Institut für Informatik
Technische Universität München

Particle-based Flow Visualization

Kai Bürger

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende:	Univ.-Prof. G.J. Klinker, Ph.D.
Prüfer der Dissertation:	1. Univ.-Prof. Dr. R. Westermann
	2. Univ.-Prof. Dr. H. Theisel, Otto-von-Guericke-Universität Magdeburg

Die Dissertation wurde am 31.08.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.11.2010 angenommen.

To my family and friends.

Abstract

Due to technological and algorithmic advances it is by now possible to simulate complicated 3D unsteady flows at a very high spatial resolution. To find relevant features in such flows and, thus, to gain insight into the underlying flow phenomena, effective exploration mechanisms are needed. Especially interactive visual exploration environments are important, since they enable putting experts and their capabilities into the center of the exploration process. This requires new possibilities to interactively guide the exploration process by exploiting the humans' perceptual and cognitive abilities.

This thesis presents a framework for the visual exploration of 3D unsteady flow fields that addresses the aforementioned requirements. A variety of techniques for interactive flow visualization has been developed, consisting of novel concepts as well as extensions of existing geometry-, texture- and feature-based flow visualization techniques.

We introduce techniques that allow to track huge particle sets and to extract a large number of characteristic lines as well as to generate adaptive integral surfaces in real time. We present a variety of advanced rendering modalities that allow to encode additional flow properties into the extracted geometric representation and, thus, to communicate even more information in a single visual event.

Particle tracing in 3D can quickly overextend the viewer due to the massive amount of visual information that is typically produced by this technique. We alleviate this problem by presenting importance-driven visualization techniques that automatically reduce the amount of presented information to a detailed view on relevant features, while at the same time preserving context information.

Streak surface extraction is a prominent tool for interactive flow exploration. However, to enable a feature-driven analysis of the flow, one is mainly interested in surfaces that show separation profiles and, thus, detect unstable manifolds in the flow. We introduce a new method to interactively reveal such features by extracting Lagrangian coherent structures in a subregion of the flow domain and employing them as seeding

structures for the generation of generalized streak surfaces that reside on the boundary layers of dynamically coherent regions. This concept allows to study large scale transport behavior intuitively, as it reveals the evolution of the global flow geometry in real time.

Furthermore, we introduce new techniques for interactive surface flow visualization, discuss a variety of geometry-based visualization techniques for such fields and present a view-independent, dense surface flow representation on the basis of line integral convolution.

To achieve an interactive exploration environment for 3D unsteady flows, we introduce an asynchronous streaming strategy for a time-resolved sequence of flow fields and present parallelization strategies that effectively exploit graphics processing units as numerical co-processor. Feature extraction and the successive mapping to renderable primitives are executed entirely on the GPU to facilitate the real-time performance.

We conclude this manuscript with a brief digression into another field in scientific visualization, namely volume rendering. Here, we develop an interactive volume editing framework. We present techniques to directly manipulate or classify the underlying data and we employ particle tracing to compute a local iso-surface parametrization, which in turn is used for advanced volume rendering and illustration techniques.

As all approaches presented in this work rely on consumer class hardware, their application is available to a wide range of users. The presented techniques allow scientists to effectively explore scientific data sets interactively, thus giving rise to new possibilities to gain insight in and communicate the findings of complex phenomena.

Zusammenfassung

Dank technologischer und algorithmischer Fortschritte ist es heutzutage möglich, komplizierte instationäre 3D Strömungen in einer äusserst hohen räumlichen Auflösung zu simulieren. Um relevante Strukturen in solchen Strömungen zu entdecken und damit Einsicht in die zugrunde liegenden Phänomene zu erlangen, werden effektive Explorationsmechanismen benötigt. Hierbei sind besonders interaktive visuelle Explorationsumgebungen wichtig, da sie es erlauben, Experten samt Ihrer Auffassungsgabe in den Mittelpunkt des Explorationsprozesses zu stellen. Dies verlangt jedoch nach Möglichkeiten den Explorationsprozess interaktiv, durch Ausnutzung der menschlichen Wahrnehmungs- und kognitiven Fähigkeiten, zu steuern.

Diese Dissertation stellt ein Framework vor, das die interaktive visuelle Exploration instationärer 3D Strömungsfelder ermöglicht und somit obige Anforderungen erfüllt. Es wurde eine Vielfalt interaktiver Strömungsvisualisierungstechniken entwickelt, bestehend aus neuen Konzepten, sowie Erweiterungen bereits existierender Ansätze, in den Bereichen der Geometrie-, Textur- und Feature-basierten Strömungsvisualisierung.

Wir stellen Techniken vor, die es erlauben, grosse Mengen von Partikeln interaktiv zu verfolgen sowie charakteristische Teilchentrajektorien und adaptive Integralflächen in Echtzeit zu extrahieren. Wir präsentieren eine Auswahl unterschiedlicher Visualisierungsmodalitäten, die zusätzliche quantitative Information intuitiv vermitteln können.

Die Partikelverfolgung in 3D tendiert dazu, aufgrund der schieren Flut erzeugter visueller Information, die menschliche Wahrnehmung zu überlasten. Wir lindern dieses Problem durch die Einführung neuer *focus+context* Techniken, welche automatisch die Menge präsentierter visueller Information reduzieren, dabei jedoch wichtige Zusammenhänge erhalten können.

Streichflächen sind ein bedeutendes Werkzeug der interaktiven Strömungsvisualisierung. Um jedoch eine Feature-getriebene Analyse der Strömung zu ermöglichen, sollten möglichst separierende Flächen extrahiert werden, die instabile Mannigfaltig-

keiten aufdecken. Wir präsentieren ein neues Verfahren, das mit Hilfe der Lagrangeschen Teilchendynamik—und Prinzipien der Morsetheorie—kohärente Strukturen aus einem Unterbereich des Strömungsfeldes extrahiert. Solch detektierte Strukturen werden daraufhin als Partikel-Saatstrukturen zur Generierung generalisierter Streichflächen verwendet, die in den Grenzbereichen dynamisch kohärenter Regionen liegen und somit besonders gut dazu geeignet sind das globale Transportverhalten von instationären Strömungen in Echtzeit zu untersuchen.

Des weiteren wird eine neue Technik präsentiert, die eine interaktive Partikelverfolgung in instationären Oberflächenströmungen ermöglicht sowie darauf aufbauende Geometrie- und Textur-basierte Visualisierungsansätze vorgestellt.

Um eine interaktive Explorationsumgebung für instationäre 3D Strömungen zu verwirklichen, wurde eine asynchrone Streaming-Technik für zeitaufgelöste Sequenzen von Strömungsfeldern enwickelt und die den Visualisierungstechniken zugrunde liegenden Algorithmen effizient, unter Einsatz von Grafikhardware, parallelisiert. Moderne Grafikkarten bieten die Möglichkeit grosse Datenmengen massiv parallel zu verarbeiten und sind somit besonders gut für Partikelbasierte Ansätze geeignet. Weiterhin hat der Einsatz von Grafikhardware zur Strömungsvisualisierung den Vorteil, dass extrahierte Strömungsmerkmale bereits im lokalen Speicher liegen und somit direkt auf darstellbare Primitive abgebildet werden können, ohne zusätzliche Datenzugriffe auf den Hauptspeicher des Systems zu benötigen.

Wir unternehmen am Ende dieser Arbeit einen kurzen Ausflug in ein weiteres wissenschaftliches Visualisierungsgebiet, die Volumenvisualisierung. Hier stellen wir Techniken vor, die eine benutzergestützte Klassifikation und Segmentierung von volumetrischen Skalarfeldern ermöglichen, und wir zeigen, wie die Partikelverfolgung in solchen Datensätzen eingesetzt werden kann, um eine lokale Parametrisierung von Isoflächen zu berechnen.

Alle Techniken, die in dieser Arbeit vorgestellt werden, benötigen lediglich Standard-PC Hardware und stehen somit einer grossen Gruppe von Benutzern zur Verfügung. Die präsentierten Techniken ermöglichen es Wissenschaftlern effektiv und interaktiv wissenschaftliche Datensätze zu explorieren (oder anzureichern) und führen somit zu neuen Möglichkeiten, Einsicht in komplexe volumetrische Erscheinungen zu erlangen sowie gewonnenes Wissen untereinander zu kommunizieren.

Acknowledgements

In 2003 Prof. Westermann joined the Technische Universität München and became the chair of the computer graphics and visualization group. While still being an undergraduate student at that time, I started to attend his lectures which rapidly aroused my interest in computer graphics and visualization related topics. I started to participate in practical courses and seminars offered by the chair and soon became even more attracted to the group due to my supervisors' friendliness and keenness to discuss. After finishing my Diploma thesis under the supervison of Dr. Jens Krüger in 2006—which lead to my first publication as an undergraduate student—I asked Prof. Westermann about the possibilities to join his chair and was happy to be offered a position in his group. The last four years shaped my being not only as a scientist but also as a person, and I am grateful for the opportunity I have been given.

First and most of all I wish to thank my advisor Rüdiger Westermann not only for providing me the opportunity to conduct the research presented in this thesis, but especially for being my biggest inspiration and helping me in evolving as a scientist. Rüdiger was always open for discussion and inspired many of the methods presented here. Without his encouragement this thesis would definitely not have been possible.

A special thanks also goes to my former colleague Jens Krüger with whom I spent a lot of time in close collaboration and who tremendously supported this work. Also, I wish to thank my current and former colleagues, namely Stefan Auer, Matthäus Chajdas, Christian Dick, Roland Fraedrich, Raymund Fülöp, Joachim Georgii, Stefan Hertel, Polina Kondratieva, Martin Kraus, Hans-Georg Menz, Tobias Pfaffelmoser, Thomas Schiwietz, Jens Schneider and Marc Treib. They have always been keen to discuss ideas and supported me in proof-reading this thesis.

I would also like to thank Prof. Holger Theisel and his visual computing group at the university of Magdeburg. Holger contributed many ideas to my recent work on interactive flow visualization and his group provided additional real-world data sets to validate techniques presented in this thesis. Thanks also go to my student Florian Ferstl for implementing and validating two methods presented here.

Last but not least, I would like to thank my family and friends for their ever ongoing support. I would like to send special thanks to my mother Elle. Besides the fact of being a single mother, she managed to fund my undergraduate studies and always supported me by any means and in any form. Without her, my academic career would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Flow fields play an important role in a wide range of scientific and industrial areas. Just to give a few examples: in fluid dynamics, flow fields are of special interest in the study of gases and liquids in motion to understand the transport behavior around obstacles or in intermixing processes. In medicine (or biology in general), flow fields are investigated to learn about basic processes appearing inside living organisms. Even in fundamental research, the evolution of dynamical system is often studied by observing the flow of interdependent parameters along trajectories in phase space.

Particle tracing is a standard tool employed in the study of such fields, and the history of its application ranges back to the first attempts in fluid flow research itself. Before the advent of non-intrusive flow measurement, computer aided reconstruction and visualization techniques, real-world flow exploration based on the observation of patterns revealed by the movement of tracer material injected into a flow was the only way to shed light on internal phenomena. Even today, it is a common technique to release nearly massless materials (such as dye, hydrogen bubbles or heat energy) into a flow and to visually track their temporal evolution.

While the direct observation of real-world flow is still of practical relevance, it can only deliver a qualitative description of flow phenomena. Detailed dynamics and precise mechanisms underlying the evolution of specific features remain rather unknown. Furthermore, “global” flow phenomena are often governed by the interaction of chaotically appearing, interacting and disappearing features (such as vortices and eddies) at a large range of scales. However, the human perception system is often overstrained in detecting such features due to visual clutter introduced by large amounts of particles performing rapid directional changes. Yet, there is no way to slow down or halt a real-world experiment under investigation and, thus, to study such features in detail.

Due to technological advances over the last decades, nowadays, it is possible to measure physical flow properties (such as velocity, density or viscosity) at a high spatio-temporal resolution and, thus, to reconstruct a quantitative flow evolution. In general, flow field measurement techniques (such as particle image velocimetry) also rely on particle motion. They record the evolution of particles over time and reconstruct a discretized time-resolved velocity vector field by matching particle sets in successively obtained snap-shots.

Other scientific areas, such as computational fluid dynamics, generate digital flow data through numerical simulations. Here, physically plausible approximations of real world flow are developed with the objective to verify fundamental theoretical models. Lagrangian models (such as smoothed particle hydrodynamics) again rely on a particle metaphor to develop equations describing the fluid dynamics.

Due to advances in flow reconstruction techniques and thanks to increasing numerical capabilities, today, digitalized 3D unsteady flow fields comprise billions of samples and the sheer amount of information renders it impossible to gain insight into complex flow phenomena through statistical analysis of the resulting data. Thus, effective techniques have to be developed to filter the information, presenting the observer an intuitive insight on the data under investigation.

Scientific visualization is the field of research associated with the question of how to map information represented in the form of numbers to visual representations. *Flow visualization* is the subarea of scientific visualization dedicated to the visual investigation of flow phenomena. Over the last decades a variety of different flow visualization classes has been developed. However, 3D unsteady flow fields have moved only recently into the focus of flow visualization and while for 2D unsteady and stationary 3D flows many interactive techniques exist, here, this aspect has barely been tapped. Yet, interactivity is of special interest in the unsteady case due to the following reasons:

Firstly, preserving the time axis as an important feature of the data set is advantageous as the evolution of flow dynamics can be comprehended most intuitively if it is visualized in a time-dependent context. Yet, to grasp the time-correlation between flow features, it is important to extract and display them in real time. An animated visualization clearly communicates the dynamics of extracted flow features and the motion parallax provides an excellent depth cue, thus, easing to understand the spatial correlation between features interacting over time.

Secondly, occlusion is a problem inherent to the visual study of 3D phenomena. Thus, it has been proven worthwhile to incorporate user guidance into the steering of the visual data analysis process. Occlusion problems can be alleviated by restricting the

visualization of phenomena to subregions of the flow domain. Yet, to determine regions of interest efficiently, features need to be extracted and displayed in real-time. Furthermore, if different views on the data set can be generated rapidly, the self-obstruction of features in the focus region can be solved effectively through camera interaction. Moreover, the most suitable visualization modality for the phenomena under investigation can be selected interactively.

Thirdly, interactive visualization techniques give scientists the possibility to immediately examine how changes to computational or experimental parameters affect the flow phenomenon under investigation. This is a highly desired goal according to the observations made by McCormick [114].

This thesis focuses on the development of interactive visual exploration techniques for 3D (unsteady) flow. The underlying velocity field and quantities derived thereof are of utmost importance in the study of flow as they reveal its dynamics. Hence, the approaches presented in this manuscript adhere to the concept of vector calculus.

The most fundamental building block of the algorithms proposed in the following is Lagrangian particle tracing. We employ this paradigm to approximate line integrals and, thus, to extract geometry- and texture-based flow representations. We will present new concepts for these visualization classes as well as new approaches for existing methods that allow to obtain such representations interactively even in large 3D unsteady flow fields.

We apply differential operators on the velocity vector field to derive further flow measures, and we incorporate these quantities into the flow visualization process. We encode such quantities not only into the visual representation to convey additional information, but we also employ them as importance measure during feature extraction to automatically reveal relevant flow features while at the same time preserving context information.

Furthermore, we will combine differentiation and integration to derive Lagrangian flow quantities which are then used to extract feature-based flow representations. As this class of methods is generally not suited for an interactive exploration environment (due to the necessary intense pre-processing), we will show how aspects of feature-based flow visualization techniques can be efficiently combined with geometry-based approaches to effectively reveal global flow phenomena in real time.

To achieve an interactive flow exploration environment, algorithms underlying the techniques presented in the following have been tailored for parallel execution on graphics processing units (GPU). GPUs have been introduced rather recently to the mainstream market and have developed rapidly from simple analogue/digital converters into

full-fledged parallel stream processors that are almost freely programmable. Latest models have a raw-performance of more than 1 teraflop at *IEEE* floating point single-precision and, thus, outperform even state-of-the-art multi-core CPU architectures by far. Even though they are still dedicated graphic chips that require programmers to follow certain paradigms tailored to the needs of real-time computer graphics and games, they present a cost-efficient alternative—on average less than 500 $ for the current high-end models—to supercomputers in terms of numerical processing power.

Next to the raw computational power, which facilitates the real-time extraction of flow features, executing visualization algorithms purely on the GPU yields another advantage. Data generated on a GPU resides in local video memory and can be rendered immediately without the need to communicate data between the CPU and GPU, thus, omitting potential bandwidth bottlenecks in the bus interface between the CPU, main memory and graphics hardware.

Moreover, the techniques presented in the following can be executed on commodity PC hardware and can, thus, be integrated into the modern scientists workflow at a reasonable cost expenditure. The suitability of our proposed methods has been approved by experts from various research areas as well as in two international contests.

1.1 Contribution

We now give an overview of the following chapters. To achieve a self-contained manuscript, we will start with two chapters providing basic knowledge to prepare the reader unfamiliar with the research topics covered by this thesis.

First, we will introduce fundamental terminology with respect to flow fields and fluid dynamics, motivate the field of scientific flow visualization and present existing classes therein. Furthermore, we introduce fundamental theoretical concepts for flow investigation and basic methods from mathematics used throughout this thesis.

As the presented techniques exploit the numerical capabilities of GPUs to attain an interactive visual flow exploration environment, we will continue with a broad overview on the development of graphics hardware architectures in Chapter 3 and introduce the rendering pipeline –i.e the underlying concepts to program on such hardware– in close relation to the graphics API employed in the validation of the proposed methods.

The remaining chapters discuss the academic contributions developed in the course of this dissertation and are closely related to work published in a series of peer-reviewed research papers:

Chapter 4 describes how the particle tracing paradigm can be mapped efficiently onto the GPU's rendering pipeline and presents a streaming approach for time-resolved sequences of 3D unsteady flow fields. It exploits multi-core CPU architectures to decouple the visualization from data handling and, thus, facilitates the real-time exploration of such data sets. We discuss fundamental particle-based rendering techniques and present new strategies to extract and visualize time-dependent characteristic lines interactively on the GPU. To emphasize the spatial relationship between flow structures and boundaries of the flow domain, we introduce focus+context visualization techniques for polygonal models. Work presented in this chapter has been developed in collaboration with Jens Schneider, Polina Kondratieva, Jens Krüger and Rüdiger Westermann and was published in [27]. Moreover, the presented concepts have been validated in the visual analysis of the Terashake 2.1 earthquake simulation data in line with the IEEE Visualization Contest 2006 [5].

Chapter 5 introduces importance driven particle techniques for flow visualization. Particle tracing in 3D quickly overextends the viewer due to the massive amount of visual information produced by this technique. Thus, this chapter focuses on strategies to automatically reduce the amount of information presented to the user while at the same time revealing important structures in the flow. We introduce an effective clustering approach for vector fields which in turn is used to generate a sparse set of static primitives depicting regions of constant motion in the flow. We employ scalar flow quantities at different scales in combination with user-defined regions of interest to control the shape, appearance and density of particles so that the user can focus on the dynamics in important regions while at the same time preserving context information. We introduce a new focus for particle tracing, so called anchor lines. These lines can be used to analyze local flow features by visualizing how much particles separate over time. This is of particular interest if the finite-time Lyapunov exponent is used to guide the placement of anchor lines. Work presented in this chapter was published in collaboration with Polina Kondratieva, Jens Krüger and Rüdiger Westermann in [24].

In Chapter 6, we present techniques for the visualization of unsteady flows using integral surfaces. We introduce new GPU-based algorithms to generate and display adaptively refined streak surfaces. Two different approaches to generate such surfaces are presented. The first approach computes a patch-based surface representation that avoids any interdependence between patches. Thus, the surface construction stage can be parallelized entirely but requires advanced rendering techniques to result in a closed surface representation. The second approach computes a particle based surface representation with particle connectivity. To preserve particle interdependence during adap-

tive refinement and coarsening, a multi-pass construction technique is employed which results in a closed surface representation that can be rendered outright as a triangle mesh. Techniques presented in this chapter allowed for the first time the construction and visualization of adaptive streak surfaces in real time and were published in collaboration with Florian Ferstl, Holger Theisel and Rüdiger Westermann in [22].

Chapter 7 presents a novel approach that extracts separating streak surfaces in 3D unsteady flow at interactive rates and, thus, facilitates a visually guided flow exploration based on the concept of Lagrangian coherent structures (LCS). Such structures confine regions of coherent dynamics and are generally of interest in the study of global transport behavior. This approach avoids computing LCS in 3D, i.e. 2D FTLE ridges. Instead, LCS computations are restricted to a 2D manifold in the flow domain. We present techniques to compute the FTLE on a planar probe interactively and introduce a new 1D ridge extraction method that is specifically tailored to the GPU. The extracted ridges are then employed as seeding structures for a generalized streak surfaces integration to reveal separating structures in the flow. The work presented in this chapter has been developed in collaboration with Florian Ferstl, Holger Theisel and Rüdiger Westermann and was published in [42].

New techniques for the visualization of flow on surfaces will be presented in Chapter 8. We introduce the Orthogonal Fragment Buffer (OFB), a sample-based data structure used to represent arbitrary surfaces. We will show how geometry- and texture-based flow visualization techniques can be applied to this GPU-friendly data structure to efficiently reveal surface flow phenomena. We present advanced rendering approaches which employ the OFB to solve rendering issues inherent to geometry-based surface flow visualization techniques. Moreover, we will present various new rendering modalities for geometric surface flow representations. Additionally, we will use the OFB to create a view-independent texture-based flow visualization on the basis of line integral convolution. On the one hand, we employ LIC to visualize flow fields living on a 2D manifold. On the other hand, we compute LIC in 3D (unsteady) flow but restrict the extraction and visualization to arbitrarily shaped clip geometry positioned in the flow domain. Work presented in this chapter has been developed jointly with Jens Krüger and Rüdiger Westermann and was published in [26].

In Chapter 9, we make an excursion into another field of scientific visualization, namely volume rendering. Here, we will show how the presented GPU-based concepts can be applied to develop a framework for interactive volume editing. We introduce a volumetric paint metaphor that can be used for a user-guided classification and segmentation of 3D scalar fields, as well as interactive volume illustration. We employ

particle tracing to place internal annotations on extracted iso-surfaces and we extend this technique to realize surface aligned cutaway-views. Such shape-aligned windows can be employed to effectively reveal internal surface structures. All the techniques underlying this framework have been developed in collaboration with Jens Krüger and Rüdiger Westermann and were published in [25].

Finally, we summarize the topics covered by this thesis and give interesting directions for future research work.

1.2 Research Publication Summary

This thesis does not cover all the research conducted in the course of my dissertation. Several publications, containing mainly research on computer graphics related topics, have been omitted. For the sake of completeness, this section provides a list of all academic research papers published during my work as a PhD student. My dissertation is supported by the following peer reviewed publications (listed in chronological order):

1. *Interactive Screen-Space Accurate Photon Tracing on GPUs*: Jens Krüger, Kai Bürger, Rüdiger Westermann; in Rendering Techniques, Eurographics Symposium on Rendering 2006 [89].

2. *Interactive Visual Exploration of Instationary 3D-Flows*: Kai Bürger, Jens Schneider, Polina Kondratieva, Jens Krüger, Rüdiger Westermann; in Proceedings of Eurographics/IEEE VGTC Symposium on Visualization 2007 [27].

3. *GPU Rendering of Secondary Effects*: Kai Bürger, Stefan Hertel, Jens Krüger, Rüdiger Westermann; in Proceedings of Vision, Modeling and Visualization 2007 [23].

4. *Importance-Driven Particle Techniques for Flow Visualization*: Kai Bürger, Polina Kondratieva, Jens Krüger, Rüdiger Westermann; in Proceedings of IEEE VGTC Pacific Visualization Symposium 2008 [24].

5. *Direct Volume Editing*: Kai Bürger, Jens Krüger, and Rüdiger Westermann; in Proceedings of IEEE Transactions on Visualization and Computer Graphics 2008 [25].

6. *Interpolating and Downsampling RGBA Volume Data*: Martin Kraus, Kai Bürger; in Proceedings of Vision, Modeling, and Visualization 2008 [86].

7. *Real-Time Approaches for Model-Based PIV and Visual Fluid Analysis*: Polina Kondratieva, Kai Bürger, Joachim Georgii, Rüdiger Westermann; in the book Imaging Measurement Methods for Flow Analysis, Series: Notes on Numerical Fluid Mechanics and Multidisciplinary Design [85].

8. *Interactive Streak Surface Visualization on the GPU*: Kai Bürger, Florian Ferstl, Holger Theisel, Rüdiger Westermann; in Proceedings of IEEE Transactions on Visualization and Computer Graphics 2009 [22].

9. *Sample-based Surface Coloring*: Kai Bürger, Jens Krüger, Rüdiger Westermann; IEEE Transactions on Visualization and Computer Graphics journal 2009 [26].

10. *Interactive separating streak surfaces*: Florian Ferstl, Kai Bürger, Holger Theisel, Rüdiger Westermann; in Proceedings of IEEE Transactions on Visualization and Computer Graphics 2010 [42].

Chapter 2

Flow Visualization Fundamentals

Instead of delving right into details of our developed techniques, we dedicate this chapter to clarify terminology used throughout the work, introduce the basic building blocks of the visualization pipeline and present a classification for various different visual flow exploration approaches. We will finalize this chapter with an introduction to basic mathematical methods and a discussion of fundamental theoretical building blocks employed throughout this work. The intention of this chapter is to deliver a self-contained manuscript. Readers unfamiliar with the field of visual flow exploration will be given a brief introduction to the most important areas—wrt. the work presented in this thesis.

2.1 Flow Field Terminology

As the term *flow field* is interpreted differently in various areas of science, we will first clarify the terminology used throughout this thesis. In the majority of cases, the term *flow* is related to the properties of a moving *fluid*. In various technical sciences, all gases, liquids and plasma are considered to be fluids and they represent matter for which even small externally applied forces cause a deformation of the underlying molecular structure. *Fluid flows* can be classified into different categories based on inherent physical properties. E.g., the *density*, the *viscosity* and the *velocity* are quantities of particular interest in the study of fluid dynamics. Each of these quantities can be used to categorize fluid flows into distinct classes as listed in the following:

- Density: Based on the behavior of the fluid density, flows can be classified into two categories, namely *compressible* and *incompressible* flow. If the fluid density stays constant under all conditions, the flow is considered incompressible. In general, all liquids as well as gases moving at slow velocities are presumed to be

incompressible. However, if changes in pressure affect the density of a fluid, it is considered to be compressible. Compressible fluid flows are commonly subject to the study of phenomena at supersonic (or hypersonic) velocities.

- Viscosity: The viscosity of a fluid is the property abstracting frictional forces acting within the flow. It is a measure of the resistance of a fluid being deformed by shear stress or tensile stress. Thus, it describes a fluid's resistance to "internal flow". If internal frictional forces are very strong, a fluid is called *viscous*. Fluids having no resistance to shear stress are known as an ideal fluid and are classified as *inviscid*. Gases are commonly regarded as inviscid fluids, whereas liquids such as oil or syrup are considered highly viscous.

- Velocity: The flow velocity, or more precisely the ratio of acting inertial forces to viscous forces, is generally used to categorize flows into *creeping*, *laminar* or *turbulent* regimes. In fluid mechanics, the Reynolds number *Re* is a dimensionless number measuring this ratio and it is generally employed to perform the classification.

  Creeping flow occurs at low Reynolds numbers ($Re \ll 1$) and it is dominated by viscous and acting pressure forces. This regime takes place in flow experiments conducted at microscopic scales, e.g. the swimming of microorganisms or in the flow of viscous polymers in general.

  Laminar flow occurs at increased velocities and is characterized by a wide range of Reynolds numbers. Within this regime, a fluid flows in parallel layers (without disruption between the layers) and it can continue to move even further due to internal inertia forces although external forces cease to exist. Creeping flow can be considered an extreme case of laminar flow where viscous effects are much greater than inertial forces.

  Turbulent flow occurs at even higher velocities and it is characterized by chaotic property changes, e.g., rapid variations of pressure and velocity in space and time. Turbulent flow is dominated by inertial forces and, consequently, frictional forces can be disregarded. Random and instable flow patterns are inherent to this regime, unsteady eddies and vortices appear on many scales and interact with each other. The flow conditions in industrial equipment (such as pipes or ducts) correspond to the turbulent flow regime. Additional examples for turbulent flow are wind-tunnel experiments (studying the external flow over all kind of obstacles such as cars or airplanes) or the mixing of warm and cold atmospheric layers.

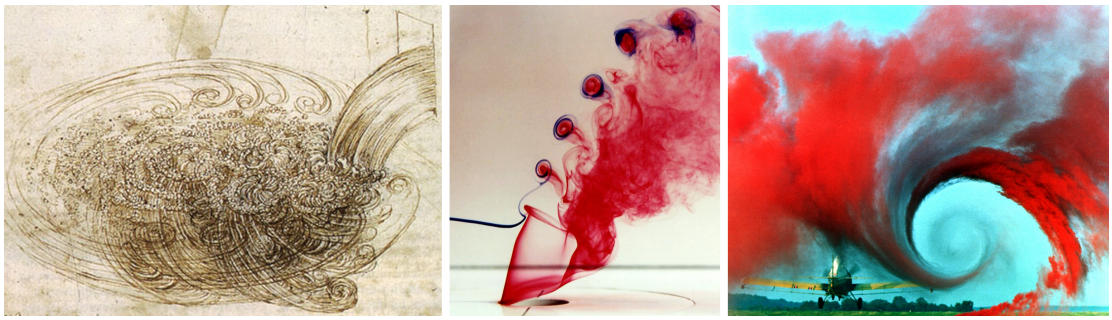


Figure 2.1: *Left: A sketch by Leonardo Da Vinci based on flow observations (Image under the Wikimedia Commons licence). Middle: Dye in water visualizes a round jet in a cross flow (Image courtesy of T.T. Lim, NUS [74]). Right: Smoke injection reveals a wake vortex behind a starting airplane (image courtesy of NASA [29]).*

In fluid mechanics the term flow field is commonly defined from the Eulerian point-of-view as the collection of all properties of a fluid defined over the whole spatio-temporal domain [128]. The collection of flow properties contains scalars (pressure, density, viscosity), vectors (velocity and acceleration) as well as tensor values (stress and strain).

Another common definition of the term flow stems from the earliest approaches to understand phenomena appearing within fluids under the influence of external forces. First scientific attempts to understand the internal structural behavior of moving liquids are based on the induction of substances—consisting of small, nearly "massless" particles, so called tracers—into the liquid and the observation of their trajectories. The advection of particles in currents within the flow sheds light on the local velocity magnitude and direction of matter. Moreover, the observation of developing patterns by the deformation of tracer material can reveal transport behavior at different scales and is, thus, well suited to gain insight into the global flow geometry. Figure 2.1 depicts three examples for flow phenomena revealed by the movement of particle tracers. From this perspective, the term flow denotes the motion appearing within a fluid (or fluid dynamics). The fluid motion in advection is described mathematically as a vector field, thus, within the context of particle tracing the term flow field is commonly related to the underlying *velocity vector field*.

Since this thesis focuses on the visualization of flow phenomena on the basis of the particle tracing paradigm, we relate to the second definition—namely velocity vector fields and eventually quantities derived thereof—whenever we will refer to flow fields in the following.

2.2 The Visualization Pipeline

The *visualization pipeline* describes the whole process of creating visual representations for scientific data. As shown in Figure 2.2 it can be broken down into four stages, starting with the generation of data and ending with a visual representation. In the following we will describe its stages in strong relation to flow visualization.

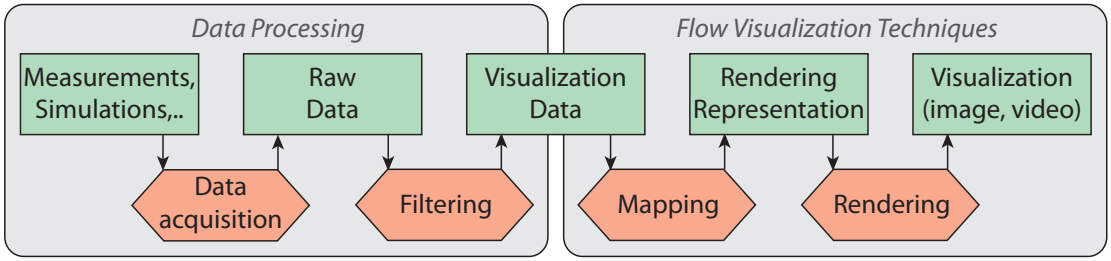


Figure 2.2: *The (Flow) Visualization Pipeline: Starting with information obtained through experimental measurements or numeric simulations, the data (green) traverses four processing stages (red) until results in the form of a visual representation are obtained.*

The pipeline starts with the *data acquisition*. With respect to the visualization of fluid flow, data is either acquired through experimental flow measurements or numerical flow simulations. The *raw data* is usually not appropriate for visualization. Only parts of the raw data might be of interest during the visual exploration process or a distinct data format is required. The *filtering* stage prepares the data for visualization. It performs tasks like clipping, segmentation or resampling to bring the data into the desired format, and it usually reduces the amount of data being processed in successive stages. Common operations in this stage employ smoothing and interpolation algorithms to determine missing values or to correct erroneous samples and result in the *visualization data*. In the *mapping* stage the input data is mapped to renderable primitives. E.g., particles are advected in the velocity field and visualized as point primitives, or a differentiation on the vector field is performed to detect certain features that are mapped to iconic shapes. The *rendering* stage performs the last step in the visualization pipeline by projecting the renderable primitives onto an image that is finally presented to the user.

2.2.1 Data Acquisition

In the context of flow visualization, the data is usually either acquired through physical flow measurements of real-world experiments or given as the result of numerical flow simulations. Flow data sets comprise multifield data representing scalar quantities like density, vector quantities like velocity or even tensor values like stress and strain.

