIDIA device architecture with memory specifications as in the GeForce GTX 1080 Ti.

data sharing within a block. Although CUDA does not natively support inter-block synchronization, semaphore-like mechanisms can be implemented using atomic operations [NVI20].

Atomic Operations Current versions of CUDA support atomic operations on common data types, including `int`, `float`, `unsigned int`, and `long` [NVI20]. One key feature of atomic operations is that they return the previous value stored at the memory location, which is useful for generating offsets or implementing locks. For example, a binary semaphore can be created using `atomicExch`: threads enter a critical section only when the returned value is 0, indicating the lock was unoccupied [Sta14].

A Simple Kernel Example Listing 2.1 illustrates a simple kernel function for adding n numbers in parallel, where the kernel is executed n times concurrently. In this example, we explicitly demonstrate device memory allocation, deallocation, and memory transfer between the host and device. However, in all subsequent code snippets, we omit the memory allocation and transfer code segments, as well as error code checks, for the sake of clarity and conciseness.

```
1  #include <iostream>
2  #include <cuda_runtime.h>
3
4  #define NUM_ELEMENTS 512  // Total number of elements
5
6  // CUDA kernel for vector addition
7  __global__ void vec_add(const float* input_a, const float* input_b, float* output, int num_elements) {
8      int thread_id = threadIdx.x;
9      if (thread_id < num_elements) {
10         output[thread_id] = input_a[thread_id] + input_b[thread_id];
11     }
```

```
12 }
13
14 int main(int argc, char *argv[]) {
15     // Host memory allocation
16     float host_a[NUM_ELEMENTS], host_b[NUM_ELEMENTS], host_c[NUM_ELEMENTS];
17
18     // Initialize input data on host
19     for (int i = 0; i < NUM_ELEMENTS; ++i) {
20         host_a[i] = static_cast<float>(i);
21         host_b[i] = static_cast<float>(2 * i);
22     }
23
24     // Device memory allocation
25     float *device_a, *device_b, *device_c;
26     cudaMalloc(&device_a, NUM_ELEMENTS * sizeof(float));
27     cudaMalloc(&device_b, NUM_ELEMENTS * sizeof(float));
28     cudaMalloc(&device_c, NUM_ELEMENTS * sizeof(float));
29
30     // Copy input data to device
31     cudaMemcpy(device_a, host_a, NUM_ELEMENTS * sizeof(float), cudaMemcpyHostToDevice);
32     cudaMemcpy(device_b, host_b, NUM_ELEMENTS * sizeof(float), cudaMemcpyHostToDevice);
33
34     // Kernel launch: 1 block, NUM_ELEMENTS threads
35     vec_add<<<1, NUM_ELEMENTS>>>(device_a, device_b, device_c, NUM_ELEMENTS);
36
37     // Optional: check for kernel launch errors
38     cudaError_t cuda_status = cudaGetLastError();
39     if (cuda_status != cudaSuccess) {
40         std::cerr << "CUDA kernel launch failed: " << cudaGetErrorString(cuda_status) << std::endl;
41     }
42
43     // Copy result back to host
44     cudaMemcpy(host_c, device_c, NUM_ELEMENTS * sizeof(float), cudaMemcpyDeviceToHost);
45
46     // Display first 10 results
47     for (int i = 0; i < 10; ++i) {
48         std::cout << "host_c[" << i << "] = " << host_c[i] << std::endl;
49     }
50
51     // Free device memory
52     cudaFree(device_a);
53     cudaFree(device_b);
54     cudaFree(device_c);
55
56     return 0;
57 }
```

Listing 2.1: A simple kernel example that adds N elements from two arrays, with GPU memory allocation, deallocation, and kernel launch performed on the CPU.

Since all experiments in the following chapters were conducted using an NVIDIA GeForce GTX 1080 Ti, its relevant technical specifications are listed in Table 2.1.

Memory Specifications	
Memory Size	11 GB
Memory Type	GDRSX
Memory Bus	352 bit
Bandwidth	484.4 GB/s

Theoretical Performance	
Pixel Rate	139.2 GPixel/s
Texture Rate	354.4 GTexel/s
FP16 (half)	177.2 GFLOPS (1:64)
FP32 (float)	11.34 TFLOPS
FP64 (double)	354.4 GFLOPS (1:32)

Graphics Features	
DirectX	12
OpenGL	4.6
OpenCL	3.0
Vulkan	1.3
CUDA	6.1
Shader Model	6.8

Table 2.1: NVIDIA GeForce GTX 1080 Ti technical specifications.

2.4 General-Purpose Computing on Graphics Processing Units (GPGPU)

In this section, we introduce the fundamentals of thread scheduling on GPUs and present parallel primitive algorithms that serve as foundational building blocks for developing more complex parallel applications.

2.4.1 Thread Management on GPU

```
1  __global__ void reverse_d(uint32_t* in_array, uint32_t* out_array, uint32_t n) {
2
3      int idx = blockDim.x * blockIdx.x + threadIdx.x;
4
5      if (idx >= n)
6          return;
7
8      // Reverse the array elements
9      out_array[idx] = in_array[n - 1u - idx];
10 }
11
12 int main(int argc, char *argv[]) {
13
14     uint32_t n = 1000;  // Example array size, should be set based on actual data
15     uint32_t *in_array, *out_array;
16
17     // Allocate device memory and fill input data
18
19     // Define block and grid sizes
20     dim3 threads_per_block(256);  // 256 threads per block
21     dim3 num_blocks = (n + threads_per_block.x - 1) / threads_per_block.x;
22
23     // Launch the kernel to reverse the array
24     reverse_d<<<num_blocks, threads_per_block>>>(in_array, out_array, n);
25
26     // Check for errors after kernel launch
27
28     // Free device memory
29
30     return 0;
31 }
```

Listing 2.2: A CUDA kernel that reverses an array of N elements into an output array using parallel threads.

Linear Thread Launch A simple way to process a set of N elements using CUDA is to launch W workgroups, each with S_W threads, where $W = \lceil \frac{N}{S_W} \rceil$. Listing 2.2 shows an illustrative example that reverses the elements of an input array into an output array. Inside the kernel, the global thread index is obtained as a flattened value computed from the group index, group size, and thread index (Line 3).

Group-Based Thread Launch In certain scenarios—such as processing tree nodes or clusters, each with a relatively small number of elements—we may wish to leverage the thread block hierarchy to process input data. In such cases, we assign each data cluster to a thread block. By doing so, we can benefit from the built-in thread synchronization and shared memory, allowing algorithms to be designed to take advantage of these features.

```
1 __global__ void process_groups_d(const uint32_t* groups_start, const uint32_t* groups_size, float* groups_value,
      const float* elements_value, float* elements_group_idx) {
2
3     int idx = threadIdx.x + blockDim.x * blockIdx.x;
4     int group_idx = blockIdx.x;
5
6     __shared__ uint32_t shared_group_idx;
7     __shared__ float shared_sum_group_value;
8
9     if (threadIdx.x == 0) {
10        shared_group_idx = group_idx;
11        shared_sum_group_value = 0.0f;
12    }
13
14    __syncthreads(); // Synchronize threads within the block
15
16    int group_start_idx = groups_start[group_idx];
17    int group_size_val = groups_size[group_idx];
18
19    // Process elements within the group
20    for (int i = threadIdx.x; i < group_size_val; i += blockDim.x) {
21        int element_idx = group_start_idx + i;
22
23        // Update the sum for the group using atomic operation
24        atomicAdd(&shared_sum_group_value, elements_value[element_idx]);
25
26        // Assign the group index to the element group index array
27        elements_group_idx[element_idx] = shared_group_idx;
28    }
29
30    __syncthreads(); // Synchronize threads before updating global memory
31
32    if (threadIdx.x == 0) {
33        groups_value[group_idx] = shared_sum_group_value;
34    }
35 }
36
37 int main(int argc, char *argv[]) {
38
39    uint32_t num_clusters;
40    uint32_t *groups_start, *groups_size;
41    float *groups_value, *elements_value, *elements_group_idx;
42
43    // Allocate device memory and fill input data
44
45    // Define block and grid sizes
46    dim3 threads_per_block(256);
47    dim3 num_blocks(num_clusters);
```

```
48
49      // Launch the kernel to handle each group of data via a thread block
50      process_groups_d<<<num_blocks, threads_per_block>>>(groups_start, groups_size, groups_value, elements_value,
         elements_group_idx);
51
52      // Check for errors after kernel launch
53      // Free device memory
54
55      return 0;
56  }
```

Listing 2.3: A CUDA kernel illustrating group-level cooperation using shared memory and atomic operations. Each thread processes a small batch of data elements and makes use of fast shared memory and thread barriers for data collection, scattering, and distribution.

We highlight these concepts in Listing 2.3. In this example, each data group is assigned to a thread block, and threads within the thread block cooperate to process the corresponding group's data elements. Elements of each group are defined by a start index and the number of elements. In this example, we emphasize the use of shared memory and local group synchronization. In Lines 9–12, we allow the first thread to get the group's index into a shared variable and clear a shared sum value to 0. Then, we call the group barrier in Line 14. This group synchronization ensures that all threads within a thread block do not proceed further with processing steps until all threads reach this point. At this stage, each thread picks an element value relative to the group start and adds the value to the shared variable, while also scattering the group index to the corresponding element index. After processing group elements, we use another barrier to ensure that all threads accumulated their elements' values to the group accumulator. In the final step in Lines 32–34, we allow the first thread to store the computed group sum into the global memory for the respective group.

2.4.2 Parallel Primitive Algorithms

Parallel algorithms can generally be broken down into consecutive steps of smaller parallel tasks. In most scenarios, these smaller parallel constructs can employ pre-implemented, standardized parallel algorithms, referred to as primitive algorithms. Since our work follows this approach, we frequently use parallel primitive algorithms as key building blocks for constructing data structures and rendering. To ensure that this thesis remains self-contained, we provide a brief review and explanation of relevant GPU parallel primitive algorithms. Whenever applicable, we leverage the NVIDIA CUB API [NL] to take advantage of optimized, existing implementations of parallel primitives, thereby avoiding redundant implementations.

CUB (CUDA Unbound) is a high-performance, flexible C++ template library for CUDA programming. It provides a collection of reusable parallel algorithms and data structures to optimize GPU performance on NVIDIA GPUs. CUB includes efficient implementations for common parallel operations such as reductions, scans, sorting, and more. Its design allows for easy integration into CUDA applications, leveraging advanced GPU capabilities to achieve high performance with minimal overhead.

Scan The scan primitive [SHZO07] is an operation that is performed on array elements to produce an array of the same size where each element in the output array is a function (e.g., summation, minimum, or maximum) of the preceding elements in the input array. A scan operation can be either exclusive or inclusive depending on the inclusion or the exclusion of the corresponding element in the input array. A segmented scan performs the scan operation on predefined separate partitions of input array independently (see Figure 2.5).

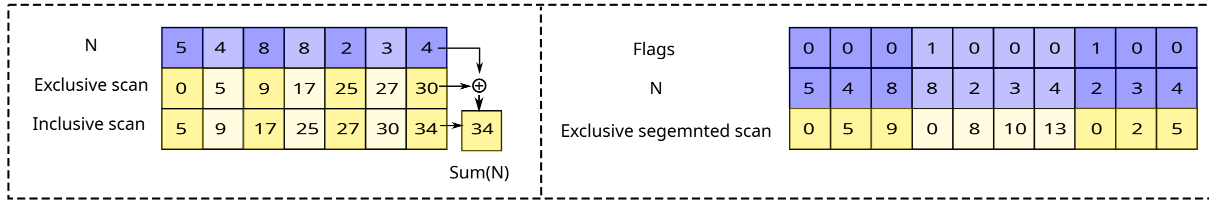


Figure 2.5: On the left, a scan example illustrates both inclusive and exclusive variants. On the right, a segmented exclusive scan is shown where each partition, marked by a head flag of 1, is scanned independently. The first partition is implicitly defined by the start of the array and does not require an explicit head flag.

Reduction A reduction operation [Har07] computes a value from an array of elements after applying a binary primitive operation between all array elements (see Figure 2.6). A segmented version of parallel reduction has been presented and employed for kd-tree construction in [ZHWG08].

List Compaction A stream compaction primitive operation [SHZO07] is used to compact a set of non-contiguous elements from the input array into a contiguous block. This operation can be implemented on top of a scan operation [SHZO07] (see Figure 2.7). List compaction is commonly used for compacting array elements after parallel processing steps that produce

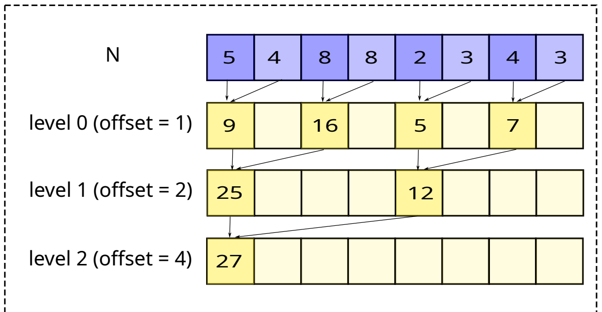


Figure 2.6: A reduction operation over N elements runs in parallel in $\log(N)$ recursive steps. In each step, half of the elements are accumulated into the other half.

arbitrary output elements with gaps of null elements. This process is useful since memory is managed by the application developer and is a limited resource on GPU.

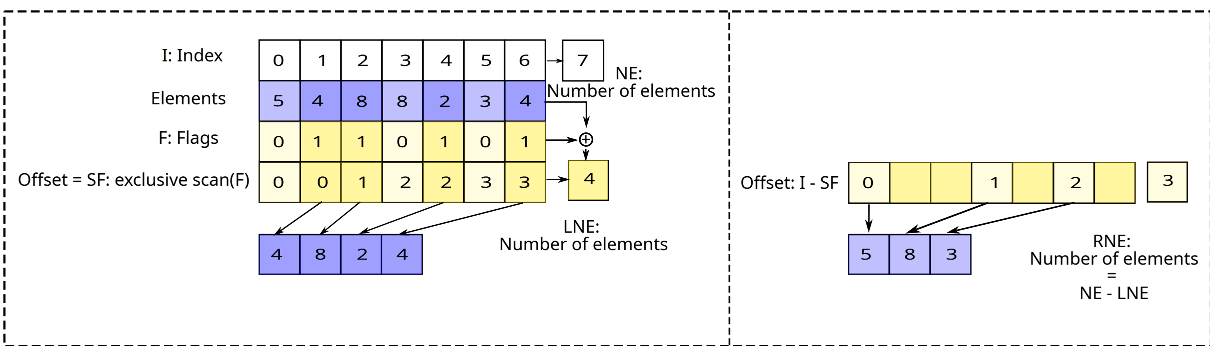


Figure 2.7: List compaction is performed by applying a scan operation over selection flags to determine the new locations of selected elements. On the right, the unselected elements are marked with flag values of 0 and are excluded from the compacted output.

Sorting Parallel sorting on GPU has been extensively studied in the past decades with many sorting variants such as bitonic sort [BP04], radix sort [SHG09, HKS09], and quick sort [KW05]. Since memory is a limited resource on the GPU, and efficient use of caching is critical, key-value sorting is employed to organize data blocks. In this approach, the value array aligns with each key and stores the index of the corresponding data block. After sorting, the data blocks can be accessed indirectly using the indices in the value array, or a parallel round of blocks relocation can be performed based on the sorted index sequence (see Figure 2.8).

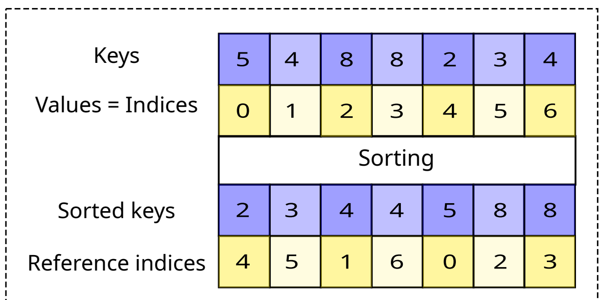


Figure 2.8: Key-value sorting.

Segmented Sort Segmented sort is a parallel sorting technique in which the input array is divided into multiple segments, and each segment is sorted independently. On Graphics Processing Units (GPUs), segmented sort is particularly useful for applications involving batched data, such as database operations or key-value grouping. Rather than sorting the entire dataset globally, the algorithm performs sorting within each segment in parallel. This enables efficient utilization of GPU resources, as each thread block can process one or more segments independently.

Merrill et al. [SPCB22] evaluated fast segmented sort implementations on GPUs, comparing multiple parallel approaches and analyzing their performance across varying segment sizes and workload distributions. A closely related technique is array sort, which involves sorting a large number of independent arrays on modern GPUs. Awan and Saeed [AS16] proposed GPU-ArraySort as a parallel, in-place solution optimized for such scenarios.

Pseudo-Random Number Generator Random numbers are sequences of values generated according to a specified probability distribution $P(x)$. They are essential components in numerical methods, particularly for approximating the integration of complex mathematical functions [PJH16]. In practice, we use pseudo-random number generators (PRNGs), which simulate the behavior of truly random numbers. However, due to numerical limitations, PRNGs exhibit periodicity. A high-quality pseudo-random number generator should have a long period and accurately follow the intended probability distribution. One widely used PRNG is the Mersenne Twister [MN98], known for its good performance in scientific applications and a period of $2^{19937} - 1$.

For a practical introduction to random sampling and its application in Monte Carlo methods, see [HM22]. Additionally, the books [PJH16, Jen01] present theoretical foundations and practical details for using Monte Carlo techniques in ray tracing and light simulation.

Listing 2.4 illustrates a simple C++ example of probabilistic element selection based on uniform random sampling. A pseudo-random number generator is used to produce uniform random values, and each input element is accepted if the generated value falls below a prescribed threshold. In this way, a fixed percentage (i.e., 10%) of the input elements is selected using an acceptance–rejection strategy. Figure 2.9 shows the rendering of several sampled particle sets from the Illustris dataset [VGS*14] generated using this approach.

```
1 #include <iostream>
2 #include <random>
3
4 int main(int argc, char* argv[]) {
5     std::random_device rd;                      // Obtain a random seed from the hardware
6     std::mt19937 generator(rd());               // Mersenne Twister engine seeded with rd()
7     std::uniform_real_distribution<> distribution(0.0, 1.0); // Uniform distribution between 0.0 and 1.0
8
9     const double sample_ratio = 0.1;            // Sampling ratio
10    const unsigned int num_elements = 100;      // Total number of elements
11
12    for (unsigned int i = 0; i < num_elements; ++i) {
13        double value = distribution(generator);  // Generate a random value
14
15        // Select element if value < sampling ratio
16        if (value < sample_ratio) {
17            std::cout << "Element " << i << " selected\n";
18        }
19    }
20
21    return 0;
22 }
```

Listing 2.4: Rejection sampling using a pseudo-random number generator on the CPU.

The CUB API [NL] also provides an implementation of a parallel pseudo-random number generator with a period greater than 2^{190}. Using this parallel random generator, each thread independently draws a random sequence from a pool based on the given distribution. Sequences generated with different seeds typically exhibit no statistical correlation. However, certain seed choices may result in statistically correlated sequences. Conversely, sequences generated using the same seed but different sequence indices are generally uncorrelated.

For optimal parallel pseudorandom number generation, each experiment should have a unique seed. Within an experiment, every computational thread should be assigned a distinct sequence number. If the experiment spans multiple kernel launches, it is advisable to maintain the same seed for threads across kernel launches, while assigning sequence numbers in a monotonically increasing order. To minimize setup time, random states can be preserved in global memory between kernel launches when the same thread configuration is used.

In Listing 2.5, we use the standard CUDA random number generator to generate random colors for cluster display. We begin by initializing random states with a random seed and

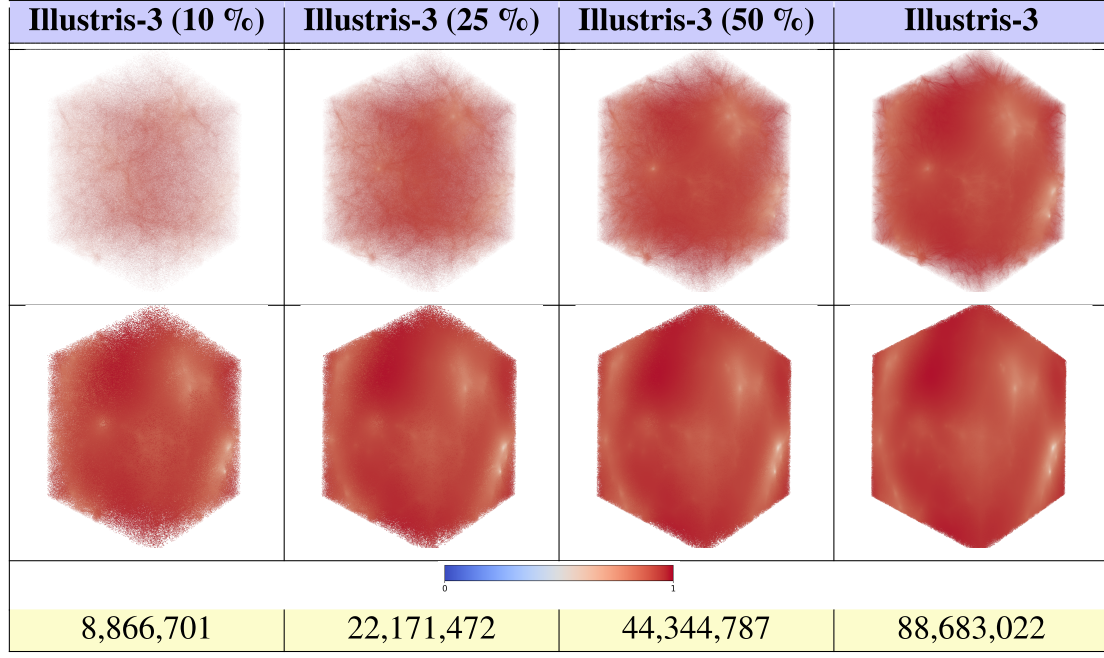


Figure 2.9: Sprite rendering of presampled sets using rejection sampling on the CPU from snapshot number 100 of the Illustris-3 dataset: top rendering using semi-transparent opacity and bottom rendering using fully opaque opacity. For each column, the number of particles in the corresponding presampled set is indicated below.

thread index. Then, we launch parallel threads to generate 3-component random colors and pack each color into an unsigned integer value. Figure 2.10 shows particles cluster colors assigned to particles based on the Morton code pattern [Mor66].

```
#include <curand_kernel.h>  // Include for CURAND kernel API

__global__ void setup_random_generators(curandState* states, uint64_t seed, uint32_t num_states) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx >= num_states)
        return;
    // Initialize the random generator state for each thread with the same seed, a different sequence number, no offset
    curand_init(seed, idx, 0, &states[idx]);
}

__global__ void generate_random_colors(curandState* states, uint32_t* cluster_colors, uint32_t num_clusters) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx >= num_clusters)
        return;

    curandState local_state = states[idx];  // Load state for this thread

    // Generate random values for RGB channels
    float rr = curand_uniform(&local_state);
    float rg = curand_uniform(&local_state);
    float rb = curand_uniform(&local_state);

    // Convert the float values (0.0 to 1.0) to 8-bit unsigned integer color channels (0-255)
    uint32_t packed_color = (clamp(uint32_t(rr * 255), 0u, 255u)) |
                            (clamp(uint32_t(rg * 255), 0u, 255u) << 8) |
                            (clamp(uint32_t(rb * 255), 0u, 255u) << 16) |
                            (clamp(uint32_t(1.0f * 255), 0u, 255u) << 24);  // Full alpha channel set to 255 (opaque)
```

```

    cluster_colors[idx] = packed_color;  // Store the packed color in the array
}

int main(int argc, char* argv[]) {
    uint32_t num_clusters = 1000;  // Example number of clusters
    curandState* states;
    uint32_t* cluster_colors;

    // Allocate memory for states and cluster_colors on the device

    dim3 threads_per_block(256);  // Define number of threads per block
    dim3 num_blocks((num_clusters + threads_per_block.x - 1) / threads_per_block.x);  // Define number of blocks
    uint64_t seed = time(0);  // Use current time as a seed for randomness
    setup_random_generators<<<num_blocks, threads_per_block>>>(states, seed, num_clusters);
    generate_random_colors<<<num_blocks, threads_per_block>>>(states, cluster_colors, num_clusters);

    // Error check for kernel execution
    // Free device memory

    return 0;
}
```

Listing 2.5: Generating random cluster colors on the GPU using the standard CUDA random number generator.

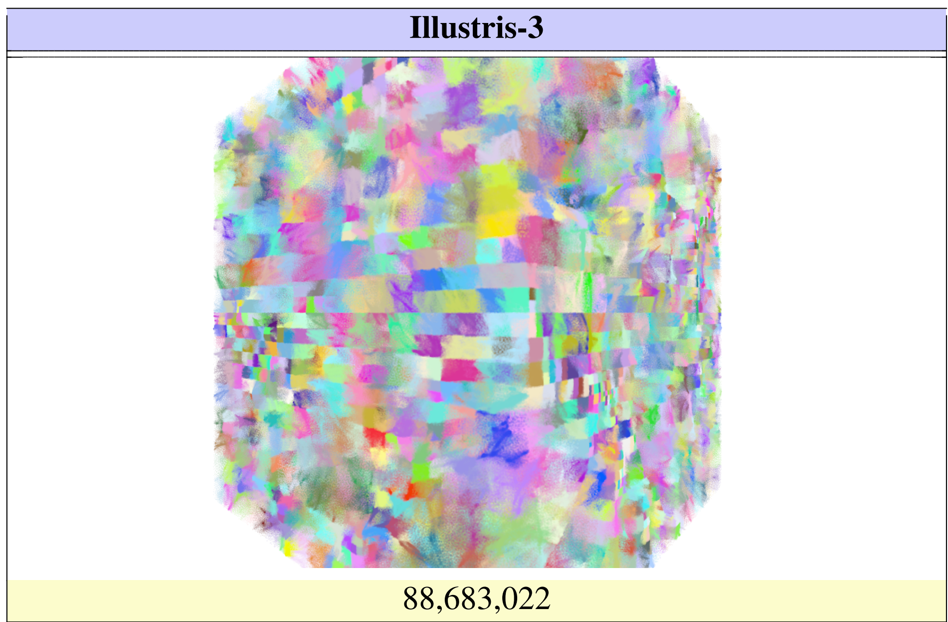


Figure 2.10: Clusters of particles with random color patterns generated on the GPU, with the total number of particles indicated below the image.

Less Frequently Used Parallel Primitive Algorithms **Multisplit** is a parallel primitive that efficiently categorizes input data into contiguous buckets. While traditionally implemented

using sorting algorithms, it can be optimized for GPUs to avoid the inefficiencies of sorting [ADMO17]. **Compress-Sort-Decompress** [GL10] is a GPU-friendly sorting algorithm that builds on run-length encoding [RC67] of array elements.

3 Parallel Generation of Flow Geometry

In this chapter, we provide the technical foundation for flow geometry, that is, the geometric representation and generation of vector field–derived structures such as streamlines and their extruded surfaces. First, in Section 3.1, we present an elementary introduction to vector field visualization. Next, in Section 3.2, we offer a basic overview of numerical integration methods for visualizing vector fields, along with implementation details for generating flow structures on the GPU. Then, in Section 3.3, we describe the implementation of streamline rasterization, followed by an explanation of geometric extrusion techniques for flow structures in Section 3.4. These geometry generation techniques are later employed in Chapter 4 for efficient vector field visualization and in Chapter 5 for opacity optimization strategies.

3.1 Introduction

Flow visualization has been an active research area for many decades, supporting scientists across a wide range of disciplines, including medicine, biology, and astrophysics. By enabling the visual exploration of flow data, these techniques help researchers develop insight into complex phenomena and underlying scientific processes. In general, flow data originates either from numerical simulations or from physical measurements.

Over the past decades, a large body of work has been dedicated to flow visualization and its applications. With the advent of GPGPU, many flow visualization techniques have achieved substantial performance improvements, often providing significant speed-ups over their CPU counterparts [Wei06, Fan08]. Recent flow visualization systems [GTG17] demonstrate real-time and interactive performance that would be difficult to achieve without GPU acceleration. At the same time, GPU-based solutions introduce several research challenges on massively parallel architectures, including limited memory bandwidth, synchronization overhead, and data dependencies.

Flow visualization is the most frequent application of vector field visualization and has yielded several specialized techniques. Vector field visualization techniques can be broadly classified into several categories. Direct visualization methods [PGL*12, dLvW93, HLNW11] represent vector field properties using graphical primitives such as points, colors, arrows, and other symbols. These techniques are relatively simple and often serve as building blocks for more advanced approaches. Geometric flow visualization techniques [MLP*10] rely on particle tracing to construct characteristic lines [GRT13] and surfaces [ELC*12] via numerical integration in the vector field. Texture-based techniques [FW08, LEG*08, WSE07, SH95, CL93] depict flow data densely using textures. A taxonomy of texture-based visualization methods is presented in the survey by Laramee et al. [LHD*04]. Feature-based, or topology-based, methods extract, track, and visualize key structures within the flow [PPF*11, LHZP07, PVH*03, HH91].

Topology-based flow visualization aims to extract structures in a vector field that govern or constrain particle motion, thereby providing compact representations that support analysis in both steady and unsteady settings [GBR21]. More generally, flow visualization often relies on identifying the most informative subsets of large datasets to obtain manageable representations that can be explored interactively. In this context, flow topology extraction summarizes a vector (or tensor) field while preserving its most salient structural features, enabling scientists and engineers to analyze topological elements such as critical points (e.g., sinks, sources, and saddles) and separatrices that emanate from and connect these points. These separatrices partition the domain into regions with coherent asymptotic behavior.

In steady fields, topology is commonly formulated in terms of asymptotic particle behavior: for any seed location, one asks where trajectories originate and where they converge. This behavior is captured by a topological skeleton composed of distinguished (including boundary-related) points linked by separatrices. For unsteady flows, topology is more nuanced; the literature distinguishes streamline-oriented and pathline-oriented perspectives and studies how topological elements evolve through events such as fold and Hopf bifurcations, as well as saddle connections, which can be analyzed in a space–time view. We refer interested readers to surveys on topology-based flow visualization for a broader overview [HLH*16, WWL16, PL09, LHZP07]. In this thesis, we present experiments on vector field visualization and direct our analysis toward steady vector fields.

As the following chapters focus on geometric flow visualization on massively parallel architectures, the upcoming sections provide a detailed discussion of parallel flow geometry generation on the GPU, serving as a foundation for the visualization techniques introduced

later. For a deeper exploration of vector field visualization, we recommend the book by Weiskopf [Wei06], the book by Johnson and Hansen [JH04], and the PhD thesis of Günther [Gü16].

3.2 Streamline Generation

3.2.1 Particle Tracing

A vector field is defined as an assignment of a tangent vector to each point in an n-dimensional manifold M, possibly with boundary N, where $N \subseteq M$. In most practical or experimental scenarios, M represents a 2D or 3D domain. A vector field may be time-independent (i.e., steady flow) or time-dependent (i.e., unsteady flow).

In this thesis, we present experiments on vector field visualization, with a particular focus on steady vector fields. Accordingly, the subsequent analysis—especially in the context of particle tracing using Euler and Runge–Kutta methods—is conducted under the assumption of time-invariant flow, ensuring that all derived equations and interpretations apply specifically to steady fields.

Integral curves play a crucial role in vector field visualization and in understanding the underlying flow phenomena. Particle tracing is a standard approach for computing integral curves by solving the ordinary differential equations (ODEs) that govern particle motion within the flow.

The tracing process begins by injecting particles into the domain and then advancing each one along a trajectory tangential to the vector field. Initial particle positions can be randomly selected or defined based on application-specific criteria. A particle proceeds in the direction of the field vector at its current position, with a speed proportional to the field magnitude. Tracing continues until one of the following termination conditions is met: (1) the particle exits the domain boundaries, (2) the particle enters an invalid or null region of the field, or (3) a stopping criterion is triggered, such as exceeding a maximum number of integration steps.

Numerical Integration for Particle Tracing The motion of a particle in a vector field is governed by the following first-order ODE:

$$\frac{dx(t)}{dt} = \vec{v}(x(t)) \tag{3.1}$$

where $x(t)$ denotes the particle position parameterized by t, and $\vec{v}(x(t))$ is the vector field evaluated at that position. Under the assumption of steady flow, the vector field $\vec{v}$ does not explicitly depend on time; thus, t serves only as a trajectory parameter describing the evolution of particle positions along integral curves.

The Euler method offers a straightforward and easily implemented approach to integration. Given an initial position $x(t_0)$, the next position is estimated as:

$$x(t + \Delta t) = x(t) + \vec{v}(x(t))\Delta t \tag{3.2}$$

Here, Δt denotes the integration step, which must be small to limit the local truncation error, typically of order $O(\Delta t^2)$.

Practically, Euler integration can accumulate significant numerical error over multiple steps. To improve accuracy, a smaller step size is often used; however, this also increases the computational cost.

The classical fourth-order Runge–Kutta (RK4) method offers improved accuracy and stability, with a global truncation error of $O(\Delta t^4)$. It computes the next position as

$$x(t + \Delta t) = x(t) + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \tag{3.3}$$

where the intermediate stage values are

$$k_1 = \vec{v}(x(t)), \tag{3.4}$$

$$k_2 = \vec{v}\left(x(t) + \frac{\Delta t}{2}k_1\right), \tag{3.5}$$

$$k_3 = \vec{v}\left(x(t) + \frac{\Delta t}{2}k_2\right), \tag{3.6}$$

$$k_4 = \vec{v}\left(x(t) + \Delta t\, k_3\right). \tag{3.7}$$

Further information on numerical integration for particle tracing is provided in [RS15].

Seed Locations We initialize particles at uniformly distributed random positions, which is one possible seeding strategy and is used here for simplicity and reproducibility. A CUDA kernel for seed initialization is presented in Listing 3.1. The vector field is represented as a 3D CUDA texture of type `float4` (i.e., `texture<float4, 3, cudaReadModeElementType> vector_field`), where the additional component is unused and serves solely to ensure proper memory alignment and efficient texture access on the GPU. We enable trilinear interpolation (`cudaFilterModeLinear`) and use clamping at volume boundaries (`cudaAddressModeClamp`) for improved precision.

To simplify texture fetches, particle positions are normalized to the unit cube $[0, 1]^3$ using `vector_field.normalized = true`. Integration is performed in this normalized coordinate space, while final particle positions are mapped back to the original grid bounds. This formulation assumes isotropic grid spacing; for anisotropic grids, the integration step size or velocity field would need to be adjusted accordingly to avoid non-uniform scaling effects.

All code examples in this and subsequent chapters use vector and matrix types (e.g., `vecn`, `matn`, `matnxm`) following the OpenGL Mathematics (GLM) library [GLM], with standard arithmetic operations.

```
1 __global__ void fill_seeds(curandState* rand_states, uint32_t num_rand_states, vec3* seed_positions, uint32_t
      num_seeds) {
2     auto thread_id = blockDim.x * blockIdx.x + threadIdx.x;
3     if (thread_id >= num_seeds)
4         return;
5
6     auto state_id = thread_id % num_rand_states;
7     auto local_state = rand_states[state_id];
8
9     float rand_x = curand_uniform(&local_state);
10    float rand_y = curand_uniform(&local_state);
11    float rand_z = curand_uniform(&local_state);
12
13    seed_positions[thread_id] = normalize(vec3(rand_x, rand_y, rand_z));
14
15    // Write back updated state (if needed in your use case)
16    rand_states[state_id] = local_state;
17 }
```

Listing 3.1: Initialization of particle tracer seeds.