Flow Measurement and Reconstruction

A common class of techniques to measure real-world flow is called *Pulsed-Light Velocimetry* (PLV). *Particle Image Velocimetry* (short PIV) is subject to this class and follows principles of photography to reconstruct a flow velocity vector field. PIV is a non-intrusive PLV technique that evaluates the displacement of particle tracers within a certain time interval Δt to construct an instantaneous velocity map of the whole flow domain. The PIV technique requires a laser—illuminating particles moving along the flow—and a camera recording at least two images at successive time steps t and $t + \Delta t$. These images are then divided into tiles of uniform flow movement and each tile will result in one velocity vector representative for the whole tile area. Each tile, commonly called interrogation window, contains a discretized function in the form of per-pixel intensity values of light scattered by particle tracers in the flow. Cross-correlation is then applied to find the matching function of an interrogation window in the successive PIV image and the velocity vector is given by the shift that is necessary to translate a function in the PIV image captured at time t to its position in the image captured at $t + \Delta t$. Two or multiple cameras and additional registration tasks are required to reconstruct 3D flow velocity fields. Besides cross-correlation, optical flow techniques or model-based approaches are commonly applied to reconstruct the velocity field. For a thorough overview on real-world flow measurement and reconstruction techniques, we refer the reader to [84].

Computational Fluid Dynamics

Over the last two centuries, scientists have developed physically plausible sets of equations describing the motion of fluids, e.g., the Euler and Navier-Stokes equations. The Navier-Stokes equations consist of a set of partial differential equations (PDEs) describing the motion of fluids based on the motivation that changes in momentum are the product of changes in pressure and dissipative viscous forces acting within the fluid. However, so far the existence of a closed form solution of the Navier-Stokes equations has not been discovered [41] and the only way to solve these equations is by means of numerical methods. Computational fluid dynamics solve the Navier-Stokes equations in numerical simulations. A detailed introduction to flow simulations is beyond the scope of this chapter, as this thesis focuses on the visual exploration of existing data. For a thorough overview on computational fluid dynamics, we refer the reader to [54]. Turbulent flow produces fluid interaction at a large range of length scales. Different solution methods exist, varying in the approach taken to abstract turbulent motion at

small scales. In the course of this thesis, various flow fields obtained with the following methods have been employed to validate the usefulness of the presented techniques:

- *Direct Numerical Simulations (DNS)*: In this technique the Navier-Stokes equations are resolved at all relevant scales of turbulent motion, so no (turbulence) model is needed for the smallest scales. This makes DNS simulations the computational most expensive, both in terms of memory consumption and arithmetic operations. The simulation resolution depends on the Reynolds number *Re*. The number of arithmetic operations required to complete the simulation is proportional to the number of spatial samples and the number of time steps, and in conclusion, the number of operations grows as Re^3. For the average Reynolds numbers encountered in industrial applications, the computational resources required by a DNS would even exceed the capacity of the largest super computers. Hence, DNS simulations are barely used in practice but are often subject to fundamental research at small scales.

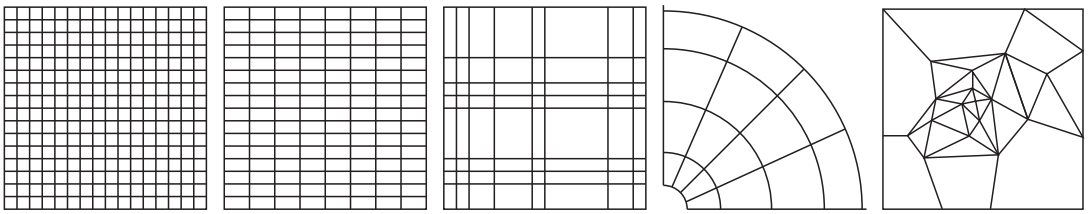
- *Reynolds-averaged Navier-Stokes Equations ((U)RANS)*: This technique is primarily used while dealing with turbulent flow, and it is based on time-averaged equations of motion for fluid flow. Through Reynolds decomposition, flow variables (like velocity) are separated into a mean (time-averaged) component and a fluctuating component. An ensemble version of the governing equations is solved, which introduces new apparent stresses. The so called Reynolds stress is a nonlinear stress term that requires additional modeling to close the RANS equation for solving. (Unsteady) RANS simulations employ turbulence models and resolve only unsteady mean-flow structures.

- *Large Eddy Simulation (LES)*: This method requires less computational effort than DNS but more effort than RANS methods. LES simulations calculate only the large scale motions of a flow. Effects on sub-grid scales are modeled using a so called sub-grid scale (SGS) model. An SGS term, which is commonly defined by the Smagorinsky model [158], is added to filtered Navier-Stokes equations. Unresolved turbulence scales are compensated by the addition of an *eddy viscosity* into the governing equations. The main advantage of LES over computationally cheaper RANS approaches is the increased level of detail it can deliver. RANS simulations provide ”averaged” results, whereas LES simulations are able to predict instantaneous flow characteristics and resolve turbulent flow structures.

2.2.2 Filtering

The filtering stage is responsible to reassess the raw data collected in the acquisition stage and to bring it into a format convenient for visualization. Especially measured data is prone to contain erroneous samples, thus, averaging or smoothing algorithms are applied to remove outliers from—or determine missing samples in—the input data. The flow domain is commonly reduced to a region of interest to decrease the amount of data processed in successive stages of the visualization pipeline. User guided clipping or segmentation algorithms can be used to determine the data under investigation.

Furthermore, resampling methods are subject to the filtering stage to change the underlying representation of a flow field as well as its resolution. Discretization methods used to solve the Navier-Stokes equations result in different flow data representations. Finite-difference methods (FDM) or the Lattice-Boltzmann-Method (LBM) deliver structured grids, whereas finite-element (FEM) or finite-volume methods (FVM) typically result in unstructured grids. On structured grids the connectivity between samples is implicitly given, whereas unstructured grids contain an irregular topology. In practice, the grid type a flow field is given on depends on the employed simulation or measurement technique applied to obtain the data. A large variety of grids is used in practice, which are commonly variations of the basic grid types shown in Figure 2.3.



(a) cartesian (b) uniform (c) rectilinear (d) curvilinear (e) unstructured

Figure 2.3: *Basic grid types.*

In *cartesian* grids, distances between grid points are constant and equal in all dimensions. Mapping grid locations to world-space is very fast, as it requires an identical scaling in all dimensions. *Uniform* grids also feature constant distances between grid points along one direction, however, the sampling distance is not equal in all dimensions. Grid cells have a cuboid shape, and a mapping to world space is given by scaling the coordinates of a sample point individually by the sample point distances in the respective directions.

These types of grids are the most appropriate choice for interactive flow visualization, as (trilinear) interpolation enables fast access to data at arbitrary locations in the flow. Due to fast point location and interpolation, the computation of numerical derivatives is also very fast. Both types of grids can be stored in 3D textures on graphics

hardware and GPU-based operations on such data are extremely efficient due to hardware supported trilinear interpolation.

In *rectilinear* grids, even distances between sample points along one direction vary. In fluid mechanics, rectilinear grids are often used to sample the flow domain adaptively, e.g., regions of interest like boundaries or vortex detachment regions are simulated/sampled at higher resolutions. Mapping a sample to world space becomes more complex as it requires a mapping function for each grid dimension.

An adequate sampling of curved surfaces requires a large resolution even for rectilinear grids. In such cases it makes more sense to employ a *curvilinear* grid that is aligned to the curved shape. However, mapping a grid point to world space becomes even more complicated.

Due to the implicit connectivity between adjacent samples, structured grids can be stored efficiently and a point location requires only few accesses to the structured data. *Unstructured* grids can be employed to sample even complex shapes efficiently as they allow the most flexible choice for the local sample density. However, point location becomes rather complex.

2.2.3 Flow Visualization Techniques

This section covers the last two steps in the visualization pipeline, namely the *mapping* of the visualization data to renderable primitives and the *rendering* into the final image. A large variety of techniques for the quantitative and visual analysis of flow phenomena has been developed over the last decades and the field of flow visualization is still a vivid research area. According to [96] quantitative flow visualization approaches can be broken down into four classes. In the following we will classify them even further into two main categories, namely *dense* and *sparse* visualization techniques. The classification is shown in Figure 2.4 and explained in more detail in the following.

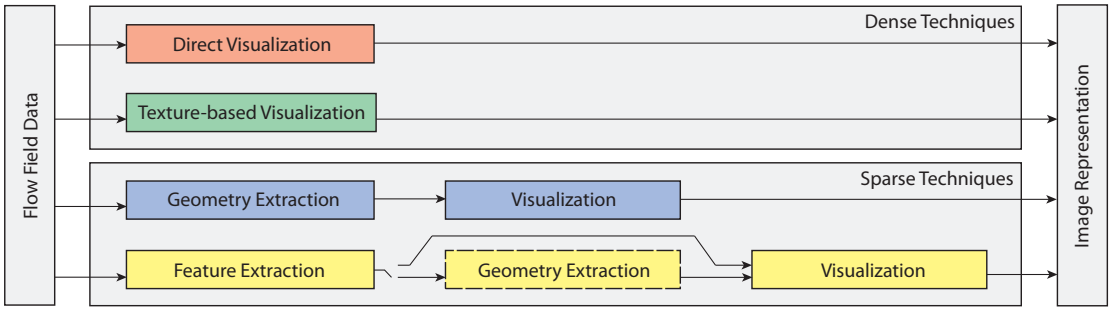


Figure 2.4: *Flow visualization classes: Dense methods (top) can be categorized into direct (red) and texture-based (green) methods. The class of sparse approaches (bottom) can be broken down into geometric (blue) and feature-based (yellow) flow visualization techniques.*

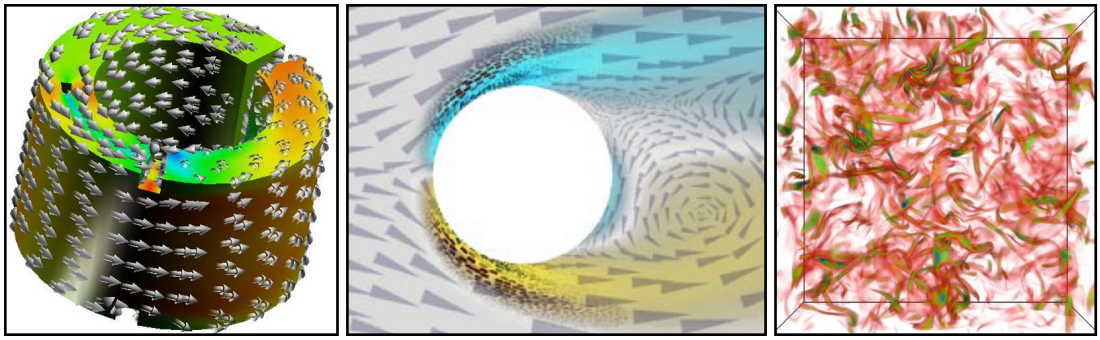


Figure 2.5: *Direct flow visualization examples: Left: Arrow shaped vector field glyphs in 3D (image courtesy of Laramee et al. [97]). Middle: Combination of arrow plots and color coding in 2D (image courtesy of Kirby et al. [80]). Right: 3D color coding of vorticity in fully developed turbulence (image courtesy of M. Wilczek et al. [189]).*

Dense flow visualization techniques deliver a single representation for the whole flow domain and can be classified into following two sub categories:

Direct Flow Visualization

Direct methods avoid extensive pre-processing and visualize the data directly. These techniques are also called *global* approaches, as they are commonly applied to the entire flow domain or a large part of it. Arrow plots depicting velocity directions or the color coding of scalar flow quantities such as the velocity magnitude fall into this category. Direct visualization techniques can deliver intuitive representations for 2D flows fields. E.g., the work by Kirby et al. [80] (see Figure 2.5 (middle)) demonstrates the effectiveness of direct visualization methods within the scope of 2D flows, as they combine arrow plots with color coded imagery to depict multiple flow properties at once.

In 3D, however, global techniques struggle to deliver an intuitive flow representation. The sheer amount of visual information contained in 3D arrow plots generally leads to self-obstruction and results in visual clutter. Occlusion is an inherent problem to the simultaneous portrayal of information at every sample point in the 3D spatial domain, thus, selective visualization strategies have to be applied. Boring and Pang [13] introduced a filtering mechanism for 3D arrow plots, where primitives pointing in a user defined direction are highlighted. Clipping geometries are also commonly applied to restrict direct methods to subregions of interest in the flow domain.

Volume rendering in 3D is the natural extension to 2D color coding. Yet, in contrast to typical scientific areas in which volume rendering is applied (such as medicine), flow field data is often very smooth. Thus, the mapping of opacity becomes much more dif-

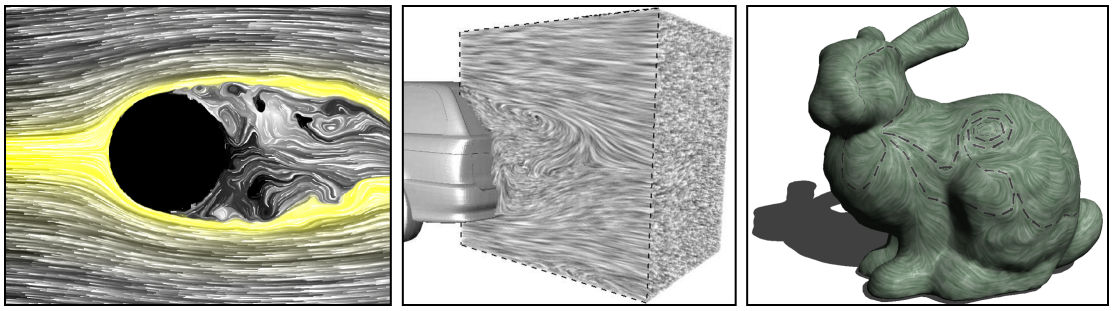


Figure 2.6: *Texture-based flow visualization examples: Left: 2D LIC of the flow around a cylinder (image courtesy of Schafhitzel et al. [144]). Middle: LIC in 3D flow in combination with a clipping plane (image courtesy of Rezk-Salama et al. [133]). Right: LIC visualization of a synthetic vector field on a surface (in combination with arrow glyphs).*

ficult as meaningful transfer functions cannot be specified easily. The first application of volume rendering in the context of flow visualization has been presented in the early nineties [34]. Later, ray casting was applied to vector fields in [45]. Non-photorealistic volume rendering techniques have been presented in [39]. The first interactive approaches, exploiting GPU hardware to speed up the volume rendering process, were introduced in [31, 51].

Texture-based Flow Visualization

Texture-based techniques employ color convolution to generate a single flow field representation revealing directional information. The general idea is to selectively blur a reference image as a function of the vector field to be displayed, where the reference image (in 2D) or volume (in 3D) usually consists of spatially uncorrelated data (e.g., a random noise distribution) defined over the whole flow domain. Spot noise [172] and line integral convolution (LIC) [28, 161] techniques fall into this category. While providing a detailed view on flow features, texture-based methods tend to require time-consuming calculations. Lately, several authors proposed to exploit the GPU to achieve significant speed-ups [71, 184, 101]. For a thorough introduction to texture-based flow visualization techniques we refer the reader to [97].

As these techniques yield a single representation for the whole flow domain, they suffer from self-obstruction in 3D. Thus, the process is usually restricted to regions of interest such as vortex regions [184] or stream surfaces [160]. The restriction to regions of interest culminates in image-based techniques [171, 98], which trade highly interactive frame rates versus artifacts due to the screen-aligned nature of the regions.

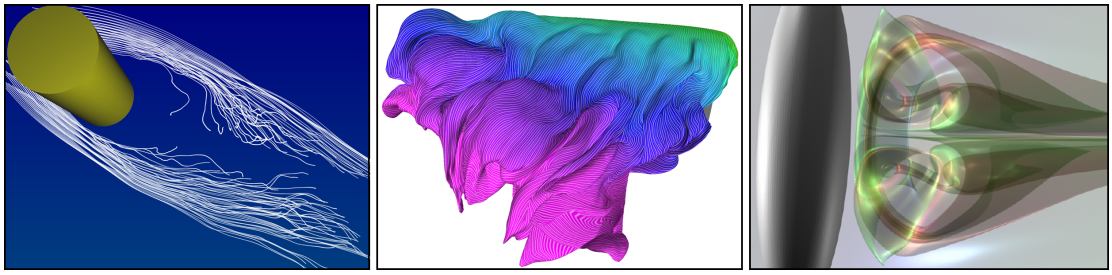


Figure 2.7: *Geometry-based flow visualization examples: Left: Streak lines, Middle: Streak surface, Right: A semitransparent path surface (image courtesy of Garth et al. [48]).*

Traditionally, unsteady fields are problematic, since it is not a priori clear how non-instantaneous characteristics such as streak or path lines can be integrated into dense methods [43, 156]. The problem of spatio-temporal coherence is mostly treated by a recent publication [184], but at considerable effort. Figure 2.6 depicts exemplary visualizations results obtained with the LIC technique.

Sparse flow visualization methods extract characteristic flow features only at specific, carefully selected locations within the flow. Following subclasses fall into this category:

Geometry-based Flow Visualization

Geometry-based methods rely on the particle tracing paradigm to integrate discretized subregions in the flow domain over time, which are then displayed using geometric objects. A wide range of visualization techniques—employing subregions of different dimensionality—has been developed over the last decades.

Particle tracing [136, 21] and the visualization of momentary tracer positions by individual representatives fall into this category. Displaying a huge amount of particles as single point primitives interactively has proven a worthwhile approach to observe dynamics in flow. Representing each particle through more advanced shapes, e.g. arrows depicting the local velocity direction, even improves to grasp flow phenomena intuitively. Next to the shape, the size, color or opacity of a primitive can additionally be used to display further scalar flow quantities.

The extraction and display of characteristic lines [95] (e.g., stream, path or streak lines) is also a prominent tool in geometry-based flow visualization. Again, advanced geometric shapes can be used to display additional flow characteristics. E.g., ribbons can be employed to show the rotation about the flow axis or tube shaped geometry can

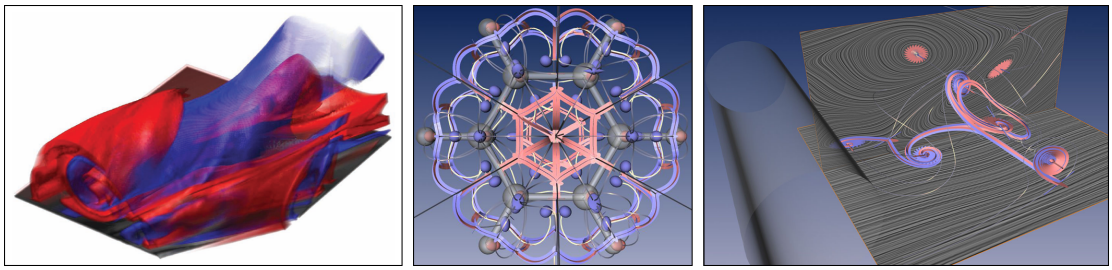


Figure 2.8: *Feature-based flow visualization. Left: Volume rendering of an FTLE field (image courtesy of Garth et al. [47]). Middle: Critical points, separation curves and saddle connectors in the Benzene data set. Right: Critical points and saddle connectors in the flow behind a circular cylinder (images courtesy of Theisel et al. [165]).*

be used to depict additional scalar flow quantities such as the velocity magnitude by adapting the tube thickness accordingly.

Thanks to increased computational resources, integral surfaces—such as path, streak or time surfaces—have moved into the focus of interactive geometric flow visualization. E.g., for stream and path surfaces, the main idea is to integrate an advancing front in the flow and apply if necessary an adaptive refinement or coarsening to it. For a thorough overview on all kinds of integral objects, we refer the reader to [115].

Geometric flow visualization methods particularly depend on proper seeding strategies [167, 123] prior to integration. Localized probing metaphors mimic the injection of external material of real-life windtunnel experiments and combined with interactive visual feedback, they have been proven to be a convenient and effective method to explore complex dynamic flow structures [90]. Detailed listings of related work in the field of geometric flow visualization methods are given in the respective sections of Chapters 4-7. Exemplary geometric flow visualizations are shown in Figure 2.7.

Feature-based Flow Visualization

This class of techniques lifts the visualization to a higher level of abstraction by extracting physically meaningful patterns such as topological structures and skeletons from the data set. Features are phenomena that are of particular interest for a certain problem. In the context of fluid flows, exemplary important features are *vortices*, *shock waves*, *recirculation zones*, *boundary layers* and *attachment* or *separation lines*. For an introduction to and a thorough overview on related work in feature-based visualization techniques, we refer the reader to [99, 129, 142] (examples are shown in Figure 2.8).

The first step in feature-based visualization is *feature extraction*, e.g. on the basis of image processing, the detection of characteristic physical patterns, selective visualiza-

tion approaches [170] or the concept of *vector field topology*. The latter approach was introduced by Helman and Hesselink [64] in 1989 and covers the concepts of critical points, separatrices and closed orbits. Since then, a multitude of related methods has been developed for steady flow fields.

In unsteady flow fields difficulties known as the *correspondence problem* arise. Here, features are objects that evolve over time, thus, the correspondence between features in successive time steps has to be determined. Moreover, the goal of feature tracking is to describe the evolution of features through time, as certain events can occur, such as the interaction of multiple features or significant shape changes of a single feature. If features are extracted in separate time steps an interdependence is generally determined on the basis of region or attribute correspondence.

Lagrangian feature detection is another prominent approach. From the Lagrangian point-of-view the fluid is described by the motion of particles. As these methods analyze trajectories, they are inherently suited for unsteady flows. The finite-time Lyapunov exponent (FTLE) is the most prominent Lagrangian feature detector and will be thoroughly introduced in Section 2.5.3.

Space-time domain approaches handle the problem of detecting features in time dependent data by lifting this problem into a higher dimension, i.e., by interpreting the time as an additional axis and thereby assuming the steady case again. This approach allows a clear definition of path lines by means of stream lines lifted to the higher-dimensional case. For example, *feature flow fields* [164] are specially designed vector fields in 4D space-time that capture parts of the topological information (critical points, periodic orbits, vortex axes) in its temporal evolution. Tracking features in unsteady flows is one of their main application [165, 166, 163].

Let us mention further feature detection classes such as stochastic and multi-field approaches or local methods. Local methods work on point-wise information, including higher order derivatives. For example, ridge extraction from FTLE data was proposed in [153] and has become and established tool for the detection of *Lagrangian coherent structures* (LCS). The theory of and extraction methods for LCS will be discussed in more detail in Section 2.5.

Feature extraction generally results in a binary data set, indicating whether points in the flow domain belong to a feature or not. This binary data set can then be visualized, e.g., with iconic oriented geometric objects or by iso-surfaces of binary regions.

Feature-based flow visualization techniques can achieve a large data reduction (in the order of three magnitudes), but generally require intense pre-computations. Since the reduction in data is generally content-based, important information does not get lost.

2.3 Mathematical Basics

This section introduces basic methods from mathematics (mainly vector calculus) and the relevant theory from flow visualization used throughout this thesis. We will start with an introduction to Lagrangian particle tracing and discuss numerical integration schemes used to approximate the solution of the ordinary differential equation underlying this method. We present trilinear interpolation to obtain continuous values from flow fields discretized by uniform grids, and finite differencing schemes to approximate derivatives in such data sets. Furthermore, we mention the approximation of first- and second-order derivatives as well as analytic solutions to the eigenvalue problem of small symmetric matrices. Let us note that this section is not intended to be a comprehensive guide to the respective topics, but rather lists methods well-suited for the development of an interactive flow exploration environment.

2.3.1 Particle Tracing

In the following we will assume that a 3D unsteady flow field is given in the form of a velocity map $\mathbf{v}$ of the fluid, which assigns a velocity vector to each point $(\mathbf{p},t)$ in its spatial (Ω) and temporal (Π) domains:

$$\mathbf{v}(\mathbf{p},t) : \Omega \times \Pi \rightarrow \mathbb{R}^3, \quad \mathbf{p} \in \Omega \subseteq \mathbb{R}^3, \quad t \in \Pi \subseteq \mathbb{R} \, .$$

Tracking a (massless) particle through the flow field corresponds to the solution of the first-order differential equation with the independent variable t (representing time):

$$\frac{\mathrm{d}\mathbf{x}(t,t_0,\mathbf{x}_0)}{\mathrm{d}t} = \mathbf{v}(\mathbf{x}(t,t_0,\mathbf{x}_0),t) \, . \tag{2.1}$$

Here, the tangent to the particle trajectory is denoted as $\frac{d\mathbf{x}(t,t_0,\mathbf{x}_0)}{dt}$. The dependent variable, i.e. the time-varying position of the particle initialized at position $\mathbf{x}_0$ in space and time t_0, is represented by $\mathbf{x}(t,t_0,\mathbf{x}_0)$. To avoid notational clutter, we will often omit explicit references to t_0 and $\mathbf{x}_0$ in the following and simply write $\mathbf{x}(t)$. In order to solve this equation we can express it in integral form:

$$\mathbf{x}(t,t_0,\mathbf{x}_0) = \mathbf{x}_0 + \int_{\tau=t_0}^{t} \mathbf{v}(\mathbf{x}(\tau,t_0,\mathbf{x}_0),\tau) \, d\tau \, . \tag{2.2}$$

In the study of dynamical systems, flow fields are often described in a closed form

and equation (2.2) can be solved analytically. Reconstructed as well as simulated flow fields, however, are commonly represented by a discrete set of samples over the flow domain. Thus, we have to rely on numerical integration schemes to find an approximation to the solution. The fastest approximation is given by Euler's method:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{v}(\mathbf{x}(t),t)\Delta t \; . \tag{2.3}$$

Here, the position of a particle at time $t+\Delta t$ is given by the sum of the previous position $\mathbf{x}(t)$ and the velocity vector at the corresponding position in the flow field scaled by an incremental time step Δt. However, by regarding the taylor expansion (2.8) of equation 2.3 around t, one can see that the Euler approximation introduces an error per step on the order of $\mathcal{O}(\Delta t^2)$. Consequently small increments in time have to be chosen. Higher order integration schemes yield smaller errors on a per step basis at increased computational costs with respect to arithmetic operations and memory access [32]. With increasing computational numerical processing power of GPUs, it has proven worthwhile to employ the explicit Runge-Kutta integrator of fourth-order to approximate the solution of the ordinary differential equation. Here, the error introduced on a per integration step basis is in the order of $\mathcal{O}(\Delta t^5)$, and it is widely accepted as optimal compromise between numerical accuracy and computational performance. It is given as:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \frac{\Delta t}{6} \cdot (\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \; , \tag{2.4}$$

where

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{v}(\mathbf{x}(t),t) \; , \\
\mathbf{k}_2 &= \mathbf{v}(\mathbf{x}(t) + \tfrac{\Delta t}{2}\mathbf{k}_1, t + \tfrac{\Delta t}{2}) \; , \\
\mathbf{k}_3 &= \mathbf{v}(\mathbf{x}(t) + \tfrac{\Delta t}{2}\mathbf{k}_2, t + \tfrac{\Delta t}{2}) \; , \\
\mathbf{k}_4 &= \mathbf{v}(\mathbf{x}(t) + \Delta t\mathbf{k}_3, t + \Delta t) \; .
\end{aligned}$$

Another class of integrators, so called embedded schemes, yield better results with respect to accuracy and speed. Within this class, the local integration error is used to adaptively change the integration step size. An exemplary integration scheme is the RK3(2) integrator [12]. However, in interactive environments, the distance a particle moves during one advection step should be in accordance to the local velocity magnitude. Therefore, adaptive schemes require a varying number of integration steps to adhere to a fixed time interval. Due to this fact, embedded schemes do not map well to the parallel data processing paradigm of GPUs as they impose a varying load on the

parallel execution units. While one unit might calculate only one integration operation, others might have to perform multiple operations, thus, stalling grouped units running in a lock-step execution mode.

2.3.2 Characteristic Flow Lines

Three distinct types of characteristic lines are commonly employed to depict flow phenomena in unsteady 3D flow:

- **Path lines**: A path line $\mathbf{x}_{path}$ represents the trace left by a particle induced into the time-varying flow field and can be obtained by the following integration:

$$\mathbf{x}_{path}(t,t_0,\mathbf{x}_0) = \mathbf{x}_0 + \int_{\tau=t_0}^{t} \mathbf{v}(\mathbf{x}_{path}(\tau,t_0,\mathbf{x}_0),\tau)\, d\tau\,. \tag{2.5}$$

- **Stream lines**: A stream line $\mathbf{x}_{stream}$ describes an instantaneous particle path, which is the trajectory of a particle in an unsteady flow frozen at time s. It is obtained as follows:

$$\mathbf{x}_{stream}(t,t_0,\mathbf{x}_0) = \mathbf{x}_0 + \int_{\tau=t_0}^{t} \mathbf{v}(\mathbf{x}_{stream}(\tau,t_0,\mathbf{x}_0),s)\, d\tau\,. \tag{2.6}$$

- **Streak lines**: In contrast to path lines and stream lines, streak lines do not depict the history of a single particle moving with the flow. This type of line originates from real-world experiments where external materials are constantly induced into the flow and the occurring patterns are observed. As streak lines describe the path traced by dye or smoke continuously released into the flow, they are defined by the current location of all particles that have passed through a fixed spatial position at a succession of previous times $[t_{start}, t_{end}]$. To obtain a streak line, particles at successive time steps $t_{start} \leq t_i \leq t_{end}$ are released from the starting location $\mathbf{x}_0$ into the flow and their current position can be obtained with equation 2.5. Connecting successively released particles forms the streak line.

In flow visualization, integral curves are commonly approximated through numerical integration, resulting in a discrete set of consecutive control points. Geometry-based techniques then use this set to construct a piecewise continuous geometric representation and texture-based approaches use it to collect intensity values along a trajectory.

2.3.3 Flow Field Interpolation

As already mentioned in Section 2.2.2, cartesian and uniform grids are the most appropriate choices to represent flow fields in an interactive flow visualization environment. In contrast to unstructured grids where the cell containing an arbitrary point in the flow domain generally has to be searched (e.g., by traversing an adaptive data structure), here, finding the indices of the corresponding grid cell requires only a per component scaling of coordinates. Furthermore to obtain continuous function values, i.e. flow field quantities at arbitrary locations in the flow domain, an interpolation has to be applied. In structured data sets, each cell is encircled by the same amount of grid nodes and their location is inherently encoded in the data structure.

The 3D unsteady flow field data sets used in this thesis are represented by a time-resolved sequence of velocity data with spatial samples on either cartesian or uniform grids. Be $\mathbf{x} = (x_p, y_p, z_p)^T$ an arbitrary point in the flow domain. To obtain the velocity vector $\mathbf{v}(\mathbf{x})$ from one of the discrete time-steps, we apply trilinear interpolation between the eight adjacent samples ($x_i \leq x_p \leq x_{i+1}, y_i \leq y_p \leq y_{i+1}, z_i \leq z_p \leq z_{i+1}$):

$$\begin{aligned}
\mathbf{v}(\mathbf{x}) = \quad & (1-\alpha)(1-\beta)(1-\gamma) \quad \mathbf{v}(x_i, y_i, z_i) \quad + \\
& \alpha(1-\beta)(1-\gamma) \quad \mathbf{v}(x_{i+1}, y_i, z_i) \quad + \\
& (1-\alpha)\beta(1-\gamma) \quad \mathbf{v}(x_i, y_{i+1}, z_i) \quad + \\
& (1-\alpha)(1-\beta)\gamma \quad \mathbf{v}(x_i, y_i, z_{i+1}) \quad + \\
& (1-\alpha)\beta\gamma \quad \mathbf{v}(x_i, y_{i+1}, z_{i+1}) \quad + \\
& \alpha(1-\beta)\gamma \quad \mathbf{v}(x_{i+1}, y_i, z_{i+1}) \quad + \\
& \alpha\beta(1-\gamma) \quad \mathbf{v}(x_{i+1}, y_{i+1}, z_i) \quad + \\
& \alpha\beta\gamma \quad \mathbf{v}(x_{i+1}, y_{i+1}, z_{i+1}) \quad .
\end{aligned}$$

where $\Delta x, \Delta y, \Delta z$ are the uniform sampling distances along either dimension and α, β and γ denote the fractional parts of $\frac{x_p}{\Delta x}$, $\frac{y_p}{\Delta y}$ and $\frac{z_p}{\Delta z}$ respectively.

To calculate an approximation $\mathbf{v}(\mathbf{x}, t)$ of the velocity vector field at an arbitrary location in space $\mathbf{x}$ and time t, we sample data from two adjacent time-steps in the sequence ($t_i \leq t \leq t_{i+1}$) and perform one additional linear interpolation:

$$\mathbf{v}(\mathbf{x}, t) = (1-\delta) \cdot \mathbf{v}(\mathbf{x}, t_i) + \delta \cdot \mathbf{v}(\mathbf{x}, t_{i+1}), \quad \delta = \frac{t - t_i}{t_{i+1} - t_i} \, . \tag{2.7}$$

As linear interpolation assumes that a function behaves linear between sample points, the data should be sampled at a reasonable rate to avoid inaccuracies. While higher or-

der schemes such as cubic or hermite interpolation deliver more accurate results, we employ only the linear scheme as it is directly supported by GPUs and can, thus, be performed in hardware at negligible costs. Let us further note that spatial flow field samples of one time-step are always stored in one GPU texture resource corresponding to a cartesian grid defined over the unit cube. Therefore, point coordinates need to be scaled accordingly before accessing data from the field.