Particle Tracing Each particle is traced by a separate GPU thread, as shown in Listing 3.2. The integration function advances the particle in steps of size Δt, storing each resulting position in a preallocated memory buffer. After all particles are traced, we perform stream

compaction [SHZO07] to gather only valid vertices into a contiguous array. We also maintain metadata that records the vertex count and starting index for each streamline.

```
1 __global__ void integrate(vec3 box_min, vec3 box_max, const vec3* seeds, vec3* streamlines, uint32_t*
      streamlines_length, uint32_t max_n_samples, float dt, uint32_t num_streamlines) {
2
3     auto thread_idx = blockDim.x * blockIdx.x + threadIdx.x;
4     if (thread_idx >= num_streamlines)
5         return;
6
7     // scale for [0,1]^3 to grid boundary
8     auto map_to_grid_bounds = [&](auto pos) {
9         auto box_extent = box_max - box_min;
10        auto out_pos = pos * box_extent + box_min; // element-wise operation
11        return out_pos;
12    };
13
14    // particle exits the unit cube
15    auto outside_volume = [&](auto pos) {
16        if (pos.x < 0 || pos.x > 1 || pos.y < 0 || pos.y > 1 || pos.z < 0 || pos.z > 1)
17            return true;
18
19        return false;
20    };
21
22    auto pos = seeds[thread_idx];
23    auto line_offset = thread_idx * max_n_samples;
24    auto num_samples = 0;
25    streamlines[line_offset + num_samples++] = map_to_grid_bounds(pos);
26
27    for (auto i = 0u; i < max_n_samples - 1; i++) {
28        pos = integrate_step(pos, dt);
29        if (outside_volume(pos))
30            break;
31        streamlines[line_offset + num_samples++] = map_to_grid_bounds(pos);
32    }
33
34    if (num_samples > 1) {
35        streamlines_length[thread_idx] = num_samples;
36    } else {
37        streamlines_length[thread_idx] = 0;
38    }
39 }
```

Listing 3.2: GPU kernel for particle tracing using numerical integration.

Depending on accuracy requirements, we apply either Euler integration (see Listing 3.3) or RK4 (see Listing 3.4) for each particle.

```
1 texture<float4, 3, cudaReadModeElementType> vector_field;
2
3 __device__ vec3 euler_integrate_step(vec3 pos, float dt)
4 {
5     const float4 velocity4 = tex3D(vector_field, pos.x, pos.y, pos.z);
6     const vec3 velocity(velocity4.x, velocity4.y, velocity4.z);
7     return pos + dt * velocity;
8 }
```

Listing 3.3: Euler integration for particle tracing.

```
1 texture<float4, 3, cudaReadModeElementType> vector_field;
2
3 __device__ vec3 sample_velocity(vec3 p)
4 {
```

```
5     float4 v = tex3D(vector_field, p.x, p.y, p.z);
6     return vec3(v.x, v.y, v.z);
7 }
8
9 __device__ vec3 runge_kutta_integrate_step(vec3 pos, float dt)
10 {
11     const vec3 lo(0.0f), hi(1.0f);
12
13     vec3 k1 = dt * sample_velocity(clamp(pos,               lo, hi));
14     vec3 k2 = dt * sample_velocity(clamp(pos + 0.5f * k1,   lo, hi));
15     vec3 k3 = dt * sample_velocity(clamp(pos + 0.5f * k2,   lo, hi));
16     vec3 k4 = dt * sample_velocity(clamp(pos + k3,          lo, hi));
17
18     return pos + (k1 + 2.0f*k2 + 2.0f*k3 + k4) * (1.0f / 6.0f);
19 }
```

Listing 3.4: Runge-Kutta integration for particle tracing.

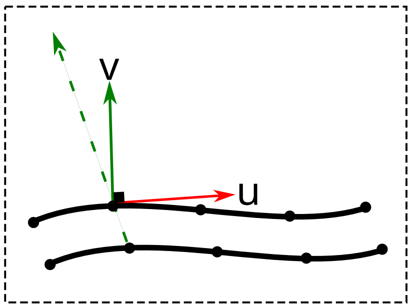


Figure 3.1: Illustration of vector field orientation using two parallel particles. The unit tangent vector u follows the flow direction. The vector v is orthogonal to u, computed by projecting the separation vector between particles onto the direction perpendicular to u.

Vector Field Orientation To estimate vector field orientation along a streamline, we trace a pair of closely spaced particles along the tangent direction. Around one particle, we define a local orthonormal frame (u, v) via Gram–Schmidt orthogonalization. Here, u is the normalized flow direction, and v is orthogonal to u, obtained by projecting the vector between the particle pair. This process is illustrated in Figure 3.1 and a code snippet is presented in Listing 3.5.

To avoid unstable frame orientation caused by small positional differences at initialization, we skip the first point when computing local frames. The resulting orthonormal frames (u, v) are used in subsequent sections for streamline extrusion.

```
1 __device__ void integrate_parallel_particles( vec3 pos_1, vec3 pos_2, float dt, float dist, vec3& pos_n_1, vec3&
      pos_n_2, vec3& u, vec3& v) {
```

```
2     // advance first particle
3     pos_n_1 = integrate_step(pos_1, dt);
4
5     // tangent direction
6     u = normalize(pos_n_1 - pos_1);
7
8     // previous separation
9     vec3 sep = pos_2 - pos_1;
10
11    // keep separation perpendicular to tangent
12    v = normalize(sep - u * dot(sep, u));
13
14    // rebuild second particle at fixed distance
15    pos_n_2 = pos_n_1 + dist * v;
16 }
```

Listing 3.5: Streamline orientation calculation using two parallel particle tracers.

3.3 Streamline Rasterization

Once the geometry of the streamlines has been generated, it is rendered using standard OpenGL line primitives (i.e., GL_LINES or GL_LINE_STRIP) [KSS16]. However, the OpenGL specification provides limited flexibility for line rendering. While support for a line width of one is guaranteed, support for thicker lines is optional and not uniformly implemented across all platforms. Additionally, standard line rendering may produce visual artifacts, such as gaps between segments, especially at sharp angles.

To address these limitations, we tessellate each streamline segment into a screen-aligned quadrilateral (quad) with a controllable width. This strategy provides greater visual control and enables more accurate shading computations.

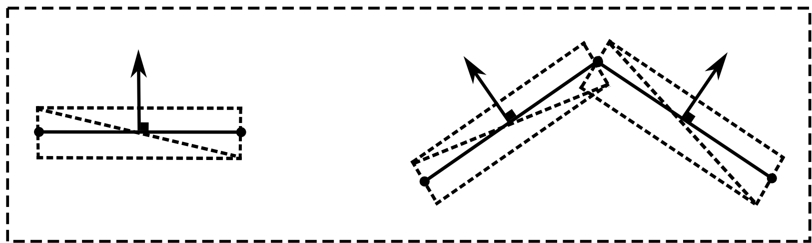


Figure 3.2: Turning a line segment into a quad (left) can result in gaps and overlaps (right) when applied to each segment independently.

Tessellating Lines into Quad Geometry A straightforward method for generating quads is to compute the normal to each segment and offset the vertices slightly along this normal. While simple, this approach fails at sharp angles between successive segments, as shown in Figure 3.2.

To mitigate such issues, the miter joint technique is adopted [Hal19a, Hal19b, Rud16], a method borrowed from woodworking. It connects adjacent segments using a shared corner, better preserving angular relationships and minimizing gaps and overlaps (see Figure 3.3). A simplified implementation of this method is provided below for completeness.

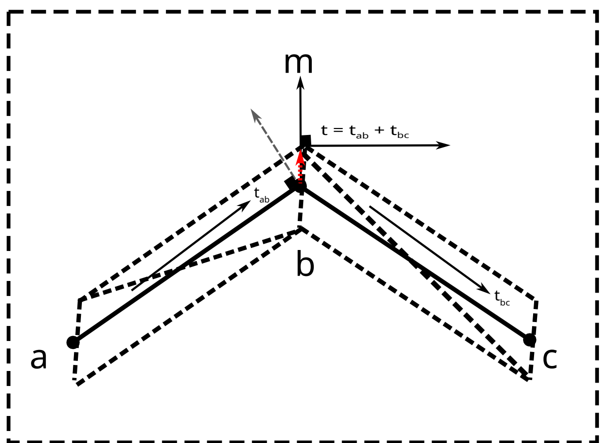


Figure 3.3: Miter joint between successive line segments using a shared edge.

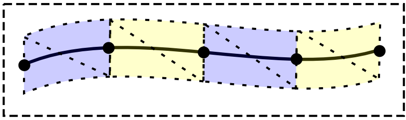


Figure 3.4: Streamline triangulation. Each pair of consecutive vertices along a streamline is expanded into a screen-aligned quad, which is then triangulated into two triangles for rendering.

As illustrated in Figure 3.4, each pair of consecutive streamline vertices generates a triangulated quad. A dedicated CUDA kernel processes the vertices and emits quad geometry as detailed in Listing 3.6. Each thread computes the local vertex offset within the streamline and emits two duplicated vertices—each tagged with a signed indicator (-1 or $+1$)—to differentiate the top and bottom corners of the quad.

In addition, each thread writes six indices to the index buffer, forming two triangles. The final thread in a streamline remains inactive since no further segments follow.

To support independent processing per vertex, the streamline index is stored for each vertex during particle tracing. This enables recovery of streamline-specific data (i.e., the start index and total length) and calculation of the vertex's local offset (Lines 7–13).

Because quads are generated between vertex pairs, the output offset in the geometry buffer depends on the vertex's absolute and local position. The formulas used are:

$$v_{oi} = v_{in} \times 2 \tag{3.8}$$

$$i_{oi} = (v_{in} - l_i) \times 6 \tag{3.9}$$

Where:
- v_{oi}, i_{oi} denote vertex and index offsets in the output buffer;
- v_{in} is the global index of the vertex in the output buffer;
- l_i is the line index of the corresponding streamline.

This deterministic indexing scheme allows predictable buffer allocation and efficient GPU parallelization.

```
1  __global__ void tessellate_lines(const vec3* lines_vertices, const uint32_t* line_indices, const uint32_t*
       lines_offsets, const uint32_t* lines_lengths, uint32_t num_lines_vertices, vec3* previous_vertices, vec4*
       quad_vertices, vec3* next_vertices, uint32_t* quad_indices) {
2
3      auto idx = blockDim.x * blockIdx.x + threadIdx.x;
4      if (idx >= num_lines_vertices)
5          return;
6
7      auto vertex_idx = idx;
8      auto line_idx = line_indices[vertex_idx];
9      auto line_offset = lines_offsets[line_idx];
10     auto line_length = lines_lengths[line_idx];
11     auto vertex_line_offset = vertex_idx - line_offset;
12     auto output_idx_offset = vertex_idx - line_idx;
13     auto out_vertex_offset = vertex_idx * 2;
14
15     vec3 current_vertex = lines_vertices[vertex_idx];
16     // Fetch neighbors while handling boundary conditions
17     vec3 previous_vertex = (vertex_line_offset > 0) ? lines_vertices[idx - 1] : current_vertex;
18     vec3 next_vertex = (vertex_line_offset < line_length - 1) ? lines_vertices[idx + 1] : current_vertex;
19
20     previous_vertices[out_vertex_offset + 0] = previous_vertex;
21     quad_vertices[out_vertex_offset + 0] = vec4(current_vertex, -1.0f);
22     next_vertices[out_vertex_offset + 0] = next_vertex;
23
24     previous_vertices[out_vertex_offset + 1] = previous_vertex;
25     quad_vertices[out_vertex_offset + 1] = vec4(current_vertex, +1.0f);
```

```
26      next_vertices[out_vertex_offset + 1] = next_vertex;
27
28
29      if (vertex_line_offset < line_length - 1) {
30          auto out_idx = output_idx_offset * 6;
31          quad_indices[out_idx + 0] = out_vertex_offset + 0;
32          quad_indices[out_idx + 1] = out_vertex_offset + 2;
33          quad_indices[out_idx + 2] = out_vertex_offset + 1;
34          quad_indices[out_idx + 3] = out_vertex_offset + 1;
35          quad_indices[out_idx + 4] = out_vertex_offset + 2;
36          quad_indices[out_idx + 5] = out_vertex_offset + 3;
37      }
38  }
```

Listing 3.6: Streamline triangulation; each consecutive pair of a streamline vertices are mapped to triangulated quad geometry.

Rendering Screen-Oriented Quad Geometry Listing 3.7 presents a shader that transforms a line vertex into a quad for rasterization. Below, we summarize its functionality and relate it to Figure 3.3 for geometric clarity.

Each vertex requires adjacency data (previous and next vertices in the streamline). Rather than relying on OpenGL's built-in adjacency features, we store this data explicitly in separate buffers (Listing 3.6, Lines 17–18).

At the start of the shader, a check determines if the vertex is degenerate (i.e., overlaps with its neighbors). In such cases, expansion is done along a perpendicular normal vector using a fixed thickness (Lines 27–36).

If the vertex connects distinct segments, we compute direction vectors to both neighbors (Lines 39–40), and derive the miter direction from the average of the two corresponding normals (Lines 48–56). Finally, the expansion magnitude along this direction is scaled based on the projection onto the normal of the incoming segment.

```
1  vec4 vertex_to_ndc(vec3 vertex) {
2      return MVP * vec4(vertex, 1.0f);
3  }
4
5  vec2 vertex_to_ndc_2d(vec3 vertex) {
6      vec4 vertex_4d = vertex_to_ndc(vertex);
7      vec2 vertex_2d = vertex_4d.xy / vertex_4d.w;  // Perspective divide
8      return vertex_2d;
9  }
10
11
12 vec2 get_vertex_offset(vec3 previous_vertex, vec3 current_vertex, vec3 next_vertex, int corner) {
13     vec2 screen_aspect_2d = vec2(screen_aspect, 1.0);
14
15     vec2 previous_vertex_2d = vertex_to_ndc_2d(previous_vertex) * screen_aspect_2d;
16     vec2 current_vertex_2d  = vertex_to_ndc_2d(current_vertex) * screen_aspect_2d;
17     vec2 next_vertex_2d     = vertex_to_ndc_2d(next_vertex) * screen_aspect_2d;
18
19     vec2 tangent = vec2(0.0f);
```

```
20     float extension_length = line_thickness;
21 
22     vec2 miter_vector = vec2(0.0f);
23 
24     const float eps = 1e-6f;
25 
26     // Robust degenerate checks to avoid float equality comparisons
27     if (length(current_vertex_2d - previous_vertex_2d) < eps) {
28         // Starting point: use (next - current)
29         tangent = normalize(next_vertex_2d - current_vertex_2d);
30         miter_vector = vec2(-tangent.y, tangent.x);
31     }
32     else if (length(current_vertex_2d - next_vertex_2d) < eps) {
33         // Ending point: use (current - previous)
34         tangent = normalize(current_vertex_2d - previous_vertex_2d);
35         miter_vector = vec2(-tangent.y, tangent.x);
36     }
37     else {
38         // Middle point with join
39         vec2 dir_previous_current = normalize(current_vertex_2d - previous_vertex_2d);
40         vec2 dir_current_next     = normalize(next_vertex_2d - current_vertex_2d);
41 
42         // Near-180-degree turn -> fall back to simple normal to avoid a large miter
43         if (dot(dir_previous_current, dir_current_next) < -0.99f) {
44             tangent = dir_previous_current;
45             miter_vector = vec2(-tangent.y, tangent.x);
46         }
47         else {
48             vec2 average_tangent = normalize(dir_previous_current + dir_current_next);
49             miter_vector = vec2(-average_tangent.y, average_tangent.x);
50 
51             vec2 normal_to_previous_current = vec2(-dir_previous_current.y, dir_previous_current.x);
52 
53             float d = dot(miter_vector, normal_to_previous_current);
54             if (abs(d) < eps) d = (d < 0.0f ? -eps : eps);
55 
56             extension_length = line_thickness / d;
57         }
58     }
59 
60     return miter_vector * extension_length * float(corner) / screen_aspect_2d;
61 }
62 
63 
64 vec4 get_line_vertex_projection(vec3 current_vertex, vec3 next_vertex, vec3 previous_vertex, int corner) {
65     vec4 vertex_4d = vertex_to_ndc(current_vertex);
66     vec2 vertex_offset = get_vertex_offset(previous_vertex, current_vertex, next_vertex, corner);
67 
68     // vertex_offset is in NDC; convert it to a clip-space offset by multiplying by w
69     return vertex_4d + vec4(vertex_offset * vertex_4d.w, 0.0, 0.0);
70 }
```

Listing 3.7: Streamline rendering.

3.4 Streamline Extrusion

Given a cross-sectional shape with s_n vertices and a streamline with l_n vertices, we extrude the shape along the streamline by generating triangulated quads between corresponding cross-section vertices at each consecutive pair of streamline points, as illustrated in (see Figure 3.5). The number of tessellated quads between two consecutive cross-sections is equal to s_n, result-

ing in a total of $(l_n - 1) \times s_n$ quads per streamline. To ensure completeness and watertightness, we also generate caps at both ends of the streamline by triangulating the cross-sectional shape. This design makes the generated triangle set deterministic, with each streamline vertex contributing to the relevant tessellated quads using a shared normal vector at each emitted quad vertex, as shown in Listing 3.8.

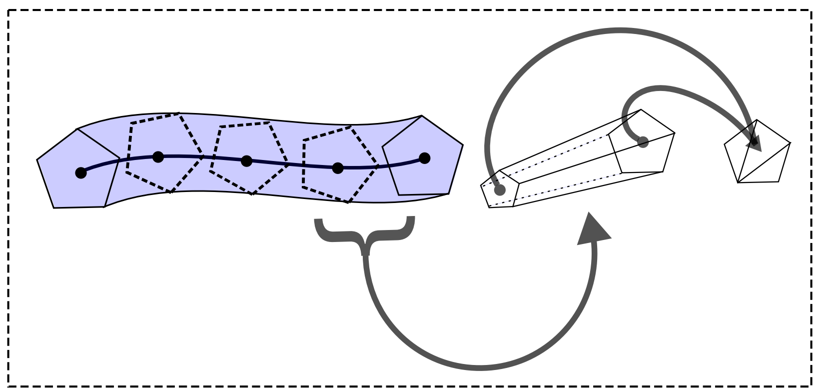


Figure 3.5: Streamline extrusion. Left: a cross-sectional glyph is extruded along the streamline vertices. Middle: each consecutive pair of vertices is expanded into a closed surface by extending corresponding parallel edges into triangulated quads. Right: the start and end of the streamline are closed with triangulated caps derived from the cross-sectional shape.

Extrusion Modes To allow more control over the visual appearance of the extruded geometry, we support two commonly used extrusion modes:

1. **Shared-vertex mode:** Neighboring quads share common vertices along the cross-section edges, and the normal at shared vertices is computed as the average of the adjacent face normals.

2. **Duplicated-vertex mode:** Vertices are duplicated along each edge, and individual normals are assigned based on the corresponding cross-section edge direction.

These two modes are depicted in Figure 3.6 for a single section of the extrusion along an arbitrary streamline.

Normals Generation We use the local coordinate frame $(\vec{v}, \vec{w})$ constructed during particle tracing (see Section 3.2) to compute surface normals for the extruded geometry. A 2D cross-

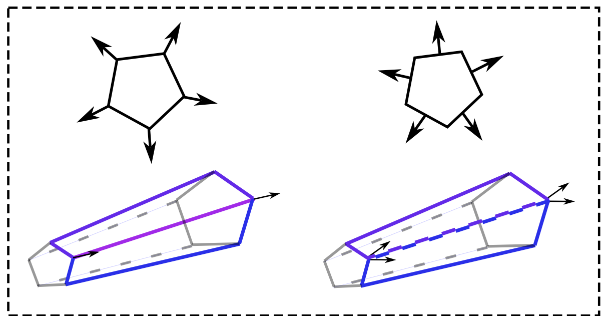


Figure 3.6: Extrusion modes; left, a shared cross-sectional shape vertex results in a shared vertex between neighbor quads, right, two vertices are emitted for each quad.

section normal $\vec{n} = (n_x, n_y) \in \mathbb{R}^2$ is defined as the vector from the shape center to a boundary vertex. The corresponding 3D normal at a streamline position p is given by:

$$\vec{N} = \vec{v} \cdot n_x + \vec{w} \cdot n_y \in \mathbb{R}^3.$$

This vector is added to p to obtain the final vertex position on the extruded shape. In duplicated-vertex mode, we compute $\vec{n}$ separately for each cross-section edge and apply the appropriate normal for each duplicated vertex, as shown in Figure 3.6.

```
1 __global__ void extrude_lines(const vec3* line_vertices, const vec3* line_tangents, const vec3* line_normals, const
      uint32_t* vertex_line_idx, const uint32_t* line_offsets, const uint32_t* line_lengths, uint32_t
      num_line_vertices, vec3* vertex_buffer, vec3* normal_buffer, uint32_t* index_buffer
2 ) {
3     auto idx = blockDim.x * blockIdx.x + threadIdx.x;
4     if (idx >= num_line_vertices)
5         return;
6
7     // Helper functions
8     auto get_vertex = [&](vec3 vertex, vec3 normal, float radius) {
9         return vertex + radius * normal;
10    };
11
12    auto get_normal = [&](vec3 vertex, vec2 shape_vertex, vec3 v, vec3 w) {
13        return normalize(vertex + v * shape_vertex.x + w * shape_vertex.y);
14    };
15
16    // Line metadata
17    auto vertex_idx = idx;
18    auto line_idx = vertex_line_idx[vertex_idx];
19    auto line_offset = line_offsets[line_idx];
20    auto line_length = line_lengths[line_idx];
21    auto vertex_offset = vertex_idx - line_offset;
22
23    // Cap geometry
24    if (vertex_offset == 0) {
25        fill_start_cap_geometry(line_idx, vertex_buffer, index_buffer);
26    }
27
28    auto vertex_buf_offset = (line_idx * 2 + 1) * cap_num_vertices;
29    auto index_buf_offset  = (line_idx * 2 + 1) * (cap_num_vertices - 2) * 3;
30    auto out_vertex_offset = vertex_buf_offset + vertex_offset * cross_shape_num_vertices;
31    auto out_index_offset  = index_buf_offset  + vertex_offset * cross_shape_num_vertices * 6;
32
33    // Geometry extrusion
34    vec3 streamline_vertex = line_vertices[vertex_idx];
```

```
35
36     // Local coordinate vectors for each streamline vertex
37     vec3 u = line_tangents[vertex_idx];  // direction tangent to the vector field
38     vec3 v = line_normals[vertex_idx];   // direction normal to the vector field tangent
39     vec3 w = cross(u, v);                // direction normal to both u and v
40
41     float radius = ...; // application-specific
42     vec2 shape_vertex;
43
44     for (size_t s = 0; s < cross_shape_num_vertices; s++) {
45         // Assume shape_vertex is accessed or computed externally for each `s`
46         vec3 normal = get_normal(streamline_vertex, shape_vertex, v, w);
47         vec3 out_vertex = get_vertex(streamline_vertex, normal, radius);
48         vertex_buffer[out_vertex_offset + s] = out_vertex;
49         normal_buffer[out_vertex_offset + s] = normal;
50     }
51
52     // Emit quad indices
53     if (vertex_offset < line_length - 1) {
54         for (size_t s = 0; s < cross_shape_num_vertices; s++) {
55             size_t sN = (s + 1) % cross_shape_num_vertices;
56
57             uint32_t v0 = out_vertex_offset + s;
58             uint32_t v1 = out_vertex_offset + s + cross_shape_num_vertices;
59             uint32_t v2 = out_vertex_offset + sN + cross_shape_num_vertices;
60             uint32_t v3 = out_vertex_offset + sN;
61
62             index_buffer[out_index_offset + s * 6 + 0] = v2;
63             index_buffer[out_index_offset + s * 6 + 1] = v1;
64             index_buffer[out_index_offset + s * 6 + 2] = v0;
65             index_buffer[out_index_offset + s * 6 + 3] = v0;
66             index_buffer[out_index_offset + s * 6 + 4] = v3;
67             index_buffer[out_index_offset + s * 6 + 5] = v2;
68         }
69     }
70
71     // End cap
72     if (vertex_offset == line_length - 1) {
73         fill_end_cap_geometry(line_idx, vertex_buffer, index_buffer);
74     }
75 }
```

Listing 3.8: Streamline extrusion using a 2D cross-section shape.

3.5 Limitations and Perspectives

During the cross-sectional shape extrusion, we limit the end caps of the extruded geometry to flat, triangulated polygons that match the shape of the cross-section. While this approach is efficient and sufficient for many use cases, more realistic representations could be achieved by incorporating smoothly blended closures. For instance, computing the intersection between a hollow cylinder and a spherical cap [CSZ*25, Gha08, CGA] could enable the generation of more expressive tessellated end geometries [HWU*19].

Furthermore, applying physically based rendering techniques [PJH16, Jen01] to the extruded geometry could enhance realism and support the exploration of specific physical

phenomena. This approach may also benefit applications in simulation, design, and visualization [OPvdWC23, GG21, CFM*13].

4 Interactive Glyph-based Flow Visualization

The work presented in this chapter is partly based on

Zeidan M., Peters C., Rapp T., Dachsbacher C.: Versatile Geometric Flow Visualization by Controllable Shape and Volumetric Appearance. In Smart Tools and Applications in Graphics - Eurographics Italian Chapter Conference (2022), The Eurographics Association.

4.1 Introduction

A large variety of techniques for 3D flow visualization has been developed over the last decades, commonly categorized into direct, geometry-based (e.g., streamlines or streamsurfaces), texture-based (such as 3D-LIC), and feature-based approaches [LHD*04].

Geometry-based techniques generate collections of geometric glyphs that can convey more information than direction alone, such as rotation (streamribbons) or divergence (streamtubes). Different glyph types are suitable for visualizing different phenomena, which often occur together within a single vector field. Therefore, we propose combining multiple methods by providing the user with fine-grained control over cross-sectional shapes along characteristic lines and allowing interpolation between them as needed. Additional attributes, such as radii and colors, are also controllable.

Texture-based approaches, which use volume rendering to generate the final images, offer another intuitive approach to flow visualization. A prominent example is 3D line-integral

convolution (LIC) [FW08]. These methods convey directionality within the volume using transparency and directionally smoothed texture features. However, volume computation and rendering are costly, scale poorly to high screen resolutions, and impose non-negligible storage demands when the volume is not computed on the fly [FW08]. Although dense visualizations do not further deteriorate performance, sparse visualizations are preferred to avoid excessive visual clutter.

In this chapter, we introduce a novel geometry-based visualization technique for vector fields that generalizes and integrates several existing methods into a flexible, GPU-accelerated framework. Our approach maps characteristic lines to a wide range of glyphs in an interactive and adaptable manner. Users can specify multiple cross-sectional shapes for extrusion along characteristic lines. The system supports smooth interpolation between these shapes, either based on vector field attributes and line characteristics or controlled globally by the user. As a result, a single characteristic line can adopt varying cross-sections along its path, facilitating the visualization of diverse phenomena. In addition, our system tracks and renders rotational behavior in the vector field and provides full control over color mapping, opacity, and radius along the characteristic lines.

We use geometric glyphs to mimic the appearance of sparse texture-based approaches. Traditionally, handling transparency has been challenging for geometry-based methods. Our system employs moment-based order-independent transparency [MKKP18] to achieve high-quality results at high performance. In addition, we propose an approximation of transmittance through glyphs, enabling us to mimic the volumetric appearance of 3D LIC.

4.2 Flow Visualization

Geometry-based flow visualization techniques [MLP*10] convey the behavior of vector fields by embedding discrete geometric primitives directly into the velocity field. A foundational example is provided by Ueng et al. [USM96], who detail the efficient construction of streamlines, ribbons, and tubes. Streamsurfaces [ELC*12] are another widely used method for exploring flow characteristics in three-dimensional domains. As flow datasets grow in size and complexity, illustrative visualization techniques [BCP*12] aim to enhance understanding through abstract, stylized representations reminiscent of hand-drawn illustrations. Complementary to this, view-dependent streamline selection and placement [MCHM10, GRT13] have emerged as effective strategies for reducing visual clutter and highlighting salient fea-

tures. Günther et al. [GTG17] further propose a global optimization framework that balances occlusion minimization with the preservation of meaningful geometry. However, such methods typically limit visual expressiveness to variations in shading, color, and opacity.

Line integral convolution (LIC), first introduced by Cabral and Leedom [CL93], has inspired extensive research in texture-based flow visualization [LHD*04, LEG*08]. An important enhancement is oriented LIC (OLIC) by Wegenkittel et al. [WGP97], which visualizes flow direction in static images using sparse spot textures that are directionally smeared along the local vector field.

Although LIC can be extended to three dimensions, such extensions are computationally demanding and prone to perceptual and occlusion challenges when rendering dense volumes. Interrante and Grosch [IG98] were among the first to explore 3D LIC, emphasizing the necessity of sparse representations in three-dimensional settings. To address feature emphasis, Suzuki et al. [SFCN02] introduce a significance map that guides the selection of visually important flow regions. Falk and Weiskopf [FW08] further advance the field with an output-sensitive 3D LIC technique that tightly integrates LIC computation with volume rendering, thereby avoiding unnecessary evaluations of the LIC integral. Motivated by these developments, we propose a fast, geometry-based rasterization approach that captures the directional and volumetric properties of vector field streaks with high efficiency and visual clarity.

4.3 Geometry Generation

We present a novel method for visualizing characteristic lines that enables flexible and intuitive generation of diverse geometric glyphs along their paths. An overview of our vector field visualization pipeline is shown in Figure 4.1. To represent local attribute variations, our method supports the construction of arbitrary geometric structures along characteristic lines using 2D cross-sectional glyphs (Section 4.3.2). Users can then interpolate between neighboring glyphs (Section 4.3.3) and adjust the width of the extruded geometry to highlight directional features (Section 4.3.4). This versatile approach not only reproduces classical geometry-based flow visualizations (e.g., streamribbons and streamtubes), but also approximates the visual characteristics of texture-based and volumetric techniques such as 3D LIC [FW08]. The entire geometry generation pipeline is GPU-accelerated and implemented using NVIDIA CUDA [NVI20].

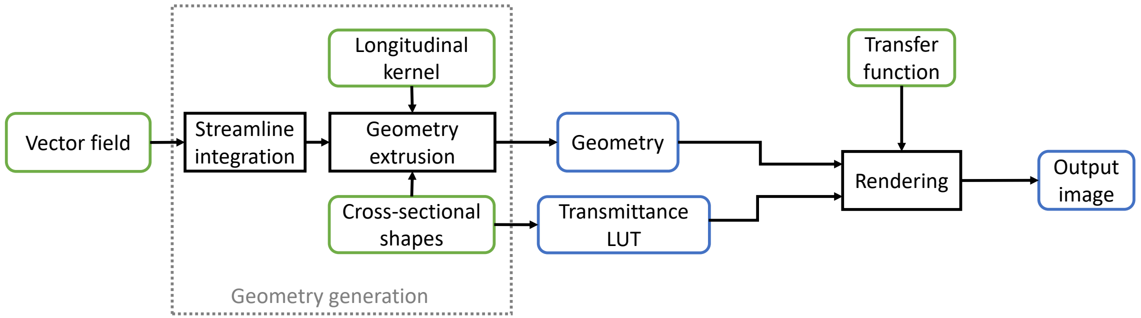


Figure 4.1: An overview of our pipeline for vector field visualization. Data in green boxes is controlled by the user interactively. Data in blue boxes is updated at run-time. All operations (black rectangles) are GPU-accelerated.

In Section 4.4, we describe our rasterization-based rendering pipeline for these extruded structures. Additionally, we approximate volumetric absorption and employ order-independent transparency [YHGT10, MKKP18] to blend semi-transparent geometry along view rays, thereby achieving a volumetric appearance.

4.3.1 Integration of Characteristic Lines

We compute characteristic lines by tracing independent particles through a vector field using a fourth-order Runge-Kutta method [BD09]. Initial positions are distributed via stratified random sampling within the domain. Tracing terminates when a particle reaches the domain boundary or exceeds a predefined number of steps. Each line is stored as a sequence of 3D points $p_0, p_1, \ldots, p_n \in \mathbb{R}^3$, each annotated with relevant attributes such as velocity magnitude for use during extrusion and rendering. To retain directional consistency, we trace pairs of particles located orthogonally to the tracing direction [Tel07].

For parallel execution, we bind the vector field as a 3D texture, benefiting from hardware-accelerated filtering. A GPU buffer is preallocated to accommodate the maximum number of steps. As particle traces vary in length, we apply parallel stream compaction [SHZO07], compute line-wise statistics, and rearrange vertices accordingly. We then prepare GPU buffers and perform glyph extrusion in parallel across line vertices.

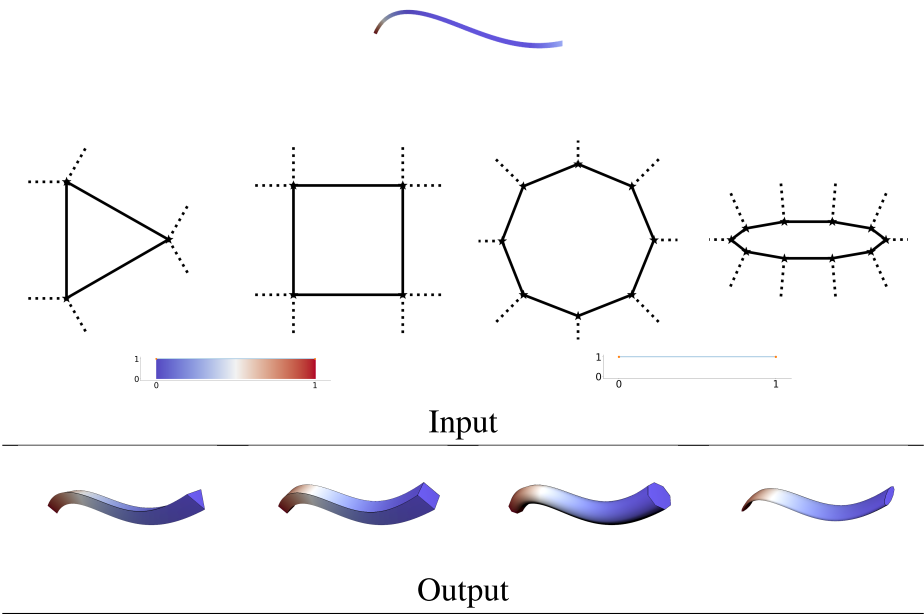


Figure 4.2: Extrusion of cross-sectional glyphs using different shapes. We use flat normals for triangle and quad glyphs, and smooth normals for circle and ellipse glyphs. We use a constant radius as shown in the kernel in the bottom right of top row.

4.3.2 Design and Placement of Cross-Sectional Glyphs

Traditional transfer functions map scalar attributes such as velocity or pressure to color and opacity. While useful, these mappings fall short in representing complex local features (e.g., rotation) or directional attributes. Our method overcomes this limitation by placing customizable 2D glyphs along characteristic lines. These glyphs encode localized flow behavior, while directional and longitudinal kernels express global structure.

All input attributes are assumed to be normalized to $[0, 1]$. Users can define a collection of closed polygonal glyphs, each associated with a range of attribute values. Glyphs are extruded along streamlets that intersect the corresponding attribute interval (Figures 4.2 and 4.3).

Our method provides an interactive panel for glyph creation, supporting standard shapes (e.g., circles, ribbons) and editable polygons with fixed vertex counts. Users may edit vertex positions via direct manipulation or input fields. Figure 4.2 shows an example in which a novice user created isotropic glyphs with minimal effort.