2.3.4 Numerical Differentiation

For 3D unsteady flow field data discretized by cartesian or uniform grids, we employ finite differencing schemes to compute numerical derivatives. According to Taylor's theorem, the value of a function f around a point x can be obtained by the series

$$f(x+h) = f(x) + \frac{f'(x)}{1!}h + \frac{f^{(2)}(x)}{2!}h^2 + \frac{f^{(3)}(x)}{3!}h^3 + \ldots + \frac{f^{(n)}(x)}{n!}h^n + R_n(x), \quad (2.8)$$

where the remainder term $R_n(x)$ denotes the difference between the original function and the Taylor polynomial of degree n. It can be proven that the absolute error in the approximation is upper bounded by the next term of the expansion. Thus, if the function at point x and in its vicinity is known, we can rearrange the Taylor expansion (regarding only the first two right hand terms) to obtain the forward (2.9), backward (2.10) or central (2.11) differences:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h), \quad (2.9)$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h), \quad (2.10)$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2). \quad (2.11)$$

For reasonable small h the error introduced by neglecting the remainder term is commonly accepted. Higher order differences can be obtained analogously. E.g., if we apply the central differencing scheme with spacing $\frac{h}{2}$ and then use above central difference formula for the derivative f' at x, we obtain the central difference approximation for the second order derivative of f:

$$f^{(2)}(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \quad (2.12)$$

Finite differences can be considered in more than one variable, some partial derivative

approximations for a function of two variables are shown in the following:

$$\begin{aligned}
\tfrac{\partial}{\partial x} f(x,y) &\approx \frac{f(x+h,y) - f(x-h,y)}{2h}, \\
\tfrac{\partial^2}{\partial x^2} f(x,y) &\approx \frac{f(x+h,y) - 2f(x,y) + f(x-h,y)}{h^2}, \\
\tfrac{\partial^2}{\partial x \partial y} f(x,y) &\approx \frac{f(x+h,y+k) - f(x+h,y-k) - f(x-h,y+k) - f(x-h,y-k)}{4hk}.
\end{aligned}$$

Throughout this work, we employ the central differencing scheme to calculate numerical derivatives. Only at boundary regions of the flow domain, where no values are available in either direction, we fall back to the forward or backward approaches.

2.3.5 Matrix Eigenanalysis

The eigendecomposition of square matrices $\mathbf{A}$ is a standard tool employed in flow visualization. A non-zero vector $\mathbf{e}$ is an eigenvector if and only if it satisfies the linear eigenvalue equation $\mathbf{Ae} = \lambda\mathbf{e}$, where λ is the eigenvalue corresponding to $\mathbf{e}$. If $\mathbf{I}$ is the identity matrix, then we can rewrite the equation as:

$$\mathbf{Ae} - \lambda\mathbf{Ie} = (\mathbf{A} - \lambda\mathbf{I})\mathbf{e} = 0. \tag{2.13}$$

If there exists an inverse $(\mathbf{A} - \lambda\mathbf{I})^{-1}$ then both sides can be left multiplied by it to arrive at the trivial solution $\mathbf{e} = 0$. Thus, we require to meet the condition that the determinant equals zero, i.e. no inverse exists

$$p(\lambda) = |\mathbf{A} - \lambda\mathbf{I}| = 0.$$

Finding the eigenvalues of $\mathbf{A}$ amounts to finding the roots of the characteristic polynomial $p(\lambda)$. According to Abel's impossibility theorem, for large polynomials (of order > 4), this problem cannot be solved by a finite sequence of arithmetic operations and radicals. Yet, to find the eigenvalues of large symmetric matrices a variety of iterative approaches exist (such as the power iteration, the Jacobi method or the popular QR algorithm [131]).

However, in the rest of this work we will only be interested in the eigenvalues of real symmetric matrices of second order and positive-definite symmetric 3×3 matrices. Here, solutions to the eigenvalue problem can be expressed in analytic form and, thus, be solved interactively even for a huge number of matrices in parallel. For a real

symmetric 2×2 matrix $\mathbf{A}$, the real eigenvalues λ_1, λ_2 and their respective eigenvector $\mathbf{e}_i$ $(i = 1 \vee 2)$ are given as

$$\mathbf{A} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}, \quad \lambda_1, \lambda_2 = \frac{(a+c) \pm \sqrt{(a-c)^2 + 4b^2}}{2}, \quad \mathbf{e}_i = \begin{pmatrix} \frac{\lambda_i - c}{\sqrt{b^2 - (\lambda_i - c)^2}} \\ \frac{b}{\sqrt{b^2 - (\lambda_i - c)^2}} \end{pmatrix}.$$

Square matrices of third order

To find the roots of the characteristic polynomial $p(\lambda)$ of an arbitrary 3×3 matrix, a cubic equation of the form $x^3 + ax^2 + bx + c = 0$ has to be solved. Here, Cardano's method [131] can be applied which starts to find a solution by moving the cubic's point of inflection to the origin. This substitution removes the quadratic term and gives a so called depressed cubic $t^3 + pt + q = 0$. This equation contains still a linear term, thus, for $p \neq 0$ it cannot be solved by means of a single cubic root. The assumption that a solution t for the depressed cubic equation can be expressed by the sum of two cubic roots $t = u + v$ leads to further substitution and the final solution. Depending on the discriminant d of the depressed cubic, the characteristic polynomial has either three distinct roots ($d > 0$), one real root and two complex conjugate roots ($d < 0$) or a multiple root and and all its roots are real. A complete derivation of Cardano's method is beyond the scope of this section. We recommend the readers interested in the concepts of this approach the article by Nickalls [119]. Here, the standard method for solving the cubic is greatly clarified by relating the solution to the cubic's geometry.

The corresponding eigenvector for an eigenvalue λ can be found by inserting it in (2.13) and solving the system of linear equations with ,e.g., an iterative Jacobi method.

Positive-definite symmetric 3×3 matrices

According to the spectral theorem, a real symmetric 3×3 matrix has three real eigenvalues λ_i $(i = 1 \vee 2 \vee 3)$ and three linearly independent eigenvectors that are mutually orthogonal. In Section 2.5.3 we will be interested in the eigenvalues of a *positive-definite* symmetric 3×3 matrix $\mathbf{D}$. This allows us to employ a more specialized analytic method to find a solution. We employ the method proposed by Hasan et al. [62] which is based on *diffusion tensor invariants* to find the eigenvalues and eigenvectors of $\mathbf{D}$. A cartesian diffusion tensor

$$\mathbf{D} = \begin{bmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{yx} & D_{yy} & D_{yz} \\ D_{zx} & D_{zy} & D_{zz} \end{bmatrix}$$

has three principal invariants I_1, I_2, I_3. They are related to the eigenvalues, and defined by the characteristic equation

$$|\mathbf{D} - \lambda \mathbf{I}| = (\lambda - \lambda_1)(\lambda - \lambda_2)(\lambda - \lambda_3) = \lambda^3 - \lambda^2 I_1 + \lambda I_2 - I_3 = 0,$$

where $\mathbf{I}$ is the 3×3 identity matrix. According to [8, 14], from this equation the three invariants are given as

$$\begin{aligned}
I_1 &= Trace(\mathbf{D}) = D_{xx} + D_{yy} + D_{zz} = \lambda_1 + \lambda_2 + \lambda_3 \ , \\
I_2 &= (D_{xx}D_{yy} + D_{xx}D_{zz} + D_{yy}D_{zz}) - (D_{xy}^2 + D_{xz}^2 + D_{yz}^2) = \lambda_1\lambda_2 + \lambda_1\lambda_3 + \lambda_2\lambda_3 \ , \\
I_3 &= |\mathbf{D}| = D_{xx}D_{yy}D_{zz} + 2D_{xy}D_{xz}D_{yz} - (D_{zz}D_{xy}^2 + D_{yy}D_{xz}^2 + D_{xx}D_{yz}^2) = \lambda_1\lambda_2\lambda_3 \ .
\end{aligned}$$

The eigenvalues and eigenvectors of $\mathbf{D}$ can now be found by an analytic diagonalization of $\mathbf{D}$ that is specific to the positive-definite symmetric cartesian tensor. The following rotational invariant variables are defined in terms of I_1, I_2, I_3:

$$v = (I_1/3)^2 - I_2/3 \quad \text{and} \quad s = (I_1/3)^3 - I_1 I_2/6 + I_3/2.$$

Since for real eigenvalues it holds that $v > 0$ and $s^2 < v^3$, we can define

$$\phi = \frac{\text{acos}(\frac{s}{v}\sqrt{\frac{1}{v}})}{3}.$$

The sorted eigenvalues $(\lambda_1 > \lambda_2 > \lambda_3)$ can then be expressed as

$$\lambda_1 = \frac{I_1}{3} + 2\sqrt{v}\cos(\phi), \quad \lambda_2 = \frac{I_1}{3} - 2\sqrt{v}\cos(\frac{\pi}{3} + \phi), \quad \lambda_3 = \frac{I_1}{3} - 2\sqrt{v}\cos(\frac{\pi}{3} - \phi).$$

According to [62] the orthonormalized eigenvector for the ith eigenvalue can be computed as follows. Define following variables

$$\begin{aligned}
A_i = D_{xx} - \lambda_i, \quad B_i = D_{yy} - \lambda_i, \quad C_i = D_{zz} - \lambda_i, \\
\mathbf{e}_i = \begin{pmatrix} (D_{xy}D_{yz} - B_i D_{xz})(D_{xz}D_{yz} - C_i D_{xy}) \\ (D_{xz}D_{yz} - C_i D_{xy})(D_{xz}D_{xy} - A_i D_{yz}) \\ (D_{xy}D_{yz} - B_i D_{xz})(D_{xz}D_{xy} - A_i D_{yz}) \end{pmatrix}.
\end{aligned}$$

The normalized eigenvector $\hat{\mathbf{e}}_i$ for eigenvalue λ_i is then given by $\frac{\mathbf{e}_i}{\|\mathbf{e}_i\|}$. Note that due to the sign ambiguity of equation 2.13, $-\hat{\mathbf{e}}_i$ is also a solution to the eigenvalue problem. Thus, the third normalized eigenvector $\hat{\mathbf{e}}_3$ can be obtained more efficiently by the cross

product between the other orthonormal eigenvectors $\hat{\mathbf{e}}_1$ and $\hat{\mathbf{e}}_2$.

2.3.6 First and Second Order Derivatives

In 3D, a scalar field f has three partial derivatives with respect to the three-dimensional cartesian coordinates x, y, z. The gradient of a scalar field points into the direction of the greatest rate of increase and is defined as the vector of its partial derivatives:

$$\nabla f(x,y,z) = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} f(x,y,z) = \begin{pmatrix} \frac{\partial}{\partial x} f(x,y,z) \\ \frac{\partial}{\partial y} f(x,y,z) \\ \frac{\partial}{\partial z} f(x,y,z) \end{pmatrix} = \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} . \tag{2.14}$$

The gradient of a 3D vector field $\mathbf{v}(x,y,z) = (u(x,y,z), v(x,y,z), w(x,y,z))^T$ is found by application of the gradient operator to each of the components of the vector field. This results in a order 2 tensor field, i.e., the gradient at an arbitrary point in the vector field is given by a 3×3 matrix of first order derivatives known as the *Jacobian*:

$$\mathbf{J}(x,y,z) = \nabla\mathbf{v} = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{pmatrix} . \tag{2.15}$$

In flow visualization the Jacobian is often used to compute a number of derived fields. Furthermore, the vector field topology is determined by an eigenanalysis of $\mathbf{J}$, as its eigenvectors and eigenvalues indicate the direction of tangent curves of the flow.

For a real-valued scalar function in Euclidean n-space ($f(x_1, x_2, \cdots, x_n) : \mathbb{R}^n \to \mathbb{R}$), its *Hessian* matrix is a square matrix of order n. Here, matrix entries contain second-order partial derivatives of f, i.e. the Hessian describes the local curvature of a function of many variables:

$$\mathbf{H}(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} = \begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n1} & f_{n1} & \cdots & f_{nn} \end{pmatrix} . \tag{2.16}$$

If the mixed differentials of function f are contiguous the order of differentiation does

not matter and, thus, its Hessian is symmetric. If f is a function from $\mathbb{R}^n \rightarrow \mathbb{R}^m$, then the array of second order partial derivatives is not a square matrix of order n, but rather a tensor of order 3 (i.e. an array of size $m \times n \times n$).

Non-degenerate critical points ($\nabla f(x_1, x_2, \cdots, x_n) = 0 \wedge |\mathbf{H}(f(x_1, x_2, \cdots, x_n))| \neq 0$) are generally studied by an eigenanalysis of the (symmetric) Hessian matrix to determine the topology of manifolds. Function f attains a local maximum at such a point if $\mathbf{H}$ is positive definite, and a minimum if $\mathbf{H}$ is negative definite. If $\mathbf{H}$ has positive and negative eigenvalues there is a saddle point at the respective location. Otherwise this test is inconclusive. For all other points, however, semi-definite Hessians can be used to determine if f is locally concave (positive semi-definite) or convex (negative semi-definite). Again eigenvalues of mixed sign indicate a saddle point. We will employ this test in Chapter 7 for the extraction of ridges from a scalar field in 2-space.

2.4 Derived Measures of Vector Fields

In Chapter 5, we will employ additional scalar quantities for importance driven flow visualization. These measures indicate certain properties of the flow field and are either directly derived from the velocity vector field (by the application of differential operators from vector calculus) or from one of its derivatives:

- *Velocity magnitude*: The magnitude of the velocity vector field $\mathbf{v} = (u, v, w)^T$ is:

$$\|\mathbf{v}\| = \sqrt{u^2 + v^2 + w^2}\,. \tag{2.17}$$

- *Divergence*: The divergence of a velocity field is the extent to which the vector field behaves like a source or sink at a given position. It measures the extent to which there is more exiting an infinitesimal region of space than entering it. If the divergence is nonzero at some location then there must be a source or sink at that position, otherwise the flow is called divergence-free. This is the common case in fluid dynamics as most fluids are incompressible. The divergence is defined as:

$$\nabla \cdot \mathbf{v} = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} \cdot \begin{pmatrix} u \\ v \\ w \end{pmatrix} = u_x + v_y + w_z\,. \tag{2.18}$$

- *Vorticity magnitude*: The *curl* $\boldsymbol{\omega}$ of a velocity field is called *vorticity*. The vorticity is a vector field that indicates the axis of rotation as well as the local angular

rate of rotation. The vorticity is perpendicular to the plane in which the locally highest amount of circulation takes place and its magnitude specifies the strength of rotation. If the vector field represents the flow velocity, then the vorticity is also referred to as the circulation density of the fluid. A vector field whose curl is zero is called irrotational or curl-free. Vortex regions in the flow may have a high *vorticity magnitude* $\|\omega\|$, thus, this quantity can be used to classify such regions [177]. The vorticity is defined as:

$$\boldsymbol{\omega} = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} = \nabla \times \mathbf{v} = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} \times \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} w_y - v_z \\ u_z - w_x \\ v_x - u_y \end{pmatrix} . \tag{2.19}$$

- *Helicity*: The helicity is a scalar quantity indicating the extent to which corkscrew-like motion occurs at a given location. If a moving parcel of fluid rotates about an axis parallel to the direction of motion, it has helicity. If the rotation is clockwise when viewed from ahead of the parcel, the helicity will be positive, if counterclockwise, it will be negative. The helicity can be used to detect vortex regions [190], and it is obtained by projecting the vorticity onto the velocity:

$$\boldsymbol{\omega} \cdot \mathbf{v} = (w_y - v_z)u + (u_z - w_x)v + (v_x - u_y)w . \tag{2.20}$$

- *Streamwise vorticity*: The streamwise vorticity is the component of vorticity that is parallel to an ambient (i.e., local mean) velocity vector $\tilde{\mathbf{v}}$:

$$\frac{\boldsymbol{\omega} \cdot \tilde{\mathbf{v}}}{\|\tilde{\mathbf{v}}\|} . \tag{2.21}$$

- λ_2*-criterion*: The λ_2-criterion [70] is the most widely used measure for vortex detection. It is based on a decomposition of the velocity field's Jacobian matrix $\mathbf{J} = \nabla\mathbf{v} = \mathbf{S} + \mathbf{A}$. Here $\mathbf{S}$ is the symmetric part (or strain tensor) and $\mathbf{A}$ the antisymmetric part (also called the vorticity tensor) of the Jacobian:

$$\mathbf{S} = \frac{1}{2}(\mathbf{J} + \mathbf{J}^T), \quad \mathbf{A} = \frac{1}{2}(\mathbf{J} - \mathbf{J}^T) .$$

  While the strain tensor $\mathbf{S}$ holds information about the local stretching of the fluid, $\mathbf{A}$ assesses rotational activity. Vortex regions are then identified by $\lambda_2 < 0$, whereas λ_2 is the second largest eigenvalue of the symmetric tensor $\mathbf{S}^2 + \mathbf{A}^2$.

2.5 Lagrangian Coherent Structures

The derived measures introduced so far adhere to the concept of viewing a flow from the *Eulerian* perspective, i.e. as a set of fixed points in the spatial domain with corresponding quantities (at varying instances in time for unsteady flow). In fluid mechanics, this is the standard approach for studying the velocity vector fields of fluid flow.

Fluid flows fall into the category of *dynamical systems*, i.e., they describe the evolution of interdependent quantities within the system's domain according to a specific set of rules. General dynamical systems are often studied from the *Lagrangian* point-of-view, i.e., in terms of particle trajectories traced in phase space [188]. Here, the system's evolution is often governed by partial differential equations.

The Lagrangian concept can also be applied in the study of fluid flows. In this specific case the system's evolution is governed by the ordinary differential equation of the particle tracing paradigm, and the Lagrangian perspective is generally used to observe large scale transport behavior and to reveal the global flow geometry.

2.5.1 Dynamical Systems

Let us first introduce a dynamical systems in its most general form. Note that the notations and descriptions used in this section are closely related to the tutorial by Shadden [152], as they are commonly used to introduce the following topics:

$$\left.\begin{aligned} \dot{\mathbf{x}}(t,t_0,\mathbf{x}_0) &= \mathbf{v}(\mathbf{x}(t,t_0,\mathbf{x}_0),t), \\ \mathbf{x}(t_0,t_0,\mathbf{x}_0) &= \mathbf{x}_0. \end{aligned}\right\} \tag{2.22}$$

Analogue to Eq. (2.1), $t \in \Pi$ represents time and $\mathbf{x}(t,t_0,\mathbf{x}_0) \in \Omega$ is the dependent variable representing the state of the system. While Ω may be more general than $\mathbb{R}^3$, in terms of fluid flows we can assume that Ω is a subset of $\mathbb{R}^3$.

With advancing time, solutions of (2.22) trace out curves in space, or in dynamical systems terminology they flow along their trajectory. Given a particle initialized at point $(\mathbf{x}_0,t_0)$ in the spatio-temporal flow domain and assuming t is a final time, we can rewrite equation (2.2) as the *flow map*, i.e., a map which takes a point in the domain at time t_0 to its location at time t:

$$\phi_{t_0}^{t} : \Omega \to \Omega : \mathbf{x}_0 \mapsto \phi_{t_0}^{t}(\mathbf{x}_0) = \mathbf{x}(t,t_0,\mathbf{x}_0). \tag{2.23}$$

According to the standard theorems on local existence and uniqueness of solutions of Eq. (2.22) the flow map satisfies the following properties [61] :

$$\left.\begin{aligned} &\phi_{t_0}^{t_0}(\mathbf{x}) = \mathbf{x}, \\ &\phi_{t_0}^{t+s}(\mathbf{x}) = \phi_s^{t+s}(\phi_{t_0}^{s}(\mathbf{x})) = \phi_t^{t+s}(\phi_{t_0}^{t}(\mathbf{x})). \end{aligned}\right\} \tag{2.24}$$

While the exact solution of Eq. (2.22) would be ideal, unless $\mathbf{v}$ is a linear function of the state $\mathbf{x}$ and independent of time t, there is no general way to determine a closed-form analytic solution of this equation. Numerical approximations of the solution yield particle trajectories, or more precisely path lines for unsteady flows and stream lines for instantaneous flow fields.

2.5.2 Invariant Manifolds and Coherent Structures

The behavior of *time-independent dynamical systems* (with a static definition over an infinite period of time) is often studied by an observation of *invariant manifolds* of fixed points in the system. A fixed point is a point in the domain where $\mathbf{v}(\mathbf{x}) = 0$. *Stable manifolds* of a fixed point are all trajectories which asymptote to it when time goes to infinity and *unstable manifolds* are those trajectories which asymptote to fixed points in inverse time. Thus, such manifolds (commonly called *separatrices*) attract or repel particles respectively, thereby dividing the domain into regions of fundamentally different dynamics (see Figure 2.9 (a)). The *Lyapunov exponents* [103] are often employed to detect such manifolds in this class of systems by quantifying the rate of separation of trajectories starting in an infinitesimally vicinity of a point in the domain. The rate of separation can vary for different combinations of such trajectories, thus, there exists a whole spectrum of Lyapunov exponents. One is often only interested in the largest rate of separation, i.e. the maximal Lyapunov exponent (MLE), as it determines the predictability of a dynamical system. Due to its asymptotic nature, the MLE can only be employed in the study of time-independent systems. For readers interested in this subject we refer to the comprehensive manuscript by Barreira et al. [94].

The transport behavior in unsteady flows is typically governed by prominent features, such as chaotic emerging and disappearing vortices or eddies in the flow around obstacles. The global flow geometry in the study of intermixing processes is generally given by the respective material surfaces separating fluids of different physical properties. In general, large-scale regions of coherent flow behavior exhibit strong correlations and are of special interest when analyzing unsteady flows. Large scale phenomena of different dynamics are confined by *coherent structures*, which are often analogous to stable and unstable manifolds in time-independent systems. The behavior of (aperiodic) *time-dependent dynamical systems*, however, is only known over a limited period

of time. Thus, a finite version of the Lyapunov exponent has to be used for a general analysis of such systems. The (maximum) *finite-time Lyapunov exponent* represents a quantity which allows the detection of coherent structures in unsteady flows, thus, giving rise to the possibility to unveil the global flow geometry.

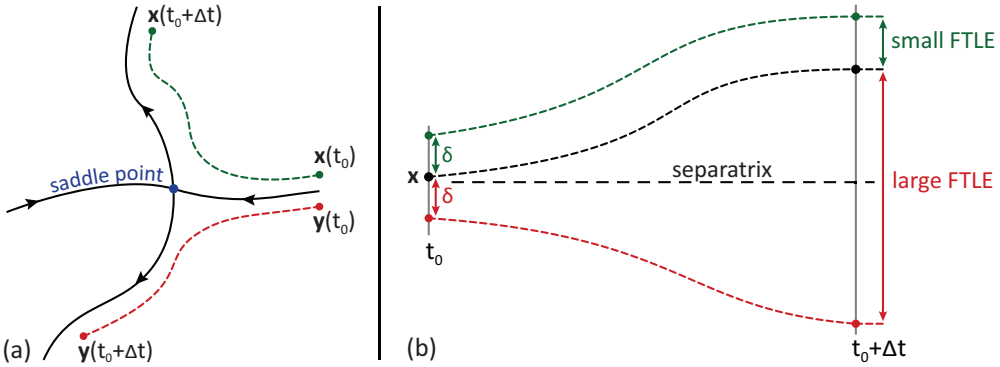


Figure 2.9: *Illustrations of separatrices and FTLE: In image (a), two particle trajectories on either side of a stable manifold are shown. Image (b) depicts the correspondence between the initial perturbation δ and the finite-time Lyapunov exponent.*

2.5.3 Finite-Time Lyapunov Exponent

The (maximum) finite-time Lyapunov exponent (FTLE) is a scalar quantity that characterizes the amount of stretching about the trajectory of particles over a finite time interval $[t, t+\Delta t]$. This notion stands for the amount of separation of integrated particles that have been released at the same time infinitesimally close to each other in space (see Figure 2.9 (b) for an illustration of the FTLE). Following the terminology of Haller [56], it can be deduced from particle tracing as follows:

Be $\mathbf{x} \in \mathbb{R}^3$ an arbitrary point in the spatial flow domain. If we release a particle from this position at time t_0 and advect it for a finite time interval Δt, it arrives at the point $\phi_{t_0}^{t_0+\Delta t}(\mathbf{x})$. Since fluid flow generally has a continuous dependence on initial conditions, we know that a particle released at the same time in the (close) vicinity of $\mathbf{x}$ will behave similar when advected in the flow (at least for a short period of time). However, as time goes by, the distance between these particles will almost certainly change. Be $\delta\mathbf{x}(t)$ an arbitrarily oriented infinitesimal distortion. With it we can express the initial position of a second particle as $\mathbf{y} = \mathbf{x} + \delta\mathbf{x}(t_0)$. After Δt time has passed, this perturbation becomes:

$$\delta\mathbf{x}(t_0+\Delta t) = \phi_{t_0}^{t_0+\Delta t}(\mathbf{y}) - \phi_{t_0}^{t_0+\Delta t}(\mathbf{x}) \; . \tag{2.25}$$

If we consider the taylor series expansion of the flow about point $\mathbf{x}$ up to the first order derivative, the perturbation can be described more generally as:

$$\delta\mathbf{x}(t_0+\Delta t) = \frac{\mathrm{d}\phi_{t_0}^{t_0+\Delta t}(\mathbf{x})}{\mathrm{d}\mathbf{x}}\delta\mathbf{x}(t_0) + \mathcal{O}(\|\delta\mathbf{x}(t_0)\|^2). \tag{2.26}$$

Since our initial assumption was that $\delta\mathbf{x}(t_0)$ is infinitesimal, the remainder term can be neglected. The flow map gradient

$$\nabla\phi_{t_0}^{t_0+\Delta t}(\mathbf{x}) = \frac{\mathrm{d}\phi_{t_0}^{t_0+\Delta t}(\mathbf{x})}{\mathrm{d}\mathbf{x}} \tag{2.27}$$

describes the deviation of trajectories started at the same time t_0 in an infinitesimally spatial vicinity of point $\mathbf{x}_0$. The tensor

$$\mathcal{C}_{t_0}^{t_0+\Delta t}(\mathbf{x}) = \left(\nabla\phi_{t_0}^{t+\Delta t}(\mathbf{x})\right)^{\mathrm{T}} \nabla\phi_{t_0}^{t+\Delta t}(\mathbf{x}) \tag{2.28}$$

known as the (finite-time) right Cauchy-Green deformation tensor, expresses the deformation of the neighborhood of $\mathbf{x}$ under the flow map. More precisely, this positive definite symmetric 3×3 matrix yields the square of local change in distances due to deformation (by exclusion of the rotation). Let us note that even though $\nabla\phi$ and $\mathcal{C}$ are functions of t_0, Δt and $\mathbf{x}$, we will occasionally omit these variables in the following for the sake of notational simplicity.

Using standard L_2-norm for vectors, the magnitude of the perturbation is then given as:

$$\|\delta\mathbf{x}(t_0+\Delta t)\| = \sqrt{\left\langle \nabla\phi\,\delta\mathbf{x}(t_0), \nabla\phi\,\delta\mathbf{x}(t_0) \right\rangle} = \sqrt{\left\langle \delta\mathbf{x}(t_0), \mathcal{C}\delta\mathbf{x}(t_0) \right\rangle}. \tag{2.29}$$

Let us assume we are interested in the maximum stretching occurring between points $\mathbf{x}$ and $\mathbf{y}$. This will occur if we align the perturbation with the eigenvector corresponding to the maximum eigenvalue λ_{max} of the deformation tensor $\mathcal{C}$. Hence, if we treat $\lambda_{max}(\mathcal{C})$ as an operator and denote the perturbation aligned with the maximum eigenvalue by $\overline{\delta\mathbf{x}}(t_0)$, the condition of maximum stretching can be expressed as:

$$\max_{\delta\mathbf{x}(t_0)} \|\delta\mathbf{x}(t_0+\Delta t)\| = \sqrt{\langle\overline{\delta\mathbf{x}}(t_0), \lambda_{max}(\mathcal{C})\overline{\delta\mathbf{x}}(t_0)\rangle} = \sqrt{\lambda_{max}(\mathcal{C})}\,\|\overline{\delta\mathbf{x}}(t_0)\|. \tag{2.30}$$

Even though $\sqrt{\lambda_{max}(\mathcal{C})}$ is the factor by which a perturbation is maximally stretched, perturbations often grow exponentially in time near (Lagrangian) coherent structures

and, thus, introducing a scaling is typically better suited for locating such structures on the basis of the FTLE. Especially for large time intervals Δt and a large spatial flow field domain, $\sqrt{\lambda_{max}(\mathcal{C})}$ can become numerically unstable. If we define the FTLE, denoted by $\sigma_{t_0}^{\Delta t}$, for a point $\mathbf{x}$ in the spatial flow domain at time t_0 and finite integration time Δt with such a scaling in mind, it can be defined as:

$$\sigma_{t_0}^{\Delta t}(\mathbf{x}) = \frac{1}{|\Delta t|} \ln \sqrt{\lambda_{max}(\mathcal{C})} \, . \tag{2.31}$$

Then equation (2.30) can be rewritten as:

$$\max_{\delta \mathbf{x}(t_0)} \|\delta \mathbf{x}(t_0 + \Delta t)\| = e^{\sigma_{t_0}^{\Delta t}(\mathbf{x})|\Delta t|} \, \|\overline{\delta \mathbf{x}}(t_0)\| \, . \tag{2.32}$$

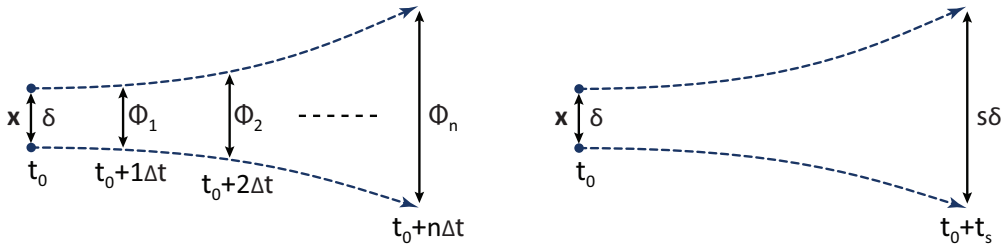
Note that using the absolute value $|\Delta t|$ makes it possible to compute the FTLE in forward and backward time. This fact gives rise to the possibility to detect coherent structures akin to stable and unstable manifold. Furthermore, it is also suited to identify hyperbolic trajectories by an intersection of coherent structures extracted from the respective (forward- and backward-time) FTLE fields [140].

Intuitively, the FTLE can be seen as a value derived from the spectral norm (i.e., matrix L_2-norm) of the flow map gradient, due to the fact that (in the reasoning above) we assumed the perturbation is aligned with the eigenvector corresponding to the largest eigenvalue of the deformation tensor.

Several modifications for the FTLE have been proposed in the literature. For the numerical computation of the FTLE, one has to estimate the flow map gradient by using a discrete set of trajectories initialized very close to the reference trajectory starting at point $\mathbf{x}$. Since trajectories tend to separate at an exponential rate from the central trajectory, Benettin et al. [9] propose a frequent renormalization for them. This can be achieved by subdividing the finite time interval into separate pieces and computing the flow map gradient as the product of the piece-wise obtained gradients.

The FTLE can exhibit finely detailed structures with a spatial variation exceeding the one of the underlying velocity field by far. Therefore, the FTLE is often not accurately computed at an arbitrary point in the domain, but rather sampled as spatial average at a resolution defined by a discretization grid. E.g., for flow fields given on a cartesian grid, a discrete version of the FTLE is commonly approximated by a scalar field exhibiting the same alignment and resolution of the underlying data set. Here, the flow map is sampled at the nodes of the grid and gradients are then estimated by finite differences [56]. The maximum separation is then given by the largest eigenvalue of

the deformation tensor, which can be found using, e.g., the method presented in section 2.3.5. Let us note that, in this setup, the initial perturbation is generally not aligned with the eigenvector associated with the largest eigenvalue of the deformation tensor, but rather axis aligned with respect to the flow field grid. However, the perturbation will typically align very quickly with this direction. The reason for this is that if an axis aligned perturbation has a component in the respective eigenvector direction, then this component will quickly dominate because it is aligned with the most unstable direction.



$FTLEM(t_0, \Delta t, \mathbf{x}, n) = \max_{k=1,\dots n} \frac{1}{|k\Delta t|} \ln(\Phi_k/\delta)$ $\quad$ $FSLE(t_0, t_s, \mathbf{x}) = \frac{1}{|t_s|} \ln(s)$

Figure 2.10: *Modifications to and variants of the FTLE. Left: The FTLEM evaluates the distortion multiple times along a trajectory. Right: The finite-size Lyapunov exponent yields the shortest necessary time it takes for two particles to separate by a given factor s.*

Sadlo et al. [137] introduced the finite-time Lyapunov exponent maximum (FTLEM), which is more suitable to detect the maximum separation along particle trajectories. They evaluate the finite time interval at ever increasing length, i.e., the flow map construction and FTLE evaluation are performed incrementally (see Figure 2.10 (left)). By taking the maximum of all FTLE values sampled at discrete samples in the time interval, this approach is able to capture high expansions along the trajectory instead of only analyzing the final flow map. This approach makes the FTLE's significance less dependent on an appropriate parameter choice for the length of the finite time interval.

Kasten et al. [73] proposed a redefinition of the FTLE to local criteria on the center trajectory, i.e., they estimate the perturbation about a trajectory by evaluating the Jacobian of the velocity vector field at discrete sample locations along the characteristic curve. This variant is known as localized FTLE (or short L-FTLE).

Aurell et al. [7] proposed the finite-size Lyapunov exponent (FSLE) as an alternative to the FTLE (see Figure 2.10 (right)). This measure yields the shortest necessary time it takes for two infinitesimally close particles to separate to a given distance. The motivation was to make the measure independent of the advection time because different regions of a system often require different parameter choices.

2.5.4 Coherent Structure Detection

If the FTLE is computed at each sample point in the flow domain, it is technically an Eulerian scalar field. In unsteady flow fields, the FTLE itself varies as a scalar function of space and time. However, since it is derived from particle trajectories, it is generally thought of as a Lagrangian quantity. Consequently, coherent structures detected on the basis of the FTLE are commonly called *Lagrangian coherent structures* (LCS).

To understand how the FTLE can be used for LCS detection, let us reiterate the meaning of stretching about the trajectory of point **x**. In the left image of Figure 2.9, point **x** and its perturbed point **y** resides on either side of an unstable manifold. If we integrate these points forward in time, they will most likely diverge from each other. Likewise, if the points are situated around a stable manifold, the distance between them will grow if we integrate backwards in time. Thus, both types of manifolds act as separatrices, i.e., they separate the flow into regions of different dynamics. The FTLE at points in the domain close to a separatrix is most likely much higher than for points residing within a region of coherent motion.