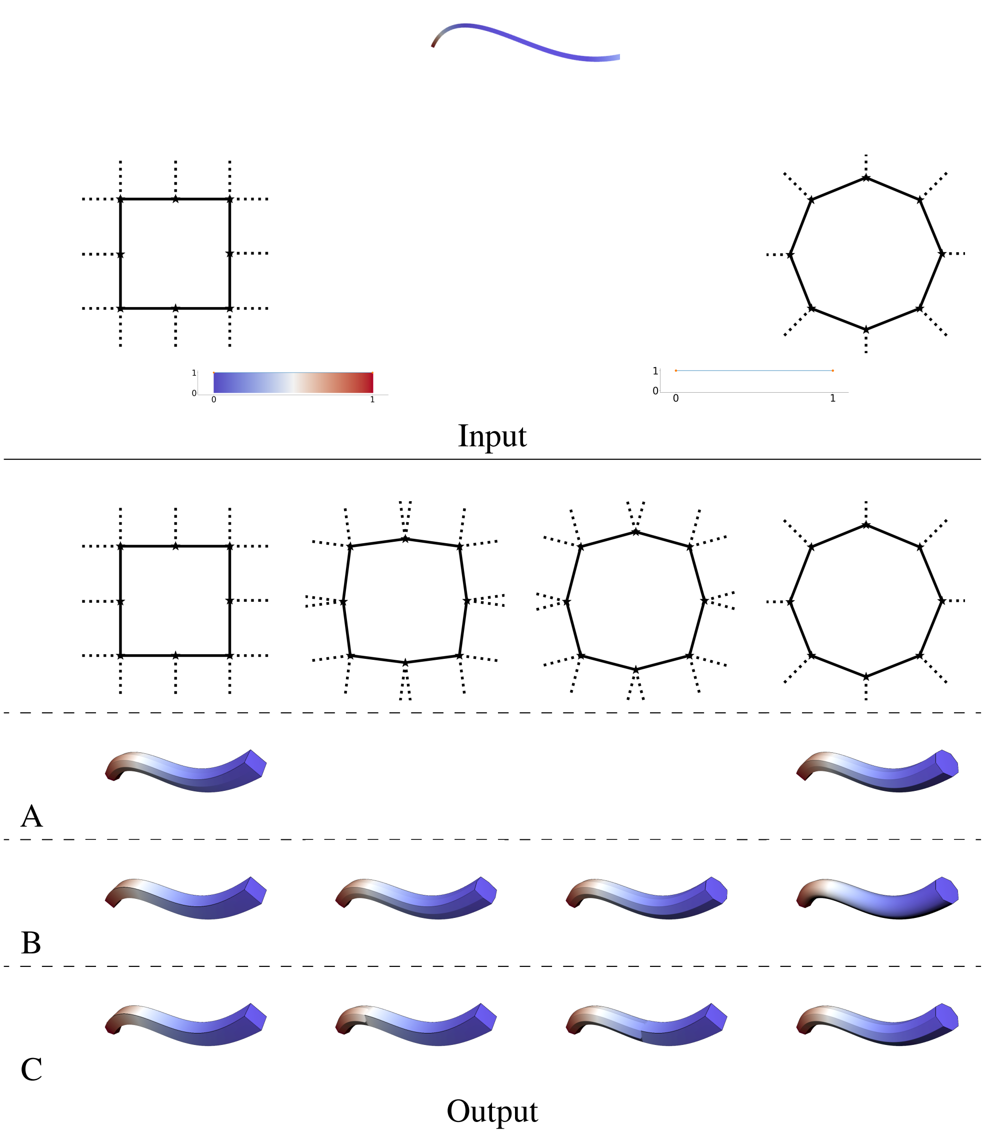


Figure 4.3: Using circle and quad glyphs, we allow several modes of interpolations along extruded geometry. In row A, we extrude all interpolation instants between quad and circle (left), and between circle and quad (right). In row B, we display four separate example instants of interpolation, where we selectively extrude an intermediate shape between quad and circle over the whole streamline. In row C, we use a user-specified threshold t along a streamline to selectively extrude the first glyph shape to all attribute values below t, and extrude the other glyph shape to the rest of the input range.

4.3.3 Morphing Cross-Sectional Glyphs

To better explore structural transitions in the data, our system supports glyph morphing via linear interpolation of vertices between glyphs with fixed correspondence. Suppose glyphs G_a and G_b are defined at scalar values a and b in the attribute domain $\Re \in [0, 1]$, and both glyphs share the same number of vertices. This setup resembles scalar transfer functions but uses shape instead of color.

Although automatic vertex correspondence methods exist [YF09], our UI allows users to manually define mappings for clarity and control. Once correspondence is established, glyph interpolation is performed vertex-wise.

The user can select from three interpolation modes, illustrated in Figure 4.3. First, attribute values along the streamline can drive continuous interpolation between neighboring shapes, as demonstrated in row A. Second, a fixed attribute value can be used to apply a uniform interpolation state across the entire streamline, allowing convenient exploration of glyph variations, as shown in row B. Finally, a threshold value can be specified to enforce a sharp transition between different glyphs, as illustrated in row C.

During morphing, we handle shading normals based on user-defined flags. Vertices marked as *flat* generate duplicated normals at side faces. Vertices marked as *smooth* are averaged with adjacent faces. When interpolating between vertices with different shading flags, the result defaults to flat shading.

4.3.4 Controlling the Longitudinal Kernel

Our system allows users to modulate the width of extruded glyphs along characteristic lines using a longitudinal kernel, implemented as a 1D texture sampled over the normalized attribute range $\Re \in [0, 1]$. Directional features can be encoded by sampling the kernel with the integration step t, while feature highlighting can be achieved using scalar attributes (Figure 4.4).

Spatially Varying Opacity In oriented LIC, the kernel $k(s)$ is typically asymmetric to encode directionality through varying density. This principle is easily integrated into our

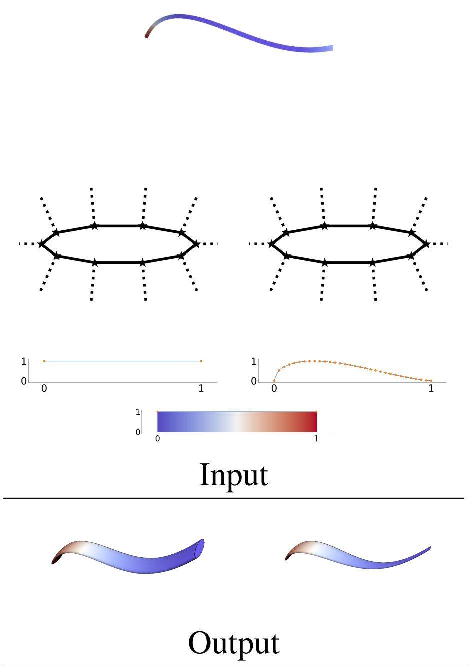


Figure 4.4: Extruding an ellipse cross-section shape using a constant kernel for extrusion (left), and ramp kernel (right).

pipeline. Opacity can be modulated using a user-defined function dependent on the characteristic line parameter $s \in \mathbb{R}$.

4.3.5 Generation of Triangle Meshes

We perform geometry generation entirely on the GPU via parallel kernel invocations across characteristic line vertices. The output is a triangle mesh consisting of vertex and index buffers, which are subsequently passed to an OpenGL rasterizer for rendering (Section 4.4).

Each extruded segment spans an attribute range bounded by two glyphs with identical vertex counts. Initially, we triangulate the cross-sectional polygons at both ends of the segment. Then, for every pair of corresponding glyph vertices, we generate quad side faces.

This process is parallelized as follows: a kernel is launched to compute the number of triangles required per vertex. A prefix sum [SHZO07] determines the global triangle allocation. A second pass emits the triangle data accordingly. Streamribbon glyphs are treated as a special case and are rendered using quads between successive vertices along the line.

4.4 Rendering

In this section, we describe the main steps involved in rendering the geometric structures produced by our framework. We begin by outlining our shading strategy in Section 4.4.1. Section 4.4.2 details how we simulate a volumetric appearance despite using geometry-based representations. Finally, in Section 4.4.3, we discuss the integration of state-of-the-art order-independent transparency techniques to correctly composite overlapping characteristic lines. Our renderer is implemented using OpenGL 4.5 with programmable shaders and supports both opaque and semi-transparent rendering modes.

4.4.1 Shading

Shading plays a critical role in conveying the shape and depth of geometric structures. We adopt a physically based reflection model [Wol18] with point light sources to compute surface illumination. Material colors can be defined arbitrarily or derived from the transfer function. The final appearance of each fragment is determined by both shading and the integrated absorption coefficient, which is used to compute opacity (Section 4.4.2).

To enhance the perception of silhouettes and edges under flat shading, we apply black outlines in the style of non-photorealistic rendering. These silhouettes and edges are identified in screen space using normal and depth buffers [ND03].

4.4.2 Opacity Computation along Volumetric Cross Sections

From the transfer function, we obtain an absorption coefficient $\sigma \geq 0$. However, in the case of volumetric structures, each rendered fragment corresponds to an entire segment of the extruded geometry traversed by a view ray. We assume that the absorption coefficient remains constant along each such segment. Therefore, the only missing component required to compute the fragment's opacity $\alpha \in [0, 1]$ is the traveled distance $d > 0$ of the ray through the extruded shape.

According to Beer's law, the resulting opacity is given by:

$$\alpha = 1 - e^{-\sigma d}.$$

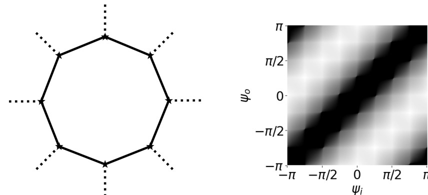


Figure 4.5: Precomputed traveled distance (right) over cross-sectional disk shape (left).

Accurate computation of d would require costly ray-triangle intersection tests. To avoid this, we introduce a simplifying assumption: the extruded geometry is sufficiently narrow to approximate the view ray's traversal as a straight segment. We project the ray direction onto the plane orthogonal to the streamline (tube) direction and compute the intersection length within this plane. This projected distance is then divided by the sine of the angle between the original ray and the tube axis to estimate the true traveled distance. To avoid instability at grazing angles, we clamp the reciprocal of the sine function to a maximum value of 5.

This formulation reduces the problem to a planar domain, enabling the use of a precomputed two-dimensional lookup table. Since the number of relevant rays is finite in the 2D plane, we construct this table only once per unique cross-sectional shape.

As established in Section 4.3.2, each glyph is defined in a coordinate frame where the yz-plane is orthogonal to the streamline direction. In this frame, we compute a bounding circle around the glyph and characterize ray orientations using their entry and exit angles $\psi_i, \psi_o \in [-\pi, \pi]$ on this circle (excluding rays that originate inside the shape). These angle pairs define the lookup coordinates (see Figure 4.5). To store the table as a texture, both angles are linearly mapped to the interval $[0, 1]$. At runtime, we compute ray-circle intersections to retrieve these angles and query the lookup table.

To construct the table, we iterate over all angle pairs at a resolution of, for example, 512^2. For each pair, we define a ray in Cartesian space and perform intersection tests with the segments that define the polygonal cross-section. The traveled distance is then derived from the sorted list of intersection points. This process is executed in parallel using a CUDA kernel, assigning one thread per lookup table entry.

To support opacity approximation for interpolated glyph shapes (as described in Section 4.3.2), we generate n intermediate cross-sectional shapes between each pair of neighboring glyphs. Corresponding lookup tables are computed and stored in a 3D texture.

During rendering, we use trilinear interpolation to query the appropriate transmittance value from this texture.

Controlling Opacity. While transmittance-based opacity is approximated automatically, our framework also provides interactive control through a transfer function widget. This allows users to directly manipulate opacity and color to emphasize specific regions of interest in the dataset. Optionally, the approximated transmittance can be modulated by the user-defined opacity values, providing a focus-plus-context rendering mechanism. This supports guided exploration by making important structures visually prominent while preserving the surrounding context.

4.4.3 Order-Independent Transparency

Since our approach is based on rasterization rather than ray marching, geometric primitives are not processed in strict front-to-back order. Therefore, rendering semi-transparent structures requires a technique for order-independent composition. We employ moment-based order-independent transparency (MBOIT) [MKKP18], which provides high-quality, scalable results under heavy overdraw and outperforms many competing techniques in terms of visual accuracy and efficiency.

MBOIT operates in two rendering passes using additive blending. In the first pass, the transparent geometry is rendered to moment buffers, which encode statistical information about fragment depths. In the second pass, these buffers are used to estimate how much light is transmitted through the preceding fragments. Each fragment's color is modulated by a computed transmittance factor and blended with other contributions. A final full-screen pass normalizes the accumulated values and composites them over a background color (constant in our case).

Rendering Opaque Geometry Since MBOIT is specifically designed for semi-transparent rendering, opaque geometry is rendered separately using traditional OpenGL blending with depth testing enabled.

MBOIT includes several quality levels depending on the number of moments stored. Given our high overdraw scenario, we adopt a high-precision variant using three trigonometric mo-

ments stored as single-precision floats. This results in a memory cost of 224 bits per pixel for the moment buffers. Intermediate color buffers also use single-precision RGBA textures to preserve visual fidelity.

4.5 Results and Evaluation

In this section, we demonstrate several visualizations that reveal important features in both real and synthetic 3D vector field datasets. We also explore relevant visualization parameters and evaluate the performance of our approach. All experiments were conducted on a machine equipped with an Intel i7-6700 CPU (3.40 GHz), 32 GB of RAM, and an NVIDIA GTX 1080 Ti GPU with 11 GB of video random-access memory (VRAM). Unless otherwise specified, velocity magnitude is mapped to color using a transfer function. All example images are rendered at a resolution of 1920×1080. Table 4.1 summarizes the input/output geometry sizes and reports timings for geometry generation and rendering. Across all cases, we achieve real-time rendering and interactive geometry editing.

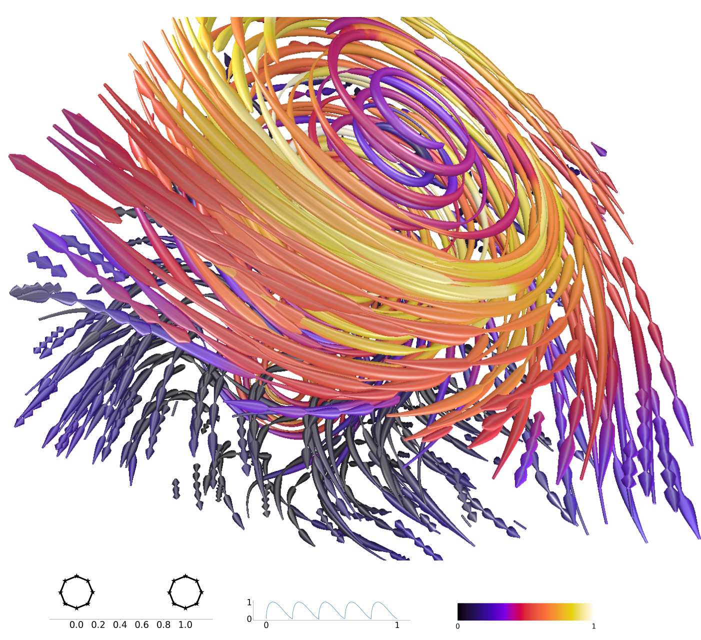


Figure 4.6: The tornado dataset with a circular cross-sectional glyph and several ramps as a longitudinal kernel.

4.5.1 Applications

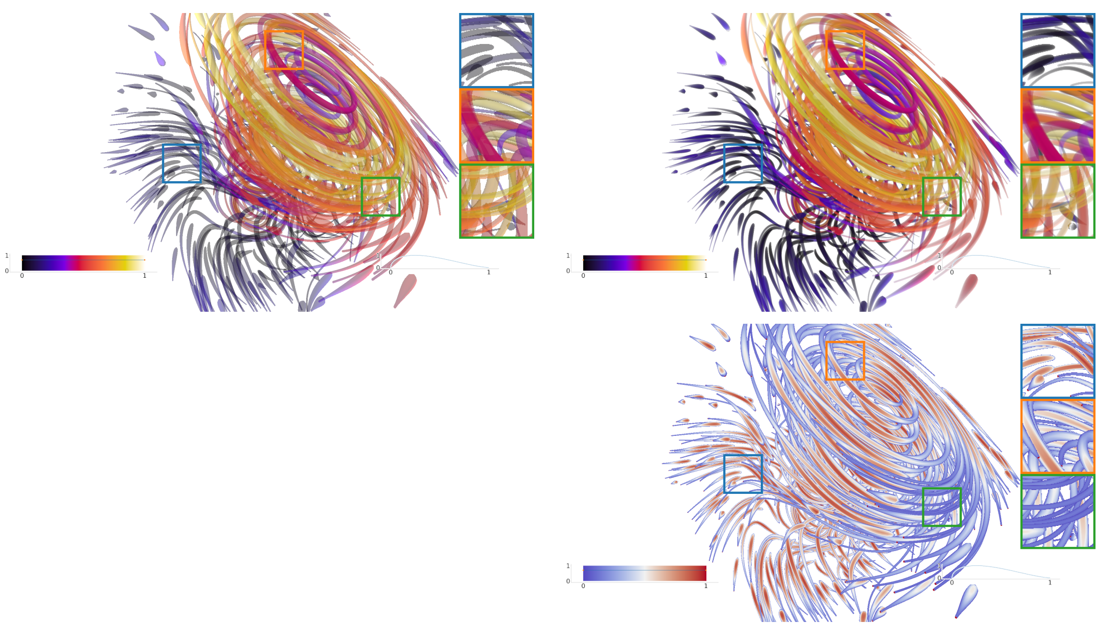


Figure 4.7: The tornado dataset with circular cross-section glyph and a ramp longitudinal kernel. Left: Geometry is rendered with a constant opacity value of 0.7 with all geometry. Right top: Approximate cross-sectional transmittance along geometry using our method. Right bottom: Approximate traveled distance using our cross-sectional transmittance approximation.

Flow Visualization Using a Directional Kernel. To visualize flow direction, we modulate the width of an isotropic cross-sectional glyph along each streamline using a directional kernel. In Figure 4.6, a multi-ramp kernel is applied with a circular cross-sectional glyph to encode flow direction in the the synthetic tornado vector field [MCHM10] dataset. To apply the longitudinal kernel during streamline tracing, we maintain a parametric distance t at each vertex, proportional to the integration step, and normalize it per streamline to the range $[0, 1]$.

Our method also enables the approximation of volumetric effects, such as those seen in 3D directional LIC. In Figure 4.7, we compare rendering using a constant opacity of 0.7 for all structures with our transmittance-based approach, which estimates opacity from traveled distances (see Section 4.4.2). As demonstrated in Table 4.1, our volumetric approximation incurs minimal performance overhead.

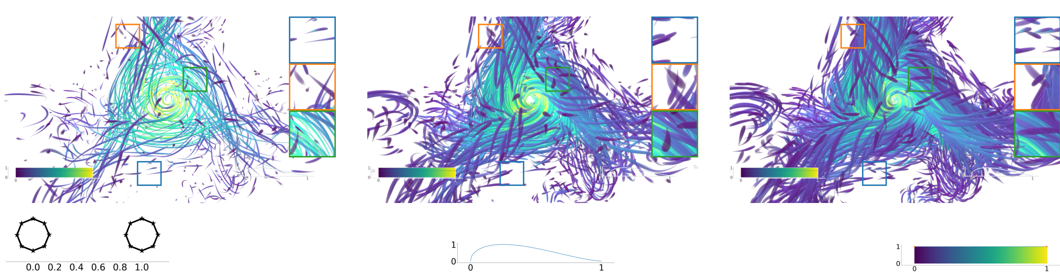


Figure 4.8: A ramp kernel and a circular cross-sectional shape are used to emulate texture sparsity with the trefoil knot dataset. From left to right we increase the number of streamlines.

An important aspect of our method is its scalability with respect to geometry generation and rendering performance as the number of integrated characteristic lines increases. Figure 4.8 presents the trefoil knot dataset with a range of LIC-like structures generated using different numbers of lines to emulate the sparsity of noise texture integration in 3D volumetric LIC visualization [FW08]. Short lines are filtered out. Table 4.1 reports the corresponding geometry size, geometry generation time, and rendering performance. Even as the number of characteristic lines grows, rendering remains real-time and geometry generation remains interactive.

In contrast to 3D LIC, our approach avoids both an expensive preprocessing step [FW08] and on-the-fly integration of characteristic lines. After geometry generation, no additional operations are required beyond rendering. Moreover, our approach is well suited for high-resolution output, as it does not rely on volume rendering. The system provides real-time feedback while avoiding the complexity of volumetric ray casting inside 3D volumes [RSHTE99, HA04].

Region of Interest Selection. In Figure 4.9, we assign cross-sectional glyphs based on velocity magnitude: low-velocity regions are represented with tube-like structures, while high-velocity regions transition to streamribbons. Here, we apply continuous interpolation from ribbon to circular shapes, providing a smooth transition that reflects changes in flow behavior and emphasizes turbulent regions.

Encoding Vector Field Rotational Attributes. Figure 4.10 shows simulated flow around an airplane model [ST69, SGH06]. The flow near the aircraft wings contains pronounced rotational components, which are critical for analyzing aerodynamic behavior. We use streamribbons to highlight regions of local streamline rotation. Streamline curvature is used as an

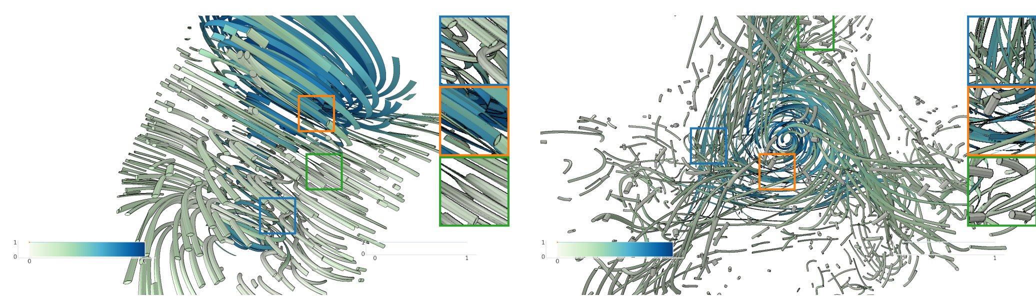


Figure 4.9: Encoding vector field attributes using different cross-section glyphs and interpolation instants. High magnitude velocities are marked with streamribbons and low values are marked with interpolation instants between a disk shape and a streamribbon.

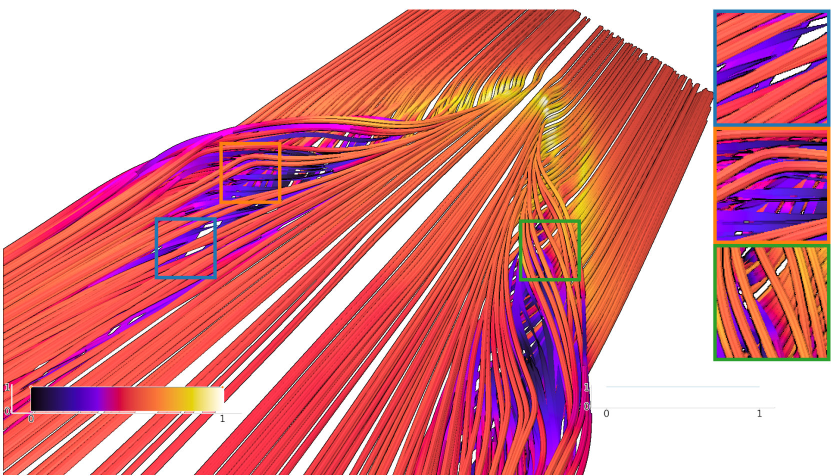


Figure 4.10: Encoding the rotational component of the vector field using different cross-sectional glyphs. We use a vertex curvature threshold ($t = 0.015$) so that high curvature values are marked with streamribbons and low values are extruded with circular cross-sectional glyphs.

input attribute to interpolate between a circular cross-section (green inset) and ribbon glyphs (orange and blue insets). Curvature is treated as a threshold parameter: areas exceeding the threshold are visualized using ribbons, while others are rendered with circular glyphs.

To ensure compatibility between glyph types, we sample equidistant vertices along the ribbon edges to match the vertex count of the corresponding circular glyphs. Only the top and bottom vertices of the ribbon are marked with flat shading flags; the remaining vertices are assigned smooth shading flags to preserve visual continuity.

Dataset	Input Geometry		Output Geometry		Run-Time	
	Streamlines	Vertices (k)	Vertices (k)	Indices (k)	Geometry (ms)	Rendering (ms)
Tornado Figure 4.6	353	21.7	179.4	1039.3	8.5	6.2
Tornado Figure 4.7 top left	353	21.7	179.4	1039.3	8.4	6.8
Tornado Figure 4.7 top right						7.3
Trefoil knot Figure 4.8 left	924	87	712	4179	32	7.7
Trefoil knot Figure 4.8 middle	1845	175	1435	8415	62	12
Trefoil knot Figure 4.8 right	2935	282	2302	13502	102	16
Tornado Figure 4.9 left	353	21.7	288.4	844.6	12.4	6.4
Trefoil knot Figure 4.9 right	3084	298.1	4740.8	14037.5	198	9
Delta wing Figure 4.10	356	307.3	4922.7	14747.2	209	9.2

Table 4.1: Frame time and geometry construction performance for various datasets and figure examples. Geometry is only regenerated if the dataset changes.

4.6 Conclusions and Outlook

In this chapter, we presented a geometry-based flow visualization technique that enables intuitive user control over visual appearance and supports the emulation of various established methods, such as streamribbons, as well as smooth transitions between them. At the core of our approach is the construction of genus-0 geometric glyphs along characteristic lines, with support for arbitrary cross-sectional shapes, user-defined radii, and customizable color and opacity mappings. By approximating volume integrals along view rays intersecting these geometric primitives, our method also replicates the appearance of texture-based and volumetric techniques such as 3D line integral convolution. Despite its flexibility, our approach remains straightforward to implement, highly parallelizable, and achieves real-time performance even at high screen resolutions.

Like many 3D flow visualization methods, our approach faces challenges related to visual clutter. Rendering a large number of characteristic lines can obscure important structures, while selecting a representative subset of lines that reveals key flow features remains a non-trivial task. Additionally, as our technique relies on geometric primitives for visualization, artifacts may occur when these primitives—such as tubes—are clipped by the near plane of the viewing frustum.

Potential extensions of the presented framework primarily focus on improving usability and automation. One such extension is the integration of automatic correspondence detection between vertices of different 2D cross-sectional glyphs [YF09], which would further reduce manual intervention. In addition, automated strategies for selecting and placing glyphs along streamlines—drawing inspiration from opacity optimization tech-

niques [GTG17, ZRPD20]—represent a promising direction for enhancing feature emphasis while mitigating visual clutter.

4.7 Acknowledgments

The tornado dataset is courtesy of Marchesin et al. [MCHM10], and the trefoil knot dataset is courtesy of Candelaresi and Brandenburg [CB11].

5 Moment-based Opacity Optimization for Geometric Structures

The work presented in this chapter is partly based on

Zeidan M., Rapp T., Peters C., Dachsbacher C.: Moment-Based Opacity Optimization. In Eurographics Symposium on Parallel Graphics and Visualization (2020), The Eurographics Association.

5.1 Introduction

In this chapter, we continue to explore flow visualization using opacity optimization techniques. Lines and surfaces [MLP*10, ELC*12] are commonly used by domain experts to detect and classify important features within the flow domain [BCP*12]. These geometric structures are generated by integrating steady and unsteady flows at specific or predefined locations. However, direct visualization often produces a large number of lines and surfaces, which can lead to visual clutter, in which important structures are obscured by less significant ones.

To better handle dense and cluttered flow structures, an efficient visualization technique aims to select a smaller yet representative set of flow structures that preserves the key features in an efficient manner. One possible approach to find this representative set is through a suitable seeding algorithm, where carefully chosen seed points are used to start the line integration. Good seeding points can be determined through density-based estimates of local

importance [MTHG03, SHH*07], by identifying locations of interesting features [YKP05, YWSC12], or based on a similarity measure of flow structures [MLP*10, MJL*13].

An alternative approach for visualizing important flow structures is to optimize the opacity of each geometric structure based on its importance. These algorithms model the opacity of each rendered segment using an optimization problem, aiming to maximize the opacity of important structures in cluttered regions while minimizing the opacity of less important surrounding structures. Günther et al. model opacity optimization as a global optimization process for 3D lines [GRT13, GRT14] and surfaces [GSE*14]. Additionally, a fast per-pixel opacity optimization for points, lines, and surfaces [GTG17] provides an analytic solution for opacity in screen space, independently for each pixel, followed by an object space smoothing step. However, these techniques often come with a significant memory footprint due to the generation of per-pixel linked lists on the GPU, and they are relatively slow due to the need for sorting a large number of linked lists in screen space.

In this chapter, we present a novel technique for opacity optimization using a moment-based representation (Section 5.3). Moment-based reconstructions [PK15, MKKP18] allow the recovery of monotonic functions from a small number of coefficients. In our context, they enable the recovery of the sum of squared importance values of surfaces that contribute to the occlusion of a given fragment. These methods are highly accurate in sparse regions of the visualization, where only a few structures are visible, and provide smooth, plausible results in denser regions. We also introduce an efficient screen space filtering approach that works directly on the moment buffers used for opacity optimization, eliminating the need for object space filtering. These desirable properties result in an efficient, reliable, and general opacity optimization technique. Our approach is not constrained by the geometric representation of the input data and can be directly applied to any geometry that fits into the rasterization pipeline. In Section 5.4, we describe the integration of opacity optimization techniques into our visualization framework. Next, in Section 5.5, we compare our method to state-of-the-art solutions for opacity optimization. Finally, we conclude the chapter and propose directions for future work in Section 5.6.

5.2 Opacity Optimization

In this section, we present the theoretical foundation of the opacity optimization problem. Since the analytic solution proposed by Günther et al. [GTG17] serves as the basis for our

approach, we briefly review it here. We also introduce the recently developed Fourier-based approximation of discrete signals [RGG20] as a complementary state-of-the-art formulation of the opacity optimization problem. In Section 5.3, we detail our moment-based opacity optimization approach.

5.2.1 Decoupled Opacity Optimization

Opacity optimization is a selection algorithm designed to emphasize important geometric structures. It adjusts opacities to ensure that dense, unimportant structures become transparent when they occlude more important ones, reducing visual clutter. For streamlines, each segment is assigned an importance value, and the optimization process computes the optimal opacity for each segment. These importance values can be derived from physical data properties, such as density, geometric attributes like curvature, or user-defined parameters.

Decoupled opacity optimization, originally proposed by Günther et al. [GRT13, GRT14], is a global least squares minimization process applied to line geometry within a 3D vector field. This technique was later extended to surface geometry [GSE*14]. Ament et al. [AZD17] further extended this process for volumetric data in ray space. Günther et al. [GTG17] adapted this approach for points, lines, and surfaces, executing the optimization process in parallel for each pixel on the GPU.

For a given pixel covered by $n \in \mathbb{N}$ fragments, each fragment $l \in \{0, \ldots, n-1\}$ is associated with a depth $z_l \in [z_{\text{min}}, z_{\text{max}}]$ and an importance $g_l \in [0, 1]$. Using an A-buffer [Car84], all this information is made available via per-pixel linked lists, though this comes at a high computational cost.

To compute the optimized opacity for fragment l, we calculate the sum of squared importance values up to depth z_l and the full sum:

$$G(z_l) := \sum_{\substack{k=0 \\ z_k < z_l}}^{n-1} g_k^2, \tag{5.1}$$

$$G_{\text{all}} := \sum_{k=0}^{n-1} g_k^2. \tag{5.2}$$

The opacity for each fragment l is optimized along the view ray for each pixel in screen space:

$$\alpha_l := \frac{p}{p + (1 - g_l)^{2\lambda} \left(rG(z_l) + q(G_{\text{all}} - G(z_l) - g_l^2) \right)}, \tag{5.3}$$

where p, q, r, and $\lambda \geq 0$ are user-controlled parameters with the following roles:

- p prevents empty renderings and controls how opacities approach one.
- q penalizes foreground clutter.
- r penalizes background clutter.
- λ adjusts the fall-off of importance from 1, allowing the user to emphasize important structures more strongly.

Decoupled opacity optimization [GTG17] solves Equation 5.3 by utilizing per-pixel linked lists, sorted by depth, for opacity optimization. This per-pixel optimization is followed by a geometric filtering pass to smooth the opacity values along adjacent vertices. While this parallel solution improves performance, it comes with the cost of a large memory footprint due to the creation of per-pixel linked lists. Furthermore, the sorting process becomes prohibitively slow for dense geometry.

5.2.2 Fourier Opacity Optimization

Fourier Opacity Optimization [RGG20] solves the opacity optimization process in the frequency domain using a Fourier series approximation of the importance function along the view ray. By using Fourier series, we replace per-pixel linked lists with a smaller number of frame buffers for storing Fourier coefficients. The cumulative importance per fragment along the view ray $G(z_l)$ is reconstructed using a truncated Fourier series.

This approach significantly reduces memory usage, and by replacing sorting with Fourier series reconstruction, it achieves a substantial processing gain. The result is a plausible approximation of the reference solution using linked lists.

Given a function $f(z)$ in the range $z \in [0,1]$, a Fourier series approximation with m frequency bands can be computed as:

$$f(z) \approx \frac{a_0}{2} + \sum_{k=1}^{m} a_k \cos(2\pi k z) + \sum_{k=1}^{m} b_k \sin(2\pi k z), \tag{5.4}$$

$$a_k = 2\int_0^1 f(z)\cos(2\pi k z)\, dz, \tag{5.5}$$

$$b_k = 2\int_0^1 f(z)\sin(2\pi k z)\, dz, \tag{5.6}$$

where a_0 and a_k, b_k with $k \in 1, \ldots, m$ are the coefficients of the Fourier basis functions. Once expressed in the Fourier basis, the integral of the function $f(z)$ in Eq. 5.4 is conveniently calculated as:

$$F(d) = \int_0^d f(z)\, dz \approx \frac{a_0}{2}d + \sum_{k=1}^{m} \frac{a_k}{2\pi k}\sin(2\pi k d) + \sum_{k=1}^{m} \frac{b_k}{2\pi k}(1 - \cos(2\pi k d)), \tag{5.7}$$

This approximation ensures that the bounds of the domain are reconstructed correctly:

$$F(0) = 0, \tag{5.8}$$

$$F(1) = \int_0^1 f(z)\, dz. \tag{5.9}$$

5.3 Moment-Based Opacity Optimization

In this section, we introduce a novel technique for opacity optimization. Figure 5.1 illustrates the main processing steps of our method, which computes optimized opacities for fragments in arbitrary rasterized geometry. To achieve this, we require a representation of the impor-

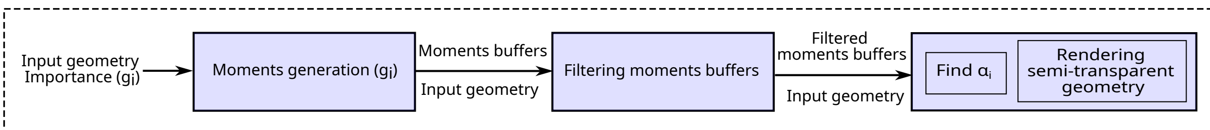


Figure 5.1: Overview of our proposed moment-based opacity optimization technique.

tance of each fragment covering a pixel. Decoupled opacity optimization [GTG17] uses an A-buffer [Car84], which incurs high overhead on graphics hardware [YHGT10]. Our approach improves upon this by adopting approximations from moment-based OIT [MKKP18].

Our technique performs opacity optimization in two passes. The first pass renders all transparent geometry into moment buffers to collect the importance of all fragments per pixel (Section 5.3.1). In the second pass, the transparent geometry is rendered again, and the information from the first pass is used to perform the opacity optimization for each fragment (Section 5.3.2). Additionally, our technique incorporates moment-based order-independent transparency (OIT), which introduces a third pass (Section 5.4.2). We also implement an A-buffer as a reference solution.

5.3.1 Accumulation of Importance

As described in Section 5.2.1, given a pixel with $n \in \mathbb{N}$ fragments, each having an importance value $g_0, \ldots, g_{n-1} \in [0, 1]$ and depth values $z_0, \ldots, z_{n-1} \in [z_{\text{min}}, z_{\text{max}}]$. A-buffers incur high costs because they explicitly store all of this information.

To optimize the opacity of a fragment $l \in \{0, \ldots, n-1\}$, we need to compute the accumulated importance up to depth z_l, which is defined as:

$$G(z_l) = \sum_{\substack{k=0 \\ z_k < z_l}}^{n-1} g_k^2. \tag{5.10}$$

The function $G(z)$, with $z \in [z_{\text{min}}, z_{\text{max}}]$, provides all the necessary information. It remains constant between adjacent fragment depths and increases by g_l^2 at depth z_l.

To achieve a more efficient opacity optimization technique, we seek a compact approximation of this monotonic function. The construction of this representation must be fast as it is performed per frame and per pixel. Moreover, it needs to be robust, especially when fragments are clustered in small parts of the depth range but should still work for more uniform distributions. Moment-based OIT [MKKP18] addresses these challenges and offers an effective solution. Therefore, we represent the function $G(z)$ using a small number of moments.

The moments describe the accumulated importance $G(z_l)$ as a function of depth z_l. For constant relative accuracy across the depth range, moment-based OIT uses a logarithmic warp of the depth values. We apply the same approach and define the warped depth as:

$$z'_l := \frac{\log(z_l) - \log(z_{\text{min}})}{\log(z_{\text{max}}) - \log(z_{\text{min}})} \cdot 2 - 1 \in [-1, 1], \tag{5.11}$$

where $l \in \{0, \ldots, n-1\}$. This formula is meaningful when z_l represents linear view-space depth, and $z_{\text{min}}, z_{\text{max}}$ are tight bounds computed from the geometry's bounding box and the near clipping plane.

Next, we define the moment of order $j \in \{0, \ldots, m\}$ as:

$$b_j := \sum_{k=0}^{n-1} g_k^2 \mathbf{b}_j(z'_k), \tag{5.12}$$

where the moment-generating function $\mathbf{b}_j$ is defined as:

$$\mathbf{b}_j(z'_k) := (z'_k)^j \in \mathbb{R} \tag{5.13}$$

for power moments or

$$\mathbf{b}_j(z'_k) := \exp\left(\left(2\pi - \frac{\pi}{10}\right) ij \frac{z'_k + 1}{2}\right) \in \mathbb{C} \tag{5.14}$$

for trigonometric moments. Power moments provide a faster but less accurate approximation.

Since Equation 5.12 defines moments as a summation over all fragments, a single rendering pass with additive blending allows us to compute all moments. We compute the warped depth z'_k in the fragment shader and output $\mathbf{b}_j(z'_k)$ for all $j \in \{0, \ldots, m\}$ to the channels of our render targets.

Note that:

$$b_0 = \sum_{k=0}^{n-1} g_k^2 \mathbf{b}_0(z'_k) = \sum_{k=0}^{n-1} g_k^2 = G_{\text{all}} \tag{5.15}$$

holds the total importance.

Typical values for the maximum order m are four or six for power moments and three or four for the complex-valued trigonometric moments. Given the high depth complexity of typical visualization datasets, we select the high-quality moment-based reconstruction, using three trigonometric moments stored in seven single-precision floats for both opacity optimization and OIT.

5.3.2 Opacity Optimization

Decoupled opacity optimization [GTG17] defines the optimized opacity for fragment $l \in \{0, \ldots, n-1\}$ as:

$$\alpha_l := \frac{p}{p + (1 - g_l)^{2\lambda}\left(rG(z_l) + q(G_{\text{all}} - G(z_l) - g_l^2)\right)} \tag{5.16}$$

where $p, q, r, \lambda \geq 0$ are user-defined parameters. We obtain the exact G_{all} from the buffer storing the zeroth moment. The remaining challenge is to approximate $G(z_l)$ as defined in Equation 5.10. For this, we rely on moment-based reconstructions used by moment-based OIT [MKKP18].

These reconstructions use the moments $b_0, \ldots, b_m$ to compute two approximations to $G(z_l)$. In the case of power moments, one approximation is guaranteed to be less than $G(z_l)$, while the other is always greater. These approximations are the best possible lower and upper bounds for $G(z_l)$, considering that we only know the moments $b_0, \ldots, b_m$ [PK15].

Since we want an approximation to $G(z_l)$ that excludes g_l^2 without being overly conservative, we weight the lower bound by 75% and the upper bound by 25%. This strategy has been proven to work well in moment-based OIT [MKKP18]. For trigonometric moments, the two approximations have different meanings, but they are used in the same manner.

The code published with moment-based OIT includes conversions to transmittance by taking the exponential of the reconstruction. Since we accumulate importance additively rather than multiplicatively, these conversions are not needed in our case. Aside from that, the reconstruction code applies to opacity optimization without modification.

Thus, we can implement opacity optimization in a shader. The shader reads the moments from the render targets of the first pass, reconstructs $G(z_l)$ as described above, and then evaluates Equation 5.16 to compute the optimized opacity α_l. The remaining steps are to filter these opacities (Section 5.3.3) and perform order-independent compositing (Section 5.4.2).

5.3.3 Screen Space Filtering for Smoothed Opacity

Decoupled opacity optimization [GTG17] performs per-pixel opacity optimization, which initially leads to discontinuities in the results. To mitigate this issue, minimal opacity values are first stored per segment, followed by object space filtering—typically Laplacian smoothing—applied across adjacent segments [GTG17, RGG20]. However, this approach requires detailed knowledge of the geometry's topology and involves a non-trivial implementation in compute shaders. Points, lines, and surfaces each demand distinct handling and additional data structures.

We introduce a more general and less complex alternative. Our method smooths the moment buffers directly. Recall that each moment b_j is stored in a separate texture channel for each screen space pixel. We apply a standard two-pass Gaussian blur to each moment individually. This operation is widely used in graphics and benefits from highly optimized implementations. Because moments are linearly dependent on the function $G(z)$, this filtering effectively smooths the function $G(z)$ across pixels.

The result of this process differs from directly filtering opacities for three key reasons. First, the moment-based reconstruction is non-linear, so the filtering introduces non-linear approximation errors. Second, Equation (5.16) is a rational function, introducing a non-linear dependency on $G(z_l)$. Third, our filtering operates in screen space using a fixed kernel size, unlike the object space filtering of previous approaches. Consequently, the radius around important structures where clutter is reduced remains constant in screen space.

Despite these differences, our method achieves the main objective of opacity optimization filtering: clutter is not only reduced for fragments directly occluding important structures, but also for nearby fragments, enabling a smooth and continuous transparency transition.

5.4 Implementation

In this section, we provide a brief overview of our visualization framework and detail how we integrate moment-based opacity optimization into the rendering pipeline. Specifically, we discuss how this is combined with moment-based OIT and A-buffers. Our framework utilizes OpenGL, and we evaluate the system on a machine equipped with an NVIDIA GeForce 1080 Ti. All rendering tasks are GPU-accelerated, using either the rasterization pipeline or compute shaders.

5.4.1 Generating Visualization Primitives

To showcase the general applicability of our approach, we use datasets comprising points, lines, and arbitrary triangle meshes. For line datasets, we focus on streamlines, which are computed using explicit Euler integration as a preprocessing step. These streamlines are derived from velocity fields, with seed points placed uniformly at random within a user-defined volume.

To define the importance of each line, we either compute the average curvature of the line [BSSZ08] or use the velocity at the seeding point in the field. For line data, adjacency information is available, but the Laplacian smoothing used in decoupled opacity optimization is not supported for points or triangle meshes in our implementation. However, our screen space filtering technique is applicable to all types of geometry.

5.4.2 Moment-Based OIT

Our opacity optimization method is compatible with any order-independent transparency (OIT) technique. We implemented both a reference solution using an A-buffer and our moment-based OIT [MKKP18]. In this section, we briefly review how moment-based OIT works and explain how it integrates with our opacity optimization approach.

Moment-based OIT involves rendering all transparent geometry twice. In the first pass, moment buffers are generated, similar to those used for opacity optimization. The moment for each fragment l is defined as:

$$b'_j := \sum_{k=0}^{n-1} -\log(1-\alpha_k)\mathbf{b}_j(z'_k). \tag{5.17}$$

Using the reconstruction methods described earlier, the moment buffers allow us to approximate the transmittance at depth z_l, which is given by:

$$T(z_l) := \prod_{\substack{k=0 \\ z_k<z_l}}^{n-1} (1-\alpha_k) = \exp\left(\sum_{\substack{k=0 \\ z_k<z_l}}^{n-1} \log(1-\alpha_k)\right). \tag{5.18}$$

In the second pass, the transmittance is estimated for each fragment, and the fragment color is multiplied by the transmittance and opacity before being added to the pixel color using additive blending.

In the context of our opacity optimization, transparent geometry that requires opacity optimization is rendered three times (Figure 5.1). All three passes use additive blending. The first pass computes the moments required for opacity optimization, rendering to multiple render targets that hold single-precision floats. For three trigonometric moments, the moments have one, two, and four channels, for a total of seven. Screen space filtering is applied to the moments generated during this first pass, using a separable Gaussian blur in two fragment shader passes, leveraging hardware-accelerated bilinear filtering [Wol18].

In the second pass, optimized opacities α_l are computed for each fragment, and the moments necessary for OIT are calculated. In the third pass, both moment buffers are read to compute the optimized opacity and transmittance. This pass also implements shading and multiplies the fragment color by transmittance and opacity. A final pass executes one thread per pixel to composite the foreground with the background color using b_0'.

5.4.3 A-Buffers

Our implementation also supports A-buffers, which enable decoupled opacity optimization and serve as a ground truth for OIT. The GPU implementation follows the approach outlined in [YHGT10], and constructing the A-buffer only requires a single draw call that fills the per-pixel linked lists.

To integrate our moment-based opacity optimization with OIT using an A-buffer, we still generate the moments in an additive render pass and filter them. In this case, one thread per pixel loads each linked list, sorts it by depth, and performs blending using opacities computed on the fly based on the moment buffers. Similarly, we can use the A-buffer to compute optimized opacities according to Equation (5.16) in real-time.

This approach differs from decoupled opacity optimization because it does not filter opacities. Screen space filtering is not applicable to A-buffers, but the A-buffer is useful for validating our moment-based approximation, as discussed in Section 5.5.2.

Decoupled opacity optimization [GTG17] follows a different path. After the A-buffer is constructed, one thread per pixel loads each linked list, sorts it by depth, and performs opacity optimization for each fragment. The A-buffer stores indices of line segments per fragment, allowing the minimal opacity for each segment to be stored using atomic instructions. Once the minimal opacities are known, a compute shader filters them along the lines. Since the

repeated Laplacian smoothing kernel eventually approximates a Gaussian kernel, we perform this smoothing in a single pass using a Gaussian filter. Afterward, the linked lists are loaded and sorted once more to perform blending.

5.4.4 Fourier Opacity Optimization

Fourier opacity optimization [RGG20] proposes using the same object space filtering as decoupled opacity optimization. Since it does not use an A-buffer, this method requires rendering all geometry four times: twice for opacity optimization and twice for OIT. Our approach, which uses screen space filtering, requires only three geometry passes, as opacity optimization is integrated into OIT. For a fair comparison, we also implement Fourier opacity optimization with screen space filtering in our system.

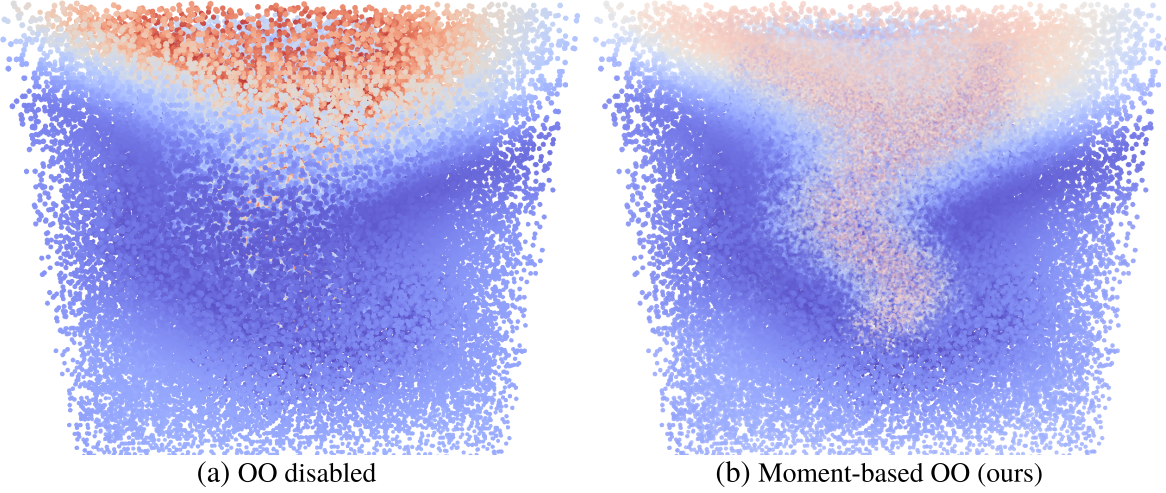
In our experiments, we use nine real Fourier coefficients for Fourier opacity optimization, as the authors of [RGG20] reported artifacts when using fewer coefficients.

5.5 Results and Evaluation

This section presents a comparison of our moment-based opacity optimization (MBOO) with decoupled opacity optimization (DOO) and Fourier opacity optimization (FOO) across several datasets containing points, lines, and surfaces. We start by evaluating how effectively each technique highlights key structures (Section 5.5.1). Next, we directly compare all methods without filtering to assess the approximation error in the moments (Section 5.5.2). Following that, we provide an in-depth analysis of the impact of our screen space filtering (Section 5.5.3). Finally, we review the runtimes of all tested techniques (Section 5.5.4).

5.5.1 Quality of Opacity Optimization

We start by examining the simple point dataset shown in Figure 5.2 to evaluate the fundamental behavior of our opacity optimization. The points are randomly distributed in a synthetic tornado vector field with a resolution of 128^3 [MCHM10]. The central vortex contains significant structures that need to remain visible, while the surrounding slower-moving air should



(a) OO disabled (b) Moment-based OO (ours)

Figure 5.2: Visualization of the tornado vector field using 100 k points. The colors and importance are based on the velocity magnitude, and the rendering utilizes moment-based OIT. Notice how opacity optimization reveals the vortex at the center while preserving the less significant structures.

not obscure them. Our opacity optimization effectively uncovers these important structures without removing the less important features.

Figure 5.3 presents a more thorough evaluation using challenging streamline datasets. Each dataset utilizes the average curvature to determine the importance of the lines for opacity optimization, with the transfer function visualizing the velocity magnitude. The first row shows streamlines for the tornado dataset discussed earlier. Lines with high curvature are predominantly found in the central vortex, and all opacity optimization techniques are successful in revealing these structures.

In the second row, we display magnetic field lines from the decay of magnetic knots [CB11]. A close-up of one of the two symmetric rings in this dataset is shown. Without opacity optimization, this ring is hidden by foreground clutter. However, opacity optimization uncovers this ring and highlights other lines with high curvature. The Rayleigh-Bénard convection shown in the third row arises when a thin layer of fluid is heated from below. Opacity optimization enhances the visibility of the resulting convection cells while preserving slower flows that do not obstruct important structures. In the fourth row, the trefoil knot dataset [CB11], another magnetic field, is shown. It consists of three interlocked magnetic rings that decay over time, and opacity optimization accentuates the structures near these rings.

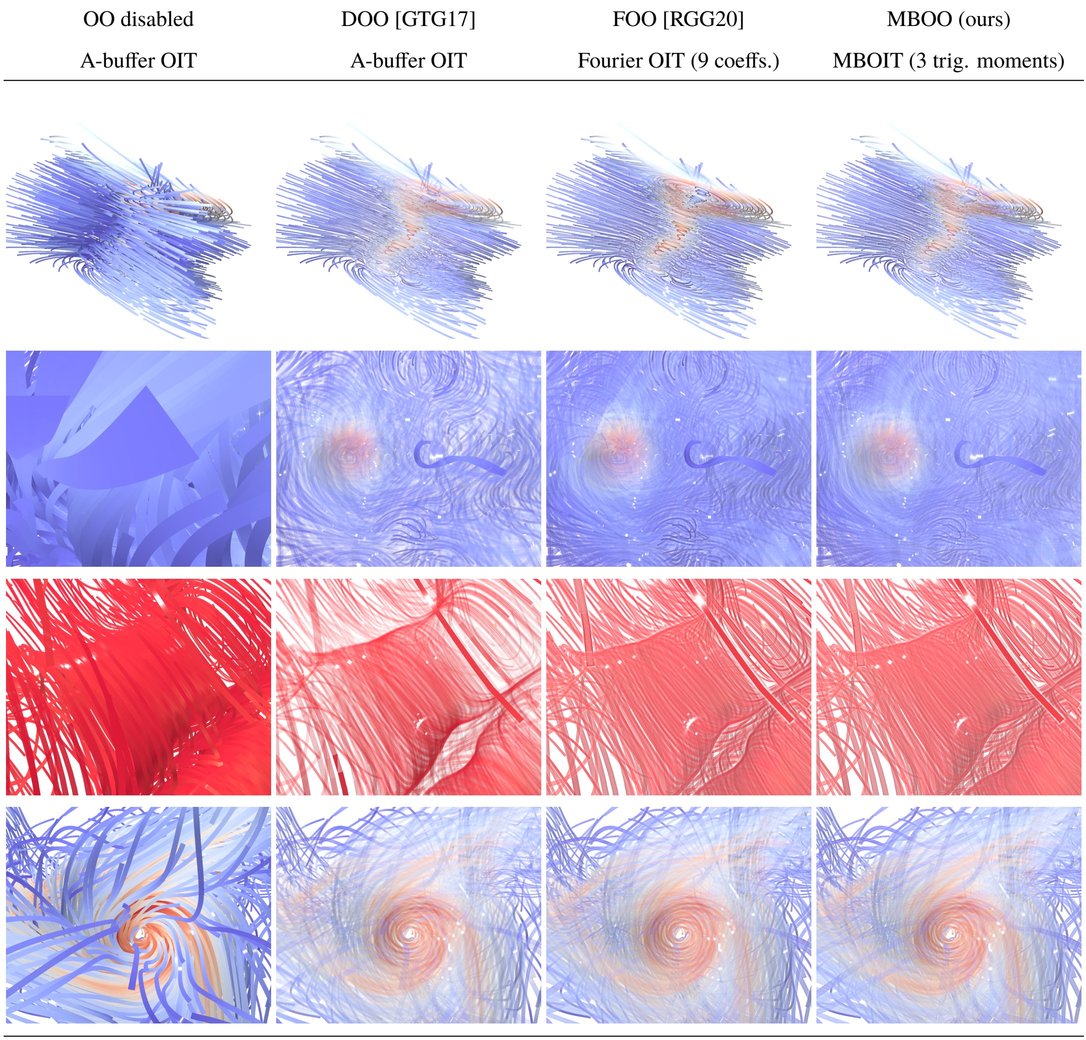


Figure 5.3: Comparison of three opacity optimization techniques using four challenging streamline datasets: Tornado, Borromean rings, Rayleigh-Bénard convection and trefoil knot. Note the overall improvement in clarity through opacity optimization and the differences in filtering between DOO and FOO or MBOO. Our technique achieves comparable quality to DOO at a substantially lower cost.

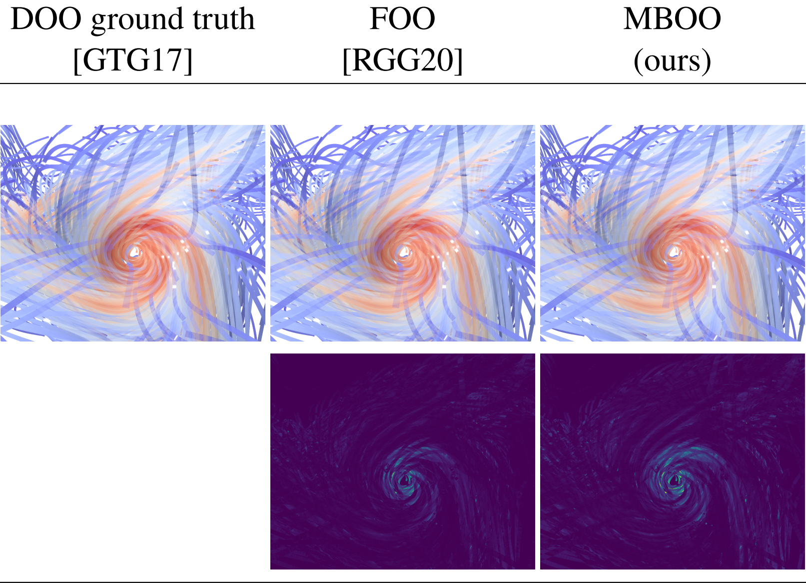


Figure 5.4: A comparison of opacity optimization techniques on the trefoil knot with both object space filtering and screen space filtering disabled. OIT uses A-buffers. In this setting, all differences are due to the approximation of the truncated Fourier series or the moment-based reconstruction. The bottom row shows L1-difference images with the Viridis color map.

Opacity optimization consistently improves the clarity in all visualizations. The results of Fourier opacity optimization and our moment-based opacity optimization are similar. The most notable differences appear in the second row, where Fourier opacity optimization fails to eliminate some foreground clutter at the top left. Decoupled opacity optimization produces significantly different results due to differences in filtering. The apparent size of the object space filtering on screen depends on the tessellation of the lines and perspective. In most cases, this filter is larger than our screen space filter. While larger filters may provide more clarity at the shown resolution, our screen space filter, with a 21×21 pixel footprint, offers finer details that are more visible when the image is viewed in full screen. The size of the filter is intuitive to control and efficient to apply.

5.5.2 Validation Against Ground Truth

To make a more direct comparison between decoupled opacity optimization and the other techniques, we disable filtering. Decoupled opacity optimization is performed dynamically during OIT compositing as detailed in Section 5.4.3. Screen space filtering is also turned off. We use the velocity magnitude as the importance measure, which is visible through

the transfer function. In this scenario, decoupled opacity optimization serves as our ground truth, and all deviations represent errors introduced by the truncated Fourier series or the moment-based reconstruction. Both techniques closely match the ground truth. The largest errors occur in areas with important lines near unimportant lines, where strong overdraw happens. In such regions, the advantage of moment-based reconstructions diminishes. Our technique introduces a slightly higher error than Fourier opacity optimization. However, note that Fourier opacity optimization uses nine coefficients, while our technique uses only seven.

5.5.3 Screen Space Filtering

Since screen space filtering for opacity optimization is a novel approach in our work, we provide a more detailed comparison with alternative methods in Figure 5.5. Without filtering, foreground clutter is removed only if it directly obstructs important structures. As a result, there is no clear distinction between clutter and essential lines, leading to a less effective visualization. The object space filtering used in decoupled opacity optimization helps address this issue. However, controlling the apparent size of the filter on screen is challenging, and the method's implementation is more complex. In contrast, our screen space filtering removes foreground clutter from a region with a fixed size on screen (in this case, 21×21 pixels). In scenarios with higher geometry density (Figure 5.5 bottom), it enhances the visualization by adding more detail, thereby improving the clarity and perception of the streamlines.

Figure 5.6 highlights another advantage of our screen space filtering. We apply it to two triangle meshes with complex topology. The bunny and the bush are assigned constant importance values of 0.9 and 0.2, respectively. Computing the adjacency information needed for Laplacian smoothing would be challenging, as the meshes are not designed with this in mind. However, our screen space filtering is straightforward to implement, without requiring any special modifications. Any geometry that can be rendered through the rasterization pipeline is compatible with our method. As a result, we achieve effective opacity optimization in scenarios that would be difficult to handle with decoupled opacity optimization.

5.5.4 Performance Evaluation

Table 5.1 presents the timing results for most of the figures, measured on an NVIDIA GeForce 1080 Ti. All images were rendered at a resolution of 1920×1080. The timings where an

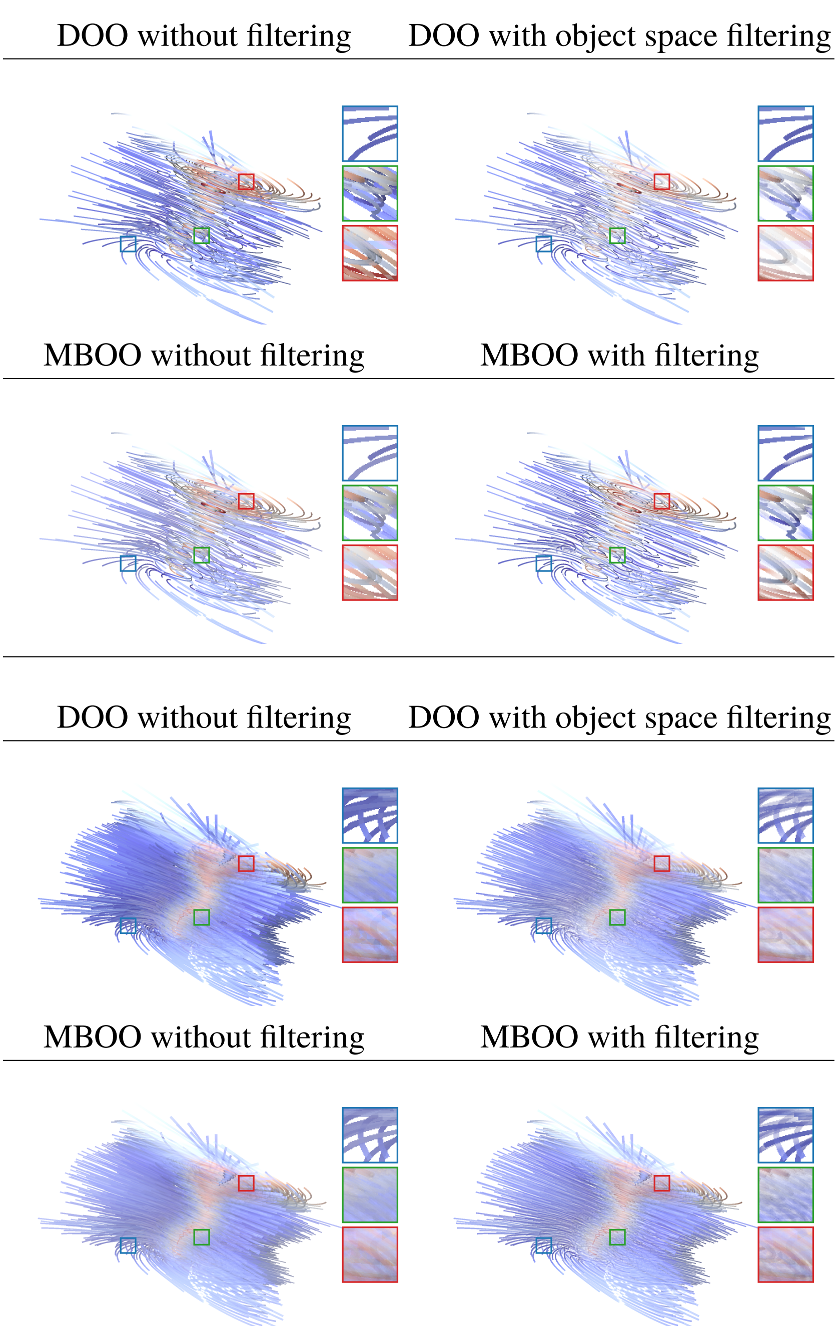
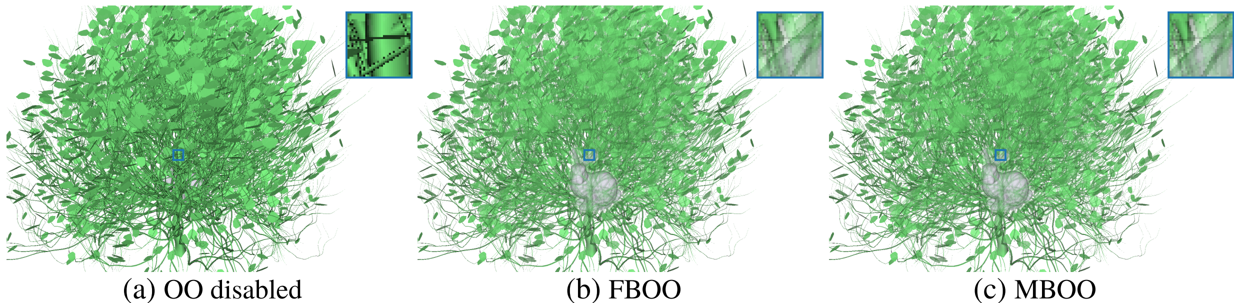


Figure 5.5: The tornado dataset with two different line counts and different methods of filtering for opacity optimization. OIT uses A-buffers. Note how screen space filtering removes clutter in a fixed radius around important structures.



(a) OO disabled (b) FBOO (c) MBOO

Figure 5.6: Two complicated triangle meshes with different importance. Opacity optimization removes foreground clutter that would otherwise hide the bunny in the bush. OIT uses an A-buffer.

Dataset	**Figure**	**Geometry**			**Run-time**			
		Num. lines	**Max. num. segments**	**Num. vertices**				
Tornado	Figure 5.3 first row	1728	110592	691200	6.8	12.0	5.8	**5.6***
	Figure 5.5 top	512	204800	32768	3.1	3.7		
					2.2*	**3.0***		
	Figure 5.5 bottom	4096	1638400	262144	32.5	35.7		
					23.1*	**24.0***		
Borromean	Figure 5.3 second row	2845	182080	2845000	60.0	100.0	30.5	**26.9***
Rayleigh-Bénard	Figure 5.3 third row	863	55232	161721	25.5	40.9	**7.9**	8.2*
Trefoil knot	Figure 5.3 fourth row	1293	82752	258600	18.7	31.3	**8.7**	9.4*
	Figure 5.4				32.2	21.2	**20.9***	

Table 5.1: Total frame times in milliseconds and primitive counts for the results in the figures above. Our techniques are marked with an asterisk. The fastest technique in each comparison is marked bold. Note that our technique clearly outperforms decoupled opacity optimization and performs similar to Fourier opacity optimization.

A-buffer is used for OIT are less informative due to its high overhead. Therefore, we focus our analysis on Figure 5.3. Our results show that moment-based opacity optimization clearly outperforms decoupled opacity optimization, with the advantage increasing as scene complexity grows. In comparison to Fourier opacity optimization, our method achieves similar performance. Although our technique uses seven channels instead of nine, Fourier opacity optimization requires fewer arithmetic operations. These two factors appear to roughly offset each other. Both techniques in our implementation use single-precision floating point values for each channel.

5.6 Conclusions and Outlook

Our moment-based opacity optimization introduces an effective new tool for visualization pipelines. It offers a faster and more straightforward implementation than decoupled opacity optimization, while achieving comparable clutter removal capabilities. In terms of both quality and speed, our technique is on par with Fourier opacity optimization, but it operates with a smaller memory footprint and benefits from the advantages of moment-based OIT. Additionally, our screen space filtering decouples the opacity optimization process from specific geometric primitives. By not requiring topological information, our method is simpler to integrate into visualization pipelines.

The method operates through three sequential additive rendering passes, all of which are inherently parallel. This parallelism is not limited to the GPU; rendering with opacity optimization for a single dataset can easily be distributed across multiple nodes, provided there is sufficient bandwidth to exchange framebuffers. Therefore, our approach could facilitate the visualization of even larger datasets across multiple GPUs in the future.

Acknowledgments

The tornado dataset is courtesy of Marchesin et al. [MCHM10]. The Borromean rings and trefoil knot datasets are courtesy of Candelaresi and Brandenburg [CB11]. The Bunny model in Figure 5.6 is courtesy of the Stanford Computer Graphics Laboratory and the bush is courtesy of Blendswap user jlnh.

6 Efficient Visualization of Large Particle Datasets on GPU

6.1 Introduction

Understanding large and complex particle simulations of the universe is a critical area of study for physicists, astrophysicists, and other domain scientists. Simulations of cosmological structures, galaxy formation, and the evolution of the universe can be effectively analyzed through dynamic particle-based simulations, such as smoothed particle hydrodynamics (SPH), which are executed on large-scale compute clusters [NPG*15, JKPT21]. However, with the continuous advancements in computational power and storage capacity, the data generated by these simulations have grown to the gigascale, terascale, and beyond. As a result, the interactive visualization of these particle datasets has become a significant challenge, especially in maintaining rapid and responsive feedback for effective data exploration.

Datasets produced by astrophysics simulations [MVP*18, NPS*18, NPS*17, PNH*17] often contain millions to billions of particles, each with multiple attributes. These datasets frequently exceed the memory capacity of available GPUs, and the limitations of data transfer between the CPU and GPU can introduce performance bottlenecks. To mitigate these challenges, an out-of-core (OOC) memory management system is essential for the efficient processing of large particle sets. Additionally, level-of-detail (LoD) structures [SKKW23, HBJP12, CNLE09] offer representative visualizations of the data based on the current viewpoint and available memory budget. These systems dynamically select suitable LoD representations to optimize rendering performance while adhering to memory constraints.

Alongside LoD methods, sampling techniques [RPD20, HBW*20, BDPA18, DWS*17] are commonly employed to provide compact and structurally faithful summaries of the origi-

nal dataset, enabling fast and responsive exploration. However, these sample sets must be precomputed, as the full dataset is often too large to be stored directly in GPU memory.

This chapter presents a visualization system designed for efficiently handling and rendering large particle sets using GPU-based OOC paging techniques. We assume that the particle set fits entirely in main memory. The proposed system combines OOC techniques with LoD methods to construct approximate yet faithful particle representations, allowing millions of particles to be rendered at interactive frame rates on the GPU.

The main contributions of this chapter are as follows:

1. A novel OOC GPU-based paging technique that is capable of handling both homogeneous blocks of data and two-level non-uniform data blocks.

2. An efficient GPU-based, chunk-based sorting technique for large data arrays, utilizing binary search to rank and merge elements into a sorted sequence.

3. A GPU-based algorithm for constructing OOC indexing structures for large particle sets, which are then applied to LoD rendering.

The chapter is organized as follows: In Section 6.2, we review related work. In Section 6.3, we present GPU OOC paging techniques for efficiently uploading large data blocks into GPU core upon request. In Section 6.4, we introduce a chunk-based OOC merge sort technique for handling large arrays of data. In Section 6.5, we describe our indexing structures. The construction of particles LoD based on the proposed indexing structures is discussed in Section 6.6. Experimental results are presented in Section 6.7, and we conclude with final remarks and future directions in Section 6.8.

6.2 Related Work

In this section, we discuss closely related work, particularly in the areas of OOC paging on the GPU, GPU parallel sorting, LoD indexing structures, and GPU-based OOC rendering and visualization.

GPU-based Out-of-Core Paging Zheng et al. [ZNZ*16] proposed architectural and system-level enhancements to improve the performance of paged memory on GPUs. They identified

bottlenecks in address translation and page fault handling, and introduced optimizations such as low-latency demand paging, improved Translation Lookaside Buffer (TLB) designs, and efficient page migration mechanisms. Their experiments revealed a $2\times$ average slowdown of traditional paged memory compared to manual data transfers and demonstrated a 12% average performance improvement with their proposed optimizations in replayable far-fault handling and prefetching. Van Beurden and Scholz [vBS23] presented a streaming-based OOC approach for GPU array operations.

NVIDIA introduced Unified Virtual Memory (UVM), a memory model that unifies device and host memory, enabling GPUs to transparently access a memory space larger than their physical video random-access memory (VRAM) by spanning multiple accelerator cards and system memory [NVI24]. Wagley et al. [WMC*24] explored Remote Direct Memory Access (RDMA)-backed paged memory for GPUs in the context of irregular workloads, highlighting key limitations and performance trade-offs when using Non-Volatile Memory Express (NVMe)-backed storage for graph-based neural network execution.