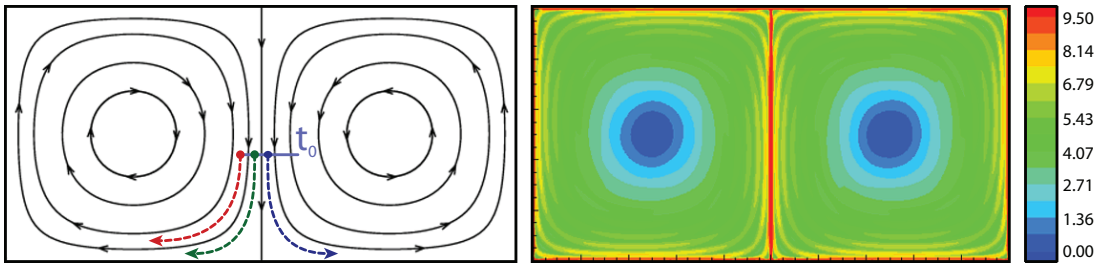


Figure 2.11: *FTLE in a stationary 2D flow field of two counter rotating vortices. Left: Streamlines in the flow domain as well as the trajectories of three points advected for the same amount of time are shown. Right: The corresponding color coded FTLE scalar field. (images courtesy of S. Shadden [152])*

Let us further imagine a simple (instantaneous) 2D flow field. The left image in Figure 2.11 depicts stream lines in the analytic *double gyre* flow field [153]. Here, at the center of the horizontal axis a separatrix divides the flow along the vertical axis into two regions of different dynamics, i.e. two counter rotating vortices. In the left image trajectories of three particles initialized in close vicinity are shown. As can be seen, after a fixed integration time, the distance between particles starting on either side of the axis of symmetry differs the most. If we calculate the FTLE at each point in the flow domain, respective values will be largest along this axis.

This example also emphasizes the advantages of the FTLEM approach, which interleaves flow map computation and FTLE evaluation. The standard FTLE might yield

spuriously small values even for trajectories initialized on either side of the separatrix if an inappropriate advection time is chosen (e.g. particles traveling around either of the vortices once, therefore ending up at their initial positions close to each other).

LCS classification

Boundaries of regions of coherent motion will be situated in regions of high FTLE, or more precisely correspond to the local maxima of the scalar FTLE field. The notion of a local maximum of a scalar field $s : \mathbb{R}^n \to \mathbb{R}$ is unambiguously defined by a vanishing gradient and negative second derivatives in all possible directions. In the literature, a variety of approaches to relax this definition in order to obtain d-dimensional maxima or minima have been proposed. *Height ridges* [60, 37, 104] are a well accepted approach to classify LCS in FTLE fields and will be discussed briefly in the following.

Height ridges are lower-dimensional (elongated) regions of relatively high values. They reside at locations where the scalar field s exhibits a maximum in at least one direction. Such ridges are d-dimensional manifolds in n-dimensional space ($n > d \geq 0$) and they can be identified by an analysis of the Hessian matrix $\mathbf{H}(\mathbf{x})$ of s at point $\mathbf{x}$. If $\mathbf{v}_i$ $(i = 1, \ldots, n)$ are the unit eigenvectors of $\mathbf{H}$ ordered by the corresponding eigenvalues $\lambda_1 \leq \cdots \leq \lambda_n$, then $\mathbf{x}$ resides on a d-dimensional height ridge if:

$$\lambda_{n-d} < 0 \quad \wedge \quad \forall j = 1, \ldots, n-d : \ \mathbf{v}_j \cdot \nabla s(\mathbf{x}) = 0 \, .$$

Sadlo et al. [137] propose to combine this ridge detection criterion with a height threshold $(s(\mathbf{x}) > s_{min})$ to exclude regions of small FTLE and an additional curvature threshold c for the second derivative $\lambda_n < c$ to suppress flat regions in the data set.

Multiple further ridge extraction approaches exists, however a detailed description is beyond the scope of this section. Let us mention topological approaches [141], techniques based on watershed segmentation [178, 116] and particle based approaches [77]. Techniques discussed in [151, 126] focus on the extraction of 2D ridges in 3-space.

Continuous Spatial LCS Representation

Since the FTLE field is commonly sampled on a discrete lattice, additional measure have to be taken into account to determine a continuous LCS representation within the cells (i.e. between the grid nodes) of the respective lattice. Commonly methods from the family of *Marching cubes* [108] algorithms are employed to obtain a linear approximation of ridges between the grid nodes. These techniques generate line segments (1D LCS) or triangles (2D LCS) to approximate LCS within the grid cells. Here, primitive

edges intersect the edges between grid nodes where the desired value is located. These techniques result in a continuous iso-contour or iso-surface, respectively.

Furthermore, since eigenvectors (of the Hessian) lack an orientation, directional derivatives at adjacent grid nodes do not exhibit a consistent orientation. This makes it impossible to determine iso-surfaces without further ado. Methods to solve this problem apply a *principal component analysis* (PCA) to make the eigenvectors of a cell consistent. For readers interested in these approaches we refer to the *Marching ridges* technique by Furst [68] and the work by Sadlo et al. [137]. For the extraction of ridges in 1D, we refer to the parallel vectors approach by Peikert [125]. The application of feature flow fields for ridge extraction is discussed in [69].

A final issue for visualization is the orientation of respective iso-surfaces. Kindlmann et al. propose to employs an additional post-processing pass after surface extraction to ensure a consistent surface orientation [78]. For non-orientable manifolds this problem can be avoided by using two-sided normals in the final rendering of the extracted surface.

Temporal Coherence

The original Lyapunov exponent (MLE) is constant along a trajectory. This property holds approximately for the FTLE if the integration time is chosen to be sufficiently long. Therefore, LCS extracted from this quantity are approximately material surfaces and are essentially advected with the flow. This makes LCS of special interest in the study of transport and mixing processes in unsteady flows.

However, this raises the issue commonly adherent to feature based flow visualization, namely the tracking of features over time (see Section 2.2.3). Techniques proposed in [139, 105] exploit the temporal coherence of LCS to efficiently compute time series of FTLE ridges by interleaving the advection of a 2D sampling grid and incremental tracking of 1D ridges on the respective grid.

To conclude this section, let us note that, in general, the techniques presented above (namely FTLE computation and LCS extraction through ridge classification and iso-surface reconstruction) are by far not suited for a real time exploration of 2D LCS in 3D unsteady flow, as they require time-consuming numerical operations. However, in Chapter 7, we will propose a technique, which—for the first time—makes an interactive exploration of unsteady flow based on the concepts of LCS concepts possible. This is achieved (on the assumption of the temporal coherence of LCS) by combining aspects from feature based flow visualization (namely FTLE computation and ridge extraction) with geometry based visualization approaches (i.e., streak surface extraction).

Chapter 3

Programmable Graphics Hardware

GPUs have rather recently been introduced into the mainstream market, but have become an inherent part of today's desktop computers. Concepts for the first PC add-on graphics accelerator cards originate from dedicated graphics workstations such as SGI's RealityEngine [6], which implemented the 3D rendering pipeline with a SIMD[1] processing paradigm in parallel vertex and pixel engines. Early GPU generations were mainly designed as accelerators for 3D computer games and provided a fixed-function pipeline for the effective rasterization of a large number of triangles. Due to the customers' growing demand for realistic computer game graphics as well as the programmers demand for higher flexibility and high level programming models, GPUs have evolved into full-fledged, almost freely programmable processors.

In this chapter, we will introduce the rendering pipeline of modern GPUs, show how the GPU hardware performance has evolved over the last decade and introduce concepts of the DirectX graphics API, which was employed in the course of this thesis to validate the proposed approaches. Readers unfamiliar with the concepts of GPUs will be given a compact overview of the programmability and inherent restriction of this platform, which in turn will help to understand certain algorithmic choices we made while developing the algorithms presented in this thesis.

3.1 The Rendering Pipeline

Todays GPUs are massively parallel SIMD processors following the stream programming model [72]. In this model, all the data is represented as an ordered set of data of the same type (commonly called a *stream*). The data type can be an arbitrarily com-

[1] SIMD = Single Instruction Multiple Data

plex combination of fundamental data types. A data stream can be of any length but operations on streams are most efficient if they are long, i.e. comprising thousands of elements. Computational operations on streams are performed with a *kernel*. A kernel takes one or more streams as input and produces one or several streams as output. Kernel output relies only on the kernel input, and within a kernel, computations on one stream element never depend on computations performed on another element inside the stream. Thus, during kernel compilation the data required for kernel executions is completely known. If an interdependence of computations on individual stream elements within a single kernel is assured, serial kernel calculations can be effectively mapped onto data-parallel hardware (i.e., the stream elements are processed in parallel). Applications following the stream programming model are designed by chaining multiple kernels together.

The rendering pipeline has been developed with respect to the stream programming paradigm and it is structured into computational stages connected by data flow between these stages. Early GPU generations provided only fixed-function kernels, allowing the programmer to change kernel behavior by a certain amount through a set of predefined state objects before processing an entire stream, but they omitted the possibility to define or change the instruction set of a kernel directly. Over the last decade, GPUs introduced the possibility to freely program certain kernels in the graphics pipeline. Figure 3.1 depicts an abstract view on the rendering pipeline, and the following list gives a rough overview with short descriptions of all stages before we will describe in detail how graphics APIs map this concept to graphics hardware.

- **Input Assembly**: This stage defines the geometric topology of the input data as well as how data given in the form of one or several input streams should be combined and scheduled into the geometry processing stage. The input assembly defines in which input stream and at which location in the data block of one stream element a distinct vertex attribute (e.g. its position in object space) is located.

- **Vertex Shader**: This kernel performs calculations on a per vertex basis. It takes all attributes issued per vertex as input and performs transformations from one reference system into another as well as additional operations such as lighting calculations.

- **Rasterizer**: The rasterizer stage computes the screen coverage of every input primitive and converts its continuous representation into a discrete set of fragments. For each pixel in the frame buffer covered by the processed primitive interpolated vertex attributes are issued to the output fragment stream.

- **Pixel Shader**: A pixel shader kernel is invoked for every fragment output by the rasterizer. It receives a set of interpolated vertex attributes per input element and computes the output value(s) written to the corresponding pixels in the target texture buffer(s), e.g., the color value in the frame buffer or the distance to a reference image plane (z-Buffer).

- **Output Merger Stage**: This final stage of the rendering pipeline controls how the values in the target buffers should be changed according to the processed fragment attributes and externally set states. It may discard fragments if they fail depth or stencil operations and, thus, performs visibility test on a per fragment basis. It also renders it possible to manipulate the color in the frame buffer according to alpha-blending instead of replacing the value in the target pixel and, thus, facilitates the display of semi-transparent objects.

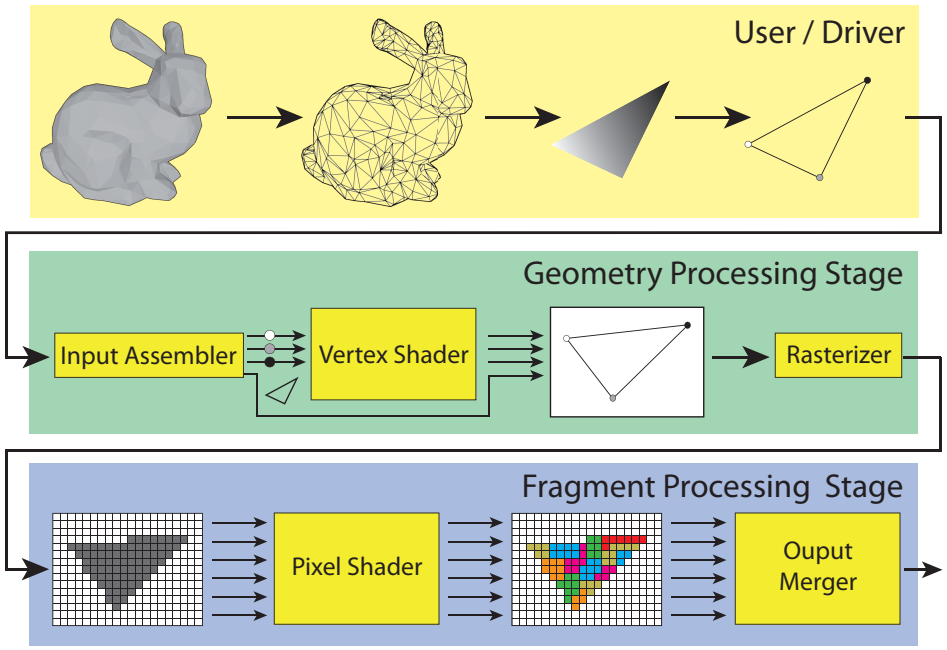


Figure 3.1: *Data flow of the rendering pipeline: A scene is decomposed into a stream of triangles and sent to the GPU. The input assembler schedules triangle vertices into the vertex shading units. A kernel transforms each vertex into screen space and calculates additional attributes, e.g. per vertex lighting. The output stream is passed to the rasterizer, which reassembles triangles based on information provided by the input assembler. The rasterizer converts the continuous representation into a set of fragments by scan-line conversion and passes the fragment stream to the subsequent pixel shader stage. For each fragment, carrying linear interpolated vertex attributes, a pixel shader is executed. This kernel calculates per fragment operations such as texturing and directs the resulting stream to the output merger stage. Within this stage, additional per fragment operations are performed to determine whether (visibility z-test) and how (alpha blending) the corresponding pixel in the frame buffer should be altered.*

3.2 Evolution of GPUs

This sections is intended to give a short overview on the development of GPU performance. For a thorough view on GPU history we refer the reader to [88]. Over three decades ago, Intel co-founder Gordon Moore observed that the amount of transistors on a single die doubled on a biannual basis [117]. Even today, this statement holds true and has led to an exponential growth in raw compute performance. In the context of GPU development, the magnitude of hardware evolution is usually quoted as Moore's law cubed. Even though performance gains do not scale linearly with increasing transistor count, enormous performance improvements from one GPU generation to its successor can be observed. The highly competitive GPU market with its rapid changes in hardware development has led to a decreasing number of competing manufacturers. Since the beginning of the 21^{st} century, graphics hardware development is mainly governed by two major competitors, namely NVIDIA and AMD/ATI. The following tables list the most important GPU chips featuring programmable components, sorted chronologically by release year. Table 3.1 shows all important GPUs released by NVDIA, table 3.2 all relevant hardware generations released by ATI, respectively. Column *GPU model* lists the name of the respective consumer-class (flagship) GPU released in the respective year, column *Memory* the maximum amount of memory (in MB) available for each card and column *Shader units* the number of programmable shader cores. Values in brackets correspond to the amount of shading units dedicated to different stages in the pipeline, namely Vertex (*v*) and Pixel Shading units (*p*). Since the introduction of unified shading hardware in 2007, programmable shading units adhere a unified programming specification, thus, provide a single computational pool of programmable resources for the programmable pipeline stages. Columns *Core clock* and *Memory clock* list the reference clock frequencies (in MHz) specified by the manufacturer, and the last column shows the *Fill rate* in million textured pixels per second (MT/s).

GPU model	Year	Memory (MB)	Shader units (v:p)	Core clock	Memory clock	Fill rate
GeForce 256	1999	64	4 (0: 4)	120	166	480
GeForce 2 Ultra	2000	64	4 (0: 4)	250	460	2000
GeForce 3 Ti 500	2001	128	5 (1: 4)	240	500	1920
GeForce 4 Ti 4600	2002	128	6 (2: 4)	300	650	2400
GeForce FX 5900	2003	256	7 (3: 4)	450	850	3600
GeForce 6800 Ultra	2004	512	22 (6:16)	400	1100	6400
GeForce 7800 GTX	2005	512	32 (8:24)	430	1200	15600
GeForce 8800 Ultra	2007	768	128	612	2160	39168
GeForce 9800 GTX	2008	1024	128	675	2200	43200
GeForce GTX 285	2009	2048	240	648	2484	51850
GeForce GTX 480	2010	1536	480	700	3696	42000

Table 3.1: *NVIDIA GPU revisions sorted by release year. Information is taken from [2].*

GPU model	Year	Memory (MB)	Shader Units	Core Clock	Memory Clock	Fill Rate
Rage 128	1999	32	2 (0:2)	125	143	250
Radeon 7200	2000	64	2 (0:2)	183	183	1098
Radeon 8500	2001	64	6 (2:4)	275	275	2200
Radeon 9700 PRO	2002	128	12 (4:8)	325	310	2600
Radeon 9800 XT	2003	256	12 (4:8)	412	365	3296
Radeon X800 XT PE	2004	256	22 (6:16)	520	560	8320
Radeon X1900 XTX	2006	512	66 (8:48)	650	775	10400
Radeon HD 2900 XT	2007	512	320	743	1000	11900
Radeon HD 4670	2008	1024	320	750	1100	34000
Radeon HD 5770	2009	1024	800	850	1200	10400
Radeon HD 5970	2010	2048	1600	725	1000	46400

Table 3.2: *ATI GPU revisions sorted by release year. Information is taken from [1].*

3.3 Graphics APIs

To facilitate a more production friendly environment, standardized graphics APIs such as OpenGL [4] or DirectX [3] abstract from the rapidly changing hardware implementation and are nowadays commonly used to communicate with a GPU. They allow programmers to write portable code that can be executed in all hardware environments fulfilling API specific standards. The first OpenGL specification was released by SGI[2] in 1992. Microsoft introduced their DirectX graphics API about three years later as a component of the *Windows 95* operating system. More than a decade later, both APIs still coexists and expose comparable concepts to communicate with the underlying hardware layer. As both APIs map to the same hardware, there usually exists a one to one mapping from one APIs functionality to the other one's. While OpenGL implementations are available for various operating systems, the application of DirectX is still restricted to Microsoft's proprietary operating systems. Thus, selecting the right API for a graphics application rather depends on the programmers preferences and the operational environment than any API related constraints.

Both APIs, however, approach different ways to add new functionality to the existing standard. New OpenGL major revisions are carefully maintained by a group of specialists from different fields of interest. The so called architecture review board (ARB) contains members from a large variety of companies. Since 2006 the Khronos group—an industrial consortium consisting of more than 100 members from different companies, e.g. AMD, Intel, NVIDIA, SGI, Google or Sun Microsystems—has taken over the supervision of further development of the OpenGL API. New functionality is added in a process consisting of multiple stages, following the concept of extensions.

[2] SGI = Silicon Graphics Incorporated

GPU manufacturers have the possibility to expose a new hardware feature immediately by releasing an extension tailored to specific hardware revisions. Functions and constants belonging to such an extensions can be identified through a manufacturer specific postfix. If multiple manufacturer decide to expose identical functionality, components are marked with the EXT postfix. If the ARB decides to assimilate an extension, its postfix is changed to ARB, and there is a high probability that the extension will become an integrated component in an upcoming major revision of the OpenGL specification. This standardization model provides instant access to new (or experimental) hardware functionality, but it makes application code vendor-dependent and there exists the risk of ceased support for (experimental) extensions.

DirectX in contrast is under close supervision of Microsoft and up-to-date versions usually require a feature set that will become available with future hardware generations. New specifications are released only after the requested feature set is met. Even though new API specifications are developed in close collaboration with GPU manufacturers, this approach enforces GPU developers to design hardware with the inflicted specifications in mind. DirectX programmers, however, have the advantage to develop applications that will (most likely) run on all upcoming GPUs.

Even though one would assume that the OpenGL specification model would deliver new features faster, the development of both APIs over the last years has taken another direction. Major DirectX API revisions, namely versions 9.0c and 10, successfully introduced well-defined sets of new GPU features faster than the rivaling OpenGL standard. Due to this reason we decided to use the DirectX API for the validation of our proposed approaches. While this restricts the application to Microsoft's proprietary family of operating systems, it ensures that the software is applicable in a wide range of heterogenous hardware environments. In the following we will take a closer look at two major API versions employed in the course of this thesis.

3.3.1 DirectX 9.0 and the Shader Model 3.0

As introduced before, the rendering pipeline comprises a set of kernel stages connected by fixed data paths (see Figure 3.1). The DirectX 9.0 standard was released in 2002 and supports two programmable stages in the rendering pipeline, namely *vertex* and *pixel* shaders. Together with this API version two Shader model (SM) standards have been introduced, laying the groundwork for scientific computing on the GPU.

Shader Model standards define all capabilities a chip of graphics hardware has to support to call itself compliant to the standard. Basically it defines the data structures, the execution processes (the pipeline) and all states and buffers.

With the SM 2.0 standard, floating point data structures and arithmetic were introduced. The Shader Model 3.0 demands access to texture resources in the vertex shader stage, thus, for the first time giving rise to the possibility to change geometry dynamically on the GPU. Outsourcing vertex attributes into texture resources (e.g. spatial coordinates) and updating the content of these textures through a separate execution of the rendering pipeline—by rasterizing into the respective textures—allows to manipulate geometry on the GPU without the need to down-/upload data to the CPU. This feature is the fundamental basis for an interactive GPU-based particle engine [90]. With the SM 3.0 standard, Microsoft also introduced the HLSL programming language (high level shading language), which allows programmers to write shader kernels with a syntax similar to the C programming language.

Data Types and Structures

Table 3.3 lists all fundamental scalar data types supported by the Shader Model 3.0 standard in all programmable shader stages of the rendering pipeline. These fundamental data types can be grouped into vectors of up to 4 components or square matrices up to an order of 4. Most intrinsic functions support 4-tuple parameters and are executed in parallel on all components.

Data Type	Representable Value
bool	true or false
int	32-bit signed integer
half	16-bit floating point value
float	32-bit floating point value
double	64-bit floating point value

Table 3.3: *Fundamental data types in the Shader Model 3.0 standard*

Data structures on the GPU reside in local video memory, which we will refer to as GPU memory from now on. Only a limited number of data structures is available on the GPU, and can be categorized as follows:

1. **Vertex Buffers** are intended to store data associated with the vertices of a geometric object. Common attributes are coordinates in 4D homogeneous space, normals or texture coordinates. Each attribute can contain up to four scalar data types of type int or float.

2. **Index Buffers** are used in conjunction with vertex buffers and enable programs to benefit from GPU mechanisms to cache intermediate results. An index is a pointer

to a vertex residing in a separate buffer. Vertices are often shared among multiple primitives—e.g adjacent triangles—describing one geometric object. With index buffers, multiple copies of the same vertex data can be avoided, thus, reducing the size of the pipeline input stream. Vertex shaders cache intermediate results and if an index points to a vertex that has already been processed and still resides in cache, these values are reused instead of invoking the vertex shader instruction set again. The application of index buffers can drastically reduce the processing time of the vertex shader stage if the vertex data is stored in a cache coherent manner. Each index buffer element consists of an unsigned integer of either 16 or 32-bit precision.

3. **Texture Resources** are 1D, 2D or 3D arrays of data, whereas each array-element can be a tuple of up to four 32-bit values. Array elements are usually referred to as *texels* (1D and 2D) or *voxels* in the case of 3D textures. Dependent on parameters specified at resource creation, textures can support read/write access to the CPU and GPU. Though all four features are never supported at once. E.g., a GPU writable texture is never CPU readable and writable.

The Shader Model 3.0 Rendering Pipeline

The Shader Model 3.0 standard realizes the rendering pipeline as a set of fixed and programmable stages connected by fixed data paths as depicted in Figure 3.2. The role and limits of each kernel stage according to the DirectX 9 standard are listed in the following:

1. **Input Assembly stage**: The input assembler stage contains setting groups to define how input streams, representing a geometric object based on a set of vertices and relations between them, are interpreted by the GPU.

   (a) *Vertex Layout setup* The vertex layout consists of an array of element descriptors defining how the data of one vertex element has to be interpreted and how it is assembled from a set of input streams. Each element descriptor entry contains following declarations: The *stream id* identifies the stream a vertex attribute resides in. The *offset* descriptor contains the byte offset from the beginning of the vertex data to the data associated with the particular vertex attribute. The *data type* descriptor as one of several predefined types is used to determine the data size of a vertex attribute. A *usage* enumerator defines what the data will be used for, e.g., if the attribute contains position

or texture coordinates. A single vertex can carry up to 64 scalar attributes or more precise 256 bytes per-vertex data.

(b) *Vertex buffer setup*: One or more vertex buffers can be bound to the pipeline prior to the execution of the rendering pipeline. This settings group is used to define the ordered set of currently active vertex buffers. Each vertex buffer consists of an array of vertices, whereas each array element contains the same amount of vertex attributes defined in the vertex layout and associated with the corresponding input buffer.

(c) *Index buffer setup*: Indices point to vertices in the input vertex stream. A set of indices is called an index buffer and can be used to address vertices that are shared among multiple primitives instead of defining the vertex multiple times explicitly in the vertex stream.

(d) *Primitive Topology*: The vertex topology defines how the input vertex and index streams are interpreted to form a certain type of primitive. Supported types are points, lines and triangles. Also, buffers can be interpreted as either lists, strips or fans. While the list type describes each primitive individually, strips and fans reuse one or more previously indexed vertices in order to save processing time with the help of vertex caching.

The input assembler can additionally be employed to create (system generated) values. These values are generated at various pipeline stages (either given on a per vertex or per fragment basis) and can be accessed by shader kernels in successive pipeline stages. Exemplary system values are vertex ids, a value mapping to the z-buffer depth value or culling information regarding the orientation of one primitive.

2. **Vertex Shader stage**: A *programmable* vertex shader kernel is executed for each vertex element sent into the rendering pipeline. It is intended to transform coordinates between reference systems and to perform general operations on a per vertex basis. The Shader Model 3.0 demands that up to 16 textures can be bound and sampled at this stage. As texture content can be modified on the GPU, but no modifications on arbitrary (vertex) buffers are possible, this gives rise to the possibility to change geometry on the GPU dynamically. Thus, the so called vertex texture fetch is a fundamental pipeline feature for the development of an interactive particle engine.

3. **Rasterizer stage**: After projection and transformation into window coordinates, all vertices are equipped with a 2D position on the target raster and a depth value. The rasterizer processes the edges of one primitive in a scan-line fashion and generates one fragment for each pixel covered by the primitive. An interpolation unit interpolates all vertex attributes demanded by the successive pixel shader stage according to the desired one of several interpolation schemes. The rasterizer stage is configured through a set of rasterizer states. With these states, a programmer can specify the rasterization scheme, i.e if only pixels covered by the edges of a primitive should be filled (*wireframe* mode) or fragments for all pixels covered by the primitive should be generated (*solid* fillmode). A primitive cull mode can be activated and used to discard all primitives facing either towards or away from the camera, thus excluding them from rasterization. Depth/stencil operations allow to discard primitive samples on a per fragment basis in the output merger stage. However, the Shader Model 3.0 introduced a mechanism called *early-z* test, which allows to perform the depth test for a fragment in the rasterizer stage if its depth value will not be modified in the pixel shader stage and stencil operations are disabled. The depth test incorporates a depth/stencil or z-buffer, which stores distances to a reference image plane for all pixels in the frame buffer. If one fragment passes the depth test the corresponding z-buffer pixel value is replaced by the fragment's depth value. Fragments failing the depth test will never contribute to the final result, thus the early-z test allows to efficiently discard fragments before they are sent into the pixel shader stage, which in turn can drastically reduce the load on the pixel shader stage.

4. **Pixel Shader stage**: A *programmable* pixel shader kernel is executed for each fragment generated by the rasterizer. It has access to all interpolated vertex attributes and can read data from external texture resources residing in GPU memory. The Shader Model 3.0 allows to bind up to eight parallel render targets to the pipeline, whereas each texture element can contain up to 4 32-bit scalar values. Thus, within one single rendering pass a pixelshader can output up to 128 bytes of data with each fragment. Furthermore a pixel shader can also modify a fragment's depth value.

5. **Output Merger stage**: This final pipeline stage is responsible to route data into multiple output buffers. Up to eight render targets can be activated at the same time. Geometry rendered into multiple render targets is projected only once and rasterized at the same position in each target. Depending on the output merger's

decision, fragment attributes are either written at the same position into all currently bound output buffers or discarded entirely.

Programmers can configure depth/stencil operations as well as blending functionality through various state objects. While depth/stencil operations allow to discard data on a per fragment basis, the blend state allows to manipulate how pixel values in the target buffer are updated according to the opacity/alpha attribute of a fragment. Blending operations are usually employed to render semi-transparent objects, updating the target color according to a opacity value of a fragment instead of simply overwriting the target value. The output merger is certainly one candidate for another programmable component in the rendering pipeline.

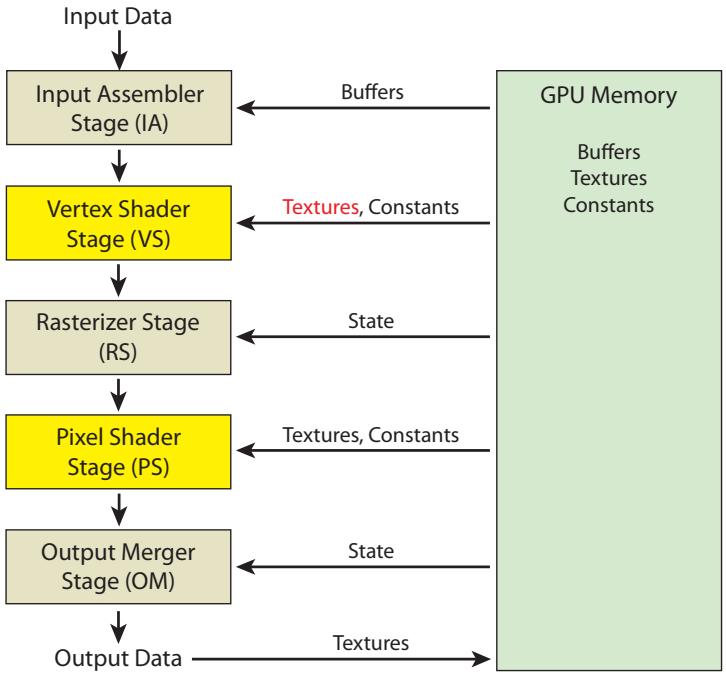


Figure 3.2: *The DirectX 9.0 / SM 3.0 rendering pipeline. Programmable kernel stages are shown as yellow boxes. One important feature introduced by the SM 3.0 standard (highlighted in red) is the possibility to access texture buffers in the vertex shader stage, a fundamental building block to realize interactive particle tracing on the GPU.*

3.3.2 DirectX 10 and the Shader Model 4.0

The DirectX 10 API with its Shader Model 4.0 standard was released in 2007 and is closely coupled to low level system layers of the Windows Vista operating system. For a thorough overview of the complete specification we refer the reader to [11]. Figure 3.3 depicts the DirectX 10 rendering pipeline, important additions are highlighted in red. The most important changes to the DirectX 9.0/SM 3.0 standard are listed in the following:

Data Types and Structures

The SM 4.0 standard loosens restrictions regarding data types that can be represented in vertex and texture buffers. Vertex attribute and texture data layouts now support up to to four components per element, whereas the components can be of type char, short, (unsigned) integer or float. Also 8-, 16- and 32-bit typeless data has been introduced as well as the possibility to reinterpret buffer and data types throughout the rendering pipeline. Furthermore, buffer resources can be grouped into arrays (e.g., a *Texture2DArray* contains multiple 2D texture slices of equal size and format). Bit arithmetic on (unsigned) integer data types throughout the whole programmable pipeline is also one new addition of the SM 4.0 standard.

The SM 3.0 standard featured (global) shader constants which can be scalars, vectors or matrices of fundamental data types. Prior to the execution of the rendering pipeline these global shader variables can be set individually through respective API calls within the application. In the SM 4.0 standard a new type of buffer, so called *constant buffers*, was introduced. Constant buffers are optimized for constant-variable usage, which is characterized by lower-latency access and frequent CPU updates. In HLSL code, these buffers are defined similar to structures in the C programming language. Grouping global shader constants according to their “update frequency“ reduces the bandwidth required to update shader constants as updates are committed at the same time rather than making individual calls to commit each constant separately.

The Shader Model 4.0 Rendering Pipeline

One important feature of the SM 4.0 standard is its demand for read access to buffers in all programmable pipeline stages. Vertex buffers can be bound as one or four component float buffers and (unfiltered) data can be read within the shader kernels. In conjunction with the new *stream output stage*, this gives rise to the possibility to directly manipulate buffer content on the GPU. Further additions to the rendering pipeline as

well as changes to existing kernel stages are listed in the following, the complete SM 4.0 rendering pipeline is depicted in Figure 3.3:

1. **Input Assembly stage**: New system generated values were added to the input assembly stage. E.g., a *primitive id* can now be issued to uniquely identify whole primitives throughout the geometry shader and pixel shader stages. Primitive fans have been removed from the pipeline. Now points, lines and triangles must be either specified as lists or strips. For line and triangle primitives new topology types where introduced in the form of lists or strips with adjacency information. This adjacency information can be used in the geometry shader stage to access attributes of vertices residing on adjacent primitives.

2. **Geometry shader stage**: With the SM 4.0 standard, a new programmable stage was added to the rendering pipeline, namely the *geometry shader* stage. The geometry shader is situated between the vertex shading and pixel shading stages and operates on the primitive level. This stage receives whole primitives, e.g. a line segment or a triangle with adjacency information, and operates on the primitive level with access to the data structures of all primitive vertices. It allows arbitrary operations on each individual vertex data structure but its intended purpose is to amplify or reduce the incoming stream by adding or removing whole primitives. A geometry shader can output up to 1024 32-bit values in the form of vertices, whereas each individual vertex can carry up to 256 bytes of attribute data. Next to amplifying or reducing an input stream, the geometry shader can also change the primitive type itself. For example, textured sprites (glyphs) can be realized efficiently by sending a stream of point primitives into the pipeline and letting the geometry shader issue one quadrilateral in the form of two triangles to the successive rasterizer stage.

   The geometry shader also introduces the possibility to direct its output into specific slices in texture arrays or 3D textures. For each slice targeted by the geometry shader stage, separate rasterizer and output merger stages are invoked, i.e. even the projection and, thus, the area covered by an output primitive can vary.

3. **Stream Output Stage**: The stream output stage renders it possible to stream data directly into buffers residing in GPU memory. It can be activated solely or parallel to the rasterization stage and, thus, allows to update geometry data residing in vertex buffers directly while (optionally) rasterizing further information into texture render targets. As the currently bound input stream buffer(s) cannot be

bound as stream out target(s) to the pipeline, a ping-pong approach using two sets of buffers has to be applied to update geometry data directly on the GPU.

4. **Output Merger stage**: Within the SM 4.0 standard, it is possible to activate blending functionality separately as well as to define individual write masks for each active texture target. 32-bit floating point precision is introduced to the blending stage, thus, offering full float support throughout the whole rendering pipeline.

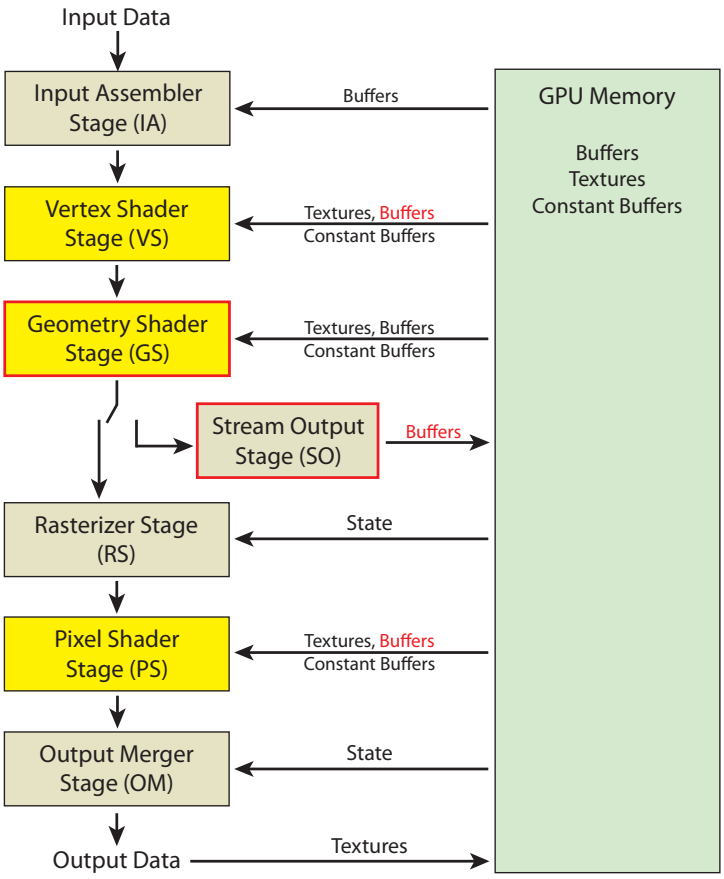


Figure 3.3: *The DirectX 10 / SM 4.0 rendering pipeline. Yellow boxes denote the three programmable kernel stages. The new stream output stage allows to manipulate vertex buffer content directly. Important new features addded to the rendering pipeline are highlighted in red.*

Chapter 4

Interactive Visual Exploration of 3D Unsteady Flows

4.1 Introduction

Interactive visual exploration of 3D unsteady flows is still one of the grand challenges in many areas of science and engineering. Popular applications where such fields arise include computational fluid dynamics and mechanics, as well as medical imaging techniques like functional CT. In the unsteady case the expert gains insight into the underlying physical phenomena especially from the dynamics of the flow. Consequently there is a dire need for real-time techniques that provide rapid visual feedback. These techniques, however, have to be supported by interactive and intuitive metaphors to enable the user to focus on relevant details and to flexibly select the most appropriate visualization option. Only then, the massive amount of 3D information provided to the user can be filtered adequately.

Despite the advances in CPU and graphics hardware technology, existing visualization techniques for reasonably sized 3D unsteady flow fields still cannot run at acceptable rates. As numerical and rendering capabilities continue to increase, so does the size of the data sets to be visualized. Today, time-resolved numerical simulations comprised of billions of grid points are available, making the visualization difficult due to memory constraints. Figure 4.1 shows such a gigantic field that consists of 227 time steps at resolution $512 \times 256 \times 64$ and requires over 20GB to store velocity information. As these requirements will continuously increase in the future, there is a dire need for flow visualization techniques that comprehensively address these issues.

In this chapter, we present a novel visualization technique for 3D unsteady flow fields that addresses the aforementioned requirements. As the size of such data sets usually exceeds the memory capacities of GPUs, additional measures have to be taken to manage the data needed during the interactive flow exploration session. We propose a novel multi-core approach to asynchronously stream such fields from the CPU. By decoupling visualization from data handling this approach results in interactive frame rates.

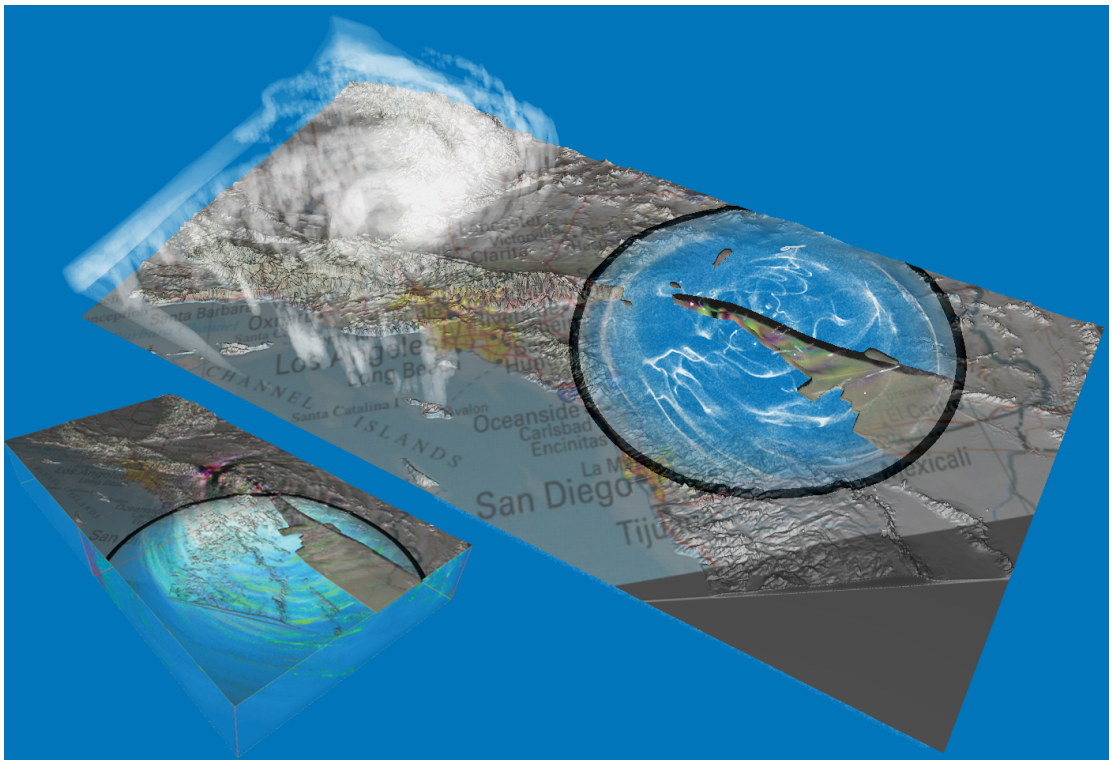


Figure 4.1: *Visualization of the time-resolved Terashake 2.1 simulation data. On a PC equipped with a dual-core CPU and a single GPU, particle-based visualization using 256K primitives in combination with volume rendering runs at over 40 fps.*

4.2 Contribution

The techniques presented in this chapter are based on a streaming approach for time-resolved sequences. In contrast to previous visualization techniques for such fields, both the mapping of visualization data onto renderable primitives and the rendering of these primitives is performed entirely on the GPU. Our approach has the following properties:

- **Memory efficiency**: Asynchronous streaming of the data allows the visualization of an unlimited amount of time steps. Recent advances of multi-core architectures are exploited to abstract from the limited size of the local GPU memory.
- **Exploration efficiency**: Since the reconstruction of local flow features—e.g. stream, streak and path lines, as well as derived scalar quantities—is integrated into the rendering process on the GPU, our system provides instantaneous visual feedback to the user. This accommodates a more efficient and better understanding of even very complex flow phenomena.
- **Visualization efficiency**: Particle tracing and the computation of characteristic lines is performed on the GPU to visualize the dynamics of unsteady flows. This results in a significant performance gain compared to previous approaches.
- **Cost efficiency**: The visualization techniques presented in this work are especially designed for off-the-shelf PC hardware.

The remainder of this chapter is organized as follows: In the next section we discuss related work. In Section 4.4 we show how 3D unsteady flow fields can be stored on the GPU to allow efficient particle tracing and we address data handling and transfer issues inherent to visualization techniques for large data sets. Section 4.5 is dedicated to GPU-based particle tracing, and Section 4.6 discusses a variety of rendering modalities for individual particles. Section 4.7 presents GPU-based integration techniques to extract characteristic lines in unsteady flow fields and Section 4.8 discusses various visualization modalities for particle trajectories. Section 4.9 introduces focus+context techniques for polygonal meshes which facilitate the integration of static boundary regions as context information into the obtained visual flow representation. Finally, we conclude this chapter with a discussion of the main contributions.

4.3 Related Work

In contrast to 3D steady and 2D unsteady flow, the literature on interactive techniques for 3D unsteady flow is amazingly sparse. In this section, we review existing approaches and motivate how our system can fill this gap. As introduced in Section 2.2, the field of flow visualization techniques can be classified coarsely into *dense* and *sparse methods*:

Dense visualization methods [97] seek to reconstruct a single representation for the whole flow domain. To overcome occlusion effects, the process is usually restricted

to regions of interest, such as vortex regions [184] or stream surfaces [160]. The restriction to regions of interest culminates in image-based techniques [171, 98], which trade highly interactive frame rates versus artifacts due to the screen-aligned nature of the regions. Traditionally, unsteady fields are problematic, since it is not a priori clear how non-instantaneous characteristics such as streak or path lines can be integrated into dense methods [43, 156].

In contrast, sparse methods reconstruct characteristic flow features only at specific locations. Particle tracing [136, 21] and the reconstruction of stream, streak, and path lines [95] fall into this category. Also, methods seeking to extract topological structures [64, 163] or features in general [130] can be considered sparse methods. Both classes are appealing in their own right, depending on which aspects of the data should be emphasized; however, Figure 4.2 clearly demonstrates that for large amounts of primitives geometry-based methods naturally converge towards dense methods.

Most sparse methods pay particular attention to proper seeding strategies [167, 123]. However, recent work by Wiebel et al. [186] indicates—maybe opposing common belief—that there is a need for a simple, controllable, and very localized probing metaphor. Mimicking the dye- and smoke-injection of real-life windtunnel experiments, such a metaphor elegantly circumvents problems naturally arising when seeding in unsteady flow fields. Krüger et al. [90] show that a probing metaphor combined with rapid visual feedback is a convenient and highly effective method to explore the complex dynamic structures present in many flow fields. Probing the flow is a very intuitive and valuable tool that gives engineers full control of the visualization process, rather than forcing them to rely on an automatic seeding algorithm.

The first version of the particle engine, constituting the foundation for the work presented in this thesis, was developed at our chair in 2004 by Krüger et al. [90]. The introduction of the vertex texture fetch to the DirectX graphics pipeline (see Section 3.3.1) inspired the development of the GPU particle engine. Before this framework was published, interactive techniques employed pre-computed particle trajectories and uploaded them to the GPU for rendering to enable interaction with the data [20]. Alternative approaches required sophisticated caching strategies [33] or expensive parallel hardware architectures [21] to achieve interactivity. The GPU-based particle engine allowed to trace a huge number of particles in parallel at two orders of magnitude faster than on state-of-the art CPUs available at the time. Since no data communication between the CPU and GPU was required they could also be displayed at interactive frame rates. Thus, virtual exploration of 3D stationary flow fields in a way similar to real-world experiments became possible even on commodity PC hardware.

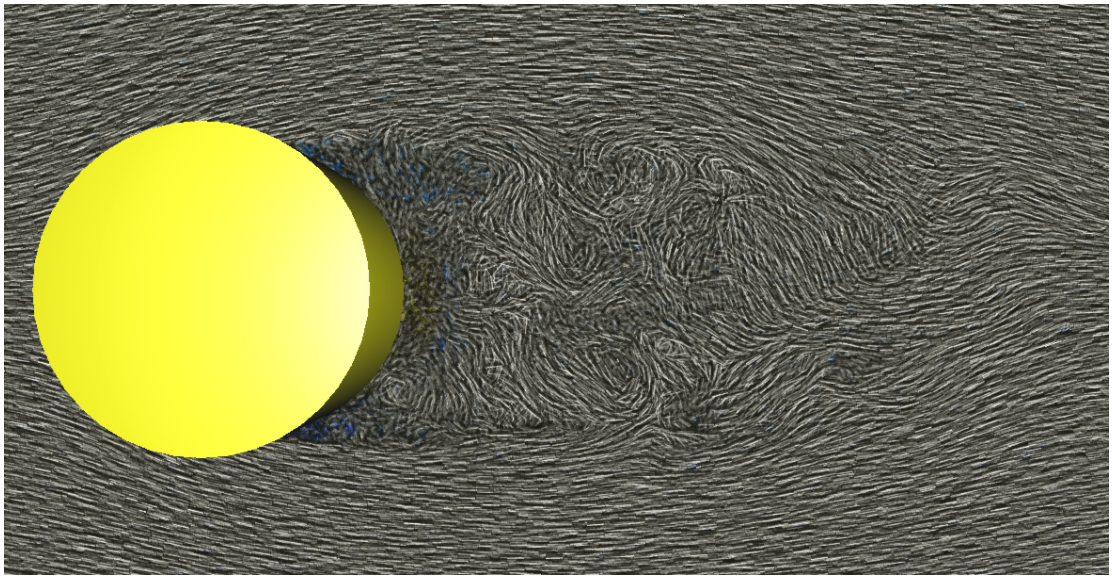


Figure 4.2: *Visualization of a large eddy simulation of the flow around a cylinder. Dense particle sets are visualized using oriented rendering primitives to achieve a "LIC-like" look.*

4.4 3D Unsteady Flow Field Data

Particle tracing on the GPU can be realized most efficiently if the 3D unsteady flow field is given as a time-resolved sequence of velocity vector fields sampled on a cartesian or uniform grid. Velocity data over the whole spatial flow domain can then be stored—for each time step individually—in the *RGB* components of 3D texture resources residing in GPU memory. Consecutive time steps are stored in separate texture resources. This setup is especially suited for GPUs for the following reasons:

Firstly, since the data resides on a structured grid, locating the velocity information for a point in the flow domain requires only a per-component scaling of its position coordinates to transform them from object-space to texture-space.

Secondly, 3D textures allow the fastest and, thus, most efficient way to sample values at arbitrary locations in the flow domain, as GPUs support automatic trilinear interpolation in hardware. A linear approximation $\mathbf{v}(\mathbf{x},t)$ of the velocity data at an arbitrary location in space $\mathbf{x}$ and time t can then be obtained by sampling the two 3D textures containing adjacent time-steps ($t_i \leq t \leq t_{i+1}$) and one additional linear interpolation manually calculated in a shader kernel as described in equation (2.7).

Thirdly, if a fixed integration step size is used, the same amount of work is imposed onto all shader kernels executing the advection of an input particle stream in parallel.

Throughout this work we employ such data structures and all timings presented in

the following adhere to this concept. However, let us note that particle tracing on GPUs is not restricted to such data sets and unstructured time-resolved 3D unsteady flows can also be employed at additional costs with respect to the location of a sample position and varying amounts of integration steps. For an efficient implementation of particle tracing in unstructured grids on the GPU, we refer the reader to [148].

4.4.1 Data Handling

As the size of 3D unsteady flow field data sets usually exceeds GPU memory capacities, an additional abstraction layer is needed to manage the data needed during the interactive flow exploration session. We propose a novel multi-core approach to asynchronously stream such fields from the CPU. This approach decouples visualization from data handling, resulting in interactive frame rates. We employ multi-threading by assigning one thread to consecutively stream one time step after another from disk to the GPU, and another thread to manage integration and visualization specific GPU calls. Since these threads are concurrent per se, the visualization process is entirely decoupled and mostly unaware of the streaming data upload. Consequently, data transfer does not block the visualization thread.

To advect particles seamlessly in an unsteady field $\mathbf{v}$ represented by a discrete set of vector fields $\{\mathbf{v}_i,\ i \in [1,n]\}$ at time steps t_i, we need to store at least three fields in GPU memory. For example, Euler integration requires read access to two fields at times t_i, t_{i+1}, and a third field t_{i+2} has to be available once time integration proceeds beyond t_{i+1}. By implementing a ring buffer, we can dynamically choose how many time steps to keep on the GPU, depending on the order of the time-integration scheme and the global integration step size. As soon as the time index t of the visualization enters the interval $[t_{i+1}, t_{i+2}]$, the memory manager is notified. The manager then advances in the sequence by overwriting the GPU container storing time step t_i with the next time step t_{i+3} (see Figure 4.3). This leads to a very smooth transition in time, and, if the time needed to stream the next time step is smaller than the physical time associated with one interval, the whole sequence can even be explored in real time.

Since graphics cards lack the ability to fetch data directly from disk, the memory manager pre-fetches as many time steps as possible from disk and stores them in CPU system memory. If the entire sequence fits into RAM, it is buffered at application startup and can then be streamed without any further disk access. Otherwise, the manager uses an additional ring buffer which provides containers for a system-specific or user-defined number of data sets. This is illustrated on the right of Figure 4.3.

If only one thread is used to implement visualization and data handling, both disk transfer and the upload of data to the GPU will block the entire application. This is because both operations are issued via blocking system calls. Decoupling the data management and particle tracing tasks into separate threads enables the particle engine to issue rendering calls even while new data is streamed to the GPU. Multi-core architectures benefit most from this implementation; yet even for single core CPUs we observe a significant gain in visualization performance. This is due to the fact that the operating system scheduler switches between the two threads, enabling parallel execution of data upload and issuing rendering calls.

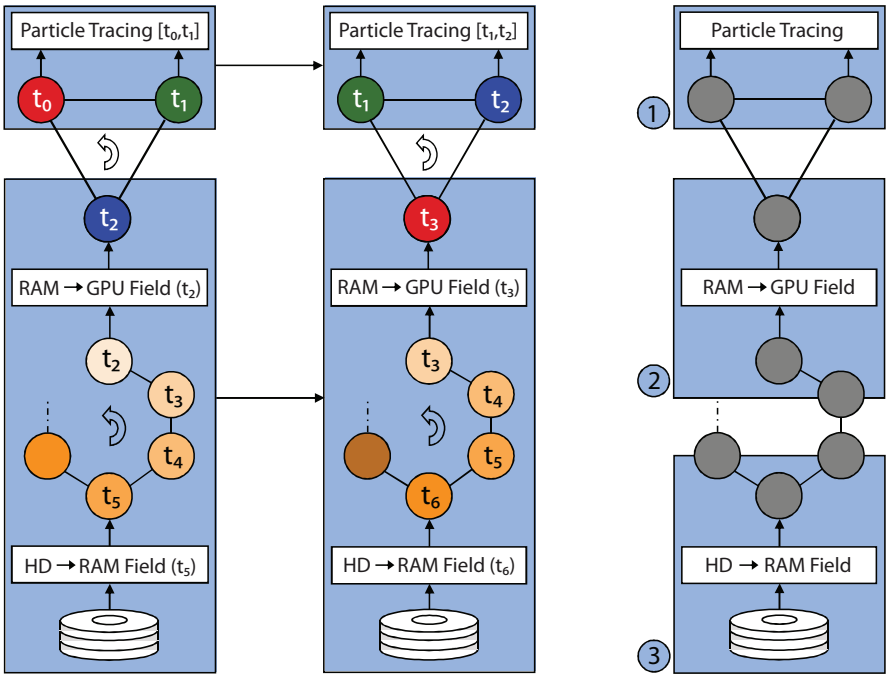


Figure 4.3: *In the left and middle images one cycle performed by the data handler when advancing in the sequence is depicted. The rightmost image illustrates the separation of the asynchronous stream manager into distinct threads.*

Currently, the GPU visualization module and the two memory managers are running in two separate threads (see Figure 4.3 (left)). Once the visualization thread enters the next time interval, it requests the next time step of the sequence that is not yet resident via standard thread communication mechanisms. The memory manager either acknowledges that this time step has already been successfully uploaded to the GPU, or the requested time step is streamed to the GPU. Afterwards, the system memory is updated, overwriting the block containing the now obsolete time step.

Table 4.1 compares the raw data throughput that is achieved for streaming two different data sequences on different architectures. Note that this throughput has been measured with the visualization thread not imposing any additional load. If visualization is enabled, including rasterization and shader operations on the GPU, our experiments have shown a loss in throughput of about 15%. Both test machines are equipped with 3GB RAM, two WD Raptor 74GB hard disks in a RAID0, and an NVIDIA GeForce 7900GTX with 512MB video memory. The single-core CPU is a P4 3.2GHZ, while the dual-core CPU is a Core2 Duo 6600. As can be seen, on the dual-core architecture using the same disk and memory system the multi-threading approach already yields a noticeable gain in throughput.

	LES Cylinder Flow (32 MB/Field)		Terashake 2.1 (96 MB/Field)	
	HD $\rightarrow$ CPU	CPU $\rightarrow$ GPU	HD $\rightarrow$ CPU	CPU $\rightarrow$ GPU
1-Core	90 MB/s	1130 MB/s	95 MB/s	1317 MB/s
2-Core	94 MB/s	1240 MB/s	100 MB/s	1590 MB/s

Table 4.1: *Performance measurements of the stream manager under various configurations.*

On quad-/multi-core architectures the memory management can be split further into separate threads to decouple streaming from disk to CPU and from CPU to GPU. Still, for off-the-shelf PCs, loading from disk is clearly the bottleneck of the system. To alleviate this problem it has proven worthwhile to pre-fetch as many time steps as possible into CPU system memory when the user restarts or pauses the application. A further increase in performance can be gained if more efficient RAID systems are employed.

4.5 GPU-based Particle Tracing

On Shader Model 4.0 compliant graphics hardware, particle tracing can be approached in different ways. Texture-based particle integration employs texture resources to store per-particle attributes and integration is performed in the pixel shader stage by rasterizing into respective texture targets (this technique is also available on SM 3.0 hardware). Buffer-based particle advection employs the geometry shader stage to update particle attributes and exploits the stream-output stage to send updated particles into vertex buffers residing in GPU memory. In the following we will present both approaches in detail and list their pros and cons in terms of performance and flexibility.

4.5.1 Texture-based Particle Tracing

Texture-based particle integration employs a technique commonly referred to as GPGPU[1] programming. In this model, data structures are stored in (the texels of) texture resources. To circumvent hardware limitations regarding the size of 1D textures, we store the attributes of a particle $p_{i,j}$ at the texel position $[i, j](i < m, j < n)$ in 2D textures of size $m \times n$. Each texture resource can contain up to 4 32-bit sized values per component. Multiple attributes can be distributed to several texture resources, and up to 8 textures can be simultaneously bound as output targets to the rendering pipeline (this is commonly referred to as *multiple render targets* or short *MRT*). Thus, up to 128 bytes of particle attribute data can be updated at once through a single invocation of the rendering pipeline.

Updates are performed in the pixel shader stage and are invoked by sending a single quadrilateral covering the whole viewport into the pipeline. In general, DirectX does not allow simultaneous read/write access to texture resources. Thus, whenever the update of an attribute relies on results calculated in a previous step (e.g., the particle position), we employ a *ping-pong* mechanism to access this information. Figure 4.4 shows a flowchart of the texture-based particle integration technique.

GPU-based particle tracing comprises following three major components:

- *Particle Setup*: Each particle is initialized with a *starting position* and a *life time* value before the particle advection loop starts. An initial starting location for all particles is pre-computed on the CPU with respect to a user-selected spatial distribution function, e.g., a uniform or random distribution over the unit cube.

  The flow exploration is coupled with a probing metaphor, allowing the user to interactively change the size and location of a rectangular seeding probe. Manipulating the probe results in a transformation matrix which is applied to the start position whenever a particle is released into the flow domain.

  Additionally, each particle gets assigned a random life time value l in the range $[1 - var, 1 + var]$, where var is a user-specified variance value. l is scaled during particle incarnation by a user-defined global value to equip each particle with an individual life time. Higher variance values result in a homogenous particle distribution over time as particles will disappear and be reincarnated at their starting locations in a random manner.

[1] GPGPU = General Purpose computation on GPUs

The starting location and life time values are copied into a four-component floating point texture (*StartTex*) and are stored as an additional resource in GPU memory. With this setup a user can change the probe location and global life time interactively without the need to re-calculate the initial values and to send the texture resource to the GPU (which would stall the exploration progress). Only if the number of particles or the life time variance are changed an update involving the CPU becomes necessary.

- *Particle (Re)Incarnation*: Whenever the particle setup stage has ended, a simple pixel shader is invoked to initialize particle attributes prior to particle integration. The following operations are performed by this pixel shader: First, a particle's starting location and life time value are read from the *StartTex* resource. Then, the starting positions are shifted from the unit cube to the flow domain object-space according to the probe transformation matrix. Particle life time values are scaled according to the global life time. The probe transformation matrix and the global life time are accessible as global shader constants (residing in GPU memory). The pixel shader writes the updated particle attributes into one of two ping-pong attribute textures used during particle advection.

  During the successive particle advection stage, whenever a particle leaves the flow domain or its life time expires, the same operations are also executed.

- *Particle Advection*: Particle advection is performed in a pixel shader (as depicted in Figure 4.4) and requires access to following resources: The *StartTex* texture is required in case a particle has to be reincarnated. If attributes depend on results from the last advection step, the respective textures also need to be available (e.g., the position of a particle or its life time). Furthermore, the set of consecutive velocity vector fields holding the flow field data confining the current position in time must be provided to the pixel shader for particle integration.

  In every update pass a set of $m \times n$ fragments is generated and processed in parallel by a shader kernel. Each shading unit executing the respective kernel performs the following operations: Current position and life time values are read from the attribute textures filled during the last invocation of the advection pass. The respective read locations are available as system generated values, namely the target texel indices in the output target. Then, the pixel shader checks if the life time of a particle has expired or whether it has left the flow domain. In both cases the particle is reincarnated as described above. Otherwise particles are advected using either the Euler (Eq. 2.3) or the fourth order Runge-Kutta

integrator (Eq. 2.4), which require multiple read operations from the flow field textures and interpolation operations (Eq. 2.7) to compute the necessary velocity field values. Updated position and life time values are written to a render target, which will become the input in the next advection step. Additional attributes, e.g. the velocity at the current particle position, can be written to additional render targets and can be used to determine the appearance of a particle during rendering.

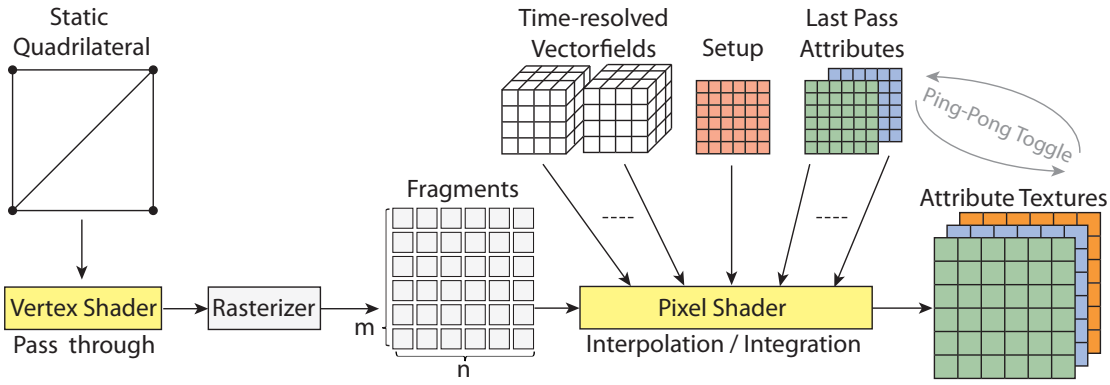


Figure 4.4: *Flowchart for the particle attribute update: A quadrilateral covering the whole target texture is sent into the vertex shader stage and passed through to the rasterizer to generate one fragment for each covered texel. The pixel shader stage accesses the flow field data as well as results from the last iteration and performs the attribute updates in parallel. Textures storing particle attributes that rely on previous results are toggled between successive update iterations.*

4.5.2 Buffer-based Particle Tracing

With the SM 4.0 standard, Microsoft introduced the *stream output stage* to the rendering pipeline. This stage allows to stream intermediate results into (multiple) vertex buffers directly (see Figure 3.3). As the geometry shader allows to output a varying number of primitives for each processed element, this advection technique is especially useful if the amount of particles needs to be changed flexibly over time (e.g., for an adaptive refinement or coarsening of the particle set). Furthermore, it is even possible to stream results into buffers residing in GPU memory and sending the elements to the rasterizer stage in parallel. Thus, particle advection and successive rendering can be performed in a single invocation of the rendering pipeline (see Figure 4.6). This is especially suited if particles are rendered as single point primitives. More importantly, the parallel rasterization can be exploited to store additional information about the buffer content in additional texture resources. For example, if an interdependence adheres to particles in disjunct parts of the linear vertex array, they can store further information by rasterizing

a point primitive into corresponding (disjunct) regions of additional texture targets (see Figure 4.5). By application of different alpha blending operations, information such as the amount of particles or the vertex buffer index of the first/last primitive in the respective bin can be captured. Alternatively, the pixel shader stage can be disabled. Then, the rendering pipeline is only executed up to the stream output stage and further stages of the rendering pipeline are omitted.

Buffer-based particle tracing can be realized with two different approaches. The first approach is more flexible as is provides the possibility to store a larger amount of data per buffer element. The second approach, on the other hand, can only cope with up to sixteen scalar components per particle in one steam output invocation but results in a minimal memory footprint. Both approaches rely on the ping-pong mechanism as described above to access results from the last update pass.

The first technique stores all attributes of one particle within one vertex-element of a single buffer. In this configuration, up to 64 scalar components of per-vertex data (256 bytes or less) can be captured by the output buffer for each processed vertex element. In this setup, the content of a single vertex buffer is sent into the rendering pipeline and updated particle attributes are written into a second buffer bound to the stream output stage. A flowchart for the single-buffer technique is given in Figure 4.5.

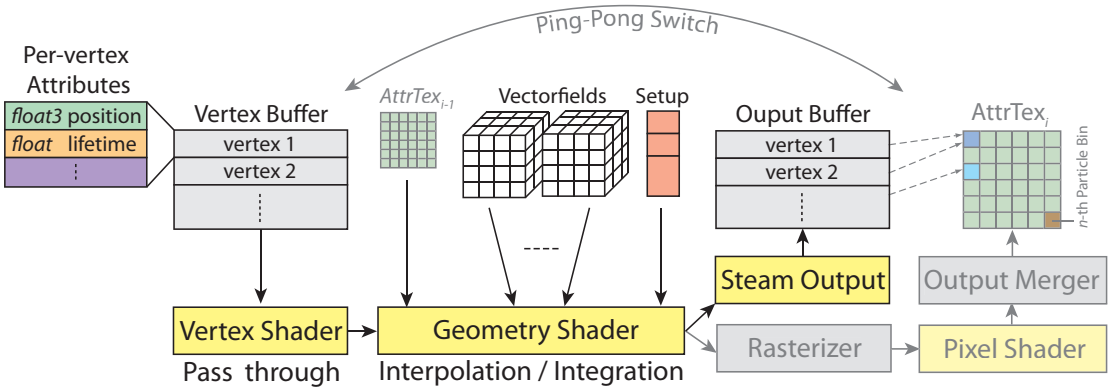


Figure 4.5: *Single-buffer particle update: Per-particle attributes are stored on a per-vertex basis in a single buffer. The particle updates are performed in the geometry shader stage. Buffer Setup contains data required for particle reincarnation. Optionally, the rasterizer and successive stages can be activated and exploited to store further particle (interdependence) information in additional texture resources . For example, here the first two particles in the output buffer are interdependent and share information inside a common texel in the AttrTex output texture.*

Alternatively, attributes can be separated into those who need access to results from the last pass and those who are independent of previous calculations. By storing up to four scalar components per vertex in separate buffers and keeping only two copies

of those buffers that require access to intermediate results during the advection stage, a minimal memory footprint can be achieved. However, streaming data from multiple input buffers to multiple output buffers restricts the total amount of per-vertex attributes to 16 scalar components as only 4 simultaneous output streams can be bound to the rendering pipeline and each output target can only capture a single element (with up to 4 components) of per-vertex data. Let us note that whenever a geometry shader appends an element to the output stream, attributes are written into all buffers bound to the output stage. Thus, it is not possible to distribute an input stream to several output buffers with the help of the stream output stage. Figure 4.6 depicts the flowchart for the multi-buffer particle update technique.