In our experiments, we opt for a simple paging strategy tailored to particle datasets, which consist of independent, unordered primitives with uniform access patterns and no explicit spatial connectivity. This property allows paging decisions to be driven primarily by spatial locality and view-dependent importance, making the strategy well suited for two-level indexing structure construction and LoD rendering. Nevertheless, we believe that many of the aforementioned paging techniques can be integrated into our system to enable further optimizations.

Parallel Sorting on GPU In our particle rendering pipeline, sorting is required for two key reasons. First, it enables the construction of indexing structures suitable for LoD rendering, where particles must be spatially organized to support hierarchical data access. Second, sorting ensures correct back-to-front ordering during transparency blending, which is essential for producing visually plausible output. Over the past three decades, the continuous advancement of GPU architectures and the emergence of general-purpose programming models such as CUDA and OpenCL have enabled the development of numerous efficient, high-performance sorting algorithms. A diverse set of sorting techniques has been implemented on GPUs, including radix sort [MG11, AM22], quicksort [CT10], merge sort [SHG09], and others.

While early work by Purcell et al. [PBMH02] demonstrated bitonic sort on GPUs using fragment shaders, modern sorting algorithms [SHG09, AM22] significantly improve performance by leveraging CUDA-based parallelism, memory coalescing, and warp-level optimizations. Sintorn and Assarsson [SA08] introduced a GPU-based hybrid sorting algorithm that begins with a single pass of bucket sort to partition the input list into sublists, which are then sorted in parallel using a vectorized implementation of merge sort. Green et al. [GMB12] proposed the GPU Merge Path algorithm, an efficient merging strategy tailored for GPUs.

For a broader perspective, comprehensive surveys on GPU-based sorting are available in [CT09, SJC18], offering detailed comparisons of algorithmic approaches, performance trade-offs, and implementation strategies across various GPU platforms.

Tanasic et al. [TVJ*13] introduced a comparison-based sorting algorithm tailored for systems with multiple GPUs, achieving scalable performance across heterogeneous environments. Maltenberger et al. [MITR22] examined multi-GPU sorting performance using modern interconnects such as NVLink and PCIe, analyzing both unified and discrete memory configurations across up to eight GPUs. Their study provides detailed insights into communication overhead, scalability, and throughput under different architectural setups. Sato et al. [SMMO16] proposed an out-of-core sorting framework that leverages I/O chunking and latency hiding to overlap data transfers between flash-based non-volatile memory and GPU computation, significantly enhancing throughput for large-scale datasets.

In our pipeline, we introduce a GPU-based merge sort algorithm suitable for large particle sets. We build our algorithm on top of an optimized sorting algorithm offered by the NVIDIA API and efficiently merge sorted chunks using a fast GPU-based merge algorithm that employs binary search to find key ranks in the sorted sequence.

Out-of-Core Indexing Structures Building OOC indexing structures for massively large models—such as BVHs, kd-trees, octrees, or sparse voxel hierarchies—has been extensively studied over the past few decades [PF03, CNLE09, SLM25]. In scenarios where the input geometry is too large to fit into device memory, indexing structures are typically constructed in several rounds, where the geometry is split into manageable units that fit within the available memory budget. As an initial step, the input geometry is partitioned into memory-resident blocks suitable for processing on the GPU, and a local tree structure is built for each block [PFHA10, WHY*13, SOW20]. The final hierarchy is then constructed on top of these blocks by treating each block's bounding box as an input primitive. However, these

methods generally require one or more passes over the input geometry to determine the block boundaries. In some cases, additional refinement is required, either by splitting large blocks or merging small neighboring ones, in order to satisfy a predefined geometric threshold—typically expressed in terms of the number of primitives, total memory footprint, or screen-space error per node.

Level-of-Detail Rendering LoD represents a simplified version of scene geometry to enable faster rendering, processing, and evaluation. LoD structures for point cloud data or tessellated geometry provide multiple representations of the model with varying resolutions, suitable for different viewing conditions and motivated by the human perception system [SZD*23, LRC*02]. In most cases, LoD is constructed as a multi-resolution representation built on top of a spatial hierarchy, such as grids, octrees, or binary trees. Inner nodes contain a simplified version of their child nodes [SKKW23, BP23]. During rendering, a selection criterion is applied to determine an appropriate level cut, often based on screen-space metrics such as the pixel footprint of projected tree nodes [SKW22]. Layered Point Clouds (LPC) [GM04] build a LoD structure by sampling the point sets of child nodes into the parent node. A similar approach was presented in [SKKW23] for large point cloud models using two sampling strategies.

Crassin et al. [CNLE09] employed an octree as a multiresolution hierarchy, where a 3D volume is represented as an octree, with each tree node corresponding to the subvolume it contains, represented as a brick of an M^3 grid. A brick pool holds in-core subvolumes that are loaded during viewpoint changes. When the view changes, new subvolumes are loaded, and the least recently used blocks are swapped out. An efficient traversal algorithm performs frustum culling, empty space skipping, LoD selection, and ray casting to guide data production and streaming based on information extracted during rendering. In [HBJP12], a virtual memory system for 3D volumes using a two-level page table hierarchy is used to subdivide the volume into small blocks of 32^3 voxels. Only the required blocks of data are stored in the resident cache block of a 3D texture, building a multi-resolution hierarchy.

6.3 Out-of-Core Paging

Since the introduction of General-Purpose Computing on Graphics Processing Units (GPGPU), GPUs have been extensively used to achieve high computational throughput in a

wide range of scientific applications. Modern GPUs offer significantly higher floating-point operations per second (FLOPS) than CPUs, often by an order of magnitude [SADK20]. However, many GPU-accelerated applications are designed with the assumption that all relevant data can fit entirely within the limited GPU device memory. This restriction imposes a significant limitation on data-intensive applications.

To overcome the limited memory capacity of GPUs, a memory management system is required to partition and load data into GPU memory on demand. These techniques are generally referred to as memory virtualization and have been extensively studied in the context of CPU systems to support memory extensions through main storage [Den96]. Although the concept appears straightforward, its practical implementation on GPUs is non-trivial. This is because managing data on the GPU involves additional challenges: data is accessed concurrently by a large number of threads executing in parallel, and any correct implementation must ensure data consistency and synchronization across all active threads.

In this section, we introduce a general-purpose paging system for memory virtualization on the GPU in Section 6.3.1. Our system assumes that the application data fits in host memory but does not need to reside entirely in GPU memory. We begin by presenting a memory system based on a simple page table implementation, which maps large application arrays into a virtual page table structure. Upon request, data blocks can be transferred from the CPU to the GPU, and parallel execution continues until all data requests have been fulfilled. Additionally, we present an extension to this paging strategy to map non-uniform data blocks into a fixed-size page table structure in a compact way in Section 6.3.2.

6.3.1 Homogeneous Data Paging

In the following, we explain the main details of the paging strategy and outline the key steps involved in the paging process and parallel processing with OOC paging. More implementation details, including code snippets, are provided in Appendix A.

We employ a simple paging strategy [ZNA15] to load data into GPU memory on demand. Given a large block of data residing in main memory, we partition it into fixed-size pages and construct a page table as an array of indices, where each element references an OOC page. Each page table entry contains two or three binary flags: one flag indicates whether a page currently resides in GPU memory, a second flag signals a page request from the GPU, and

optionally, a third flag marks pages that require write-back when data modified on the GPU must be synchronized with host memory (see Figure 6.1).

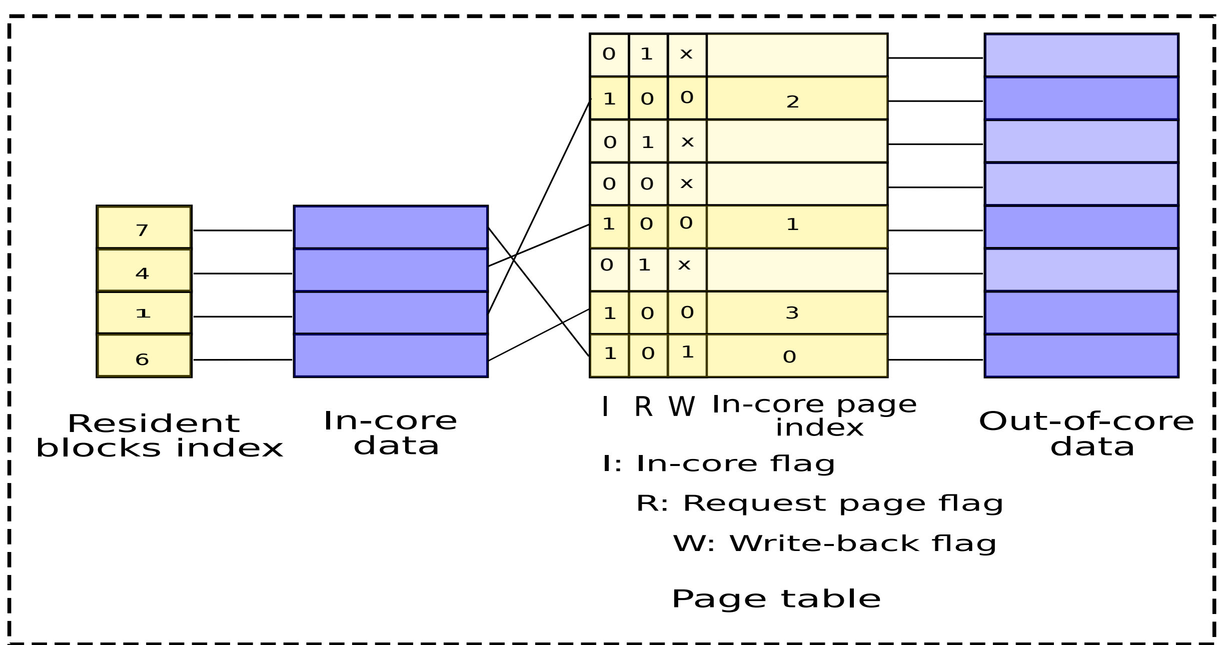


Figure 6.1: The page table structure includes the I bit, which indicates whether a page resides in GPU memory; the R bit, which represents a page request; and the W bit, which is used for page write-back. The resident block index array is used during page evacuation to invalidate the corresponding I flag for a previously loaded page.

We always maintain two consistent copies of the page table on the CPU and GPU. After each kernel call that accesses OOC data blocks, we copy the page table from the GPU to the CPU, and then iterate over the page table page by page to copy the required pages from the host to the device. Meanwhile, we update the corresponding request and residence bit flags. We also maintain an indirection array that holds references to previously loaded pages from past kernel calls. Upon page replacement, we invalidate the resident bit flag of the corresponding previously loaded page. In certain scenarios, some data blocks may need to be copied back from the GPU to the CPU; in such cases, we iterate over these pages and write them back to the CPU before supplying any new page requests.

Once the required pages are transferred to the GPU, in full or in part depending on the available GPU memory budget, we copy the page table to the GPU and call the kernel again to continue processing. To save bandwidth, we concatenate all pages of the current round on the CPU and perform a single CPU-to-GPU copy. Our paging strategy uses a first-in, first-

out (FIFO) approach by always iterating from the last accessed page index in the previous call in a circular manner. Since it is not guaranteed that all data requests are satisfied in one iteration, a do-while loop continuously calls the kernel and supplies page requests, exiting only when no more page requests or write-backs are needed.

6.3.2 Two-Level Hierarchical Data Paging

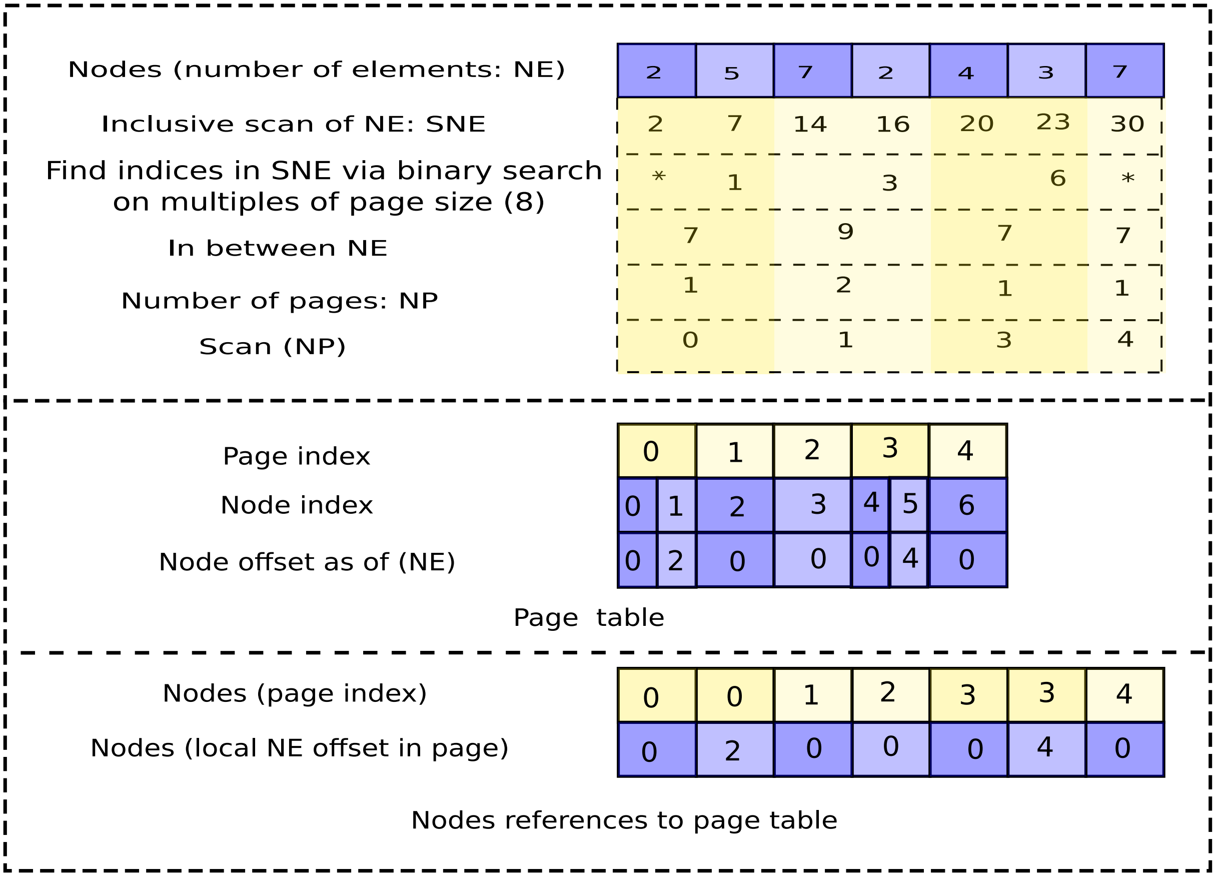


Figure 6.2: Dynamic page table structure. Given a set of nodes, each containing a different number of elements, we compute splits at the nodes so that each consecutive set of node elements can fit into a single page. In this illustrative example, we assume that each page can hold up to 8 elements.

In our pipeline, we build a two-level indexing structure for the visualization of large particle sets. The first few levels are built until each leaf node contains a relatively small number of particles, which can be managed by a single workgroup to construct a bottom-level treelet. We assume that the top-level hierarchy is small enough to reside in GPU memory throughout

the entire rendering process. However, this is not the case for the bottom-level hierarchy. The bottom-level hierarchy is neither homogeneous nor split into fixed-size blocks, as each emitted treelet has a different number of nodes and particles. To optimize GPU memory usage and reduce bandwidth consumption for the bottom-level hierarchy, we compact several consecutive treelets into a single page block, rather than mapping each treelet onto a separate page with empty padding in the remaining space.

As illustrated in Figure 6.2, we first count the number of nodes in each treelet and then apply a binary search over page-size multiples in the prefix sum of node counts. These marks provide a rough estimate of where treelets can be compacted into a single page. Such splits ensure that intermediate data blocks fit into at most two pages in the page table. In the next step, we count the intermediate pages and, using a prefix sum of this count, fill the page table with entries storing each consecutive set of treelets in one page. At this stage, we also construct an array mapping treelet indices to page indices, along with an offset array that stores, for each treelet, the position of its data relative to the start of its page.

6.4 Out-of-Core Chunk-Based Merging for Large Array Sorting

Sorting is an elementary step in many hierarchical construction algorithms [GPM11, LGS*09]. By sorting input primitives according to their spatial location, building indexing structures is simplified through recursive splitting until a termination criterion is met, such as reaching a small number of primitives in a split, the spatial extent becoming smaller than a predefined threshold, or a maximum recursion depth being reached. On the GPU, due to limited memory resources, index-based sorting is typically employed. This involves indirect referencing using sorted indices to load data in an ordered sequence (see Section 2.4).

Additionally, sorting is a vital component for correct 3D rendering and semi-transparent surface blending. Techniques such as order-independent transparency (OIT) [YHGT10] rely heavily on pixel-based sorting to correctly display semi-transparent surfaces. In Section 6.7, we adopt object-based sorting based on screen depth, as we use OpenGL rasterization to display particle sets. This allows us to leverage ordered rasterization to achieve a correct screen-space ordering of particles.

During our experiments, we found that the NVIDIA CUB API [NL] has a limitation on the number of keys it can sort. Thus, we use a modified version of the GPU-based OOC merge sorting strategy presented in [ZNA15] to sort the particles before hierarchy construction. We start by partitioning the particles into smaller chunks of a size suitable for the CUB library sorting. After sorting each chunk separately, we recursively merge neighboring pairs of chunks into progressively larger sorted chunks until only a single sorted chunk remains. This chunk merging process is a variant of the classical merge sort algorithm and follows a divide-and-conquer strategy to sort the array of elements.

In this section, we explain the main details of the proposed OOC merge sorting process, which uses dynamic scheduling to partition keys into manageable units that fit within the allowed memory budget for the merging process. More implementation details, including code snippets, are provided in Appendix B.

As a first step, we partition the input elements into chunks of a maximum size. In our experiments, we use a chunk size of 32M keys. Each chunk is loaded independently onto the GPU, where it is sorted using the NVIDIA CUB API [NL]. After sorting all chunks, we perform $\lceil \log_2(n) \rceil$ merge rounds, where n denotes the number of chunks. In each merge round, every consecutive pair of chunks is merged. This process continues until the final round, where a single chunk remains, containing all elements in sorted order.

Merging of keys is done in a ping-pong fashion, where input keys come from the input buffers and output keys are written to the output buffer. These buffers are swapped after each merge round. At the beginning, we compute the number of merge rounds based on the number of keys, the chunk size, and the input and output buffers. Then, we iterate through several merge rounds, starting with the number of chunks from the initial sorting step and the initial chunk size. During each merge round, we take chunks in pairs and merge them into larger chunks. After sorting all chunks in a given round, we double the chunk size and halve the number of chunks.

6.4.1 Dynamic Scheduling for Out-of-Core Chunk Merging

Given two sorted chunks that need to be merged into a larger, sorted one, we adaptively split the input chunks into partitions that can fit into the available GPU budget for the merge process. This process is shown in Figure 6.3. We begin by selecting keys from both arrays at fixed positions. We refer to these selected keys as splitters, and the region between two

consecutive splitters as a block. For each splitter, we keep track of its corresponding rank in the original array where it was selected. Next, we merge the splitters into a single sorted array. We also maintain an array of flags indicating whether each key was selected from the first or second input array, and compute the prefix sum scan of these flags. The splitters, their original ranks, the corresponding flags, and the prefix sum scan of the flags are used as memory-bound guidance for identifying the partial regions of the original arrays to be merged together into the sorted array.

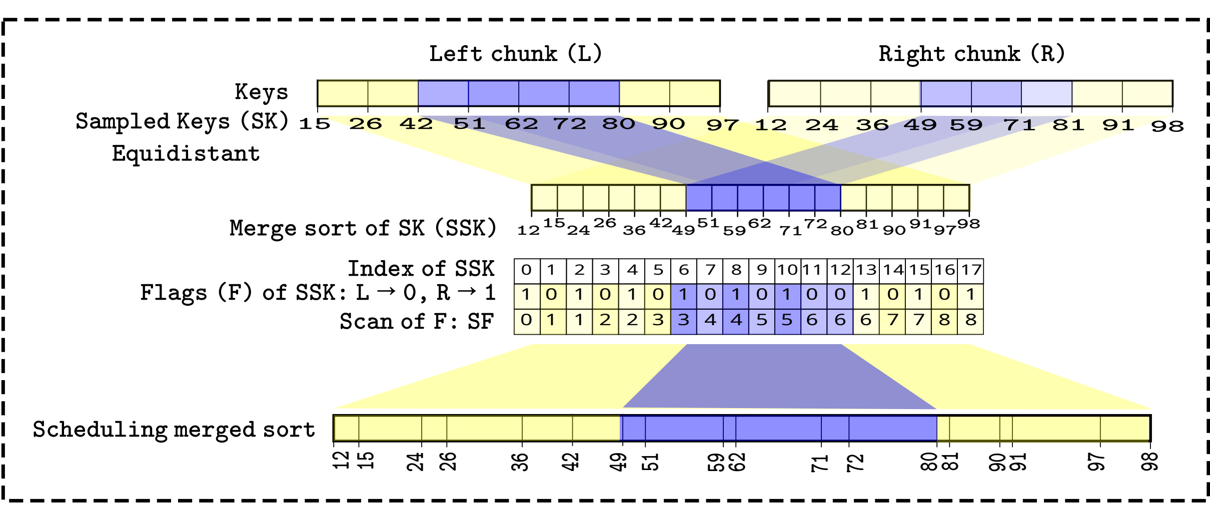


Figure 6.3: Merging two sorted arrays using sampling merge sort. The sorted splitters in the middle provide guidance for identifying block boundaries in the source arrays.

Given any two bounds in the sorted splitters array, we have n_l and n_r splitters from the first and second arrays, respectively. In the merge step, we need to load $n_l - 1$ and $n_r - 1$ blocks from the corresponding arrays, and additionally at most two blocks around the bounding splitters. These two blocks come from the other array in which the corresponding bounding splitter was not selected. Using the prefix-scan of the flags array, we determine the blocks in each array that need to be loaded for the next round, including the blocks around the boundary splitters.

We select a partition of the splitters array according to the available memory budget and load the corresponding data blocks from the left and right arrays, merging all elements into the output array. For each input block, we use the prefix sum array to find the corresponding region in the other array and perform a binary search within a limited range of keys. We then

place the key into the output array at an index equal to its original rank in its array plus its rank in the other array.

Our implementation presents a stable sorting algorithm, which ensures that keys with the same value preserve their order from the input sequence [PT92]. We employ binary search to find a key's rank in the opposite chunk of sorted keys. We use two slightly modified versions of the standard binary search: one for keys from the left chunk and one for keys from the right chunk. Each of these functions returns the key's rank relative to the start of the search range. Each key is stored in the output chunk at a position equal to the sum of its rank in the left chunk and its rank in the right chunk.

6.5 Two-Level Indexing Structure Construction for Level-of-Detail Particle Sets

Inspired by modern GPU ray tracing APIs such as DirectX Raytracing (DXR) [Mic18], NVIDIA's OptiX framework in CUDA [PBD*10], and Vulkan's ray tracing extension [Gro20], we build a two-level indexing structure for the visualization of large particle sets. This approach has also been employed for constructing two-level hierarchies for large point cloud data [SOW20].

As a preparation step, we extract the Morton code [Mor66] for each particle and sort particle indices accordingly. We build the hierarchy over the sorted particle sequence in two main steps. In the first step, the top tree levels are constructed in breadth-first order until each leaf node contains fewer particles than a certain threshold (i.e., 10K particles). In the following step, each CUDA workgroup fetches a leaf node from the first step and constructs a treelet until the end criteria for node splitting are met.

We use OOC paging for loading particle data that are too large to fit into GPU memory. After hierarchy emission, we relocate the particle data from input arrays alongside their corresponding treelet hierarchy. The two-level hierarchy emission is an efficient way to achieve better caching during bottom-level hierarchy construction. Additionally, by relocating the particle data, we aim for improved caching during rendering.

Figure 6.4 shows the main building block of the tree construction algorithm. In the following, we explain in detail the main processing steps of the indexing structure construc-

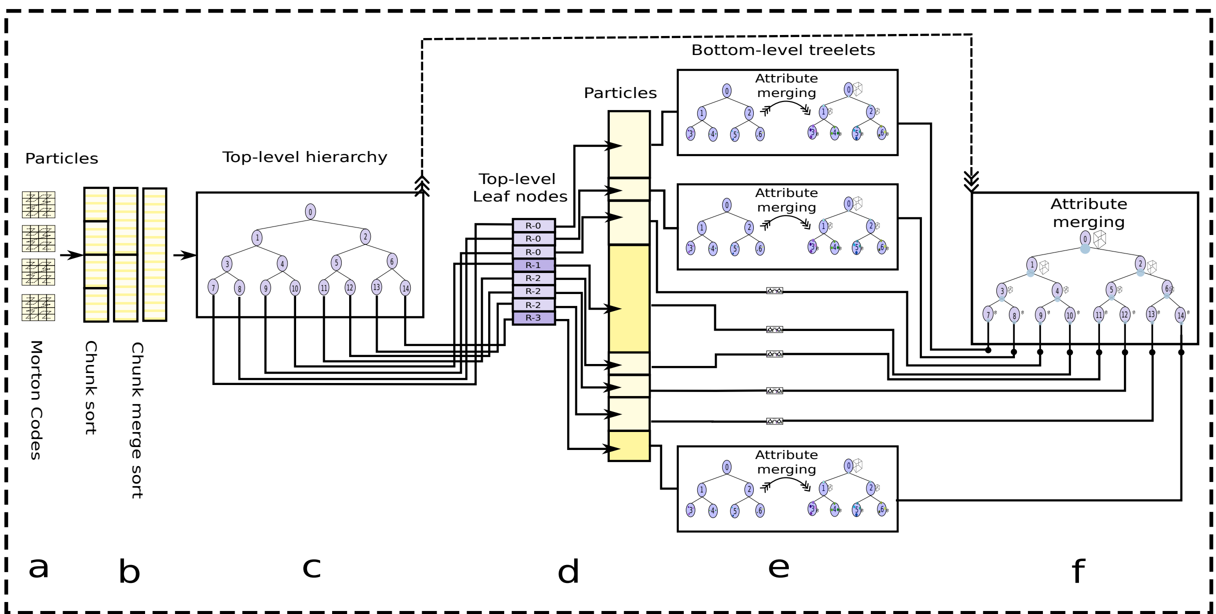


Figure 6.4: Main building blocks of our tree construction algorithm: (a) generating Morton codes for the input particles; (b) sorting the particles using a chunk-based merge sort; (c) constructing the top-level hierarchy; (d) scheduling large leaf nodes from the top-level hierarchy to construct bottom-level hierarchies and merging particle attributes by adaptively loading subsets of nodes and their particles into the GPU in multiple rounds; (e) propagating particle attributes in the bottom-level treelets, with attribute merging performed at each level; and (f) propagating particle attributes from the bottom hierarchies back to the top-level hierarchy, with attribute merging performed at each level.

tion. Firstly, in Section 6.5.1, we describe a scheduling mechanism for processing tree data. Next, in Section 6.5.2, we present a two-level hierarchy construction algorithm for particle data. In Section 6.6, we explain how to construct LoD of particle attributes using the constructed indexing structure.

6.5.1 Dynamic Scheduling for Node Processing

In our tree construction pipeline, tree nodes and their contained geometry are stored in two separate arrays. Each node references a contiguous set of particles, marked by a start index and the number of particles. The number of particles contained within each node varies, and either one or both of the arrays may not fit entirely within the GPU memory budget. When processing a sequence of nodes and their contained geometry, a challenge arises in partitioning the nodes array into chunks that fit precisely within the available GPU memory budget. Uniformly splitting the nodes array is not feasible, as the storage requirements of the contained geometry vary between nodes, since each node may reference a different number

of particles. However, we have developed a method to partition the nodes into chunks that fit optimally within the available GPU memory budget using binary search.

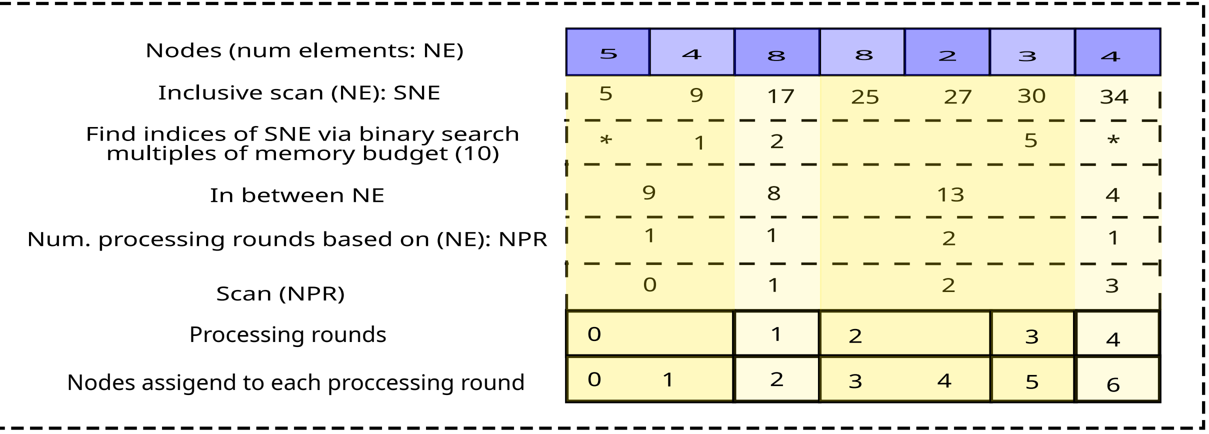


Figure 6.5: We use binary search in the prefix sum of the number of particles in each node to find the partitions of nodes that fit within the GPU memory budget. In this illustrative example, we assume that the memory budget can accommodate up to 10 geometry elements, and each node does not contain more than 8 geometry elements.

As shown in Figure 6.5, we start by filling an array with the number of particles in each node and performing a prefix sum scan of this array. Given the available GPU memory budget, we launch a kernel to find positions in the prefix sum array where the number of particles is as close as possible to multiples of this budget. Then, we count the number of particles between each pair of consecutive marks found in the binary search step. Based on the number of particles between these marks, we determine how many processing rounds are needed to load these particle blocks. By using the prefix scan of the number of required processing rounds, we iterate several times and call the processing kernel. For each round, we load a subset of nodes and their corresponding particles.

6.5.2 Two-Level Hierarchy Construction

As we mentioned earlier, our CUDA-based tree construction method creates the tree hierarchy in two main steps. In the first step, we build the top tree levels until reaching leaf nodes that contain a relatively small number of particles, which can be processed by a single workgroup on the GPU. In the next step, each workgroup processes a leaf node from the first step and emits the corresponding treelet locally on the GPU.

Top-Level Hierarchy Construction We build the hierarchy in a breadth-first order and use binary search to find bit splits among Morton codes [GPM11]. To accelerate the construction process, we select Morton code splitters at fixed locations, similar to what we did during the merge sort (see Figure 6.6 bottom). The splitter array is small enough to reside in the GPU core during the construction process and provides good indicators of OOC blocks that contain the actual node splits.

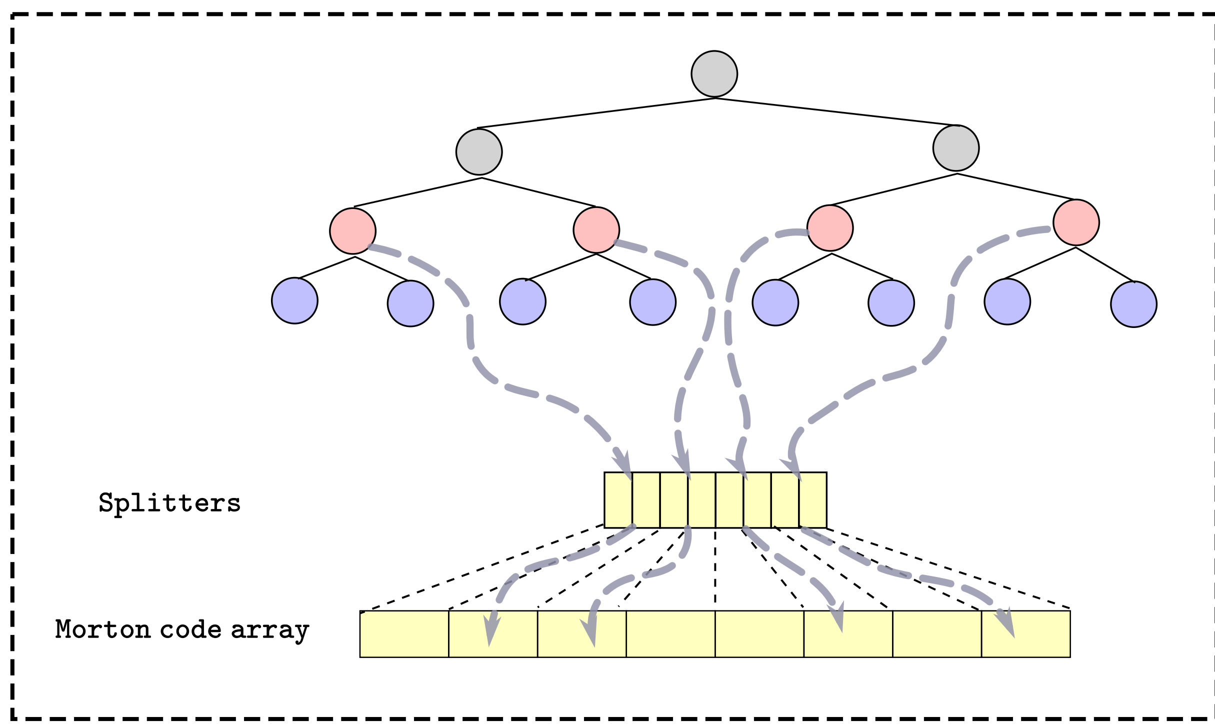


Figure 6.6: For any red node to be split, we search in the Morton code splitters array to find the relevant Morton code block that contains the exact node split. Once the relevant Morton code block is loaded into the GPU, we locate the exact split point to divide the parent red node into two child blue nodes.

We use two running queues for input and output node indices [LGS*09] for each tree level construction. Each thread is responsible for splitting a specific node and uses binary search to find the first bit difference between the first and last Morton code of the node within the splitters array. This split is used to locate the block where the bit difference occurs in the original sorted Morton codes array. Upon identifying this block, we issue a page request for it if it resides OOC memory. Once the page is loaded, we use binary search again within the requested block to locate the exact bit split index that divides the node into two child nodes (see Figure 6.6).

Bottom-Level Hierarchy Construction Starting with a list of leaf nodes created in the previous step, we process each leaf node using a single workgroup and emit the corresponding treelet locally. Since the GPU memory is not sufficient to process all top leaf nodes simultaneously, we conservatively estimate the required memory for each leaf node and its resulting treelet. We then initiate several kernel launches to process blocks of nodes from chunks of the roots array.

Our conservative estimation of treelet geometry comes from the fact that a binary tree with a maximum level l would contain at most $2^{(l+1)} - 1$ nodes, and a tree with n_l leaf nodes would contain at most $n_l - 1$ internal nodes. A conservative build method can assume that each leaf node contains only one particle, or that the node splitting process stops at a maximum level. We use the latter criterion and stop splitting whenever a node contains τ particles, where $\tau > 1$. Based on this, we estimate the memory budget for each leaf node and determine how many leaf nodes can fit into the GPU memory budget. We then initiate several iterations: in each iteration, we load a set of leaf nodes onto the GPU, and each thread emits the corresponding treelet locally. After treelet construction, we relocate the relevant geometry and corresponding particle data for improved caching and data locality.

6.6 Level-of-Detail Construction

Multi-resolution hierarchies built on top of indexing structures such as grids, kd-trees, and octrees have been extensively used in visualization frameworks [HBJP12]. Upper levels of these hierarchies represent coarser approximations of the lower, more detailed levels and the data they contain. Due to limited computing resources and memory constraints, coarser levels are often used to adaptively display proxies of the original input upon request. The appropriate level of detail is selected based on the viewing distance, as detailed in Section 6.7. In practice, multi-resolution structures and LoD rendering have proven effective, and in many scenarios, users perceive little to no difference between the original data and its coarser representations [SKKW23, SKW22, SOW20, jedban13, DVS03].

We construct the multi-resolution hierarchy over the indexing structures in a bottom-up manner. Initially, the leaf nodes store the raw particles, while internal nodes reference two child nodes. Data summaries are propagated upward from the leaf nodes through the internal nodes, across all levels of the tree, until reaching the root (see Figure 6.7). As we compute attribute summaries level by level, we maintain indexing data for nodes at each tree level,

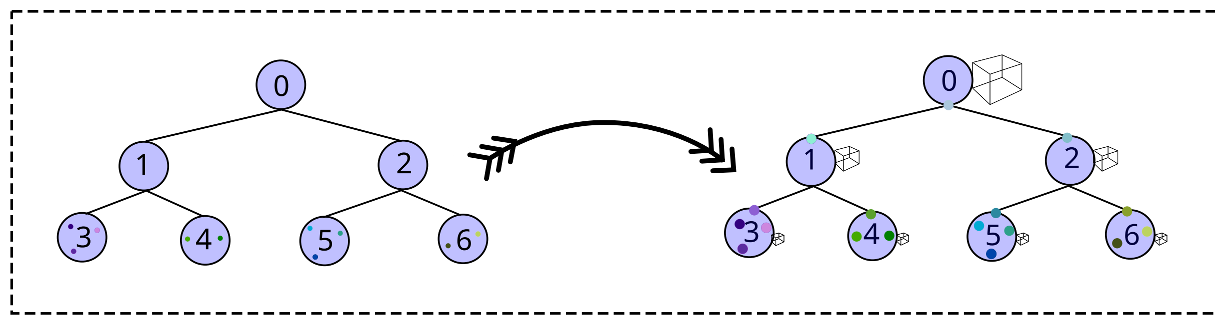


Figure 6.7: Bottom-up LoD construction. Left: Initial treelet where particles reside at leaf nodes, and each internal node references two child nodes. Right: After LoD construction, starting from the leaf nodes, we merge particle attributes and bounding boxes upwards until we reach the root node.

which remains local to each treelet in the lower hierarchy. Each treelet is assigned to a work-group, and parallel threads process nodes in a bottom-up fashion. Each thread fetches a node and computes the attribute summary of its child nodes or enclosed particles. Finally, the attribute summary of each treelet's root node is propagated to the corresponding leaf node in the top-level hierarchy.

In our experiments, we use the potential energy as a representative particle attribute and compute the internal node summary as the arithmetic average of the contained particles.

Additionally, we compute conservative bounding boxes around the particles of the leaf nodes and propagate these bounding boxes upward. Each node's bounding box serves as a geometric proxy for the enclosed particles during LoD selection and rasterization.

We employ OOC paging to load nodes and their associated particle data during propagation. Once the multi-resolution hierarchy is constructed for all treelets, we propagate both attribute summaries and bounding boxes similarly through the upper hierarchy. It is worth noting that the particle attribute summary is domain-dependent; in our implementation, this computation is designed to be interchangeable and pluggable.

6.7 Results and Discussion

In this section, we report the implementation details of our tree construction algorithm and LoD rendering. We demonstrate our technique on several datasets and present key design metrics, including memory consumption across the main processing stages and timing measurements.

Evaluation Setting Our pipeline is implemented using C++, NVIDIA CUDA 12.5, and OpenGL 4.5. We use C++ for loading particles, managing GPU resources, and saving indexing structures. Tree construction, LoD generation, frustum culling, and LoD selection are implemented in CUDA, while particle rendering is handled using OpenGL. We extensively utilize the NVIDIA CUB API [NL] for parallel primitives wherever applicable. All experiments were conducted on a PC running Ubuntu 24.04, equipped with an Intel Core i7 processor, 32 GB of RAM, and a NVIDIA GTX 1080 Ti GPU with 11 GB of VRAM.

We pre-allocate conservative GPU memory blocks for intermediate and persistent data structures, releasing them immediately when no longer required. In some cases, the memory block size is derived from a preceding step. For instance, after emitting bottom-level hierarchy treelets, all node-related structures are bounded by the treelet size, and particle-related structures are similarly constrained by the number of particles within the treelet.

Dynamic memory requirement estimation for an array of elements follows a classical two-step process: first, computing the required size per element in parallel; then, applying a parallel prefix scan to determine the memory offsets. The total memory demand is obtained by summing the final entries of the count array and its prefix scan [SHZO07].

Hierarchy emission proceeds in a breadth-first manner. For top-level hierarchy emission, we use a global node queue [LGS*09], while for bottom-level hierarchy treelets, a shared-memory queue is used within each thread-block group. The queue size at each step is bounded by the number of leaf nodes at the current level, with each node emitting zero or two children [CLRS09]. We apply list compaction [SHZO07] for node emission in the top-level hierarchy and employ an atomic counter to manage queue interactions during bottom-level construction [AL09].

The constructed tree is serialized to external storage using the HDF5 [The97] file format, providing efficient data access and persistence. Each tree structure includes a local index that, for each level, records the offset and size of the corresponding nodes and particles, thereby enabling fast and structured access during rendering.

Datasets We evaluated our pipeline on the Illustris dataset [VGS*14], a large-scale cosmological simulation project developed by physicists and astrophysicists to model the evolution of the universe. The simulation consists of three separate runs, each comprising 135 distinct time steps. Each snapshot contains millions to billions of particles, where each particle is

represented by a 3D coordinate along with several physical attributes such as mass, velocity, and density.

In our setup, each particle is defined by its 3D coordinate, radius, and a single floating-point attribute. For our evaluation, we use the potential energy attribute provided in the Illustris dataset to simulate various visual and computational tasks.

To assess the scalability of our algorithm, we present results on two representative subsets of the Illustris-3 particle data, containing 20% and 50% of the original particle set, respectively.

Execution times were measured on the GPU using two CUDA events placed before and after the kernel call, with the reported value taken as the median of five runs.

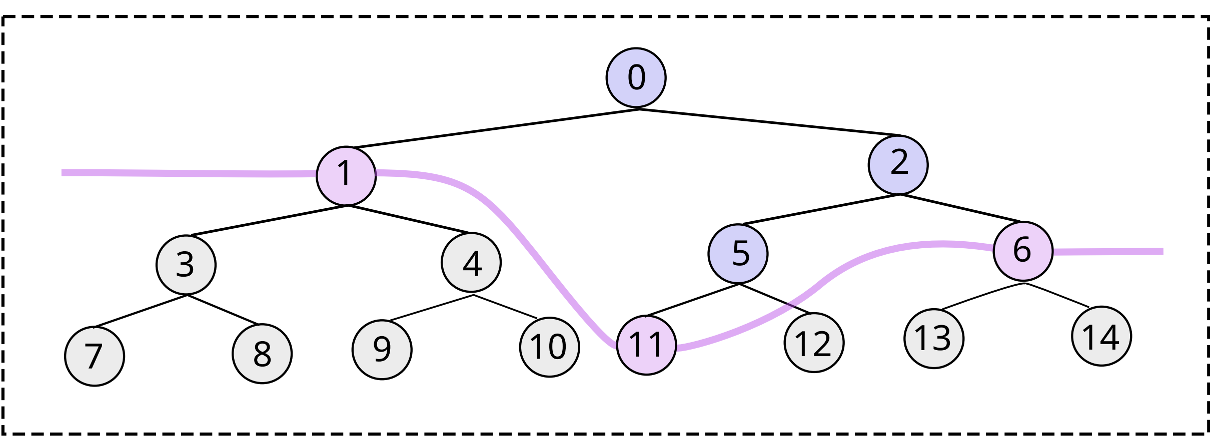


Figure 6.8: Frustum culling and LoD selection. The selected nodes forming the tree cut are shown in red. These nodes have passed the frustum check and represent the most appropriate LoD relative to the viewer's position. Blue nodes indicate already visited tree nodes, while grey nodes are either outside the viewing frustum or have a projected size below a defined threshold.

Particles Display and Rendering We use point sprites [Wol18] for particle rendering. Each particle is expanded into a screen-oriented quad centered at its position in the geometry shader, where texture coordinates are assigned to the quad vertices. In the fragment shader, the particle's color and opacity are determined using a colormap applied to a selected particle attribute. To enhance particle display, the fragment opacity is modulated using a precomputed Gaussian texture, producing a spike-shaped appearance for each particle.

Adaptive Level-of-Detail Rendering As previously described, the top-level hierarchy is maintained in GPU memory. Depending on the current viewing parameters, the corresponding bottom-level hierarchies are selectively loaded into the GPU to support adaptive rendering (see Figure 6.8). In Appendix C, we provide implementation details for LoD selection and frustum culling.

The tree hierarchy is traversed using a breadth-first strategy to perform frustum culling and LoD selection. Our traversal algorithm can be viewed as an extended and parallelized version of the rendering approach proposed by Rusinkiewicz and Levoy [RL00]. During traversal, each node's bounding box is projected onto the screen. If the projected size falls below a predefined threshold, the node is considered suitable for display and is added to a separate list for rendering in the subsequent stage.

If not, we test the node's bounding box against the viewing frustum. If the bounding box lies entirely within the frustum, we skip further frustum checks for its children but continue evaluating them against the screen-projection threshold to determine an appropriate tree cut. Conversely, if the node only partially intersects the frustum, its children are queued for both frustum and projection evaluations in the next traversal round.

Similar to tree construction, we identify the tree cut in two main rounds. In the first round, we traverse the top-level hierarchy and select leaf nodes that satisfy both the frustum and LoD criteria. This stage employs global queues to manage the tree traversal efficiently. In the second round, we traverse the bottom-level hierarchies, utilizing OOC paging to adaptively load treelets on demand. Each treelet is assigned to a separate workgroup, where LoD selection is performed in parallel.

To coordinate traversal within each workgroup, we use a shared memory atomic variable to track the traversal queue size. Nodes selected as part of the treelet cut are written to a preallocated array in global memory. A shared memory variable within each workgroup maintains the local cut size. Once a treelet is processed, an atomic operation on a global memory variable records the number of nodes selected for that treelet's cut [AL09]. The value returned by this atomic update is also used to compute the offset in a global array, enabling compact storage of all cut nodes across treelets.

After the traversal of each treelet, a separate kernel is launched to collect the selected nodes into global memory. Each treelet is assigned to an independent workgroup, and the previously

computed offset is used to correctly place its cut nodes in the global array. During this process, if a leaf node is reached, we record its particle range relative to the treelet.

Once all treelets have been processed, a prefix scan over the leaf-node particle counts is performed to expand particle indices into a dedicated array. We then use OOC paging to efficiently load particle data and their associated attributes. The collected internal nodes and leaf-node particles are finally sorted based on their screen space depth to ensure correct back-to-front rendering order when using the rasterizer.

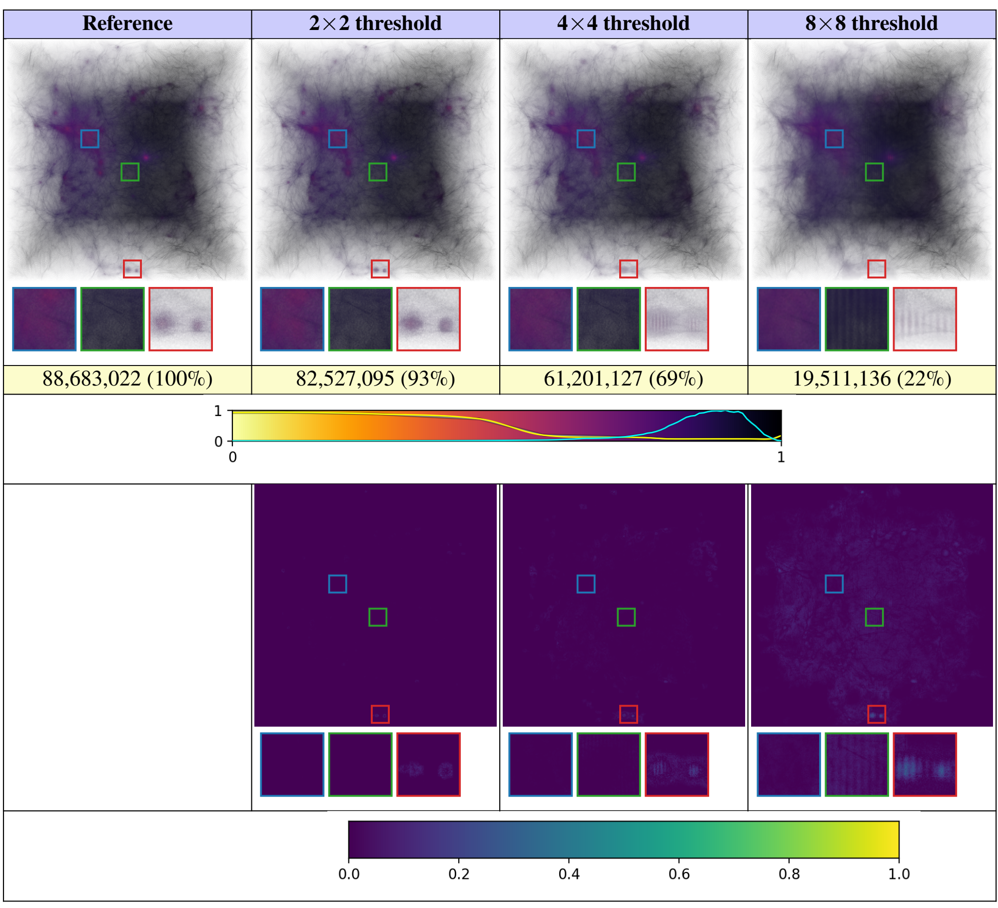


Figure 6.9: LoD rendering of snapshot 100 from the Illustris-3 dataset. The first column shows the full reference rendering, while the next three columns correspond to screen space thresholds of 2×2, 4×4, and 8×8 pixels. The cyan curve overlaid on the colormap represents the particle distribution within the dataset, while the yellow curve depicts the transfer function profile that controls opacity modulation. The bottom row shows the corresponding L1-difference images using the Viridis colormap. All images are rendered at a resolution of 1024^2 pixels.

Screen Space Projection Threshold Screen space area is a widely used metric for LoD control. We compare the screen projection of a node's Axis Aligned Bounding Box (AABB) against a predefined screen space threshold. If the node's projected area exceeds this threshold, we traverse its children; otherwise, the node is used as a LoD representative for rendering. To compute the screen space size of an AABB, we transform each of its bounding box vertices using the model-view-projection matrix and measure the area of the resulting 2D projected bounding box.

Figure 6.9 demonstrates our LoD rendering of the *Illustris-3* dataset compared to reference rendering, using various screen space selection thresholds. Our results exhibit visually plausible quality with minimal error at small pixel thresholds, and only minor noticeable deviations in certain cluttered regions when larger thresholds are applied.

Figure 6.10 shows renderings at different viewing distances using varying screen space thresholds. The results indicate that a threshold between 2×2 and 4×4 pixels yields visually plausible LoD representations at moderate cost. Increasing the threshold reduces the number of rendered particles; however, excessively large values may result in visibly blurry outputs, as demonstrated in Figure 6.10 right.

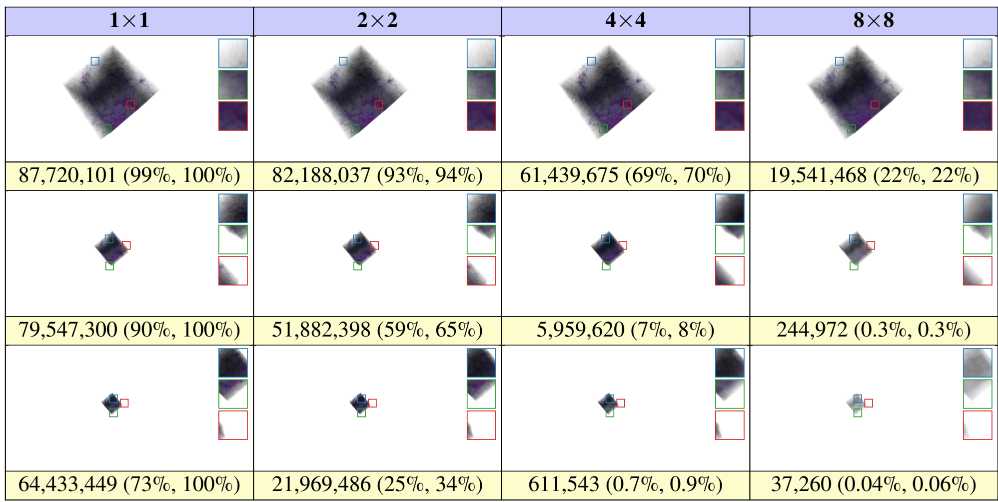


Figure 6.10: LoD rendering of snapshot 100 from the Illustris-3 dataset using different screen space projection thresholds, indicated at the top of each column. From top to bottom, the viewing distance increases. The numbers below each image show the total number of displayed particles (or particle proxies). The first percentage indicates the ratio relative to the full reference rendering of 88,683,022 particles, while the second percentage indicates the ratio relative to the number of particles in the first column (i.e., with 1×1 pixel threshold) of the same row. All images are rendered at a resolution of 1920×1080 pixels.

Furthermore, the number of selected particles scales effectively with the LoD selection strategy. As the viewing distance increases, coarser representations are chosen, significantly lowering the rendering cost while preserving the overall structural integrity of the dataset. These findings validate the effectiveness of screen space LoD control in balancing visual fidelity and performance.

OOC Sorting We employ an OOC merge sorting strategy for both tree construction and back-to-front particle rasterization. As previously described, we begin by dividing the input keys into chunks of 32 million elements and perform in-core GPU sorting on each chunk independently. Next, we recursively merge each pair of sorted chunks until all keys are fully sorted. After each merge step, main memory serves as the resident platform. This is implemented using input-output buffer ping-ponging at each merge round.

Table 6.1 presents the sorting time for the *Illustris-3* snapshot containing 88 million keys. We report the timing of each primary step in the merge process, including sampling equidistant splitters from the input keys, sorting the splitters, and merging the keys using the sorted splitters as guides for OOC key loading.

Phase	Phase details	In-core Time (ms)	OOC Time (ms)
Chunk sorting	C_0^0 (32M) / C_1^0 (32M) / C_2^0 (20.6M)	28 / 28 / 18	197 / 197 / 126
Merging rounds (2)			
Round ^{0}			460
Merge chunks	C_0^0 (32M) $\cup$ C_1^0 (32M) $\rightarrow$ C_0^1 (64M)		350
Sample splitters			87
Sort splitters		0.16	
Merge elements	Dynamic scheduling (see Figure 6.3)		$88 + 88 + 88 + 88$
Copy chunk	C_2^0 (20.6M) $\rightarrow$ C_1^1 (20.6M) (CPU $\rightarrow$ CPU)		23
Round ^{1}			576
Merge chunkss	C_0^1 (64M) $\cup$ C_1^1 (20.6M) $\rightarrow$ C_0^2 (84.6M)		461
Sample splitters			115
Sort splitters		0.16	
Merge elements	Dynamic scheduling (see Figure 6.3)		$87 + 87 + 87 + 87 + 87 + 25$

Table 6.1: OOC sorting is performed on 88 million Morton codes, each 64 bits in length. Since the keys are divided into three chunks, two merge rounds are required to sort the entire key sequence. Please note that for the first row, the reported in-core time is included in the OOC time. In this table, each chunk C_i^r is marked by the merge round r and chunk index i in the respective merge round.

Dataset	Size	Morton code	Sorting	Top-level hierarchy		Bottom-level hierarchy	
	Number of particles	Time* (ms)	Time* (ms)	NL / NN / NLN	SMC* / CT* / AMT	NN	Time* (ms)
Illustris-3 (S 100) (20%)	17,735,402	49	198	22 / 5,809 / 2,905	23 / 28 / 0.304	1,647,465	895
Illustris-3 (S 100) (50%)	44,344,787	116	554	24 / 14,479 / 7,240	57 / 68 / 0.425	4,150,620	2,479
Illustris-3 (S 100)	88,683,022	229	1,506	25 / 28,985 / 14,493	114 / 133 / 0.629	8,391,553	6,036
SS : Snapshot				NL : Number of levels NN : Number of nodes NLN : Number of leaf nodes	SMC : Sample Morton code CT : Construction time AMT : Attribute merging time	NN : Number of nodes	

Table 6.2: Indexing structure details and build time for the main construction stages. Numbers marked with * indicate the inclusion of OOC paging on the GPU and are expanded with further details in Table 6.3.

Construction Time Table 6.2 summarizes the construction time of the main indexing structure stages. We report results for the full *Illustris-3* dataset, as well as for 20% and 50% pre-sampled subsets of the same dataset (see Figure 6.11). The results indicate that our construction algorithm scales well with the input particle set size. However, OOC paging remains a performance bottleneck, primarily due to the limited memory bandwidth and transfer latency between the CPU and GPU devices—a known issue in hybrid memory architectures.

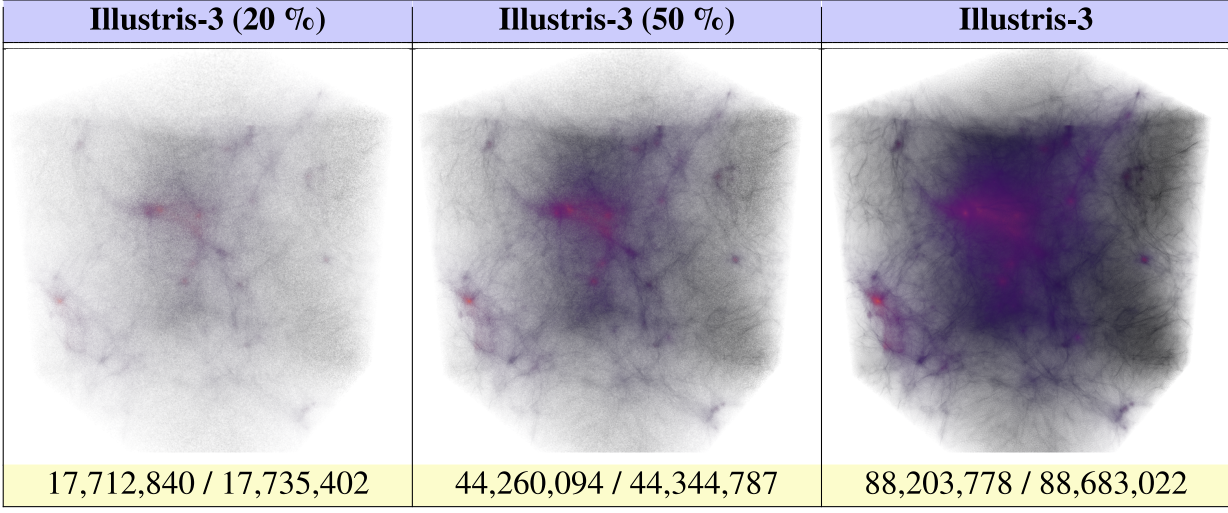


Figure 6.11: Rendering of snapshot 100 from the Illustris-3 dataset (right), shown alongside 50% (middle) and 20% (left) uniformly sampled particles, using a 1×1 pixel threshold for the LoD cut. Below each image, we list the number of selected particles relative to the total number of particles in the dataset. All images are rendered at a resolution of 1024^2 pixels.

To investigate paging performance, we report the OOC paging parameters in Table 6.3. We distinguish three main phases: loading particles for Morton code sorting, building the top-level hierarchy, and constructing the bottom-level treelets.

Dataset	Morton code RSS: 2GB / ES: 28B PS / NPL	Top-level hierarchy RSS: 512M / ES: 16B / PS: 4K NPL	Bottom-level hierarchy RSS: 512M / ES: 36B / PS: 4K ES / NPL
Illustris-3 (20%)	16M / 0.231	2.9K[1]	36B (8) / 4.6K[1] 36B (8) / 4.6K[2] 36B (20) / 8.5K[3]
Illustris-3 (50%)	42M / 0.578	7.24K[1]	36B (8) / 12.9K[1] 36B (8) / 12.9K[2] 36B (20) / 31.1K[3]
Illustris-3	73M / 1.156	14.45K[1]	36B (8) / 28.47K[1] 36B (8) / 28.47K[2] 36B (20) / 97.06K[3]
SS : Snapshot		[1]: Paging of sorted Morton code	[1]: Paging of sorted indices [2]: Paging of Morton code [3]: Paging of particle data (Position, Radius, Attributes)

Abbreviation	Meaning	Notes
RSS	Resident Set Size	Physical memory used
ES	Element Size	Depends on structure layout
PS	Page Size	Typically 4k entries
NPL	Number of Page Loads	Pages loaded into memory

Table 6.3: Out-of-core paging statistics of particle data during different stages of tree construction. In the second column, Morton codes are generated by chunking particle positions directly to GPU memory based on the allocated memory block size (i.e., page size).

Please note that during the bottom-level treelet emission, we perform three OOC paging rounds. First, we load the sorted index sequence for each treelet node. Second, we load the corresponding Morton code sequence. Finally, using these indices, we fetch the scattered particle data from the original particle set. This multi-stage loading process is necessary because we employ key–value sorting without physically relocating the input particle data after sorting. During this stage, we also merge particle attributes and compute bounding boxes for each treelet in a bottom-up manner.

Rendering Performance Figure 6.12 shows the rendering of the datasets from both near and far perspectives. We report the hierarchy traversal times in Table 6.4, which include frustum culling and appropriate LoD selection.

Subsequently, using OOC paging and dynamic scheduling, we adaptively load treelets into the GPU and assign each loaded treelet to a workgroup for traversal.

Once this stage is complete, we obtain a list of particles and their associated attributes relevant to the current view. A per-particle depth value is computed as a sorting key and key-value sorting is applied. This sorting process also utilizes OOC paging to load particle positions,

Illustris-3 (20%)	Illustris-3 (50%)	Illustris-3 (100%)
16,882,656 / 17,735,402 (95%)	39,477,986 / 44,344,787 (89%)	72,153,788 / 88,683,022 (81%)
4,614,169 / 17,735,402 (26%)	5,684,528 / 44,344,787 (13%)	4,933,109 / 88,683,022 (6%)

Figure 6.12: Rendering of near and far views of snapshot 100 from the Illustris-3 dataset (right), shown together with subsampled versions containing 50% (middle) and 20% (left) of the particles. A 4×4 pixel resolution is used for the LoD cut threshold, and all images are rendered at 1024×1024 pixels. The values reported below each image indicate the number of selected particles relative to the total particle count in the dataset, with the corresponding percentages given in parentheses.

as the selected particles may not entirely fit into GPU memory. Each depth key is computed by transforming the particle position with the model-view matrix.

Dataset	Time (ms)				Output
	Top hierarchy traversal	Bottom hierarchy traversal *	Particle key-value sorting *	Particle data sorting *	Referencing Figure 6.12
Illustris-3 (20%) near view	6	297	199	238	Top left
Illustris-3 (20%) far view	0.9	253	55	100	Bottom left
Illustris-3 (50%) near view	6.8	576	824	549	Top middle
Illustris-3 (50%) far view	0.9	388	68	112	Bottom middle
Illustris-3 near view	8.7	1001	1860	1052	Top right
Illustris-3 far view	1.0	563	59	106	Bottom right

Table 6.4: Time measurements of LoD rendering using near and far views of several datasets with a 4×4 pixel threshold. Numbers marked with * indicate the inclusion of OOC paging on the GPU.

After sorting the depth keys, particles are relocated according to the sorted sequence using OOC paging. As a final step, the sorted particles are streamed to the rasterizer to ensure correct back-to-front rendering.

6.8 Conclusions and Outlook

In this chapter, we presented a GPU-based, two-level hierarchical approach for visualizing large particle datasets. Our indexing structure incorporates a dynamic scheduling strategy for out-of-core (OOC) node emission and particle distribution, enabling scalable and efficient rendering. We demonstrated that a two-level hierarchy is sufficient for the current requirements of our visualization pipeline.

While our implementation employs a straightforward paging strategy, there remains considerable room for optimization. One potential enhancement involves using an atomic variable as a voting mechanism for page requests, thereby eliminating the need to inspect each request individually. Another promising direction is the collection of distinct page requests directly on the GPU, allowing for the preparation of paging scenarios without CPU intervention. Additionally, selectively assigning page availability checks to specific threads within a workgroup could help reduce redundant computations and improve overall efficiency.

A major limitation of our algorithm is the reliance on main memory as the central host between all GPU processing stages. This design choice was made to ensure that data produced by GPU stages can be serialized without exceeding available memory limits. However, in certain scenarios, this main memory platform may become a bottleneck depending on system state. We leave further optimization of memory management and pipeline decoupling to future development of our visualization framework.

Future work will focus on testing and optimizing the proposed sorting algorithm under more demanding conditions, particularly with larger arrays. This includes evaluating its scalability and performance as data size increases and analyzing how different GPU memory configurations affect throughput. Maximizing the utilization of available GPU memory will be a key objective, ensuring that our approach remains efficient in large-scale data processing scenarios.

Currently, our implementation performs particle selection and sorting for every view change. In cases of minor viewport updates, where the visible particle set remains largely unchanged, our merge sort strategy offers an efficient mechanism for incremental updates—handling both particle additions and removals. For even larger datasets, extending the hierarchy with additional levels, similar to the method proposed in [HBJP12], could further improve selection and rendering performance. Moreover, incorporating compression techniques such as those

described in [BP23] at the treelet level could help reduce memory bandwidth consumption and enhance streaming efficiency.

Looking forward, we aim to extend our dynamic OOC scheduling pipeline to additional domains, including geometry compression and ray tracing. We believe the techniques introduced in this work can provide a solid foundation for building scalable, GPU-driven systems aimed at large-scale scientific and visual data exploration.

7 Summary and Future Directions

In this chapter, we summarize our research contributions and highlight opportunities for future work related to the proposed solutions. Additionally, we provide an overview of broader future directions in the fields of scientific visualization, interactive and real-time rendering, and massively parallel processing.

7.1 Conclusions, Limitations, and Future Directions

This dissertation presents efficient visualization techniques tailored for large and complex scientific datasets, with a focus on vector fields and large particle simulations. All proposed methods are optimized for modern Graphics Processing Units (GPUs), enabling real-time interaction and scalable performance.

Our contributions include the following:

Interactive Geometry-Based Flow Visualization Technique We introduced a unified framework for geometry-based vector field visualization that supports a variety of glyph types and interpolated shapes, offering both flexibility and expressiveness. This method approximates the appearance of 3D LIC while maintaining high performance in Chapter 4.

As future work, combining our flexible extrusion framework with constructive solid geometry techniques [CSZ*25, Gha08, CGA] offers the potential to extend our system toward generalized implicit primitives, in the spirit of generalized tube primitives presented in [HWU*19]. This would enable fast and accurate rendering of complex tube-like structures (e.g., streamlines, fiber bundles) directly via GPU-based ray tracing. Furthermore, integrating our geometry-based flow visualization methods with ray tracing and physically based

rendering [PJH16] represents a promising direction for enhancing realism and scientific insight in high-fidelity visualization.

Interactive Opacity Optimization Technique We proposed a novel opacity optimization strategy based on moment-based signal reconstruction. This approach enhances visual clarity by reducing occlusion and supports fast, per-pixel opacity evaluation, even in dense scenes in Chapter 5. We also presented a hardware-accelerated filtering method that operates directly on the moment buffers and delivers results comparable to object space opacity smoothing.

We believe that exploring alternative variations of moment-based approximation methods for opacity functions [MKKP18] represents a promising direction for future work. Another avenue is to apply our opacity optimization technique to challenging scenarios involving sparse and geometrically tight regions, where we expect moment-based approaches to yield more plausible and effective approximations.

Interactive LoD Particle Rendering Techniques We presented an interactive rendering system for large-scale particle data using OOC GPU paging and level-of-detail (LoD) structures. This system enables interactive navigation and visualization of datasets containing millions of particles, demonstrated using the Illustris cosmological simulation in Chapter 6.

Several directions and extensions of our work are possible. First, extending our memory management techniques to support fully GPU-resident data structures is a promising direction. Coupling GPU-based rendering with CPU-side memory management could lead to hybrid memory systems capable of handling even larger datasets. Furthermore, integrating low-level memory management techniques—such as caching strategies and advanced replacement policies—is worth exploring [SMMO16, Sta14].

Ray tracing of particles is another promising direction for improving visualization realism and the understanding of large datasets [KMW*19, GWG*20].

In addition, exploring opacity optimization methods [GTG17, ZRPD20] in conjunction with our LoD data structures for large particle datasets and large-scale scenes offers a valuable research opportunity. Applying OOC paging techniques with asynchronous and stream-based processing on the GPU could further enhance responsiveness and system throughput. Finally, investigating advanced LoD-based rendering techniques [DVS03] presents a compelling op-

portunity for improving both visual quality and computational efficiency in large-scale scientific visualization.

Overall, our contributions advance the state of scientific visualization by enabling detailed, responsive, and scalable exploration of complex data on the GPU.

Appendices

A Implementation Details of Homogeneous Data Paging

In this and the next appendices, we present selected code snippets and key code sections for LoD indexing structures and LoD rendering. First, we describe the main code structures of OOC paging in Appendix A. Next, in Appendix B, we outline important code parts for out-of-core merging of key sorting. Finally, in Appendix C, we cover LoD frustum traversal and culling. In most cases, we focus on essential logical code sections, and for the sake of readability and reproducibility, we omit low-level code optimizations. Additionally, for conciseness, we use meaningful variable names that match the algorithm explanation and related figures.

In the following, we explain the basic implementation of the paging strategy and provide details on the key components of the paging process. The bit flags and masks used to facilitate understanding of the attached code snippets are listed in Table A.1, while the main variables and their types used in the code snippets below are listed in Table A.2.

Flag / Mask Name	Hex Value	Description
BIT_DATA_NOT_PROCESSED	0x80000000	Marks a data index as *not yet processed*. Used to defer processing until the page is in-core.
BIT_PAGE_LOADED	0x80000000	Indicates that a page is currently loaded into in-core memory.
BIT_PAGE_REQUEST	0x40000000	Marks that a page has been requested for loading in the next iteration.
BIT_WRITE_BACK	0x20000000	Indicates that the page has been modified and must be saved (written back) to host or backing storage.
ADDRESS_CLEAR_CTRL_FLAGS	0x1FFFFFFF	Bitmask to extract the raw page address or index from a control-packed address word.
BIT_CLEAR_WRITE	0xDFFFFFFF	Clears the write-back bit (Bit 29) once the page has been saved.
PAGE_OFFSET_MASK	(1u << pageSizeNumBits) - 1	Mask to compute the intra-page offset from a flat global index.
Note: Bit positions are used contextually. The same bit (e.g., Bit 31) may represent different states depending on whether it marks a data index or a page table entry.		

Table A.1: Unified bit flags and masks used for out-of-core page control, including loading, eviction, and write-back.