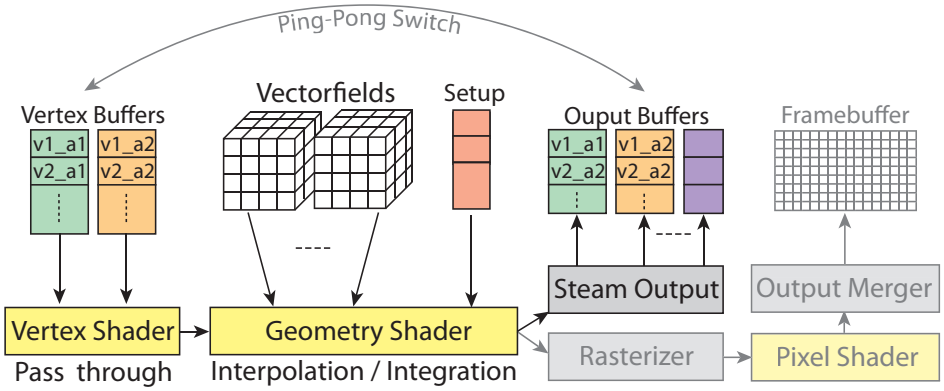


Figure 4.6: *Multi-buffer particle update: Per-particle attributes are grouped depending on whether they need to be reused or not. Up to 4 scalar quantities are stored in one vertex buffer element. Multiple input streams—holding different particle attributes—are sent into the pipeline in parallel. After the attribute update in the geometry shader stage, results are distributed into several output buffers by the stream output stage. Only attributes that rely on previous results need two buffer copies, thus the memory footprint on the GPU can be reduced. Again, the rasterizer and successive stages can be activated optionally to store further information in additional textures (shown in Figure 4.5) or to render the resulting particle set. This illustration demonstrates how particles might be updated and rendered in a single invocation of the rendering pipeline.*

The texture-based particle update approach is the fastest method to perform particle integration as it imposes the least load onto the GPU. Only four vertices (spanning the texture-filling quadrilateral) are processed in the geometry stage. The rasterizer then generates all necessary fragments to invoke a pixel shader kernel per particle. The output merger finally stores the results in the respective texels of the output target(s). The texture-based approach has proven most suitable if a static number of particles is used for visualization, and timings presented in the rest of this chapter correspond to this technique.

The buffer-based technique, on the other hand, requires the execution of (at least) two shader kernels per particle as attributes have to be passed through the vertex shader stage before a geometry shader can send the updated values to the stream output stage. Furthermore, while primitives are processed in parallel in the geometry shader stage, the stream output stage has to maintain the order of the input stream in the output buffer. The buffer-based approach will be employed in Chapters 6 and 7, where an adaptive refinement and coarsening is applied to the particle set to ensure a uniform sampling of integral surfaces.

4.6 Particle Visualization

The simplest way to visualize a particle set is by rendering every particle as a single point primitive covering one pixel in the frame buffer. If the texture-based particle update technique is employed, the particle data cannot be rendered outright. Particle attributes residing in texture resources need to be mapped onto renderable primitives. With the SM 3.0 standard the so-called vertex texture fetch ability was introduced, opening up the possibility to access texture data in the vertex shader stage and, thus, to displace vertices according to position information stored in texture resources. Instead of explicitly storing a static vertex buffer in GPU memory, one can exploit features of the input assembly stage to generate the needed renderable primitives on-the-fly. The input assembly provides optional system generated values to the rendering pipeline, which can be demanded in one of the programmable shader stages. E.g., a *VertexId* can be requested by the vertex shader stage or a *PrimitiveId* can be issued to the geometry shader stage. Binding no input buffer to the rendering pipeline but issuing a draw call from within the application leads the input assembly stage to generate a vertex stream with increasing vertex ids. This vertex stream is sent to the vertex shader stage and texture coordinates to access particle attributes can be calculated through modulo and division operations on the *VertexId*. Particle attributes are then gathered within the vertex shader kernel through the vertex texture fetch ability and additional arithmetic operations are executed to determine the particle's appearance and position in screen space. Then, the results are issued to the rasterizer.

On SM 4.0 capable hardware the vertex texture fetch can be circumvented by reinterpreting the texture data as vertex buffers and, thus, to directly bind the separate attribute textures as input buffers to the rendering pipeline.

If the buffer-based advection technique is used, all the necessary information is already inherently present in the attribute list of each vertex. Thus, by sending this

stream into the pipeline a vertex shader can compute all necessary output values directly on the basis of the input element data without the need to perform read operations on attribute textures residing in GPU memory. If particles are rendered as single points, the buffer-based advection can efficiently be combined with the successive rendering by sending vertices not only to the stream output stage but passing them to the rasterizer in parallel. However, as positions projected into screen space are required to render particles into the frame buffer, the geometry shader must provide these coordinates to the rasterizer stage.

The rendering of point primitives does not necessarily require any special operations in the pixel shader stage, and in particular no texture fetch has to be performed. Using this modality allows for the integration and rendering of millions of particles at interactive rates. Even if each particle is represented by a single pixel, the sheer amount of tiny primitives enables to mimic real-world tracer substances like smoke or dye injected into the flow domain effectively (see Figure 4.7).

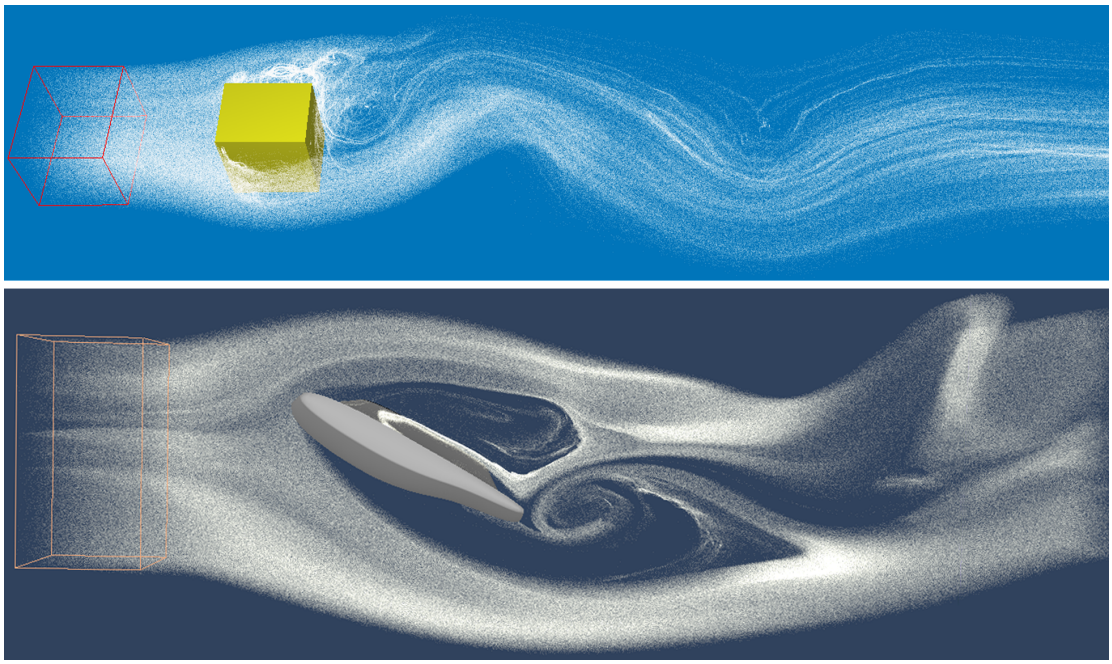


Figure 4.7: *Particle tracers rendered as single point primitives.*

4.6.1 (Oriented) Point Sprites

Rendering complex geometric objects instead of simple points primitives makes it possible to incorporate additional flow properties into the visual representation. However, representing each individual particle primitive by a mesh comprising multiple vertices on its own can drastically decrease the rendering performance due to the increased ver-

tex load. Mimicking volumetric objects by *point sprites* represents an efficient alternative. Here, objects are approximated by screen-aligned, textured quadrilaterals centered at the position of a particle in the frame buffer. Rotationally invariant 3D objects can be simply represented by attaching a photograph of an object to a point sprite.

Complex shapes can be approximated in the form of *virtual geometry*. This approach takes a discrete set of views (under different angles) on a real three dimensional object and projects them into disjunct regions of a 2D texture. Such a texture is commonly referred to as *sprite texture atlas* [55, 90] and, following the parametrization proposed in [90], can be constructed as follows.

To convey directional information we need only two degrees of freedom to align a 3D object with the respective vector direction. If we do not want to encode information into the object's rotation about the direction vector, we can use a shape that is symmetric along one direction—e.g., the x-axis—to reduce the amount of information that has to be stored in the texture atlas. Furthermore, as a point sprite is aligned with the x- and y-axes of the view-space, the rotation of the object around the z-axis takes place in the screen plane and can, thus, be obtained by rotating the texture coordinates of the quadrilateral. Therefore, we only need to parameterize views on the object with respect to the rotational angle about the y-axis in the range $[0, \pi]$ and store discrete snapshots in disjunct columns along one dimension (u-direction) in the 2D sprite texture atlas. To get all rotations from 0 to 2π we access the atlas with the texture wrap mode *mirror*.

Virtual geometry usually has an elongated shape in order to emphasize the velocity direction. By scaling an object along its major axis and encoding (view-dependent) "longitudinal deformations" in the rows of a texture atlas (v-direction), it can be employed to depict directional information as well as the local velocity magnitude. Here, the scaling parameter domain ranges from 0 to 1. An exemplary texture atlas storing virtual geometry of different lengths is shown in Figure 4.8.

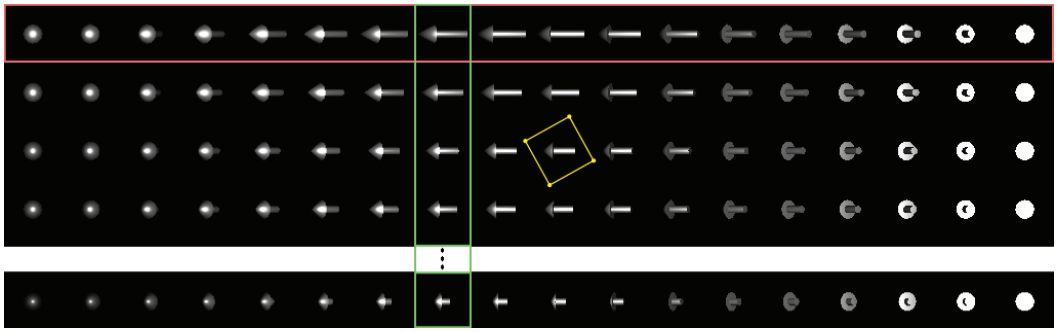


Figure 4.8: *Views on a geometric object under different angles are stored in the columns of a sprite texture atlas (red). Geometries of different length are stored in separate rows (green). The yellow square depicts an exemplary layout for the texture coordinates of a point sprite.*

To render a point sprite, an appropriate subregion from the whole sprite texture atlas has to be selected. This selection is performed on a per-particle basis via a transformation of the uniform texture coordinates at the quadrilateral's vertices. Be $\hat{\mathbf{v}} = (\hat{x}, \hat{y}, \hat{z})^T$ the normalized local velocity vector transformed into view-space. To select the correct sub-image the magnitude of the local velocity vector is used as v-offset and the arc sine of $\hat{z}$ is used as u-offset. The rotation of the virtual geometry around the z-axis is taken into account by a 2D rotation of the texture coordinates about the center of the quadrilateral. The rotation matrix

$$\mathbf{M}_{rot} = \begin{pmatrix} \frac{\hat{x}}{n} & \frac{\hat{y}}{n} \\ \frac{-y}{n} & \frac{\hat{x}}{n} \end{pmatrix} \quad \text{where} \quad n = \sqrt{\hat{x}^2 + \hat{y}^2},$$

is thereby given by the angle between the x-axis and the normalized projection of $\hat{\mathbf{v}}$ into the xy-plane of the view-space. We employ the geometry shader stage to construct a screen-aligned quadrilateral patch. For every particle the kernel receives a single point primitive with corresponding particle attributes as input and computes the four screen-aligned vertices spanning the quadrilateral patch. A global shader constant determines the size of all point sprites, however, to achieve the impression of perspective foreshortening the geometry shader adjusts the area covered by a point sprite according to the z-component of the particle position projected into screen-space. The quadrilateral is then tesselated into a triangles strip of two primitives and issued to the rasterizer stage. The pixel shader finally fetches the virtual geometry from the sprite texture atlas. Figure 4.9 compares two results obtained with the (oriented) sprite rendering technique.

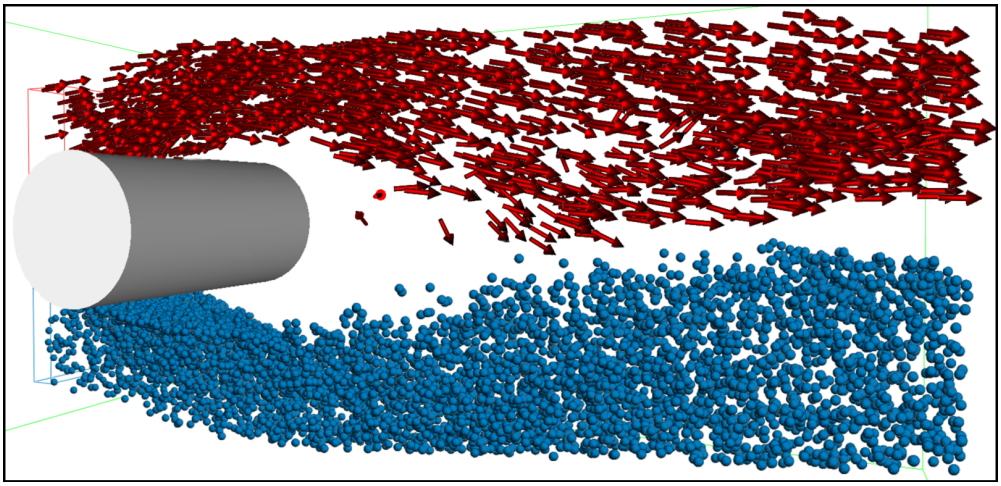


Figure 4.9: *(Oriented) Point Sprites: Two probes are positioned in the flow in front of the cylinder. Particles released from the blue probe are rendered as rotationally invariant point sprites whereas particles released from the red probe are rendered as oriented point sprites.*

4.6.2 Clip Planes

Rendering reams of moving particles often results in an obstruction of features of interest within the flow. For this reason, we have integrated functionality to specify arbitrarily oriented clip planes in the flow domain and to restrict the display of particles to respective regions. This metaphor allows to reduce the massive amount of 3D information and eases the problem of occlusions typically inherent to 3D flow visualization.

For each clip plane, the four coefficients of the general plane equation are stored in one element of a shader constant array residing in GPU memory. Furthermore, we equip the particle set with an additional attribute indicating whether a particle primitive should be displayed or not.

During the advection pass, we compute for each particle the minimum of the shortest distances to all clip planes. If this distance falls below a user defined threshold, we project the particle onto the corresponding plane and, thus, start the integration from the respective location in the next advection iteration. Furthermore we mark the particle as valid for display. All other particles move along the flow as usual (until they are captured by a clip plane) and are flagged invalid for rendering.

In the successive rendering stage, we position all particles with an invalid render flag outside the view frustum. By doing so, they are excluded from successive rasterization and, thus, do not contribute to the final image.

As can be seen in Figures 4.2 and 4.10, internal flow structures can be revealed effectively with the presented clip plane approach.

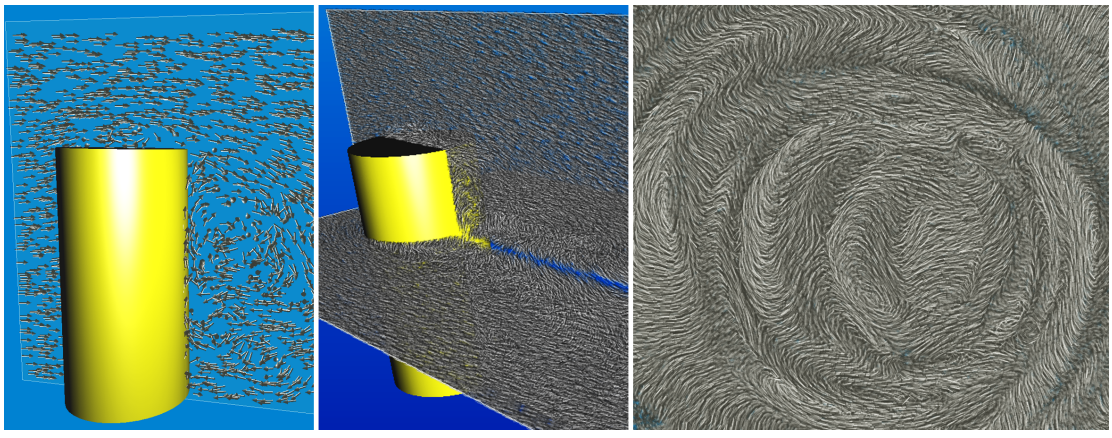


Figure 4.10: *The application of clip planes in two 3D unsteady flow fields is shown. In the right image the presented approach is able to reveal shock waves in the TeraShake data set.*

4.7 Characteristic Line Extraction

To offer additional visualization modes for unsteady flow fields we have developed GPU-based techniques for the construction of stream, path, and streak lines. Figure 4.11 shows such lines in an unsteady flow around a cylinder. For the construction of characteristic lines, particles are released from a user-defined probe and tracked over time. The construction of stream and path lines essentially uses the texture-based particle advection as described before. For the construction of streak lines, however, we perform a slightly different strategy. Throughout the following discussion we will assume that characteristic lines starting at $m \times n$ sample positions are to be computed. Each characteristic line is represented through a discrete set of len control points. To store the control points for all lines—next to the particle attribute textures—an additional texture atlas, large enough to store len blocks of $m \times n$ entries, is needed.

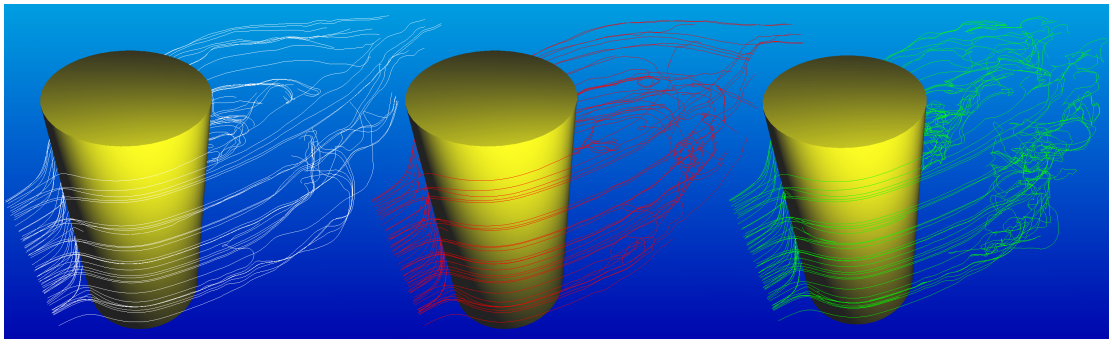


Figure 4.11: *Comparison between stream (white), path (red) and streak lines (green).*

4.7.1 Stream Lines

A stream line describes an instantaneous particle path, which is the path of a particle in an unsteady flow frozen at time t. To construct stream lines, the trajectories of all particles traveling through an instantaneous snap-shot of the flow field are computed in len advection iterations whenever the time-sequence advances (i.e., in *every* frame). Particle advection is performed as described in Section 4.5.1 with respect to numerical stream line integration (Eq. 2.6). However, after each advection step the content of the output (particle position) texture is copied into the respective sub-region of the atlas texture, determined by the current advection step and the size $m \times n$ of the particle texture. We do not transfer the resources manually in a pixel shader, but use an API-supported copy operation. If the size of the texture atlas exceeds the maximum hardware supported texture size, multiple atlases might have to be stored.

4.7.2 Path Lines

A path line describes a particle trajectory over time in an unsteady flow (Eq. 2.5). GPU construction of path lines differs from the construction of stream lines as only one advection step per frame is computed in the time-varying field. If the number of positions along the path line exceeds *len*, the texture atlas is accessed in a ring-like manner. This means that in each frame the oldest of all stored positions of a particle is overwritten by the current position. Since in this way the start vertex of the lines to be rendered is shifted, texture coordinates have to be adapted in the vertex shader according to a constant shader variable indexing the start block location in the texture atlas. As a result, line primitives of growing length are constructed and displayed. As soon as the amount of traced positions (frames) exceeds *len*, the traces start to move with the flow.

Whenever the "advancing" particle of a path line trajectory leaves the flow domain we begin the calculation of a new trajectory at the respective start position in the probe. However, if a particle dies it cannot simply be reincarnated, as this will create an incorrect line segment (from the last position before the reincarnation to the new seed position) in the successive rendering stage. Instead, invisible line segments are generated in this case as follows. The fragment shader copies the old position but marks the particle by setting its α-component to 0. Then the next advection step determines that the particle dies, but also that it has been marked during the last pass. In this case the initial seed position is read and the α-component is left at 0. In the next step, the shader recognizes that the particle has been properly reincarnated during the last pass, and sets the α-channel of the respective entry back to 1. The particular line segments can finally be masked out in the rendering stage by using α-blending.

4.7.3 Streak Lines

Streak lines do not depict the history of particles moving in an unsteady flow, but rather describe the paths traced by dye continuously injected into the flow at a fixed position. In this case, all the positional information stored in the texture atlas has to be updated every frame. Thus, instead of using two ping-pong particle advection textures of size $m \times n$—as in the construction of stream and path lines—these two buffers now have to be as large as the entire texture atlas.

In each time step, a pixel shader copies a block of $m \times n$ start locations from the setup texture into a sub-region of the texture atlas, thus, releasing a new set of particles into the flow. Then, an update is performed on the whole texture atlas to advect all $len \times m \times n$ particles in a single rendering pass. Again, the texture atlas is employed

in a ring-like manner and, during the rendering stage, modulo arithmetic on system generated values is used to address the starting location of a line in the texture atlas.

4.7.4 Performance

The computation of stream lines in unsteady flow fields comes at the expense of recalculating the whole texture atlas, i.e., the entire set of lines within the frozen time step. Path lines, on the other hand, only cause a slightly higher computational load than particle tracing, because copying per-frame results into the atlas can be realized without noticeable performance loss. For streak lines, numerical integration in the 4D field has to be performed for each position stored in the atlas in every frame. We have measured the performance of the proposed characteristic line extraction techniques on an NVIDIA Geforce 8800 GTX equipped with 768 MB local video memory. A comparative performance analysis between stream, streak, and path lines using the 4th order Runge-Kutta integrator (Eq. 2.4) is given in Table 4.2. The timings were obtained with a disabled rendering stage to minimize additional load on the GPU. Thus, only the asynchronous streaming of flow field data and the extraction of characteristic lines were performed during the measurements.

# Lines	L=100	L=500	L=1000
128	133 / 872 / 870	30 / 835 / 330	15 / 388 / 175
512	125 / 586 / 400	29 / 238 / 88	15 / 125 / 47
1024	98 / 252 / 208	27 / 114 / 45	15 / 60 / 24

Table 4.2: *Performance measurements (in fps) for stream/path/streak lines of varying length L.*

4.8 Characteristic Line Visualization

Once all particle trajectories have been computed, we employ the Direct3D instanced drawing API to render the characteristic lines. Similar to the particle rendering technique, a dummy vertex buffer containing *len* primitives is sent to the pipeline and rendered $m \times n$ times employing an instanced draw call. A vertex shader fetches corresponding control point positions from the texture atlas based on modulo div operations on the system generated values *InstanceId* (addressing the respective line primitive) and *VertexId* (addressing the current position on the line). The application takes care of setting appropriate values as uniform shader constants to correctly access the atlas (this includes the values *m*, *n*, *len*, the resolution of the texture atlas and one shader constant indexing the line start location in the texture atlas).

4.8.1 Control Points

If a small global step size is used during line integration, the impression of line primitives can be obtained even without connecting adjacent control points residing on one line explicitly. By binding the dummy vertex buffer with a point list topology in the input assembly stage and rendering it with one of the previously mentioned particle visualization techniques, intuitive representations can be obtained. By rendering individual control points as textured point sprites and scaling the sprite size according to a scalar flow quantity, further information can be communicated. For example in Figure 4.12 (left) the velocity magnitude was used to adjust the size of ball-shaped point sprites residing on stream line trajectories. In Figure 4.12 (right) streak lines were visualized by rendering their control vertices as (unshaded) oriented ellipsoids. Additive alpha blending was employed to blend overlapping primitives, thus, mimicking the appearance of dye injected into the flow domain.

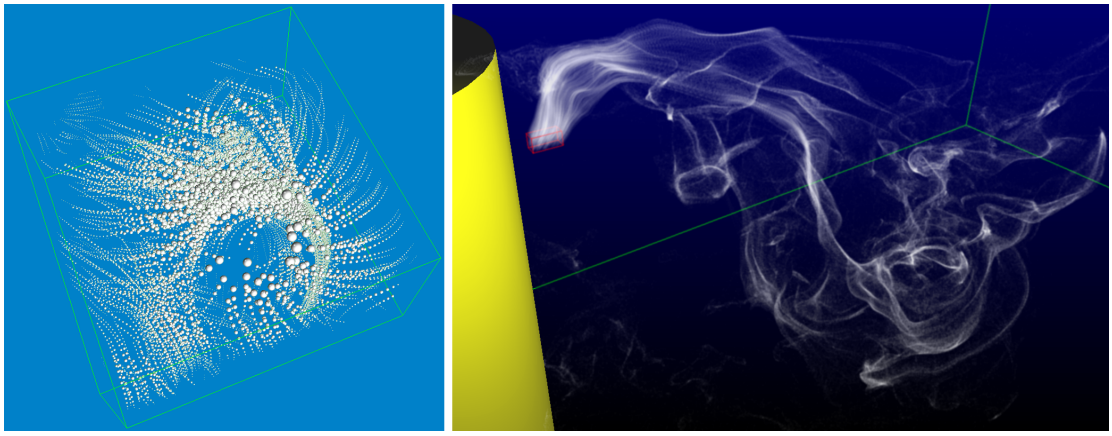


Figure 4.12: *Particle visualization techniques applied to the control points of characteristic lines. Left: Ball shaped, textured point sprites residing on stream line trajectories are scaled according to the local velocity magnitude. Right: Control points of streak lines are rendered as (unshaded) oriented ellipsoids, thereby mimicking the appearance of dye injected into the flow.*

4.8.2 Continuous Line Segments

To render the lines as a strip of linear line segments connecting adjacent trajectory positions we bind the vertex buffer with a line strip primitive topology to the rendering pipeline. After the position displacement in the vertex shader stage, the rasterizer generates fragments for each pixel covered by the line segment spanned by two consecutive control points and sends them to the pixel shader stage. If the line primitives are rendered with a low opacity value, converging flow regions can be depicted intuitively by

disabling the z-test and accumulating color through alpha blending in the output merger stage (see Figure 4.13).

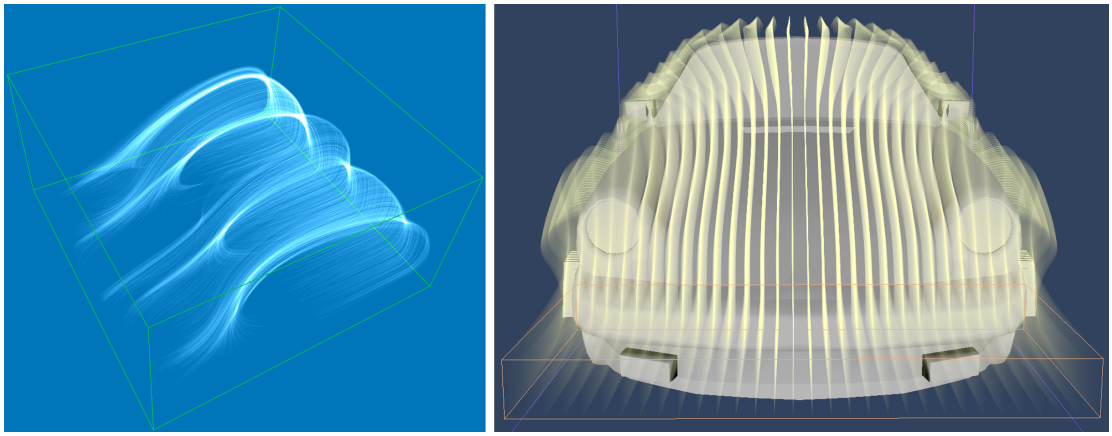


Figure 4.13: *Continuous characteristic lines. Left: Semi-transparent stream lines in a GPU-based DNS simulation of a cavity driven flow. Right: Path line trajectories are employed to visualize an interactive GPU-based fluid simulation based on the Lattice-Boltzmann method.*

4.8.3 Shaded Lines

To improve the depth perception of characteristic lines, Zöckler et al. [192] propose the application of a local illumination model during rendering. As lines have codimension 2 in $\mathbb{R}^3$, no unique normal vector is defined. Thus, they introduce a generalization of the Phong reflection model [127] by choosing a normal vector coplanar to the incident light direction and the tangent at a point on the characteristic line. The Phong lighting model breaks illumination down into three components, namely a global (constant) *ambient*, a *diffuse* reflection and a *specular* reflection term. Be $\mathbf{l}$ the incident light direction, $\mathbf{v}$ the viewing direction and $\mathbf{r}$ the reflection vector. Then according to the Phong model, the light intensity I at a point on the characteristic line is given by

$$I = I_{ambient} + I_{diffuse} + I_{specular} = k_a + k_d(\mathbf{l}\cdot\mathbf{n}) + k_s(\mathbf{v}\cdot\mathbf{r})^s \tag{4.1}$$

Here, the diffuse term—approximating a rough surface structure—obeys Lambert's diffuse reflection law (i.e., this term is equal from all view directions). The specular term—approximating a smooth surface texture—, however, is centered around the light's reflection direction and decreases with increasing angle between $\mathbf{r}$ and $\mathbf{v}$. The extent of the highlight is controlled by the shininess parameter (exponent s). The three constants $k_a, k_d, k_s \in [0,\ldots 1]$, $k_a + k_d + k_s \leq 1$ are used to weight the terms according to material-specific properties. If we choose from all possible normal and reflection vec-

tors those that are coplanar to **l** and the tangent **t**, then according to [192], the diffuse and specular terms are given as

$$I_{diffuse} = k_d \sqrt{1-(\mathbf{l}\cdot\mathbf{t})^2}, \quad I_{specular} = k_s \sqrt{1-(\mathbf{l}\cdot\mathbf{t})^2}\sqrt{1-(\mathbf{v}\cdot\mathbf{t})^2}$$

By extending the line segment render technique with a geometry shader, the tangent is implicitly given by the control vertices of the line segment stream. The local illumination model is then evaluated on interpolated vertex attributes through Phong shading in the pixel shader stage. If a directional light is used, then most parts of the lighting model can also be evaluated by the geometry shader and only those components that are dependent on the position of a point along the line segment (i.e., the view direction and parts of the specular intensity term) have to be calculated on a per-fragment basis in the pixel shader stage. By doing so, the arithmetic load imposed onto the shading units of the GPU can be minimized. An exemplary result obtained with this technique is shown in Figure 4.14.

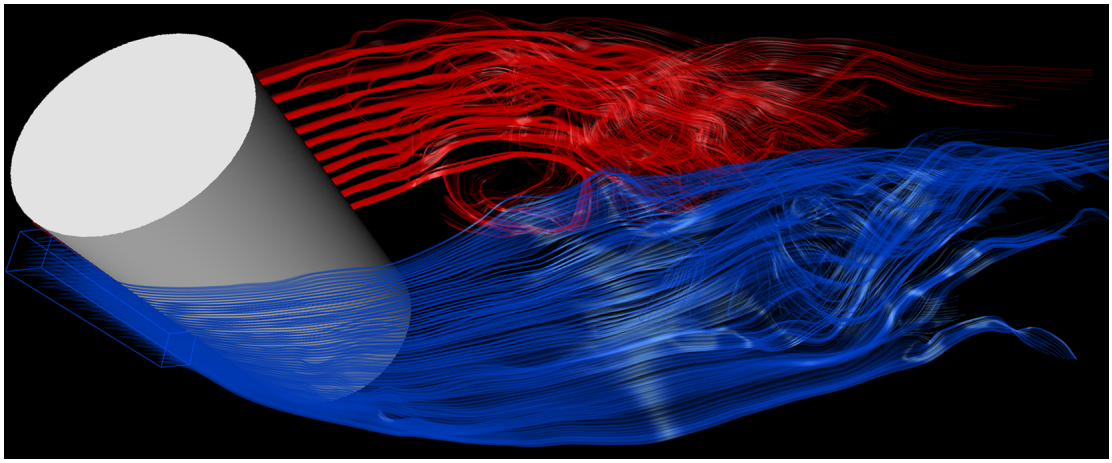


Figure 4.14: *Shaded lines: Two probes with varying sample distributions are placed in front of the cylinder. Illuminating the characteristic lines greatly enhances the depth perception.*

4.8.4 Ribbons

By extruding the line segments into two- or three-dimensional geometry, additional flow quantities can be incorporated into the visual representation of characteristic lines. 2D representations are especially suited to depict the rotation about the flow axis by twisting a ribbon-shaped primitive [169]. To construct ribbon-shaped characteristic lines we employ an additional one-component texture atlas storing incremental rotation angles for a random extrusion direction **b** defined per probe start position (and initialized perpendicular to the local velocity direction). During line integration the in-

cremental rotation angle for this vector at subsequent control points—according to the rotation about the flow direction—is calculated as

$$\theta_{i+1} = \theta_i \cdot \frac{1}{2}(\omega \cdot \hat{\mathbf{v}}) \,,$$

where ω is the curl of the vector field (see Eq. 2.19) and $\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$ is the normalized flow velocity direction. To construct the ribbon geometry in the rendering stage we employ a geometry shader. By sending an instanced dummy vertex buffer containing *len* control vertices with a line list topology into the pipeline, we proceed as follows. First, a vertex shader fetches the corresponding position and rotation values from the texture atlas and calculates two new vertex coordinates by displacing the control point residing on the line along **b** (rotated about the corresponding rotation angle θ) and its inverse, respectively. These positions are then passed to the geometry shader stage as attributes on a per-vertex basis. The geometry shader, receiving a line segment as input, then calculates a normal for each control vertex based on the plane spanned by the vectors to adjacent vertices of the quadrilateral ribbon patch. Each ribbon patch connecting two successive points on the line is then sent to the rasterizer stage in the form of two triangles. As most calculation are performed in the vertex shader, vertex caching is exploited to avoid redundant operations on a per-control point basis. Local illumination according to the classic Phong illumination model (Eq. 4.1) is then performed on a per-fragment basis in the pixel shader stage.