Listing A.1 illustrates the processing logic of a kernel launch that accesses and processes out-of-core (OOC) data blocks. In this example, each thread accesses an index and retrieves the

Variable Name	Memory Scope	Type	Description
`host_page_table`	Host	`std::vector<uint32_t>`	Host-side page control flags (load requests, write-back, etc.).
`page_table`	Device	`dev_ptr<uint32_t>`	Shared page table reflecting device page status.
`resident_blocks_indices`	Host	`std::vector<int32_t>`	Maps each in-core slot to the currently loaded page index.
`page_size`	Host	`uint32_t`	Number of elements (e.g., vertices) per page.
`num_pages`	Host	`uint32_t`	Total number of pages in the virtual address space.
`paged_set_num_pages`	Host	`uint32_t`	Number of in-core page slots (ring buffer size).
`page_table_size`	Host	`uint32_t`	Capacity of the page table (typically = paged_set_num_pages).
`page_size_num_bits`	Constant	`__constant__ uint32_t`	Number of bits used to compute the page shift ($\log_2$(pageSize)).
Note: Page control variables must be synchronized between the host and the device.			

Table A.2: Host and device variables used in the out-of-core paging system.

corresponding vertex from an OOC vertex array. At the start of execution, the most significant bit (MSB) of each index is set to indicate that the associated element has not yet been processed. Before invoking the processing kernel, this bit is initialized for all elements (Listing A.1, Line 7), as extended in Listing A.2. Additionally, all page table entries are cleared prior to launching the processing kernel, and the page table is copied to the GPU (Listing A.1, Lines 9–10). Upon processing a given element, this bit is cleared. In subsequent kernel launches, any thread encountering a cleared bit flag immediately exits, thereby avoiding redundant computation.

```
bool has_load_requests = false;
bool has_write_back_requests = false;

dim3 threads_per_block(256);
dim3 blocks_per_grid = (num_elements + threads_per_block.x - 1) / threads_per_block.x;

flag_indices_kernel<<<blocks_per_grid, threads_per_block>>>(data_indices, num_elements);

reset_page_table();
copy_page_table_to_device();

do {
    ooc_processing_kernel<<<blocks_per_grid, threads_per_block>>>(data_indices, out_vertices, page_table, num_pages,
      in_core_vertices, num_elements);

    copy_page_table_to_host();

    has_write_back_requests = check_write_back_requests();
    if (has_write_back_requests) {
        do_write_back_requests();
    }

    has_load_requests = check_load_requests();
    if (has_load_requests) {
        do_load_requests();

        copy_page_table_to_device();
    }

} while (has_load_requests);
```

Listing A.1: This procedure involves multiple GPU kernel launches, where each iteration issues page requests and, if necessary, writes all modified pages back to host memory.

```
__global__ void flag_indices_kernel(uint32_t* data_indices, uint32_t num_elements) {
    uint32_t idx = blockDim.x * blockIdx.x + threadIdx.x;

    if (idx < num_elements) {
        data_indices[idx] |= BIT_DATA_NOT_PROCESSED;
    }
}
```

Listing A.2: A kernel that sets a flag indicating each element is unprocessed before any subsequent kernel launch, allowing selective processing in future iterations.

Since it is not guaranteed that all data requests are satisfied in a single iteration, we employ a `do-while` loop that repeatedly launches the kernel and supplies page requests, exiting only when no further page requests or write-backs are required (Listing A.1, Lines 12–29).

```
__constant__ uint32_t page_size_num_bits = 12;

__device__ void ooc_processing_kernel(uint32_t* data_indices, vec3* out_vertices, uint32_t* page_table, uint32_t
    num_pages, vec3* in_core_vertices, uint32_t num_elements) {
    uint32_t idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= num_elements)
        return;

    bool data_not_processed = (data_indices[idx] & BIT_DATA_NOT_PROCESSED);

    if (data_not_processed) {

        uint32_t raw_index = data_indices[idx] & ~BIT_DATA_NOT_PROCESSED;
        uint32_t page_idx = raw_index >> page_size_num_bits;
        uint32_t page_entry = page_table[page_idx];

        bool page_loaded = (page_entry & BIT_PAGE_LOADED);

        if (page_loaded) {
            uint32_t local_page_idx = page_entry & ADDRESS_CLEAR_CTRL_FLAGS;
            uint32_t page_offset = raw_index & PAGE_OFFSET_MASK;
            uint32_t local_address = (local_page_idx << pageSizeNumBits) | page_offset; // address translation

            out_vertices[idx] = in_core_vertices[local_address];

            page_table[page_idx] |= BIT_WRITE_BACK;

            data_indices[idx] &= ~BIT_DATA_NOT_PROCESSED;
        }
        else {
            page_table[page_idx] |= BIT_PAGE_REQUEST;
        }
    }
}
```

Listing A.3: A processing kernel responsible for performing address translation to access OOC data stored in paged memory.

Listing A.3 illustrates the kernel that processes array elements residing in OOC memory. Initially, we check the MSB of each index to determine whether the element has already been processed in a previous kernel call (Lines 9–11). If not, we perform address translation and check whether the corresponding vertex page resides in GPU memory (Lines 13–15). If the page is available in-core (Lines 17–29), we proceed to process the element, issue a page write to store the output data, and reset the index flag to prevent redundant processing in future iterations. Otherwise, a page request is issued (Lines 30–32), and processing of that element is deferred until the page is loaded in a subsequent kernel launch.

After each kernel launch, we check whether any page write or page request is needed. Based on this check, we supply the necessary page writes (Listing A.1, Lines 17–20), as extended in Listing A.6 and in Listing A.7, or issue page requests according to the available memory budget (Listing A.1, Lines 22–27), as extended in Listing A.4 and in Listing A.5.

Once the required pages are (partially) loaded onto the GPU—based on the available GPU memory budget—we copy the updated page table back to the GPU (Listing A.1, Line 26) and re-launch the kernel in the following iteration to continue processing.

```
1 bool check_load_requests(const std::vector<uint32_t>& host_page_table, uint32_t num_pages) {
2     for (uint32_t page_idx = 0; page_idx < num_pages; ++page_idx) {
3         if (host_page_table[page_idx] & BIT_PAGE_REQUEST) {
4             return true;
5         }
6     }
7     return false;
8 }
```

Listing A.4: Page request check loop.

We always maintain two consistent copies of the page table—on both the CPU and the GPU. After each kernel call that accesses OOC data blocks, we copy the page table from the GPU to the CPU (Listing A.1, Line 15), then supply the necessary page requests Listing A.4. We iterate over the page table, page by page, to copy the required pages from host to device (see Listing A.5). During this process, we update the relevant request and residency flags. We also maintain an indirection array that holds references to previously loaded pages from earlier kernel calls. Upon page replacement, we invalidate the residency flag of the corresponding previously loaded page.

```
void do_load_requests() {
    static uint32_t page_index = 0;         // ring cursor
    static uint32_t in_core_page_idx = 0;   // ring cursor

    uint32_t num_loaded_pages = 0;          // number of pages loaded in this call

    auto start_in_core_idx = in_core_page_idx;
    for (uint32_t iter = 0; iter < num_pages; ++iter) {

        if (host_page_table[page_index] & BIT_PAGE_REQUEST) {

            int32_t previously_loaded_page = resident_blocks_indices[in_core_page_idx];
            if (previously_loaded_page != -1) {
                // Mark for unload (clear the in-core bit)
                page_table[previously_loaded_page] &= ~BIT_PAGE_LOADED;
            }

            // Stage into the ring slot
            memcpy(&host_paged_vertices[in_core_page_idx * page_size], &ooc_vertices[page_index * page_size],
     page_size * sizeof(vec3));

            // Request served -> clear request bit
            host_page_table[page_index] &= ~BIT_PAGE_REQUEST;

            // Page is now loaded -> set in-core bit
            host_page_table[page_index] |= BIT_PAGE_LOADED;

            // Update mapping (slot -> page)
            resident_blocks_indices[in_core_page_idx] = page_index;

            // Advance ring cursor by one
            in_core_page_idx = (in_core_page_idx + 1) % page_table_size;

            num_loaded_pages++;
            if (num_loaded_pages == paged_set_num_pages) { // reached the maximum number of in-core pages
                page_index = (page_index + 1) % num_pages;
                break;
            }
        }

        page_index = (page_index + 1) % num_pages;
    }

    if (num_loaded_pages > 0) {
        // Transfer paged vertices from host memory (host_paged_vertices) to the in-core cache, loading up to
     num_loaded_pages into in_core_vertices
    }
}
```

Listing A.5: Loading requested pages into GPU core.

To save bandwidth, we concatenate all required page blocks on the CPU and perform a single CPU–GPU transfer. Our paging strategy follows a first-in, first-out (FIFO) approach by always iterating from the last accessed page index in the previous call, in a circular fashion.

In some scenarios, certain data blocks must be copied back from the GPU to the CPU. In such cases, before handling any new page requests, we iterate over these pages and write them back to the CPU (see Listing A.7). We use the indirection array to check for any page

that requires a write-back Listing A.6, and iterate over this list to write all updated pages to the CPU.

```
bool check_write_back_requests() {
    for (uint32_t in_core_slot_idx = 0; in_core_slot_idx < paged_set_num_pages; ++in_core_slot_idx) {

        int32_t page_idx = resident_blocks_indices[in_core_slot_idx];

        if (page_idx != -1 && (host_page_table[page_idx] & BIT_WRITE_BACK)) {
            return true;
        }
    }
    return false;
}
```

Listing A.6: Page write-back check loop.

```
void do_write_back_requests() {
    for (uint32_t in_core_slot_idx = 0; in_core_slot_idx < paged_set_num_pages; ++in_core_slot_idx) {

        int32_t page_idx = resident_blocks_indices[in_core_slot_idx];

        if (page_idx != -1 && (host_page_table[page_idx] & BIT_WRITE_BACK)) {

            // Copy the page indexed by page_idx from the in-core block to its corresponding CPU block

            page_table[page_idx] &= BIT_CLEAR_WRITE;
        }
    }
}
```

Listing A.7: Writing back modified pages to CPU memory.

B Implementation Details of Out-of-Core Chunk-Based Merging for Large Array Sorting

In this section, we present key code snippets illustrating the core functionalities of our proposed chunk-based merge sorting algorithm on the GPU. At the start, we divide the input keys into chunks, where each chunk can be independently sorted on GPU. We then recursively merge each pair of sorted chunks until we obtain a single chunk containing the complete set of keys.

Merging of chunk pairs is performed in a ping-pong fashion, where input keys are read from input buffers and output keys are written to an output buffer. These buffers are swapped after each merge round. Listing B.1 illustrates the logic of chunk merging. Initially, we compute the number of chunks based on the total number of keys and the predefined chunk size, as well as determine the number of merge rounds and the input/output buffer configuration (Lines 1–6).

The process then proceeds through several merge rounds, starting with the number of chunks produced by the initial chunk sorting step and the corresponding chunk size. During each round, chunk pairs are selected and merged into larger chunks (Lines 24–32). After all chunk pairs are merged in a round, the chunk size is doubled and the number of chunks is halved (Lines 35–37). In cases where there is an odd number of chunks, the remaining unpaired chunk is directly copied to the output buffer (Lines 15–22).

```cpp
uint32_t num_chunks = (num_elements + chunk_size - 1) / chunk_size;
uint32_t num_merge_rounds = log2_uint32(next_pow2(num_chunks));
uint32_t current_chunk_size = chunk_size;
uint32_t current_chunk_count = num_chunks;
uint32_t in_idx = 0u;
uint32_t out_idx = 1u - in_idx;

for (uint32_t round = num_merge_rounds; round > 0u; round--) {
    out_idx = 1u - in_idx;

    for (uint32_t chunk = 0; chunk < current_chunk_count; chunk += 2) {

        // Special handling for an odd number of chunks to merge
        // Last input chunk is copied from input buffer to output buffer
        if (chunk + 1 == current_chunk_count) {
            uint32_t src_chunk_start = chunk * current_chunk_size;
            uint32_t src_chunk_end = num_elements;
            uint32_t src_chunk_size = src_chunk_end - src_chunk_start;
            memcpy(&keys[out_idx][src_chunk_start], &keys[in_idx][src_chunk_start], src_chunk_size * sizeof(uint64_t
    ));
            memcpy(&indices[out_idx][src_chunk_start], &indices[in_idx][src_chunk_start], src_chunk_size * sizeof(
    uint64_t));
            break;
        }

        uint32_t left_chunk_start = chunk * current_chunk_size;
        uint32_t left_chunk_end = std::min((chunk + 1) * current_chunk_size, num_elements);
        uint32_t left_chunk_size = left_chunk_end - left_chunk_start;

        uint32_t right_chunk_start = left_chunk_end;
        uint32_t right_chunk_end = std::min((chunk + 2) * current_chunk_size, num_elements);
        uint32_t right_chunk_size = right_chunk_end - right_chunk_start;

        merge_chunks(keys[in_idx][left_chunk_start], indices[in_idx][left_chunk_start], left_chunk_size, keys[in_idx
    ][right_chunk_start], indices[in_idx][right_chunk_start], right_chunk_size, keys[out_idx][left_chunk_start],
    indices[out_idx][left_chunk_start]);
    }

    current_chunk_size <<= 1;
    current_chunk_count >>= 1;
    in_idx = 1u - in_idx;
}

uint32_t sorted_array_index = out_idx;
```

Listing B.1: Chunk merging loop that alternates the roles of the input and output buffers in each iteration (ping-pong buffering).

B.1 Merging Two Sorted Chunks

Our proposed merge sort strategy employs both out-of-core (OOC) paging and adaptive splitting of input chunks into partitions that fit within the available GPU memory budget. This approach is illustrated in Figure 6.3. We begin by selecting keys from both input arrays at fixed intervals; these selected keys are referred to as *splitters*, and the region between two consecutive splitters is defined as a *block*. For each splitter, we track its rank in the chunk from which it was selected. The splitters are then merged into a single sorted array.

Additionally, a flag array is used to record the origin of each splitter (i.e., whether it came from the first or second input chunk), and a prefix scan is computed over this flag array. Collectively, the splitters, their original ranks, the flag values, and the prefix scan results are used to identify the partial regions in the original arrays that must be merged to form the final sorted output.

Given any two bounds in the sorted splitter array, we denote by n_l and n_r the numbers of splitters from the first and second arrays, respectively. In the merge step, we load $n_l - 1$ and $n_r - 1$ blocks from their corresponding arrays. Additionally, two more blocks are loaded around the boundary splitters—these extra blocks originate from the chunk from which the corresponding boundary splitter was not selected. A prefix scan over the flag array allows us to determine how many preceding blocks from each chunk need to be loaded near the boundary splitters.

Listing B.2 illustrates a merging step between two sorted chunks. Several code-level extensions to the basic merging logic are highlighted and discussed in subsequent sections. The function takes as input the keys and indices chunks and their lengths and merges them into a larger sorted chunk. A partition of the splitter array is selected based on the memory budget, and the corresponding data blocks from the left and right chunks are loaded. These blocks are then merged and copied into the output chunk. For each input block, the prefix scan array is used to determine the corresponding region in the opposite chunk, and a binary search is performed within that limited range. Each key is then placed in the output chunk at the index computed as the sum of its original rank and its rank in the opposite chunk.

```
void merge_chunks(vector<uint64_t> keys_left_host, vector<uint64_t> indices_left_host, size_t left_chunk_size,
    vector<uint64_t> keys_right_host, vector<uint64_t> indices_right_host, size_t right_chunk_size, vector<uint64_t
    > keys_out, vector<uint64_t> indices_out) {

    dim3 threads_per_block(256);

    // Sample splitters from the left and right chunks using OOC paging and prepare corresponding ranks and flags
    // Merge splitters and sort them into a single array

    // num_merge_rounds is computed based on the available memory budget and the total number of keys across both
    chunks
    dim3 num_blocks = (num_merge_rounds + threads_per_block.x - 1) / threads_per_block.x;
    find_merge_rounds_blocks_bounds<<<num_blocks, threads_per_block>>>(left_flags, left_flags_scan,
    total_left_blocks, total_right_blocks, total_blocks, num_blocks_per_round, left_blocks_bounds_scan,
    right_blocks_bounds_scan, blocks_bounds_scan, num_merge_rounds);

    // Apply the inverse process of the prefix sum of left_blocks_bounds_scan, right_blocks_bounds_scan, and
    blocks_bounds_scan to obtain the corresponding values
    // Get host versions of these arrays for processing the rounds below

    for (uint32_t round_idx = 0u; round_idx < num_merge_rounds; round_idx++) {

        // Get the block bounds for the left and right chunks in the current round
        get_blocks_ranges(round_idx, total_left_blocks, total_right_blocks, start_block, end_block,
    num_blocks, left_blocks_start, right_blocks_start, left_blocks_start_with_padding,
    right_blocks_start_with_padding, left_blocks_end_with_padding, right_blocks_end_with_padding, num_left_blocks,
    num_right_blocks);

        // Get the key ranges in the left and right chunks for the current round
        get_round_chunks_sub_ranges(left_blocks_start_with_padding, right_blocks_start_with_padding,
    left_blocks_end_with_padding, right_blocks_end_with_padding, left_chunk_size, right_chunk_size,
    left_blocks_keys_start, right_blocks_keys_start, left_blocks_keys_end, right_blocks_keys_end,
    left_blocks_keys_size, right_blocks_keys_size);

        // Load the key-value pairs of this round from the input chunks on the CPU to the GPU,
        // guided by the current block boundaries on the left and right chunks.

        // Find the ranks of each splitter in the left and right chunks relative to the current round.

        // Start index in the left and right chunks based on the first splitter ranks in the current round
        size_t left_keys_range_start = ...;
        size_t right_keys_range_start = ...;

        // Start index of keys in the output chunk
        size_t out_chunk_start = ...;

        merge_keys(left_keys_range_start, right_keys_range_start, splitters_ranks_in_left_chunk,
    splitters_ranks_in_right_chunk, keys_left, keys_right, indices_left, indices_right, keys_out, indices_out,
    out_chunk_start, start_block, num_blocks);

        // Copy sorted keys of the current round back to the output buffer on the CPU

    }
}
```

Listing B.2: A merge round of two sorted chunks is performed using dynamic scheduling, guided by splitters and blocks, according to the allowed memory budget.

Finding Block Boundaries for Sub-Merge Rounds At the beginning of the merging phase, and in accordance with the available GPU memory, the total number of merge rounds is computed. For each round, the number of blocks to be loaded from the left and right chunks

is determined, along with the number of merged output blocks (Listing B.2, Line 10). This procedure is implemented by the kernel shown in Listing B.3.

The number of blocks from the left chunk is obtained via a prefix scan over the flag array, while the number of right-side blocks is implicitly derived from the block index and the number of left-side blocks. Special conditions are applied to handle boundary cases, as demonstrated in the code snippet.

```
1 __global__ void find_merge_rounds_blocks_bounds(const uint32_t* left_flags, const uint32_t* left_flags_scan,
      uint32_t total_left_blocks, uint32_t total_right_blocks, uint32_t total_blocks, uint32_t num_blocks_per_round,
      uint32_t* left_blocks_bounds_scan, uint32_t* right_blocks_bounds_scan, uint32_t* blocks_bounds_scan, uint32_t
      num_merge_rounds) {
2     uint32_t idx = blockDim.x * blockIdx.x + threadIdx.x;
3     if (idx >= num_merge_rounds) return;
4
5     uint32_t round_idx = idx;
6     uint32_t last_block_idx = min((round_idx + 1u) * num_blocks_per_round - 1u, total_blocks - 1u);
7     uint32_t left_flag = left_flags[last_block_idx];
8     uint32_t left_flag_scan = left_flags_scan[last_block_idx];
9     uint32_t num_left_blocks = min(left_flag_scan + left_flag, total_left_blocks);
10    uint32_t num_right_blocks = min(max(int32_t(last_block_idx + 1u) - int32_t(num_left_blocks), 0) + (1u -
      left_flag), total_right_blocks);
11
12    // The values stored below represent an inclusive scan of the number of blocks up to the current block.
13    // The number of blocks can be extracted by computing the difference between the current block and the previous
      block in the scan array.
14    left_blocks_bounds_scan[round_idx + 1u] = num_left_blocks;
15    right_blocks_bounds_scan[round_idx + 1u] = num_right_blocks;
16    blocks_bounds_scan[round_idx + 1u] = last_block_idx + 1u;
17
18    if (round_idx == 0) {
19        left_blocks_bounds_scan[0] = 0;
20        right_blocks_bounds_scan[0] = 0;
21        blocks_bounds_scan[0] = 0;
22    }
23 }
```

Listing B.3: Estimating the merge round bounds for a single merge step of two sorted chunks based on the allowed memory budget. This process estimates, for each merge round, the number of blocks from each chunk based on the flags array.

Extracting Sub-Merge Round Boundaries We iterate over several rounds and adaptively merge key-index pairs based on the computed splits (Listing B.2, Lines 15–39). For a given round, we retrieve the block ranges for the left and right chunks from the previously computed array in Listing B.3. Each of these ranges is extended by one or two blocks on both sides to ensure the inclusion of keys when merging near chunk boundaries. This step is illustrated in Listing B.4, noting that the range array is first copied to host memory to enable efficient access during the merge iterations.

```
1 void get_blocks_ranges(const uint32_t& round_idx, const uint32_t& total_left_blocks, const uint32_t&
      total_right_blocks, const uint32_t& total_blocks, uint32_t& start_block, uint32_t& end_block, uint32_t&
      num_blocks, uint32_t& left_blocks_start, uint32_t& right_blocks_start, uint32_t& left_blocks_start_with_padding
      , uint32_t& right_blocks_start_with_padding, uint32_t& left_blocks_end_with_padding, uint32_t&
      right_blocks_end_with_padding, uint32_t& num_left_blocks, uint32_t& num_right_blocks)
2 {
3     start_block = round_idx * num_blocks_per_round;
4     end_block = min((round_idx + 1) * num_blocks_per_round, total_blocks);
5     num_blocks = end_block - start_block;
6
7     // left_blocks_bounds_host and right_blocks_bounds_host represent host-side copies obtained after extracting
      differences between consecutive elements of left_blocks_bounds_scan and right_blocks_bounds_scan, respectively.
8     // left_blocks_bounds_scan and right_blocks_bounds_scan represent inclusive scans of the values in
      left_blocks_bounds_host and right_blocks_bounds_host.
9     left_blocks_start = left_blocks_bounds_host[round_idx];
10    right_blocks_start = right_blocks_bounds_host[round_idx];
11
12    left_blocks_start_with_padding = max(int32_t(left_blocks_start) - int32_t(2), int32_t(0));
13    right_blocks_start_with_padding = max(int32_t(right_blocks_start) - int32_t(2), int32_t(0));
14
15    auto left_blocks_end = left_blocks_bounds_host[round_idx + 1];
16    auto right_blocks_end = right_blocks_bounds_host[round_idx + 1];
17
18    left_blocks_end_with_padding = min(left_blocks_end + 2u, total_left_blocks);
19    right_blocks_end_with_padding = min(right_blocks_end + 2u, total_right_blocks);
20
21    num_left_blocks = left_blocks_end - left_blocks_start;
22    num_right_blocks = right_blocks_end - right_blocks_start;
23 }
```

Listing B.4: Extracting block ranges for each sub-merge round.

Loading Sub-Merge Round Key-Index Pairs We then extract the actual key-index ranges of each chunk by multiplying the block indices by the block size for the left and right partitions, respectively (Listing B.2, Line 21), as detailed in Listing B.5.

```
1 void get_round_chunks_sub_ranges(const size_t& left_blocks_start_with_padding, const size_t&
      right_blocks_start_with_padding, const size_t& left_blocks_end_with_padding, const size_t&
      right_blocks_end_with_padding, const size_t& left_chunk_size, const size_t& right_chunk_size, size_t&
      left_blocks_keys_start, size_t& right_blocks_keys_start, size_t& left_blocks_keys_end, size_t&
      right_blocks_keys_end, size_t& left_blocks_keys_size, size_t& right_blocks_keys_size) {
2
3     // merging_block_size represents the block size (i.e., the number of keys) between two consecutive sampled
      splitters
4     left_blocks_keys_start  = left_blocks_start_with_padding  * merging_block_size;
5     right_blocks_keys_start = right_blocks_start_with_padding * merging_block_size;
6
7     left_blocks_keys_end  = std::min(left_blocks_end_with_padding  * merging_block_size, left_chunk_size);
8     right_blocks_keys_end = std::min(right_blocks_end_with_padding * merging_block_size, right_chunk_size);
9
10    left_blocks_keys_size  = left_blocks_keys_end  - left_blocks_keys_start;
11    right_blocks_keys_size = right_blocks_keys_end - right_blocks_keys_start;
12 }
```

Listing B.5: Getting key-index pair boundaries based on block boundaries.

Loading Key-Index Pairs In the next step, we load the key-index pairs into GPU buffers to compute the rank of each key in the left and right chunks, respectively.

Finding the Two Ranks of Each Sampled Key For each sampled key, we determine its rank in the opposite chunk—i.e., the chunk from which it was not selected—using binary search. Two separate functions are used to compute these ranks: one for keys originating from the left chunk and another for keys from the right chunk. Both functions implement slightly modified versions of the standard binary search algorithm. These ranks are later used to restrict the search space for the in-between keys to a narrow interval within the opposite array.

```
1 __device__ uint32_t find_right_rank(const uint64_t& key, const uint64_t* keys, uint32_t start, uint32_t end, bool
      ascending = true) {
2     int32_t start_ = int32_t(start), end_ = int32_t(end);
3     while (start_ <= end_) {
4         auto mid = (start_ + end_) >> 1;
5         if (ascending) {
6             if (key <= keys[mid]) end_ = mid - 1;
7             else start_ = mid + 1;
8         } else {
9             if (key < keys[mid]) start_ = mid + 1;
10            else end_ = mid - 1;
11        }
12    }
13    return start_ - start;
14 }
15
16 __device__ uint32_t find_left_rank(const uint64_t& key, const uint64_t* keys, uint32_t start, uint32_t end, bool
      ascending = true) {
17    int32_t start_ = int32_t(start), end_ = int32_t(end);
18    while (start_ <= end_) {
19        auto mid = (start_ + end_) >> 1;
20        if (ascending) {
21            if (key < keys[mid]) end_ = mid - 1;
22            else start_ = mid + 1;
23        } else {
24            if (key <= keys[mid]) start_ = mid + 1;
25            else end_ = mid - 1;
26        }
27    }
28    return start_ - start;
29 }
```

Listing B.6: Finding the key's rank within a certain range of a sorted array of keys. At the top, it involves finding the rank of a key from the left chunk within the keys of the right chunk, and at the bottom, finding the rank of a key from the right chunk in the left chunk.

Our implementation ensures stability, meaning that keys with the same value maintain their original relative order in the output [PT92]. In Listing B.6, we show the binary search functions used to find the rank of a key in the opposite sorted chunk. These functions return the rank relative to the beginning of the search range.

```
1 __global__ void merge_keys_kernel(size_t left_keys_range_start, size_t right_keys_range_start, const uint64_t*
      splitters_ranks_in_left_chunk, const uint64_t* splitters_ranks_in_right_chunk, const uint64_t* keys_left, const
      uint64_t* keys_right, const uint64_t* indices_left, const uint64_t* indices_right, uint64_t* keys_out,
      uint64_t* indices_out, size_t out_chunk_start, size_t start_block, size_t num_blocks) {
2     auto block_idx = blockIdx.x;
3     auto thread_idx = threadIdx.x;
4     auto threads_per_block = blockDim.x;
5
6     if (block_idx < num_blocks) {
7         auto data_block_idx = start_block + block_idx;
8         auto start_range_left = splitters_ranks_in_left_chunk[data_block_idx];
9         auto end_range_left = splitters_ranks_in_left_chunk[data_block_idx + 1];
10        auto num_elements_left = end_range_left - start_range_left;
11        auto start_range_right = splitters_ranks_in_right_chunk[data_block_idx];
12        auto end_range_right = splitters_ranks_in_right_chunk[data_block_idx + 1];
13        auto num_elements_right = end_range_right - start_range_right;
14
15        for (auto ti = thread_idx; ti < num_elements_left; ti += threads_per_block) {
16            auto left_rank = start_range_left + ti;
17            auto local_idx = left_rank - left_keys_range_start;
18            auto key = keys_left[local_idx];
19            auto value = indices_left[local_idx];
20            size_t other_rank = 0;
21            if (num_elements_right) {
22                auto start_search = start_range_right - right_keys_range_start;
23                auto end_search = end_range_right - right_keys_range_start - 1lu;
24                // keys_right is loaded in part based on the current round boundaries
25                other_rank = right_binary_search(key, keys_right, start_search, end_search);
26            }
27            auto right_rank = start_range_right + other_rank;
28            auto out_idx = left_rank + right_rank - out_chunk_start;
29            keys_out[out_idx] = key;
30            indices_out[out_idx] = value;
31        }
32
33        for (auto ti = thread_idx; ti < num_elements_right; ti += threads_per_block) {
34            auto right_rank = start_range_right + ti;
35            auto local_idx = right_rank - right_keys_range_start;
36            auto key = keys_right[local_idx];
37            auto value = indices_right[local_idx];
38            size_t other_rank = 0;
39            if (num_elements_left) {
40                auto start_search = start_range_left - left_keys_range_start;
41                auto end_search = end_range_left - left_keys_range_start - 1lu;
42                // keys_left is loaded in part based on the current round boundaries
43                other_rank = left_binary_search(key, keys_left, start_search, end_search);
44            }
45            auto left_rank = start_range_left + other_rank;
46            auto out_idx = left_rank + right_rank - out_chunk_start;
47            keys_out[out_idx] = key;
48            indices_out[out_idx] = value;
49        }
50    }
51 }
52
53 void merge_keys(size_t left_keys_range_start, size_t right_keys_range_start, const uint64_t*
      splitters_ranks_in_left_chunk, const uint64_t* splitters_ranks_in_right_chunk, const uint64_t* keys_left, const
      uint64_t* keys_right, const uint64_t* indices_left, const uint64_t* indices_right, uint64_t* keys_out,
      uint64_t* indices_out, size_t out_chunk_start, size_t start_block, size_t num_blocks) {
54
55    dim3 threads_per_block(256);
56    dim3 num_thread_blocks = num_blocks;
57    merge_keys_kernel<<<num_thread_blocks, threads_per_block>>>(left_keys_range_start, right_keys_range_start,
      splitters_ranks_in_left_chunk, splitters_ranks_in_right_chunk, keys_left, keys_right, indices_left,
      indices_right, keys_out, indices_out, out_chunk_start, start_block, num_blocks);
58 }
```

Listing B.7: Merging blocks' keys based on the two ranks of each key.

Storing Keys into Their Respective Locations As a final step in the merging routine, each key is placed into the output buffer at a location equal to the sum of its rank within its originating chunk and its rank in the opposite chunk. This operation is illustrated (Listing B.2, Line 35) and further detailed in Listing B.7. In this step, for each processed block, we find the range of keys in the left and right chunks using the ranks arrays of the splitters for both chunks. Using these ranks arrays, we restrict the search range of each key in the other chunk, from which it was not taken, to a narrow range.

Lastly, the key-index pairs are copied to their designated positions in the output buffer.

C Implementation Details of Level-of-Detail Selection and Frustum Culling

To perform level-of-detail (LoD) selection, we start in Line 4, where we extract the most significant bit of a node index to check whether the node previously passed the frustum test and was marked as entirely inside the viewing frustum. In such a case, we avoid further frustum intersection tests for all of the node's descendants.

Next, we retrieve the node data and bounding box, either directly from GPU memory for the top-level hierarchy or using out-of-core (OOC) paging for the bottom-level hierarchy (Line 9). In Line 11, we compute the screen projection of the node's bounding box; if it passes a certain threshold and the node is marked as an internal node, we traverse one more level in the hierarchy and re-check the child nodes for frustum culling (Lines 12–33).

A node that was previously marked as entirely inside the frustum does not need to be tested again for frustum intersection. Conversely, a node that is not marked as being inside the frustum must still undergo frustum intersection testing (Line 17). If the bounding box lies entirely inside the frustum, we set a flag to mark all of the node's descendants as being entirely inside the frustum in the following steps (Line 18). After the frustum-box intersection test, the children of the current node are added to the next round of cut tests if the node is at least partially inside the viewing frustum. Otherwise, we stop traversal at that node and add it to the list of cut nodes.

```
1  __constant__ vec4 frustum_planes[6];
2
3  __device__ void node_frustum_test(uint16_t node_idx_flag) {
4    auto inside_flag = node_idx_flag & 0x8000;
5    auto node_idx = node_idx_flag & 0x7FFF;
6
7    uint16_t node_data[2];
8    vec3 node_aabb[2];
9    get_node_data(node_idx, node_data, node_aabb);
10
```

```
11   pass = check_projection_th(node_aabb);
12   if (pass) { // node is larger than the projection threshold
13     auto intersect_state = 2; // outside frustum
14
15     if (!inside_flag) {
16       // intersect_state: 0 = intersecting frustum, 1 = inside frustum
17       intersect_state = frustum_aabb_intersect(node_aabb, frustum_planes);
18       inside_flag = intersect_state == 1 ? 0x8000 : 0;
19     }
20
21     if (inside_flag || intersect_state == 0 || intersect_state == 1) {
22       if (node_data[1] != 0xFFFF) { // leaf node
23         // __shared__ atomic
24         add_to_render_list(node_idx);
25       }
26       else {
27         // __shared__ atomic
28         // add left and right child nodes to the next round
29         add_to_frustum_test(node_data[0]     | inside_flag);
30         add_to_frustum_test(node_data[0] + 1 | inside_flag);
31       }
32     }
33   }
34   else { // node is small enough to be rendered
35     // __shared__ atomic
36     add_to_render_list(node_idx);
37   }
38 }
```

Listing C.1: Adaptive Level-of-Detail Selection and Frustum Culling.

Bibliography

[ABDVC87] AUGUIN M., BOERI F., DALBAN J. P., VINCENT-CARREFOUR A.: Experience using a SIMD/SPMD Multiprocessor Architecture. *Microprocessing and Microprogramming 21*, 1–5 (1987). doi:https://doi.org/10.1016/0165-6074(87)90034-2.

[ADMO17] ASHKIANI S., DAVIDSON A., MEYER U., OWENS J. D.: GPU Multisplit: An Extended Study of a Parallel Algorithm. *ACM Transactions on Parallel Computing 4*, 1 (2017). doi:10.1145/3108139.

[Adv69] ADVANCED MICRO DEVICES, INC.:. https://www.amd.com, 1969. Santa Clara, California, USA. Accessed: 3 January 2026.

[AL83] AUGUIN M., LARBEY F.: OPSILA: an Advanced SIMD for Numerical Analysis and Signal Processing. In *Microcomputers: Developments in Industry, Business, and Education, 9th EUROMICRO Symposium on Microprocessing and Microprogramming, Madrid, September 13* (1983), vol. 16.

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics* (2009), HPG ’09. doi:10.1145/1572769.1572792.

[AM22] ADINETS A., MERRILL D.: Onesweep: A Faster Least Significant Digit Radix Sort for GPUs, 2022. Accessed: 3 January 2026. URL: https://arxiv.org/abs/2206.01784.

[AMHM21] AKENINE-MÖLLER T., HAINES E., MARRS A. (Eds.): *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Apress, Berkeley, CA, 2021.

[AS16] Awan M. G., Saeed F.: GPU-ArraySort: A Parallel, In-Place Algorithm for Sorting a Large Number of Arrays. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)* (2016). `doi:10.1109/ICPPW.2016.27`.

[AZD17] Ament M., Zirr T., Dachsbacher C.: Extinction-Optimized Volume Illumination. *IEEE Transactions on Visualization and Computer Graphics 23*, 7 (2017). `doi:10.1109/TVCG.2016.2569080`.

[BCP*12] Brambilla A., Carnecky R., Peikert R., Viola I., Hauser H.: Illustrative Flow Visualization: State of the Art, Trends and Challenges. In *Eurographics 2012 - State of the Art Reports* (2012). `doi:10.2312/conf/EG2012/stars/075-094`.

[BD09] Boyce W. E., DiPrima R. C.: *Elementary Differential Equations and Boundary Value Problems*, 8th ed. John Wiley & Sons, Hoboken, NJ, 2009. Special Edition.

[BDPA18] Biswas A., Dutta S., Pulido J., Ahrens J.: In Situ Data-Driven Adaptive Sampling for Large-Scale Simulation Data Summarization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2018). `doi:10.1145/3281464.3281467`.

[BP04] Buck I., Purcell T.: A Toolkit for Computation on GPUs. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Fernando R., (Ed.). Addison-Wesley, 2004, ch. 37, pp. 621–636.

[BP23] Benthin C., Peters C.: Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail. *Computer Graphics Forum 42*, 8 (2023). `doi:https://doi.org/10.1111/cgf.14868`.

[BSSZ08] Bobenko A. I., Schröder P., Sullivan J. M., Ziegler G. M.: *Discrete Differential Geometry*, vol. 38. Springer Science & Business Media, 2008.

[Car84] Carpenter L.: The A-Buffer, An Antialiased Hidden Surface Method. *ACM SIGGRAPH Computer Graphics 18*, 3 (1984). `doi:10.1145/964965.808585`.

[CB11] Candelaresi S., Brandenburg A.: Decay of Helical and Nonhelical Magnetic Knots. *Physical Review E 84* (2011). doi:10.1103/PhysRevE.84.016406.

[CFM*13] Carnecky R., Fuchs R., Mehl S., Jang Y., Peikert R.: Smart Transparency for Illustrative Visualization of Complex Flow Surfaces. *IEEE Transactions on Visualization and Computer Graphics 19*, 5 (2013). doi:10.1109/TVCG.2012.159.

[CGA] CGAL, Computational Geometry Algorithms Library. https://www.cgal.org, Accessed: 3 January 2026.

[CL93] Cabral B., Leedom L. C.: Imaging Vector Fields Using Line Integral Convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (1993), SIGGRAPH '93. doi:10.1145/166117.166151.

[CLRS09] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.: *Introduction to Algorithms*, 3rd ed. MIT Press, Cambridge, MA, 2009.

[CNLE09] Crassin C., Neyret F., Lefebvre S., Eisemann E.: GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (2009), I3D '09. doi:10.1145/1507149.1507152.

[CSZ*25] Chen J., Shen Z., Zhao M., Jia X., Yan D.-M., Wang W.: FR-CSG: Fast and Reliable Modeling for Constructive Solid Geometry. *IEEE Transactions on Visualization and Computer Graphics 31*, 9 (2025). doi:10.1109/TVCG.2024.3481278.

[CT09] Cederman D., Tsigas P.: On Sorting and Load Balancing on GPUs. *ACM SIGARCH Computer Architecture News 36*, 5 (2009). doi:10.1145/1556444.1556447.

[CT10] Cederman D., Tsigas P.: GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. *ACM Journal of Experimental Algorithmics (JEA) 14* (2010). doi:10.1145/1498698.1564500.

[Dar01] Darema F.: The SPMD Model: Past, Present and Future. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2001). doi: https://doi.org/10.1007/3-540-45417-9_1.

[Den96] Denning P. J.: Virtual Memory. *ACM Computing Surveys (CSUR) 28*, 1 (1996). doi:10.1145/234313.234403.

[DGNP88] Darema F., George D. A., Norton V. A., Pfister G. F.: A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. *Parallel Computing 7*, 1 (1988). doi:https://doi.org/10.1016/0167-8191(88)90094-4.

[dLvW93] de Leeuw W., van Wijk J.: A Probe for Local Flow Field Visualization. In *Proceedings of Visualization '93* (1993). doi:10.1109/VISUAL.1993.398849.

[DRPS87] Darema-Rogers F., Pfister G. F., So K.: Memory Access Patterns of Parallel Scientific Programs. *ACM SIGMETRICS Performance Evaluation Review 15*, 1 (1987). doi:10.1145/29904.29912.

[DVS03] Dachsbacher C., Vogelgsang C., Stamminger M.: sequential Point Trees. *acm transactions on graphics (tog) 22*, 3 (2003). doi:10.1145/882262.882321.

[DWS*17] Dutta S., Woodring J., Shen H.-W., Chen J.-P., Ahrens J.: Homogeneity Guided Probabilistic Data Summaries for Analysis and Visualization of Large-Scale Data Sets. In *2017 IEEE Pacific Visualization Symposium (PacificVis)* (2017). doi:10.1109/PACIFICVIS.2017.8031585.

[ELC*12] Edmunds M., Laramee R. S., Chen G., Max N., Zhang E., Ware C.: Surface-Based Flow Visualization. *Computers & Graphics 36*, 8 (2012). doi:10.1016/j.cag.2012.07.006.

[Fan08] Fan Z.: *Flow Simulation and Visualization on GPU Clusters*. PhD thesis, State University of New York at Stony Brook, USA, 2008.

[FLB*09] Fatahalian K., Luong E., Boulos S., Akeley K., Mark W. R., Hanrahan P.: Data-Parallel Rasterization of Micropolygons with Defocus and

Motion Blur. In *Proceedings of the Conference on High Performance Graphics* (2009), HPG '09. doi:10.1145/1572769.1572780.

[Fly66] FLYNN M. J.: Very High-Speed Computing Systems. *Proceedings of the IEEE 54*, 12 (1966). doi:10.1109/PROC.1966.5273.

[Fly72] FLYNN M. J.: Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers C-21*, 9 (1972). doi:10.1109/TC.1972.5009071.

[FW08] FALK M., WEISKOPF D.: Output-Sensitive 3D Line Integral Convolution. *IEEE Transactions on Visualization and Computer Graphics 14*, 4 (2008). doi:10.1109/TVCG.2008.25.

[GBR21] GÜNTHER T., BAEZA ROJO I.: Introduction to Vector Field Topology. In *Mathematics and Visualization* (2021), Springer Science and Business Media Deutschland GmbH. doi:https://doi.org/10.1007/978-3-030-83500-2_15.

[GG21] GROSS D., GUMHOLD S.: Advanced Rendering of Line Data with Ambient Occlusion and Transparency. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2021). doi:10.1109/TVCG.2020.3028954.

[Gha08] GHALI S.: *Constructive Solid Geometry*. Springer London, 2008, pp. 277–283. doi:10.1007/978-1-84800-115-2_30.

[GL10] GARANZHA K., LOOP C.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum 29*, 4 (2010). doi:10.1111/j.1467-8659.2009.01598.x.

[GLM] OpenGL Mathematics (GLM). https://glm.g-truc.net/, Accessed: 3 January 2026.

[GM04] GOBBETTI E., MARTON F.: Layered Point Clouds. In *SPBG'04 Symposium on Point - Based Graphics 2004* (2004). doi:10.2312/SPBG/SPBG04/113-120.

[GMB12] GREEN O., MCCOLL R., BADER D. A.: GPU Merge Path: a GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on*

Supercomputing (2012), ICS '12. doi:10.1145/2304576.2304621.

[GPM11] Garanzha K., Pantaleoni J., McAllister D.: Simpler and Faster HLBVH with Work Queues. In *Proceedings of the Conference on High Performance Graphics* (2011), HPG '11. doi:10.1145/2018323.2018333.

[Gro20] Group K.: Ray Tracing in Vulkan. https://www.khronos.org/blog/ray-tracing-in-vulkan, 2020. Accessed: 3 January 2026.

[GRT13] Günther T., Rössl C., Theisel H.: Opacity Optimization for 3D Line Fields. *ACM Transactions on Graphics (TOG) 32*, 4 (2013). doi:10.1145/2461912.2461930.

[GRT14] Günther T., Rössl C., Theisel H.: Hierarchical Opacity Optimization for Sets of 3D Line Fields. *Computer Graphics Forum 33*, 2 (2014). doi:10.1111/cgf.12336.

[GSE*14] Günther T., Schulze M., Esturo J. M., Rössl C., Theisel H.: Opacity Optimization for Surfaces. *Computer Graphics Forum 33*, 3 (2014). doi:10.1111/cgf.12357.

[GTG17] Günther T., Theisel H., Gross M.: Decoupled Opacity Optimization for Points, Lines and Surfaces. *Computer Graphics Forum 36*, 2 (2017). doi:10.1111/cgf.13115.

[GWG*20] Gralka P., Wald I., Geringer S., Reina G., Ertl T.: Spatial Partitioning Strategies for Memory-Efficient Ray Tracing of Particles. In *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)* (2020). doi:10.1109/LDAV51489.2020.00012.

[Gü16] Günther T.: *Opacity Optimization and Inertial Particles in Flow Visualization*. PhD thesis, University of Magdeburg, 2016. Accessed: 3 January 2026. URL: http://dx.doi.org/10.25673/4432.

[HA04] Helgeland A., Andreassen O.: Visualization of Vector Fields Using Seed LIC and Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 10*, 6 (2004). doi:10.1109/TVCG.2004.49.

[Hal19a] Halber M.: Drawing Lines with WebGL. https://blog.scottlogic.com/2019/11/18/drawing-lines-with-webgl.html, 2019. Accessed: 3 January 2026.

[Hal19b] Halber M.: Lines, 6 Different Implementations of Doing Wide Line Rendering in OpenGL. https://github.com/mhalber/Lines, 2019. Accessed: 3 January 2026.

[HAM19] Haines E., Akenine-Möller T. (Eds.): *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, Berkeley, CA, 2019.

[Har07] Harris M.: Optimizing Parallel Reduction in CUDA. *Nvidia developer technology 2*, 4 (2007).

[HBJP12] Hadwiger M., Beyer J., Jeong W.-K., Pfister H.: Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012). doi:10.1109/TVCG.2012.240.

[HBW*20] Hazarika S., Biswas A., Wolfram P. J., Lawrence E., Urban N.: Relationship-Aware Multivariate Sampling Strategy for Scientific Simulation Data. In *2020 IEEE Visualization Conference (VIS)* (2020). doi:10.1109/VIS47514.2020.00015.

[HH91] Helman J., Hesselink L.: Visualizing Vector Field Topology in Fluid Flows. *IEEE Computer Graphics and Applications 11*, 3 (1991). doi:10.1109/38.79452.

[HKS09] Ha L., Krüger J., Silva C. T.: Fast Four-Way Parallel Radix Sorting on GPUs. *Computer Graphics Forum 28*, 8 (2009). doi:https://doi.org/10.1111/j.1467-8659.2009.01542.x.

[HLH*16] Heine C., Leitte H., Hlawitschka M., Iuricich F., Floriani L. D., Scheuermann G., Hagen H., Garth C.: A Survey of Topology-Based Methods in Visualization. *Computer Graphics Forum 35*, 3 (2016). doi:10.1111/cgf.12933.

[HLNW11] Hlawatsch M., Leube P., Nowak W., Weiskopf D.: Flow Radar Glyphs—Static Visualization of Unsteady Flow with Uncertainty. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011). doi:10.1109/TVCG.2011.203.

[HM90] Haber R. B., McNabb D. A.: Visualization Idioms: A conceptual Model for Scientific Visualization Systems. *Visualization in scientific computing 74* (1990).

[HM22] Hanada M., Matsuura S.: *MCMC from Scratch: A Practical Introduction to Markov Chain Monte Carlo*. Springer, 2022. doi:10.1007/978-981-19-2715-7.

[HWU*19] Han M., Wald I., Usher W., Wu Q., Wang F., Pascucci V., Hansen C. D., Johnson C. R.: Ray Tracing Generalized Tube Primitives: Method and Applications. *Computer Graphics Forum 38*, 3 (2019). doi:10.1111/cgf.13703.

[IG98] Interrante V., Grosch C.: Visualizing 3D Flow. *IEEE Computer Graphics and Applications 18*, 4 (1998). doi:10.1109/38.689664.

[Int68] Intel Corporation:. https://www.intel.com, 1968. Santa Clara, California, USA, Accessed: 3 January 2026.

[jedban13] jan elseberg, dorit borrmann, andreas nüchter: One Billion Points in the Cloud – an Octree for Efficient Processing of 3D Laser Scans. *isprs journal of photogrammetry and remote sensing 76* (2013). doi:https://doi.org/10.1016/j.isprsjprs.2012.10.004.

[Jen01] Jensen H. W.: *Realistic Image Synthesis using Photon Mapping*. A. K. Peters, Ltd., USA, 2001.

[JH04] Johnson C., Hansen C.: *Visualization Handbook*. Academic Press, Inc., USA, 2004.

[JKPT21] Joó A. P., Koncz B., Pinter S., Tóth L. V.: Star Formation History in the Illustris TNG Simulation. *Proceedings of the International Astronomical Union 17*, S373 (2021). doi:10.1017/S1743921323000157.

[Khr22] KHRONOS GROUP: Mesh Shading for Vulkan. https://www.khronos.org/blog/mesh-shading-for-vulkan, 2022. Accessed: 3 January 2026.

[Khr24a] KHRONOS GROUP: *OpenGL® 4.6 Specification*. Khronos Group, 2024. Accessed: 3 January 2026. URL: https://www.khronos.org/opengl/.

[Khr24b] KHRONOS GROUP: *Vulkan® 1.3 Specification*. Khronos Group, 2024. Accessed: 3 January 2026. URL: https://www.khronos.org/vulkan/.

[KLM25] KOSAREVSKY S., LATYPOV V., MEDVEDEV A.: *Vulkan 3D Graphics Rendering Cookbook: Implement Expert-Level Techniques for High-Performance Graphics with Vulkan*. Packt Publishing, Birmingham, UK, 2025.

[KMW*19] KNOLL A., MORLEY R. K., WALD I., LEAF N., MESSMER P.: *Efficient Particle Volume Splatting in a Ray Tracer*. Apress, Berkeley, CA, 2019, ch. 29, pp. 533–541. doi:10.1007/978-1-4842-4427-2_29.

[KSS16] KESSENICH J., SELLERS G., SHREINER D.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016.

[KW05] KIPFER P., WESTERMANN R.: Improved GPU Sorting. *GPU gems 2* (2005).

[Lau10] LAURITZEN A.: Deferred Rendering for Current and Future Rendering Pipelines. In *Beyond Programmable Shading Course, ACM SIGGRAPH* (2010).

[LD12] LIKTOR G., DACHSBACHER C.: Decoupled Deferred Shading for Hardware Rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2012), I3D '12. doi:10.1145/2159616.2159640.

[LEG*08] LARAMEE R. S., ERLEBACHER G., GARTH C., SCHAFHITZEL T., THEISEL H., TRICOCHE X., WEINKAUF T., WEISKOPF D.: Applications of Texture-Based Flow Visualization. *Engineering Applications of Computational Fluid Mechanics 2*, 3 (2008). doi:10.1080/19942060.2008.11015227.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2009). doi:10.1111/j.1467-8659.2009.01377.x.

[LHD*04] LARAMEE R. S., HAUSER H., DOLEISCH H., VROLIJK B., POST F. H., WEISKOPF D.: The State of the Art in Flow Visualization: Dense and Texture-Based Techniques. *Computer Graphics Forum 23*, 2 (2004). doi:10.1111/j.1467-8659.2004.00753.x.

[LHZP07] LARAMEE R. S., HAUSER H., ZHAO L., POST F. H.: Topology-Based Flow Visualization, The State of the Art. In *Topology-based Methods in Visualization* (2007).

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the Conference on High Performance Graphics* (2013), HPG '13. doi:10.1145/2492045.2492060.

[LKB20] LTIFI H., KOLSKI C., BEN AYED M.: Survey on Visualization and Visual Analytics Pipeline-Based Models: Conceptual Aspects, Comparative Studies and Challenges. *Computer Science Review 36* (2020). doi:10.1016/j.cosrev.2020.100245.

[LRC*02] LUEBKE D., REDDY M., COHEN J. D., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., 2002.

[MCHM10] MARCHESIN S., CHEN C., HO C., MA K.: View-Dependent Streamlines for 3D Vector Fields. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010). doi:10.1109/TVCG.2010.212.

[MG11] MERRILL D., GRIMSHAW A.: High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters 21*, 02 (2011). doi:10.1142/S0129626411000187.

[Mic18] MICROSOFT: DirectX Raytracing (DXR) - Developer Blog. https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/, 2018. Accessed: 3

January 2026.

[Mic23] MICROSOFT CORPORATION: *Microsoft DirectX*. Microsoft, 2023. Accessed: 3 January 2026. URL: `https://learn.microsoft.com/en-us/windows/win32/directx`.

[MITR22] MALTENBERGER T., ILIC I., TOLOVSKI I., RABL T.: Evaluating Multi-GPU Sorting with Modern Interconnects. In *Proceedings of the 2022 International Conference on Management of Data* (2022), SIGMOD '22. `doi:10.1145/3514221.3517842`.

[MJL*13] MCLOUGHLIN T., JONES M. W., LARAMEE R. S., MALKI R., MASTERS I., HANSEN C. D.: Similarity Measures for Enhancing Interactive Streamline Seeding. *IEEE Transactions on Visualization and Computer Graphics 19*, 8 (2013). `doi:10.1109/TVCG.2012.150`.

[MKKP18] MÜNSTERMANN C., KRUMPEN S., KLEIN R., PETERS C.: Moment-Based Order-Independent Transparency. *Proceedings of ACM on Computer Graphics and Interactive Techniques 1*, 1 (2018). `doi:10.1145/3203206`.

[MLP*10] MCLOUGHLIN T., LARAMEE R. S., PEIKERT R., POST F. H., CHEN M.: Over Two Decades of Integration-Based, Geometric Flow Visualization. *Computer Graphics Forum 29*, 6 (2010). `doi:10.1111/j.1467-8659.2010.01650.x`.

[MN98] MATSUMOTO M., NISHIMURA T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998). `doi:10.1145/272991.272995`.

[Mor66] MORTON G. M.: *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Tech. rep., International Business Machines Corporation (IBM), New York, 1966.

[Mor13] MORELAND K.: A Survey of Visualization Pipelines. *IEEE Transactions on Visualization and Computer Graphics 19*, 3 (2013). `doi:10.1109/TVCG.2012.133`.

[MRG18] Meier R., Rigo A., Gross T. R.: Virtual Machine Design for Parallel Dynamic Programming Languages. *Proceedings of the ACM on Programming Languages 2*, OOPSLA (2018). doi:10.1145/3276479.

[MTHG03] Mattausch O., Theussl T., Hauser H., Gröller E.: Strategies for Interactive Exploration of 3D Flow Using Evenly-Spaced Illuminated Streamlines. In *Proceedings of the 19th Spring Conference on Computer Graphics* (2003), SCCG '03. doi:10.1145/984952.984987.

[MVP*18] Marinacci F., Vogelsberger M., Pakmor R., Torrey P., Springel V., Hernquist L., Nelson D., Weinberger R., Pillepich A., Naiman J., Genel S.: First Results from the IllustrisTNG Simulations: Radio Haloes and Magnetic Fields. *Monthly Notices of the Royal Astronomical Society 480*, 4 (2018). doi:10.1093/mnras/sty2206.

[ND03] Nienhaus M., Döllner J.: Edge-Enhancement - An Algorithm for Real-Time Non-Photorealistic Rendering. In *International Conference in Central Europe on Computer Graphics and Visualization* (2003). URL: https://api.semanticscholar.org/CorpusID:16175206.

[NL] NVIDIA-Labs: Cuda Unbound (CUB) Library. https://nvlabs.github.io/cub/. Accessed: 3 January 2026.

[NPG*15] Nelson D., Pillepich A., Genel S., Vogelsberger M., Springel V., Torrey P., Rodriguez-Gomez V., Sijacki D., Snyder G., Griffen B., Marinacci F., Blecha L., Sales L., Xu D., Hernquist L.: The Illustris Simulation: Public Data Release. *Astronomy and Computing 13* (2015). doi:https://doi.org/10.1016/j.ascom.2015.09.003.

[NPS*17] Nelson D., Pillepich A., Springel V., Weinberger R., Hernquist L., Pakmor R., Genel S., Torrey P., Vogelsberger M., Kauffmann G., Marinacci F., Naiman J.: First Results from the IllustrisTNG Simulations: the Galaxy Colour Bimodality. *Monthly Notices of the Royal Astronomical Society 475*, 1 (2017). doi:10.1093/mnras/stx3040.

[NPS*18] Naiman J. P., Pillepich A., Springel V., Ramirez-Ruiz E., Torrey P., Vogelsberger M., Pakmor R., Nelson D., Marinacci F., Hernquist L., Weinberger R., Genel S.: First Results from the IllustrisTNG Simula-

tions: a Tale of Two Elements – Chemical Evolution of Magnesium and Europium. *Monthly Notices of the Royal Astronomical Society 477*, 1 (2018). doi:10.1093/mnras/sty618.

[NVI93] NVIDIA CORPORATION:. https://www.nvidia.com, 1993. Santa Clara, California, USA, Accessed: 3 January 2026.

[NVI20] NVIDIA CORPORATION: CUDA Toolkit, 2020. Accessed: 3 January 2026. URL: https://developer.nvidia.com/cuda-toolkit.

[NVI24] NVIDIA CORPORATION: CUDA C++ Programming Guide: Unified Memory Programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#unified-memory-programming, 2024. Accessed: 3 January 2026.

[OPvdWC23] OSMAN B., PEREIRA M., VAN DE WETERING H., CHAMBERLAND M.: Voxlines: Streamline Transparency Through Voxelization and View-Dependent Line Orders. In *Computational Diffusion MRI: 14th International Workshop, CDMRI 2023, Held in Conjunction with MICCAI, Proceedings* (2023). doi:10.1007/978-3-031-47292-3_9.

[Par99] PARHAMI B.: *Introduction to Parallel Processing: Algorithms and Architectures*. Springer, 1999.

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: a General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (TOG) 29*, 4 (2010). doi:10.1145/1778765.1778803.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (TOG) 21*, 3 (2002). doi:10.1145/566654.566640.

[PF03] PASCUCCI V., FRANK R. J.: Hierarchical Indexing for Out-of-Core Access to Multi-Resolution Data. In *Hierarchical and Geometrical Methods in Scientific Visualization* (2003). doi:https://doi.org/10.1007/978-3-642-55787-3_14.

[PFHA10] Pantaleoni J., Fascione L., Hill M., Aila T.: PantaRay: Fast Ray-Traced Occlusion Caching of Massive Scenes. *ACM Transactions on Graphics (TOG) 29*, 4 (2010). doi:10.1145/1778765.1778774.

[PGL*12] Peng Z., Grundy E., Laramee R. S., Chen G., Croft N.: Mesh-Driven Vector Field Clustering and Visualization: An Image-Based Approach. *IEEE Transactions on Visualization and Computer Graphics 18*, 2 (2012). doi:10.1109/TVCG.2011.25.

[PJH16] Pharr M., Jakob W., Humphreys G.: *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., 2016.

[PK15] Peters C., Klein R.: Moment Shadow Mapping. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (2015), i3D '15. doi:10.1145/2699276.2699277.

[PL09] Peng Z., Laramee R. S.: Higher Dimensional Vector Field Visualization: A Survey. In *Theory and Practice of Computer Graphics* (2009), Tang W., Collomosse J., (Eds.). doi:10.2312/LocalChapterEvents/TPCG/TPCG09/149-163.

[PNH*17] Pillepich A., Nelson D., Hernquist L., Springel V., Pakmor R., Torrey P., Weinberger R., Genel S., Naiman J. P., Marinacci F., Vogelsberger M.: First Results from the IllustrisTNG Simulations: the Stellar Mass Content of Groups and Clusters of Galaxies. *Monthly Notices of the Royal Astronomical Society 475*, 1 (2017). doi:10.1093/mnras/stx3112.

[PPF*11] Pobitzer A., Peikert R., Fuchs R., Schindler B., Kuhn A., Theisel H., Matković K., Hauser H.: The State of the Art in Topology-Based Visualization of Unsteady Flow. *Computer Graphics Forum* (2011). doi:10.1111/j.1467-8659.2011.01901.x.

[PT92] Press J. W., Teukolsky S. A.: *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, 1992.

[PVH*03] Post F. H., Vrolijk B., Hauser H., Laramee R. S., Doleisch H.: The State of the Art in Flow Visualisation: Feature Extraction and Tracking. *Com-*

puter Graphics Forum 22, 4 (2003). doi:https://doi.org/10.1111/j.1467-8659.2003.00723.x.

[RC67] Robinson A., Cherry C.: Results of a Prototype Television Bandwidth Compression Scheme. *Proceedings of the IEEE 55*, 3 (1967). doi:10.1109/PROC.1967.5493.

[RGG20] Rojo I. B., Gross M., Günther T.: Fourier Opacity Optimization for Scalable Exploration. *IEEE Transactions on Visualization and Computer Graphics 26*, 11 (2020). doi:10.1109/TVCG.2019.2915222.

[RL00] Rusinkiewicz S., Levoy M.: QSplat: a Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (2000), SIGGRAPH '00. doi:10.1145/344779.344940.

[Roo99] Roosta S. H.: *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, New York, 1999.

[RPD20] Rapp T., Peters C., Dachsbacher C.: Void-and-Cluster Sampling of Large Scattered Data and Trajectories. *IEEE Transactions on Visualization and Computer Graphics 26*, 1 (2020). doi:10.1109/TVCG.2019.2934335.

[RS15] Ramaswamy R., Sbalzarini I. F.: Particle Methods. TU Dresden Lecture Notes, 2015. Accessed: 3 January 2026. URL: https://tu-dresden.de/ing/informatik/ki/wr/ressourcen/dateien/vorlesung/pm/script.pdf.

[RSHTE99] Rezk-Salama C., Hastreiter P., Teitzel C., Ertl T.: Interactive Exploration of Volume Line Integral Convolution Based on 3D-Texture Mapping. In *Proceedings of the Conference on Visualization* (1999), VIS '99. doi:10.1109/VISUAL.1999.809892.

[Rud16] Rudakova V.: GLSL shader for 3D Bezier curves with added fog effect (using OpenSceneGraph). https://github.com/vicrucann/shader-3dcurve/, 2016. Accessed: 3 January 2026.

[SA08] Sintorn E., Assarsson U.: Fast Parallel GPU-Sorting using a Hybrid Algorithm. *Journal of Parallel and Distributed Computing 68*, 10 (2008).

doi:https://doi.org/10.1016/j.jpdc.2008.05.012.

[SADK20] Sun Y., Agostini N. B., Dong S., Kaeli D.: Summarizing CPU and GPU Design Trends with Product Data, 2020. Accessed: 3 January 2026. URL: https://arxiv.org/abs/1911.11313.

[SFCN02] Suzuki Y., Fujishiro I., Chen L., Nakamura H.: Case Study: Hardware-Accelerated Selective LIC Volume Rendering. In *IEEE Visualization, VIS 2002* (2002). doi:10.1109/VISUAL.2002.1183811.

[SGH06] Schwamborn D., Gerhold T., Heinrich R.: The DLR TAU-Code: Recent Applications in Research and Industry. In *ECCOMAS CFD CONFERENCE* (2006). URL: https://elib.dlr.de/22421/.

[SGS10] Stone J. E., Gohara D., Shi G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering 12*, 3 (2010). doi:10.1109/MCSE.2010.69.

[SH95] Stalling D., Hege H.-C.: Fast and Resolution Independent Line Integral Convolution. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques* (1995), SIGGRAPH '95. doi:10.1145/218380.218448.

[SHG09] Satish N., Harris M., Garland M.: Designing Efficient Sorting Algorithms for Manycore GPUs. In *IEEE International Symposium on Parallel & Distributed Processing* (2009). doi:10.1109/IPDPS.2009.5161005.

[SHH*07] Schlemmer M., Hotz I., Hamann B., Morr F., Hagen H.: Priority Streamlines: A Context-Based Visualization of Flow Fields . In *Eurographics/ IEEE-VGTC Symposium on Visualization* (2007). doi:10.2312/VisSym/EuroVis07/227-234.

[SHZO07] Sengupta S., Harris M., Zhang Y., Owens J. D.: Scan Primitives for GPU Computing. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2007), GH '07. doi:10.2312/EGGH/EGGH07/097-106.

[SJC18] Singh D. P., Joshi I., Choudhary J.: Survey of GPU Based Sorting Algorithms. *International Journal of Parallel Programming 46*, 6 (2018).

doi:10.1007/s10766-017-0502-5.

[SK10] SANDERS J., KANDROT E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[SKKW23] SCHÜTZ M., KERBL B., KLAUS P., WIMMER M.: GPU-Accelerated LOD Generation for Point Clouds. *Computer Graphics Forum 42*, 8 (2023). doi: https://doi.org/10.1111/cgf.14877.

[SKW22] SCHÜTZ M., KERBL B., WIMMER M.: Software Rasterization of 2 Billion Points in Real Time. *Proceedings of ACM Computer Graphics Interactive Techniques 5*, 3 (2022). doi:10.1145/3543863.

[SLM25] SCHMEISSER J., LUTZ C., MARKL V.: Efficiently Indexing Large Data on GPUs with Fast Interconnects. In *Proceedings 28th International Conference on Extending Database Technology, EDBT* (2025). doi:10.48786/EDBT.2025.53.

[SMMO16] SATO H., MIZOTE R., MATSUOKA S., OGAWA H.: I/O Chunking and Latency Hiding Approach for Out-of-Core Sorting Acceleration Using GPU and Flash NVM. In *2016 IEEE International Conference on Big Data* (2016). doi:10.1109/BigData.2016.7840629.

[SOW20] SCHÜTZ M., OHRHALLINGER S., WIMMER M.: Fast Out-of-Core Octree Generation for Massive Point Clouds. *Computer Graphics Forum 39*, 7 (2020). doi:https://doi.org/10.1111/cgf.14134.

[SPCB22] SCHMID R. F., PISANI F., CÁCERES E. N., BORIN E.: An Evaluation of Fast Segmented Sorting Implementations on GPUs. *Parallel Computing 110*, C (2022). doi:10.1016/j.parco.2021.102889.

[ST69] SCHLICHTING H., TRUCKENBRODT E.: *Tragflügel Endlicher Spannweite bei Inkompressibler Strömung*. Springer Berlin Heidelberg, 1969, pp. 1–132. doi:10.1007/978-3-662-05619-6_1.

[Sta14] STALLINGS W.: *Operating Systems: Internals and Design Principles*, 8th ed. Pearson, 2014.

[SZD*23] Sarton J., Zellmann S., Demirci S., Güdükbay U., Alexandre-Barff W., Lucas L., Dischler J. M., Wesner S., Wald I.: State-of-the-Art in Large-Scale Volume Visualization Beyond Structured Data. *Computer Graphics Forum 42*, 3 (2023). doi:https://doi.org/10.1111/cgf.14857.

[Tel07] Telea A. C.: *Data Visualization: Principles and Practice*, 2nd ed. AK Peters/CRC Press, 2007.

[The97] The HDF Group: Hierarchical Data Format, Version 5. https://github.com/HDFGroup/hdf5, 1997. Accessed: 3 January 2026.

[TVJ*13] Tanasic I., Vilanova L., Jordà M., Cabezas J., Gelado I., Navarro N., Hwu W.-m.: Comparison Based Sorting for Systems with Multiple GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (2013), GPGPU-6. doi:10.1145/2458523.2458524.

[USM96] Ueng S.-K., Sikorski C., Ma K.-L.: Efficient Streamline, Streamribbon, and Streamtube Constructions on Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (1996). doi:10.1109/2945.506222.

[vBS23] van Beurden P., Scholz S.-B.: On Generating Out-Of-Core GPU Code for Multi-Dimensional Array Operations. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages (IFL '22)* (2023). doi:10.1145/3587216.3587223.

[VGS*14] Vogelsberger M., Genel S., Springel V., Torrey P., Sijacki D., Xu D., Snyder G., Nelson D., Hernquist L.: Introducing the Illustris Project: simulating the coevolution of dark and visible matter in the Universe. *Monthly Notices of the Royal Astronomical Society 444*, 2 (2014). doi:10.1093/mnras/stu1536.

[Wal04] Wald I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Accessed: 3 January 2026. URL: https://www.sci.utah.edu/~wald/PhD/wald_phd.pdf.

[Wei06] WEISKOPF D.: *GPU-Based Interactive Visualization Techniques*. Springer-Verlag, Berlin, Heidelberg, 2006.

[WGP97] WEGENKITTL R., GROLLER E., PURGATHOFER W.: Animating Flow Fields: Rendering of Oriented Line Integral Convolution. In *Proceedings. Computer Animation '97* (1997). doi:10.1109/CA.1997.601035.

[WHY*13] WANG R., HUO Y., YUAN Y., ZHOU K., HUA W., BAO H.: GPU-Based Out-of-Core Many-Lights Rendering. *ACM Transactions on Graphics (TOG) 32*, 6 (2013). doi:10.1145/2508363.2508413.

[WMC*24] WAGLEY B., MARKTHUB P., CREA J., WU B., BELVIRANLI M. E.: Exploring Page-Based RDMA for Irregular GPU Workloads: A Case Study on NVMe-Backed GNN Execution. In *Proceedings of the 16th Workshop on General Purpose Processing Using GPU (GPGPU '24)* (2024). doi:10.1145/3649411.3649413.

[Wol18] WOLFF D.: *OpenGL 4 Shading Language Cookbook: Build High-Quality, Real-Time 3D Graphics with OpenGL 4.6, GLSL 4.6 and C++17*, 3rd ed. Packt Publishing, 2018.

[WSE07] WEISKOPF D., SCHAFHITZEL T., ERTL T.: Texture-Based Visualization of Unsteady 3D Flow by Real-Time Advection and Volumetric Illumination. *IEEE Transactions on Visualization and Computer Graphics 13*, 3 (2007). doi:10.1109/TVCG.2007.1014.

[WWL16] WANG W., WANG W., LI S.: From Numerics to Combinatorics: a Survey of Topological Methods for Vector Field Visualization. *Journal of Visualization 19*, 4 (2016). doi:10.1007/s12650-016-0348-8.

[YF09] YANG W., FENG J.: Technical Section: 2D Shape Morphing via Automatic Feature Matching and Hierarchical Interpolation. *Computers & Graphics 33*, 3 (2009). doi:10.1016/j.cag.2009.03.007.

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-Time Concurrent Linked List Construction on the GPU. In *Proceedings of the 21st Eurographics Conference on Rendering* (2010), EGSR'10. doi:10.1111/j.1467-8659.2010.01725.x.

[YKP05] YE X., KAO D., PANG A.: Strategy for Seeding 3D Streamlines. In *IEEE Visualization* (2005). doi:10.1109/VISUAL.2005.1532831.

[YWSC12] YU H., WANG C., SHENE C.-K., CHEN J. H.: Hierarchical Streamline Bundles. *IEEE Transactions on Visualization and Computer Graphics 18*, 8 (2012). doi:10.1109/TVCG.2011.155.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree Construction on Graphics Hardware. *ACM Transactions on Graphics (TOG) 27*, 5 (2008). doi:10.1145/1409060.1409079.

[ZNA15] ZEIDAN M., NAZMY T., AREF M.: GPU-Based Out-of-Core HLBVH Construction. In *Eurographics Symposium on Rendering - Experimental Ideas & Implementations* (2015). doi:10.2312/sre.20151165.

[ZNZ*16] ZHENG T., NELLANS D., ZULFIQAR A., STEPHENSON M., KECKLER S. W.: Towards High Performance Paged Memory for GPUs. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016). doi:10.1109/HPCA.2016.7446077.

[ZPRD22] ZEIDAN M., PETERS C., RAPP T., DACHSBACHER C.: Versatile Geometric Flow Visualization by Controllable Shape and Volumetric Appearance. In *Smart Tools and Applications in Graphics - Eurographics Italian Chapter Conference* (2022), STAG' 22. doi:10.2312/stag.20221259.

[ZRPD20] ZEIDAN M., RAPP T., PETERS C., DACHSBACHER C.: Moment-Based Opacity Optimization. In *Eurographics Symposium on Parallel Graphics and Visualization* (2020), EGPGV' 20. doi:10.2312/pgv.20201072.