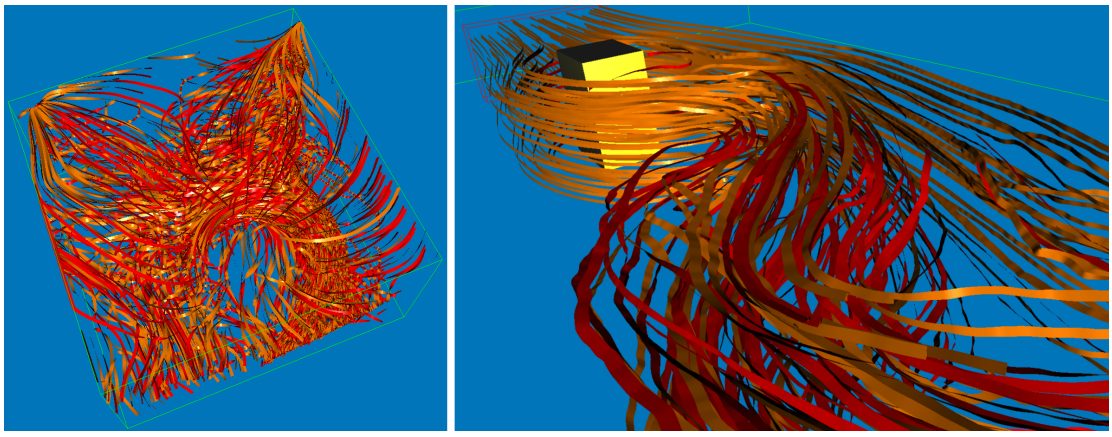


Figure 4.15: *Ribbon shaped characteristic lines are shown. Left: Path lines in a 3D unsteady flow field. Right: Stream lines in an instantaneous flow.*

On graphics hardware without geometry shader support, ribbons can be realized by storing the triangle strip vertex buffer explicitly. For a detailed implementation, we refer the reader to [90]. In Figure 4.15, stream ribbons extracted from 3D flow are shown.

4.8.5 Tubes

Extruding line segments into 3D geometry is a common technique employed to emphasize the speed of flow along stream and path lines. Ueng et al. [169] propose using generalized cylinders to visualize stream lines as tube-shaped objects and encoding the velocity magnitude into the visual representation by adapting the stream tube diameter accordingly.

Various rendering approaches for generalized cylinders have been introduced in the literature. Fuhrmann and Gröller [46] use a simple tessellation scheme with a fixed amount of subdivisions along the cylinder. Such an approach can efficiently be mapped onto Shader Model 4.0 capable hardware by application of a geometry shader to perform the cylinder tessellation. Alternative approaches render proxy geometry enclosing the characteristic line and employ ray-casting to determine a per-pixel precise intersection with the generalized cylinder [162]. While these approaches benefit from a reduced load on the GPU triangle setup stage, they introduce an increased load onto the pixel shader stage and are, thus, less preferable in an interactive environment due to the following reason: Characteristic lines are generally rendered order-independent to avoid time-consuming sort operations. Thus, the superfluous pixel shader load introduced due to ray-casting of proxy geometry—that will be occluded by successively rendered primitives—generally outweighs the geometry load of tesselated generalized cylinders. For that reason we have integrated a tessellation approach as described in the following.

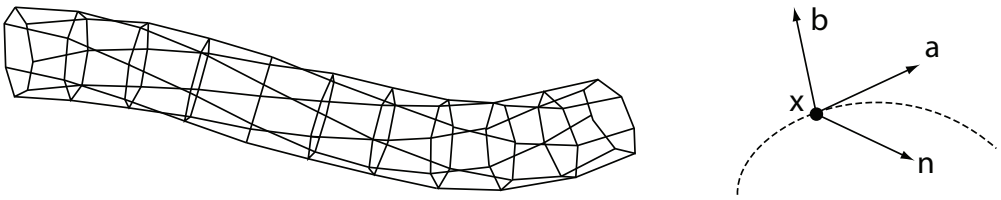


Figure 4.16: *Tube Construction: Left: A twisting reference frame can lead to a highly distorted cylinder tessellation. Right: During particle integration, a frenet frame is propagated along the trajectory to construct an undistorted polygonal representation of the generalized cylinder.*

To construct an undistorted polygonal representation of a generalized cylinder, the cross-section of each control vertex on the line must be properly aligned with its neighbors so that the structure does not twist. Such an alignment is usually provided in the form of a frenet reference frame, consisting of a tangent vector $\mathbf{a}_i$, a principal normal $\mathbf{n}_i$ and a binormal $\mathbf{b}_i$ specified per control point $\mathbf{x}_i$. We ensure the correct alignment by propagating the binormal along the line and adjust it iteratively during particle advection according to the change in curvature as proposed by Sloan [10]. The tangent for a start point $\mathbf{x}_0$ along every trajectory is set to the normalized velocity direction. We de-

termine the initial binormal through the cross-product of $\mathbf{a}_0$ and a vector perpendicular to the plane spanned by the two largest components of the tangent vector. The tangent at successive control points $\mathbf{x}_{i+1}$ along the line get assigned the averaged velocity at the previous and current position on the line. The frame is then given as [10]

$$\mathbf{n}_{i+1} = \mathbf{b}_i \times \mathbf{a}_{i+1},$$
$$\mathbf{b}_{i+1} = \mathbf{a}_{i+1} \times \mathbf{n}_{i+1}.$$

For subsequent rendering, we store the tangent, binormal and one scalar flow quantity used to determine the local diameter of the generalized cylinder (e.g., the local velocity magnitude) in additional texture atlas resources.

During rendering, we generate the tessellated cylinder with the help of the geometry shader stage. To exploit vertex caching, a vertex shader first fetches all necessary attributes (i.e., position coordinates, reference vectors and velocity magnitude) from the texture atlas and reconstructs the local frenet frame. Furthermore, to polygonize a generalized cylinder segment into s subdivisions, it extrudes the control vertex s times along the corresponding cross section. These position values are calculated by accessing a constant buffer (holding s position values radially aligned around the origin of an extrusion axis), rotating these positions with respect to the reference frame and scaling them accordingly. The set of control vertices is then sent to the geometry shader stage as attributes on a per-vertex basis. The geometry shader stage then performs the tessellation of the cylinder on a per line segment basis and issues a triangle strip consisting of $2s+2$ vertices to the rasterizer stage. Here, normals are computed based on the vector spanned by the central position on the trajectory and the extruded position s, respectively. Some exemplary results are shown in Figure 4.17.

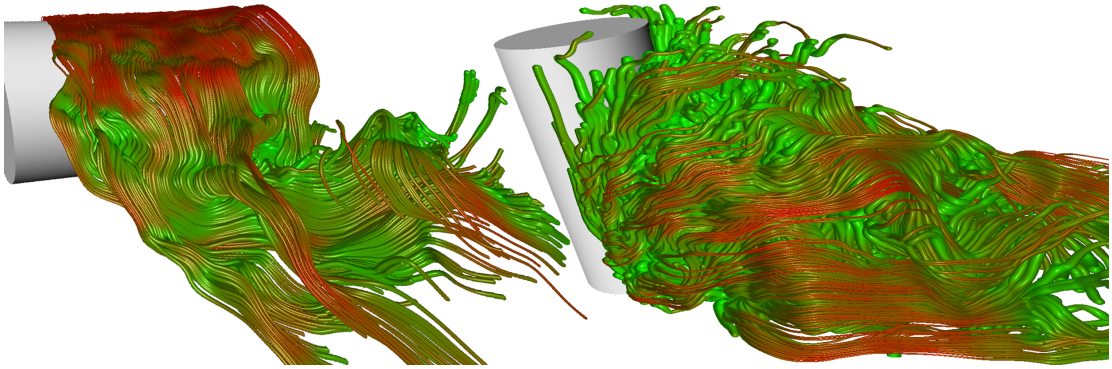


Figure 4.17: *Stream tubes in an unsteady flow around a cylinder are shown. According to the local velocity, the appearance of a generalized cylinder smoothly changes along its trajectory from thick/green (slow velocity) to thin/red (high velocity).*

4.9 Focus+Context Boundary Visualization

Our system supports the visualization of polygonal models to better reveal the spatial relationships between flow structures and boundaries of the flow domain. The study of flow behavior close to solid boundary regions plays an important role in various scientific areas. E.g., in aerodynamics one wants to minimize the drag of and turbulence behind obstacles placed within the fluid flow. In medicine, the behavior of fluids in vessels or the transport characteristics of neuro transmitters within the brain are of special interest. Seismology studies the propagation of shock waves with respect to transport media at varying density. Hence, in scenarios of practical relevance, static boundary regions often partially obstruct or totally enclose the flow. For that reason, advanced rendering techniques have to be employed to compose a single visual representation from images obtained through flow visualization techniques and renderings of the boundary geometry (Figure 4.18 depicts such a scenario).

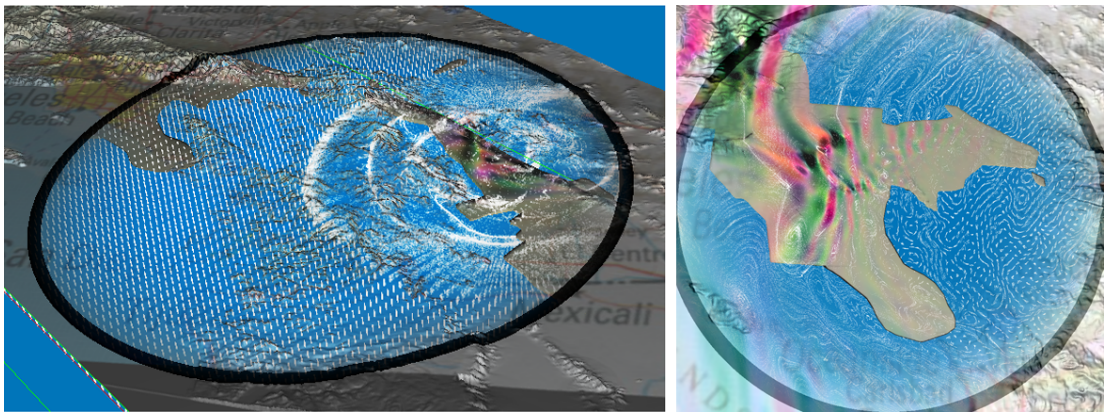


Figure 4.18: *Focus+context boundary visualization based on the ClearView paradigm. A particle-based flow visualization in the radially symmetric focus region is shown. Additionally, parts of the boundary mesh are rendered fully opaque to reveal the relation between the shock wave propagation behavior and high-density regions of the transport medium. Within the focus region, important features of the terrain are emphasized to reveal the spatial correspondence.*

Focus+context techniques address this issue by combining a view on a region of interest (the focus) with an abstract view on its surrounding (the context). Especially the *ClearView* [91] metaphor has proven suitable to be integrated in our interactive flow exploration environment. With respect to this application, the ClearView technique consists of the following building blocks to deduce a focus+context visualization.

The *focus* region contains information obtained by a flow visualization technique. Optionally, pre-selected parts of the polygonal boundary model (which should always stay fully opaque) are included in the focus information.

One or multiple *context* layers contain semi-transparent information extracted from the boundary mesh. We adapt the transparency of the boundary with respect to two criteria. Firstly, the user can specify a point of interest $\mathbf{c}$ and a radius r to determine a spherical focus region. With decreasing distance to the center of interest we linearly fade out points $\mathbf{p}$ on the boundary region. Secondly, important features on the boundary mesh (within the focus sphere) are emphasized on the basis of a curvature measure to convey the global shape of the object. Be *curv* a function that evaluates the local curvature at a given point on the boundary and *saturate* an operator that clamps its parameter to the range $[0\ldots1]$, then the transparency is given as

$$trans = 1 - saturate\left(max\left(\frac{\|\mathbf{c}-\mathbf{p}\|}{r}, curv(\mathbf{p})\right)\right).$$

The final image is composed in a multi-pass rendering approach as follows:

First, we render the geometry-based flow visualization into the back buffer as well as parts of the boundary that have been classified as boundary focus regions and, thus, should always stay fully opaque.

Before a context layer is added on top of the final image, we have to extract all surface attributes needed to evaluate the curvature at a given point on the boundary surface. For this, we employ deferred rendering, i.e., we render the mesh from the current view into a three-component floating point texture—at a resolution equal to the back buffer—and store the normals of the context boundary surface. In a second render pass, we draw the boundary mesh into the back buffer (with z-test and alpha-compositing enabled) and determine the transparency on a per-fragment basis in the pixel shader stage. The curvature importance feature is thereby computed with respect to an image-based umbrella operator [83], i.e., for each fragment a pixel shader fetches attribute values from the corresponding as well as four adjacent pixels in the deferred render target. The summed distance from the center normal to the adjacent normals is then used as a curvature criterion (see Figure 4.19). This results in low values in planar regions on the boundary and large values otherwise. The spherical distance criterion of the transparency equation is evaluated on the basis of an interpolated vertex attribute, i.e., the world-space position.

To generate multiple context layers, we repeat the two passes generating the context information several times and employ depth peeling [40] for a slice-by-slice extraction of the boundary mesh in back-to-front order.

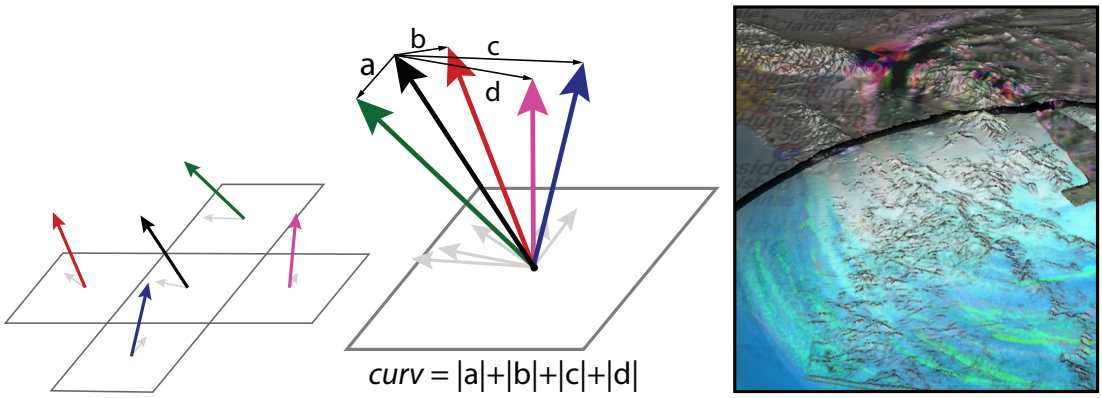


Figure 4.19: *The transparency of boundary meshes in the context region is determined on the basis of a curvature importance measure. Left: The normal at a surface pixel and its four neighbors is illustrated. Middle: The sum of the distances from the center normal to its adjacent normals is used to estimate the local curvature. Right: The presented measure reveals the global shape of the overlying terrain, enabling the user to intuitively establish a connection between the propagation behavior of earthquake shock waves and the surrounding transport medium.*

4.10 Summary

In this chapter, we have presented interactive techniques for the visualization of large unsteady 3D flows. We introduced a new multi-core streaming approach for time-resolved flow fields that allows the exploration of high-resolution data sets interactively. We discussed how particle tracing and the extraction of characteristic lines can be performed in real-time, presented a multitude of rendering modalities for such geometric flow representations and discussed efficient implementation strategies for recent GPUs. The presented techniques allow tracing millions of particles and extracting thousands of characteristic lines interactively and, thus, enable the virtual exploration of high resolution fields in a way similar to real-world experiments.

At the time the underlying research paper was published, the presented techniques allowed for the first time an interactive visualization of unsteady 3D flows on consumer class PCs. The effectiveness of these techniques for the purpose of visual data analysis has been acknowledged by researchers from various field and even been thoroughly validated in a benchmark of the visualization community [5].

The presented interactive flow exploration environment allows scientists to obtain rapid visual feedback even while the data to be visualized is generated in parallel. This allows not only to intuitively grasp the flow phenomena under investigation but also to immediately use obtained findings to (computationally) steer the data generating process.

Chapter 5

Importance-Driven Particle Techniques for Flow Visualization

Particle tracing has been established as a powerful visualization technique to show the dynamics of 3D (unsteady) flows. Particle tracing in 3D, however, can quickly overextend the viewer due to the massive amount of visual information that is typically produced by this technique. In this chapter, we address this problem by presenting various strategies which reduce the amount of information while preserving important structures in the flow.

As an importance measure for stationary 3D flow, we introduce a simple, yet effective clustering approach for vector fields. For the visualization of unsteady flow fields, we use scalar flow quantities at different scales in combination with user-defined regions of interest. These measures are used to control the shape, the appearance, and the density of particles in such a way that the user can focus on the dynamics in important regions while at the same time context information is preserved. Furthermore, we introduce a new focus for particle tracing, so called *anchor lines*. Anchor lines are used to analyze local flow features by visualizing how much particles separate over time and how long it takes until they have separated to a fixed distance. It is of particular interest if the finite-time Lyapunov exponent (FTLE) is used to guide the placement of anchor lines. The effectiveness of our approaches for the visualization of 3D flow fields is validated using synthetic fields as well as real simulation data.

5.1 Introduction and Related Work

In principle, geometric flow visualization techniques suffer from similar problems as texture-based methods in that rendered primitives overlap and occlude each other. If large sets of primitives are seeded, the same perceptual problems inherent to dense global techniques arise and important information might be obscured. These limitations can be partially overcome by real-time techniques, i.e., by enabling the user to interactively control the number of seeded particles and their starting positions. Other approaches restrict the visualization to particles moving on or close to specific surfaces in the flow [173, 113]. These techniques effectively restrict the visualization to a focus region, or in this particular case to a focus surface, but by doing so important context information as well as relevant structures outside this region might be lost.

Most inspirational for our work was previous work on focus+context techniques for scientific visualization as well as feature-based flow visualization methods. For flow visualization, Fuhrmann and Gröller [46] proposed the combination of a user-controlled focus region and a uniform stream line placement strategy. Within the focus region the flow field is visualized at the highest resolution level, and contextual information is preserved by visualizing a sparse set of primitives outside this region. Löffelmann and Gröller [107] presented a feature-based focus for 3D dynamic systems. By visualizing short stream lines, so-called streamlets, only close to a base trajectory in a 3D vector field, occlusion problems could be avoided, thus, providing a detailed view of particular regions in the field. For 2D flow visualization, Kirby et al. [80] defined a focus by combining visual elements of different size, shape and texture into a multi-layer representation. Doleisch and Hauser [36] presented non-discrete 3D regions of interest including techniques to blend between differently shaped primitives. Mattausch et al. [112] introduced more flexible and interactive focusing strategies as well as multiple options to adaptively modulate the density and appearance of stream lines in 3D flow fields.

5.2 Contribution

In this chapter, we propose a number of improvements for particle-based 3D flow visualization. Common to all techniques we present is a significant reduction of the amount of visual information presented to the user. Consequently, these techniques are less prone to perceptual artifacts like occlusions, and they can avoid visual clutter introduced by frequent positional changes of large amounts of particles. Relevant structures

in the flow are emphasized by integrating user-controlled and feature-based importance measures. The suggested techniques extend previous approaches for particle-based visualization of 3D flows as follows (see Figure 5.1 for a graphical illustration):

- We present techniques to automatically adapt the shape, the appearance, and the density of particle primitives with respect to user-defined and feature-based regions of interest. We also provide means for smooth blends between differently shaped primitives. Thus, the proposed techniques can effectively be used in combination with continuous focus+context and importance measures. Figure 5.1 (c) demonstrates the possibilities these techniques offer.

- We propose a clustering approach to determine regions of coherent motion in an instantaneous snapshot of the flow, and we use a sparse set of static cluster arrows to emphasize these regions. Figure 5.1 (a) shows such arrows in combination with an importance-driven rendering of primitives in the focus region. As can be seen, occlusion problems in the visualization of contextual information can be avoided, thus enabling a flexible integration of detail information into selected focus regions.

- In addition to scalar flow quantities derived locally from the velocity vector field, we consider the finite-time Lyapunov exponent (FTLE) as an importance measure. In particular, we employ this measure for the selection of characteristic trajectories in the flow. We call these trajectories anchor lines, and we seed particles close to the starting points of these lines. By only visualizing those particles that leave the anchor, the amount of visual information can be reduced significantly (see Figure 5.1 (b)). Furthermore, we use this approach to allow quantitative statements about the particle movement over time and space.

- All visualization techniques have been integrated into the GPU-based particle engine. This enables the user to interactively select visualization parameters and rendering modes, thus allowing an effective visual analysis of 3D flow structures.

The remainder of this Chapter is organized as follows. In the next Section we discuss the focus+context metaphor underlying our approach, and we present implementation specific details. We then introduce our clustering approach and detail how it can be integrated into the importance-driven visualization approach. Next, we describe the meaning of anchor lines as well as the used particle seeding and rendering strategy. An analysis of the performance of the proposed techniques is given in Section 5.6. We

conclude this chapter with an outline of future research in the field of particle-based flow visualization.

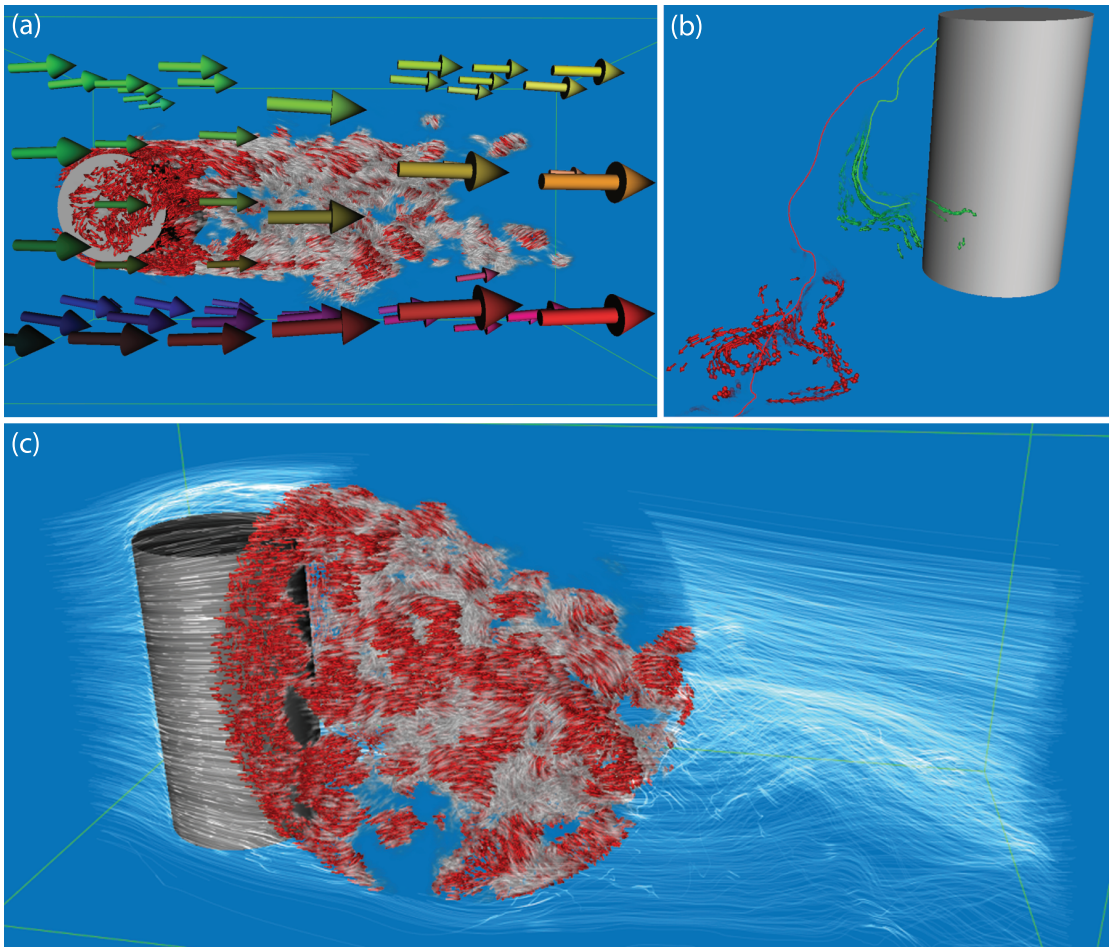


Figure 5.1: *Importance-driven particle techniques are used to visualize 3D flow. (a) Cluster arrows show regions of coherent motion. (b) Particles seeded in the vicinity of anchor lines show the extent and speed at which particles separate over time. (c) Focus+context visualization using an importance measure based on helicity and a user-defined region of interest.*

5.3 Importance-based Particle Visualization

One important goal in particle-based flow visualization is to reduce the amount of visual information presented to the viewer. This is due to the following observations: Firstly, many interesting flow structures are typically occluded by the primitives rendered in non-interesting regions of the flow. Secondly, a large amount of moving particles, often performing rapid directional changes, produces visual clutter that quickly overloads the human perceptual system.

While it is easy in general to simply restrict the visualization of particles to user-defined focus regions, this approach typically results in the loss of contextual information necessary to understand the global relationships between flow structures. The focus+context paradigm seeks to combine both aspects into a single visual event by presenting a detailed region in combination with a surrounding context. The visual information used to represent the context region must not occlude details in the focus regions, but at the same time it should indicate characteristic structures in the data. For an overview of focus+context techniques in scientific visualization we refer the reader to the tutorial by Viola et al. [179].

To use focus+context techniques in particle-based flow visualization, two different strategies have to be pursued: Firstly, the spatial density of visualized particles should be adapted according to the importance classification. Secondly, the appearance of rendered particle primitives should reflect the importance of the region they are traveling through. In the following, we will first describe how to flexibly adjust the density, the shape and the appearance of rendered particle primitives based on their importance.

5.3.1 Scale-space Particles

In the following, we assume that an importance mapping can be evaluated at every point in space $\mathbf{x}$ and time t within the flow domain to yield the local importance of the vector field at this point, i.e. a function

$$Imp(\mathbf{x},\mathbf{t}) = ImpPos(\mathbf{x},\mathbf{x}_f) \oplus ImpVol(\mathbf{x},t), \quad Imp(\mathbf{x},\mathbf{t}) \mapsto [0,1] \tag{5.1}$$

Such a function can either be a (radially symmetric) attenuation function $ImpPos(\mathbf{x},\mathbf{x}_f)$ defining the decrease in importance with respect to a user-defined focus point $\mathbf{x}_f$, or an importance volume $ImpVol(\mathbf{x},t)$—storing pre-computed importance values based on physical flow properties—that can be sampled at the respective location. The larger the value of *Imp* is, the higher is the importance given to this point. Some possible importance measures directly derived from the flow velocity vector field, and their evaluation at varying levels in scale space, will be discussed in section 5.3.2. Furthermore, both measures can be combined (operator $\oplus$ in Eq. 5.1) to achieve more flexibility in the focus+context configuration.

Once an importance mapping is given, the reduction of the amount of information displayed can be achieved by adaptively reducing the number of rendered particle primitives based on the local importance. Therefore, we employ a similar approach as used in [191] for the selection of hatches in illustrative volume rendering. Every particle

seeded into the flow is assigned a random value in the range of $[0,1]$, and a particle contributes only to the rendered image if the importance at its current position is higher than this random value. By this approach, more and more particles are removed with decreasing importance. Figure 5.2 (b) shows this effect for a user-controlled focus region positioned right behind the cylinder in the flow and in Figure 5.2 (c) the density was adjusted according to an importance volume. As can be perceived, this approach neither emphasizes characteristic structures in the context region nor does it allow a clear distinction between what is in focus and what is not.

To overcome this problem we increase the particle size by a factor inversely proportional to importance. In Figure 5.2 (c), where the vorticity magnitude was used as an importance measure, the resulting visual effect is shown. Although we now obtain a better understanding of the context information, focus and context can still not be clearly distinguished because of the same shape and appearance of the particles being rendered. We thus transform the primitives continuously from a particular shape used to depict the focus region into a shape that indicates the context. In the current example we transform an arrow glyph into an ellipsoid as shown in Figure 5.2 (d). Shape morphing allows us to quickly obtain an image of both the focus and the context information, but the visualization still suffers from occlusions due to a few large context particles overlaying the focus region. This problem is finally alleviated by using transparency to fade out particles in the context regions. As can be seen in Figure 5.2 (e), we do not remove particles entirely, but we make them highly transparent. Furthermore, we change the particle color from a light shade of grey in the context to saturated red in the focus.

Overall, the following transfer functions are used to adjust the visual attributes of the ith particle:

$$\begin{aligned} show_i &= (rand_i > Imp(\mathbf{x}_i,t)) && \leftarrow \text{visibility} \\ size_i &= s + (1 - Imp(\mathbf{x}_i,t)) \cdot C_s && \leftarrow \text{size} \\ opac_i &= Imp(\mathbf{x}_i,t) \cdot C_o && \leftarrow \text{opacity} \\ color_i &= LUT(opac_i) && \leftarrow \text{color} \end{aligned}$$

Here, $show_i$ and $size_i$ correspond to the visibility and size of the ith particle. Parameter s specifies the base size for all particles. $rand_i$ stores a random value in the range [0,1]. $opac_i$ determines the particle opacity. User-defined constants C_s and C_o specify how fast particles are fading out according to decreasing importance. The color of every particle can be modulated by means of a user-defined color transfer function LUT.

By means of the proposed transfer functions the amount of information that is displayed can effectively be reduced. In addition, with increasing size and transparency of the particles being shown, their spatial movements appear increasingly smooth. Thus, visual clutter as it is typically observed when rendering small and opaque moving primitives can mostly be avoided.

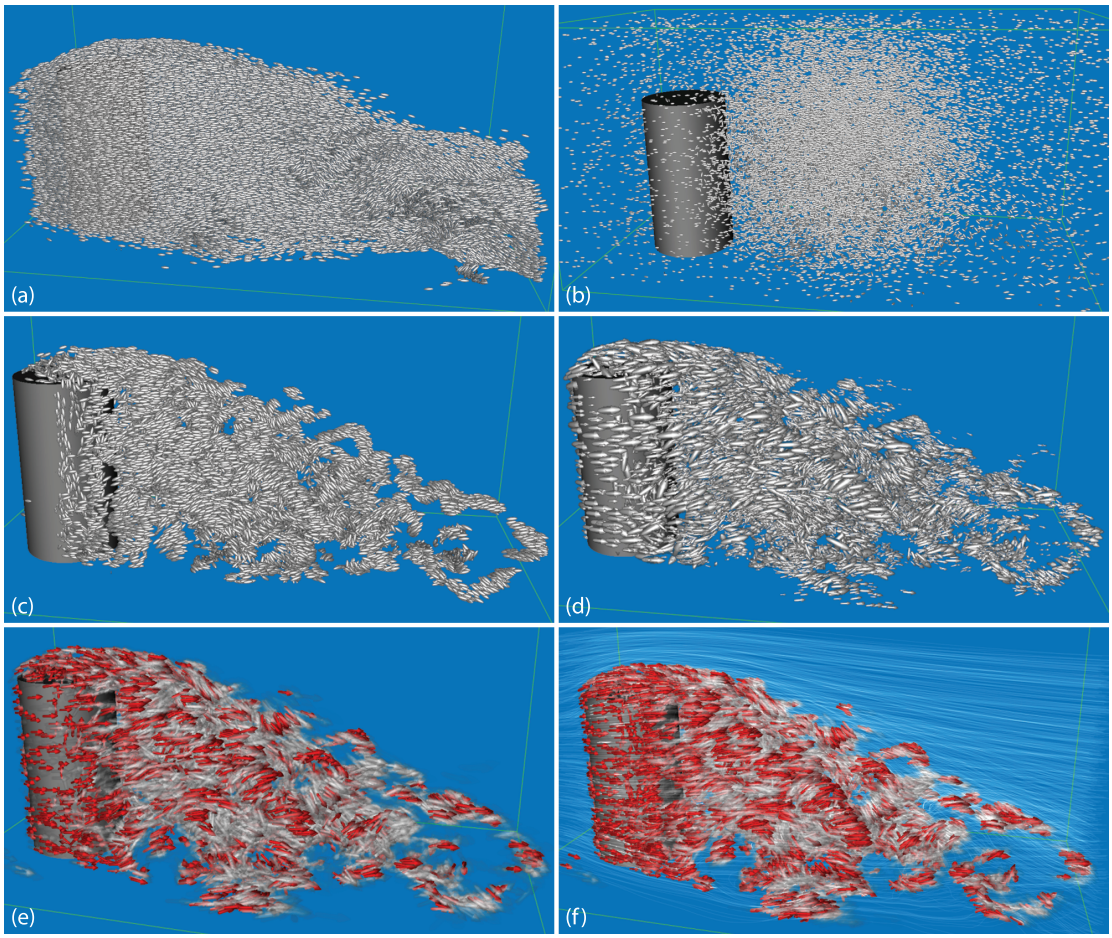


Figure 5.2: *Different approaches for 3D flow visualization using particles are shown. (a) Unsteady flow around a cylinder, visualized by a large amount of particles. (b) A region of interest has been selected, and with increasing distance to the center of this region the particle density is decreased. (c) Importance-driven density adjustment (the vorticity magnitude was used as importance measure in this example). Particles out of focus regions are removed. (d) The size and shape of particles is adjusted according to the importance of the region they are traveling through. In this example, the shape is morphed from a small arrow (high importance) to a large ellipsoid (low importance). (e) In addition to the shape transformation the transparency and color of the particles are transformed. (f) Transparent stream lines were integrated to sketch the flow structure in less important regions.*

5.3.2 Feature-based Importance Measures

To enable the visualization of characteristic structures in the flow we have integrated a number of different importance measures based on scalar physical properties of the flow. In addition to the velocity and vorticity magnitude, we employed the helicity, the λ_2-criterion as well as the maximum finite-time Lyapunov exponent (FTLE) to test the suitability of our importance-driven particle visualization technique. The meaning of each quantity and mathematical definitions were presented in Section 2.4.

These quantities are pre-computed and stored in a separate scalar volume for each time step of the unsteady flow field. During an interactive flow exploration session, the data is streamed in conjunction with the velocity vector fields onto the GPU. Next to the scalar quantities given at the spatial sample resolution of the flow velocity field, we encode additional information hierarchically in a pyramidal data structure (i.e. a mip-mapped 3D texture resource). We use a *min-max* and an *average* pyramid of volumes where the first level is the original scalar field and each successive volume is reduced about a factor of two in each dimension. Thus, only a small memory overhead is introduced. Each sample in the nth level of the pyramid stores the minimum/maximum or average importance of its eight children in the $(n-1)$st level of the pyramid. Thus, the pyramid maintains the minimum/maximum or average importance in ever increasing regions of the domain. The kind of pyramid and the level that should be considered as importance measure can be specified by the user.

By using a feature hierarchy and trilinear interpolation to reconstruct values form this hierarchy, three different effects can be achieved: First, spurious features can be suppressed by letting the importance be sampled from coarser levels in the pyramid. This is especially useful in the context region to avoid frequent changes of the particle appearance. Second, there is a smooth transition between regions of different importance. Third, continuous regions of interest are supported by smoothly interpolating between different levels in the hierarchy. To show these effects we have conducted experiments with different importance measures encoded in a multi-resolution hierarchy. Figure 5.3 shows some results indicating the suitability of feature-based importance measures in combination with a user-defined (scale-space) focus region. In particular, it can be seen that even in context regions important features are still emphasized, effectively guiding the visual exploration process.

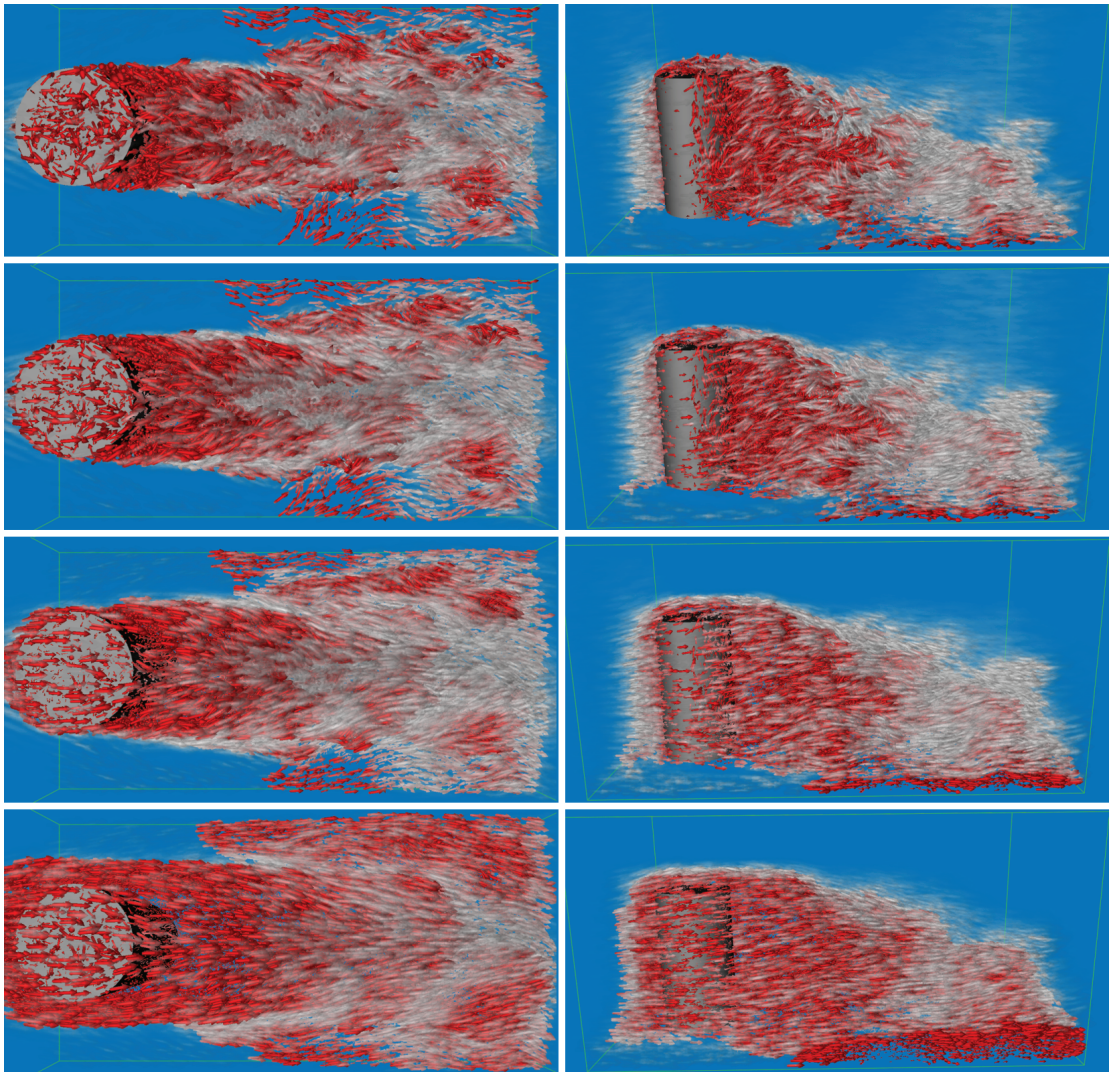


Figure 5.3: *The velocity magnitude at different scales is used as an importance measures (increasing scale space levels from top to bottom). The focus was set to low velocity regions (small, opaque and red primitives). With increasing velocity the particle density decreases and primitives smoothly transition into the context region (large, transparent and grey).*

5.3.3 Cluster Arrows

To further assist the user in the visual analysis of stationary 3D flows we propose *cluster arrows* as a sparse and static visualization metaphor. Cluster arrows are geometric primitives that represent regions of constant motion in the flow. The positions at which these primitives are placed are computed in a preprocess using a region growing approach. To find a cluster, i.e. a region in which the velocity directions do not differ by more than a given angle, we randomly select a grid point that has not yet been processed, and we inspect the velocities of all of its 26 neighbors in the grid. If none of the

velocities in the subtended region diverges more than a given angle from the average of all velocities in that region, we continue to grow the cluster until no further expansion is possible. The average velocity of all grid points in a cluster is stored as a representative for the entire region. This process is continued until the entire domain is partitioned into clusters.

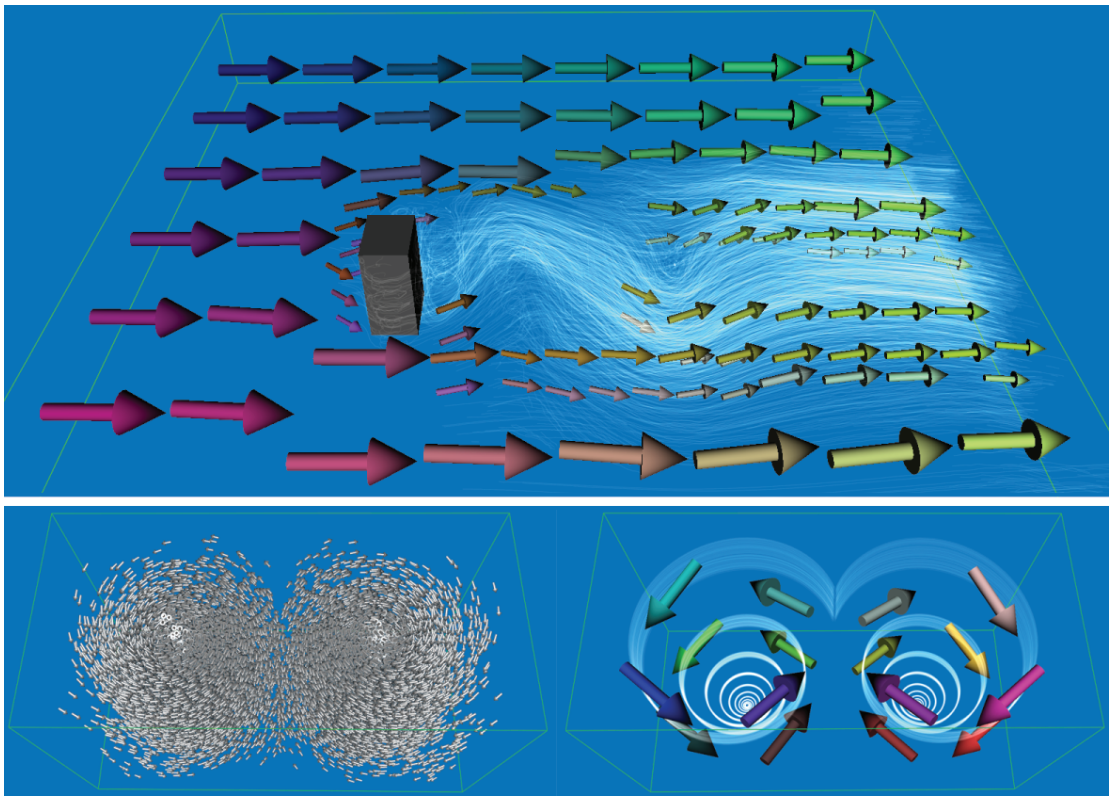


Figure 5.4: *Top: Cluster arrows and transparent lines are used to indicate coherent and less coherent motion in the flow, respectively. The size of the arrows corresponds to the size of the cluster they represent. In the bottom image (right), the same visualization technique is used. It is compared to a visualization of the same double-vortex flow using particle tracing (left).*

For every cluster, the average velocity, the cluster center position and its size are stored in a single element of a vertex buffer. During rendering, this buffer is then used to draw an oriented geometric primitive for each cluster. Our system also allows the user to select a minimum and maximum size of the clusters to be visualized. This makes it possible to hide large arrows that would otherwise occlude relevant information, as well as small clusters that would clutter into focus regions in which dynamic particles are shown. The cluster information is also used as an additional importance metric for the rendered particle primitives. Therefore, for every sample point in the grid we store the size of the corresponding cluster in an importance volume, and we fade out

primitives passing through regions of coherent motion with increasing cluster size as demonstrated in Figure 5.4.

5.4 Anchor Lines

As can be seen in the images presented so far, importance-driven visualization techniques for 3D flows using scale-space particles can effectively be employed to focus on particular regions and features while at the same time maintaining context information. On the other hand, these techniques are problematic, because in the focus region and in regions of high importance the amount of visual information is still high. To overcome this problem we propose *anchor lines*, a new focus for particle tracing that enables the user to emphasize characteristic information about particle divergence and convergence.

The idea behind anchor lines stems from the observation that one is often not interested in a detailed visualization of flow regions in which the trajectories of particles do not diverge. Instead, such regions should only be outlined by a few representative primitives. It is of interest, however, to emphasize regions in which trajectories diverge, for instance at saddles, sources or separatrices.

A scalar quantity that can be used to give evidence for the rate of divergence or convergence of neighboring trajectories in a flow is the finite-time Lyapunov exponent (FTLE). As already discussed in detail in Section 2.5.3, the FTLE is a measure for the amount of stretching of a fluid element over a fixed time. It allows to locate transport barriers and it has been studied for the analysis of transport and mixing characteristics in multi-dimensional flows.

In the following, we will introduce anchor lines as a means to locally analyze the FTLE measure. In particular, anchor lines can be used to interactively visualize how much particles separate over time and how long it takes until they have separated to a fixed distance. We extend the idea proposed in [107], where short stream lines are placed in the vicinity of characteristic trajectories to show the local flow behavior along these trajectories. To do so, we first define a set of path lines in the vector field—the anchor line center trajectories. The user can select these lines by placing their starting points in the domain. Then, additional particles are seeded in close vicinity of these starting points, with the amount of scatter around these points being selected by the user. The particles' transparency is set according to their deviation from the corresponding anchor line, i.e. particles close to the line are faded out while they are rendered more and more opaque once they start to diverge.

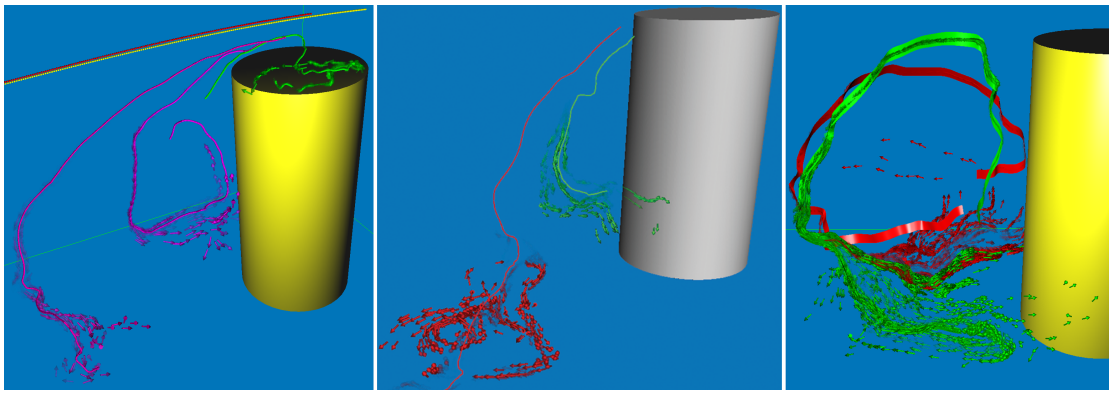


Figure 5.5: *Anchor lines (path lines) and particles seeded close to their starting points are shown. Left: While particles exactly follow some of the lines (red, yellow), along other lines the particles diverge from their anchor lines at different speed (green, purple). To improve the visual perception of the correspondence between anchor lines and particles, every anchor line gets assigned a unique color that is inherited by the particles seeded close to it. Particle transparency is inversely proportional to the separation distance from the anchor line. Right: Anchor line center trajectories are rendered as ribbon shaped geometry as described in Section 4.8.4.*

Technically speaking, anchor lines are always traced in parallel, and in every integration step the Euclidean distance between a particle and the corresponding point on the (central) anchor line trajectory is used as a measure for the deviation. Since a particle trajectory and an anchor line can deviate from each other and again approach each other, it makes sense to consider the maximum deviation of a particle along its path. This means that once the particle deviates more than a specified threshold from the corresponding anchor line it is rendered opaque along the remaining path.

Since high transparency is given to particles that remain close to the anchor line, particles are automatically faded out in regions where there is a high similarity between neighboring vector field values. In such regions only the respective anchor line center trajectory is shown. Particles in highly heterogeneous regions where the separation is high are emphasized (see Figure 5.5). From the transparency of a particle it can directly be derived how much this particle separates from the anchor line over time. The time a particle has traveled until it deviates to a fixed distance from the anchor is not directly encoded as a visual attribute, but it can be determined from the animation of particles over time and could also be encoded as an additional attribute like color or size.

In addition to the user-controlled placement of anchor lines, we propose to select the starting points of these lines automatically. In particular, we let points be positioned in the interior of a user-defined probe, but we only accept a point as a starting point if the FTLE at its position is above a certain threshold. Otherwise we randomly select

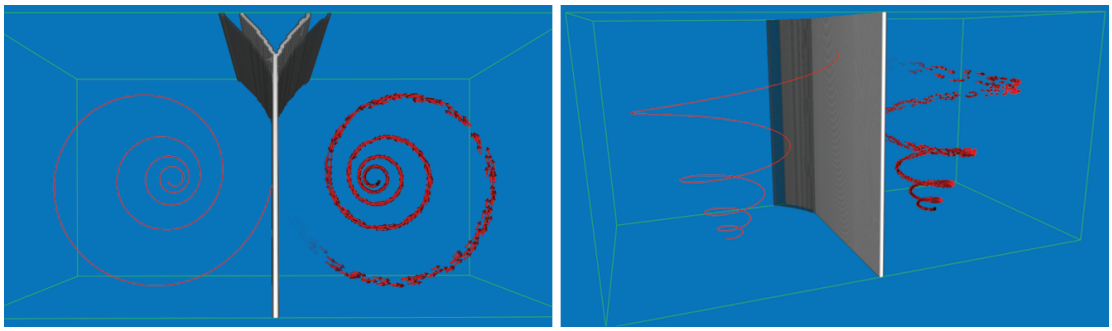


Figure 5.6: *An anchor line placed in regions of high FTLE can effectively describe why a separatrix in the dual vortex flow has been detected.*

another start location within the probe. It is worth noting here that the FTLE is precomputed at every point of the given sampling grid and adequate starting locations are also determined on the CPU on demand (and prior to GPU-based integration and visualization). Furthermore, we set the displacement distance for particles positioned around the center trajectory according to the grid spacing of the FTLE importance volume. Then, the particles correspond to the initial perturbation used during FTLE flow map computation.

The reason for restricting the placement of anchor lines to regions of high FTLE is as follows. While the FTLE characterizes the rate of separation of particles, it does neither indicate into which direction particles separate nor does it tell where the particles separate along a trajectory. Anchor lines placed in regions of high FTLE, on the other hand, are able to answer both questions and can, thus, be used for an improved analysis of the flow. Figure 5.6 demonstrates this property.

The deviation of particles from their anchor lines, and thus their transparency, is computed as follows. The anchor (path) lines are traced as described in Section 4.7. The particles scatted around all anchor lines are stored in a separate pair of 2D textures, which are alternatively updated in every advection step (as described in Section 4.5.1). These particles get assigned an additional index i that is used to reference the corresponding anchor line. In the jth advection step every particle looks up the respective position along the ith line and computes the distance between this position and its own position. This value is then used to determine the transparency of a particle.

5.5 Rendering Aspects

In this section we describe the necessary extensions of the GPU particle engine to allow for importance driven particle rendering. Furthermore, we will discuss rendering-specific performance aspects with respect to α-blending of the particle set.

5.5.1 Particle Morphing

The rendering of oriented point sprites of different size, shape and appearance is accomplished by extending the concept of the 2D sprite texture atlas as described in Section 4.6.1. Such an atlas contains a 2D array of different views of a 3D particle primitive. Views are parameterized with respect to scaling and rotation around an axis orthogonal to the viewing axis. To support differently shaped primitives, we build multiple of these atlases, each of which contains pre-computed images of a particular primitive. Each atlas is stored in a single slice of a 3D texture.

To continuously morph from one primitive into another one, we interpolate between the respective views of both primitives using 3D texture interpolation. Such an image-based blending between the same view of two different primitives is shown in Figure 5.7. The color and transparency of the interpolated views can be further modulated using the transfer functions described in Section 5.3.1. It should be mentioned here that the proposed technique can only be used if the views of two primitives that are morphed into each other are stored in successive 3D texture slices, as we employ hardware-supported trilinear filtering to achieve a smooth transition between shapes.

Figure 5.7: *Image-based morphing from an arrow into an ellipsoid.*

Although it is obvious that the proposed technique yields different results compared to geometry-based shape morphing and the construction of an atlas using the transformed geometry, our approach does not result in any noticeable artifacts. The reason for this is that particles are usually rendered as oriented primitives which show a very similar basic shape. The problem is further alleviated because we first blend between two views and then perform the scaling of the result to the adequate size of the primitives. It is clear, on the other hand, that we can easily build separate atlases for arbitrary primitives in-between the given basic shapes and store them in a 3D texture. This will result in even more flexibility to select particular shapes and their appearance in regions that can not clearly be classified in terms of importance and unimportance.

5.5.2 Blending

A critical aspect in the presented particle-based techniques is the use of transparency to visualize individual primitives. If particles are rendered as transparent sprites the order of their rendering becomes important. To guarantee a correct back-to-front or front-to-back order with respect to the viewer we use a GPU-based bitonic merge sort algorithm as proposed in [90]. The sorting algorithm essentially re-organizes the set of particles in such a way that they can be rendered in the order they are stored in local GPU memory.

As for a reasonable number of primitives sorting can quickly become the performance bottleneck, we also provide an additional rendering mode that entirely avoids sorting. This mode is inspired by the observation that in a typical interactive exploration session the majority of particles is assigned very high or very low transparency, either manually by focusing on a particular region or automatically by the proposed feature-based criterion.

The approach is similar to the standard approach used to render opaque and transparent objects in that first all the opaque particles are rendered, and in a second pass the remaining transparent particles are blended into the color buffer.

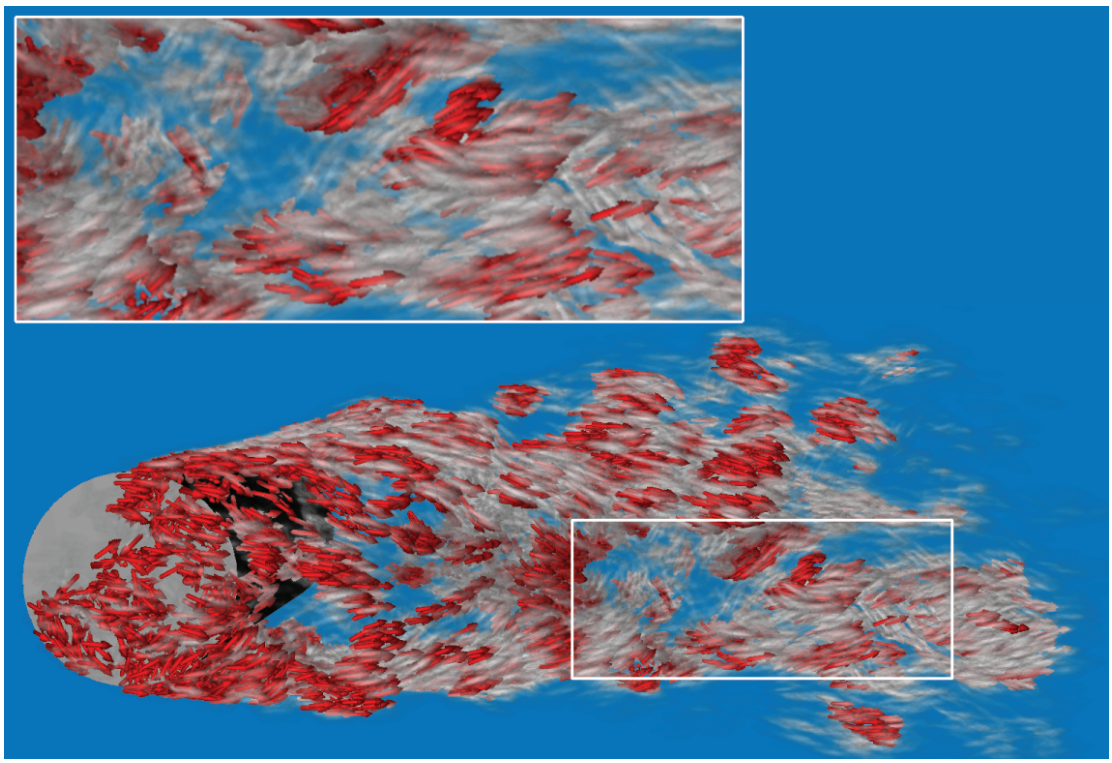


Figure 5.8: *α-Compositing of unsorted transparent particle primitives.*

In the first pass, the depth test is enabled and the depth value of a fragment surviving the depth test is written into the depth buffer. All particles are sent into the rendering pipeline, however, within the geometry shader—inflating the particles into point sprites—we remove all particles that are not fully opaque from the stream. In the second pass, writing to the depth buffer is disabled, and transparent particles are rendered in the order they are stored in GPU memory. To avoid brightness saturation as it is typically observed when accumulative blending is used, fragments are blended into the color buffer using alpha-compositing. Figure 5.8 demonstrates that the proposed rendering of opaque and transparent particles does not produce any noticeable artifacts, even though the visibility is not resolved correctly.

5.6 Results and Performance Analysis

We have used the proposed GPU techniques for the visualization of a number of real-world and synthetic 3D flow fields on uniform grids:

- *Flow around a box*: Result of a 3D time-dependent simulation of an incompressible turbulent flow around a square cylinder at $\mathrm{Re} = 22,000$. The simulation was performed using a spectro-consistent discretization of the Navier-Stokes equations [176] and it was carried out on a rectilinear grid of size $256\times448\times64$.

- *Flow around a cylinder*: Large eddy simulation of an incompressible unsteady turbulent flow around a wall-mounted finite cylinder at $\mathrm{Re} = 200,000$ [44]. 22 time steps were simulated. The size of the data grid is $256\times128\times128$.

- *Kármán vortex street*: Result of a 3D simulation of an incompressible unsteady flow over an immersed thin cuboid obstacle at $\mathrm{Re} = 100$. The simulation was performed via numerical solution of the Navier-Stokes equations according to [54]. The data set contains 30 time steps, each of which is of size $256\times64\times64$.

- *Double-vortex flow*: A steady axisymmetric flow with two counter-rotating vortices, which was computed using the following analytical expression for velocity:

  $$\mathbf{F}(\mathbf{x},\mathbf{y},\mathbf{z}) = ((-y+0.5)+(0.5-2x)/10,\ (2x-0.5)+(0.5-y)/10,\ -z/10).$$

  The computed velocity field corresponds to a spiral-like flow along the z-axis with the velocity magnitude decreasing towards the main axis of the spiral. The velocity field was mirrored to obtain the two symmetric vortices.

To validate the effectiveness of the proposed techniques, in Figure 5.9 we show additional visualizations of the described data sets using different importance-based visualization methods. With respect to the generated images, we should note here that the benefits of particle-based flow visualization can best be perceived in an animation. In a still image, oriented particles can show the direction of the flow quite clearly, but in contrast to LIC, for example, coherent particle trajectories can hardly be observed.

All of our tests were run on a dual core Core2 Duo 6600 equipped with a NVIDIA Geforce 8800 GTX graphics card with 786 MB local video memory. In terms of performance it can be observed that on recent GPUs the particle advection step only consumes a negligible fraction of the overall time. For instance, in a steady field about 100 million particles can be integrated per second on our target architecture.

The performance of the technique, thus, strongly depends on the number and the size of the rendered particles. In particular, as soon as many large particles are rendered the application quickly becomes rasterization bound and the overall performance can decrease considerably. On the other hand, as the proposed importance-driven approaches can effectively reduce the amount of rendered particles, in none of our experiments did the performance drop below 100 frames per second.

5.7 Summary

In this chapter, we have presented importance-driven particle techniques for 3D flow visualization. These techniques incorporate a number of importance measures to enable an improved visual analysis of the flow. The user controls the appearance of the visualization by a few parameters such as the size and location of a focus region, weights for the context region, and the size, shape and transparency of particles traced through the flow. In addition, feature measures that are directly derived from the flow are considered to adaptively modify the visual attributes of the particles. In this way, a better and faster understanding of complex flow structures is supported. As the proposed techniques run at interactive rates they can provide rapid visual feedback and, thus, allow for an effective visual exploration of the flow.

Finally let us mention that the proposed techniques can effectively be used for uncertainty visualization. By simply replacing focus by certainty and context by uncertainty the proposed techniques can be used to distinguish between regions containing reliable and non-reliable information. In the future we will investigate in more detail the application of the techniques proposed in this chapter for uncertainty visualization.

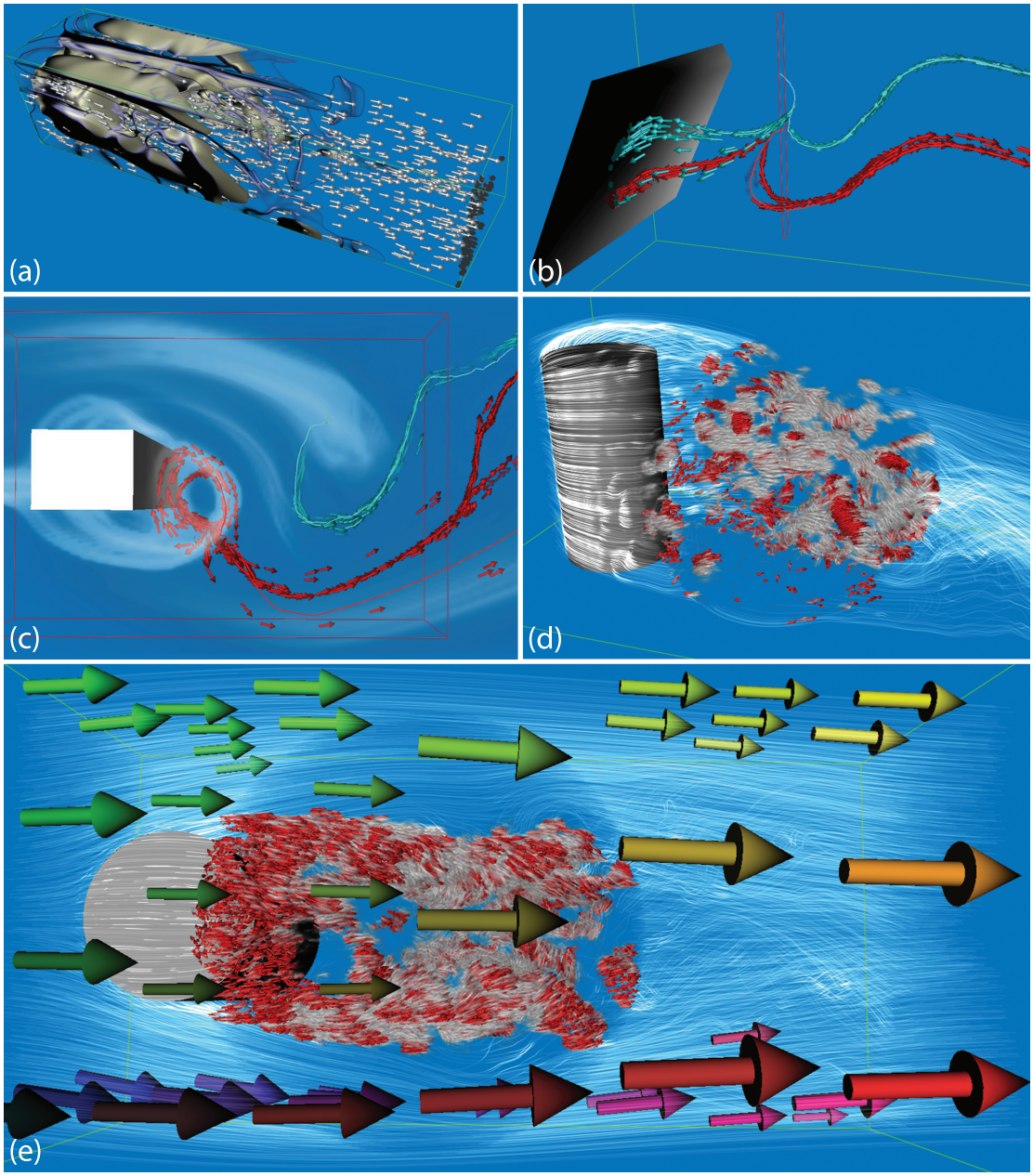


Figure 5.9: *Importance-driven particle visualization results: Image (a) depicts an iso-surface of the FTLE in the Kármán vortex street. The visualization of this data set using anchor lines is shown in image (b). Two anchor lines have been placed in the region of high FTLE. From the particle distribution one can see where particles start to separate from their anchor, and the transparency coding shows how fast they separate. Image (c) shows two anchor lines seeded in the region of high FTLE in the flow around a box. The distribution of the Lyapunov exponent is visualized using volume rendering. Images (d+e) depict two views on an unsteady flow in the large eddy data set. Here, a radially symmetric focus region was used to apply different visualization modalities. The particle appearance inside the focus region was adapted according to an importance volume based on helicity.*

Chapter 6

Interactive Streak Surface Visualization

In this chapter we present techniques for the visualization of unsteady flows using streak surfaces, which allow for the first time an adaptive integration and rendering of such surfaces in real-time. The techniques consist of two main components, which are both realized on the GPU to exploit computational and bandwidth capacities for numerical particle integration and to minimize bandwidth requirements in the rendering of the surface. In the construction stage, an adaptive surface representation is generated. Surface refinement and coarsening strategies are based on local surface properties like distortion and curvature. We compare two different methods to generate a streak surface: a) by computing a patch-based surface representation that avoids any interdependence between patches, and b) by computing a particle-based surface representation including particle connectivity, and by updating this connectivity during particle refinement and coarsening. In the rendering stage, the surface is either rendered as a set of quadrilateral surface patches using high-quality point-based approaches, or a surface triangulation is built in turn from the given particle connectivity and the resulting triangle mesh is rendered. We perform a comparative study of the proposed techniques with respect to surface quality, visual quality and performance by visualizing streak surfaces in real flows using different rendering options.

6.1 Introduction and Related Work

In geometry-based flow visualization, the integration and visualization of stream lines has been a standard tool from its very beginning. With the consideration of time-

dependent flows, path lines and streak lines have moved into the focus of research because they reflect important properties of the flow.

The visualization of integral surfaces has been proven to be common and useful in visual flow exploration. In the case of stream and path surfaces, their extraction is well-understood. The main idea is to integrate the front line of the surface and apply if necessary an adaptive refinement/coarsening to it. After the front has passed, the generated surface remains unchanged.

Streak surfaces have a strong relation to experimental flow visualization where external materials such as dye, hydrogen bubbles or heat energy are injected into the flow. The advection of these external materials creates streak lines and shows the flow patterns. Due to this reason, analogues to these experimental techniques have been adopted by researchers in computer-aided scientific visualization for flow exploration. However, up to now streak surfaces are rarely applied because of the computational complexity of streak surface generation. Since streak surfaces may change their shape everywhere and at any time of the integration, every part of the surface has to be monitored at any time of the integration for adaptive refinement/coarsening. Due to this fundamental difference to stream and path surfaces, the consideration of streak surfaces makes only sense if their evolution over time is shown, e.g., in a pre-computed video sequence or in interactive applications with a real-time performance. However, due to the computational complexity of streak line integration and adaptive integral surface construction, streak surfaces have only rarely been used in practice.

Hultquist [65] presented the first adaptive stream surface integration approach which was later extended in different ways: The approach by Stalling [160] uses local topological information to increase accuracy. Scheuermann et al. [147] compute exact solutions of stream surfaces inside piecewise linear vector fields. In the work by van Wijk [174] a global implicit approach for certain stream surfaces is given. Recently, a construction method for stream surfaces of high polynomial precision has been introduced by Schneider et al. [150]. Garth et al. [50] discussed a number of enhancements in the context of vortex extraction. In another work by Garth et al. [48], improved integral surface accuracy was achieved by separating characteristic line integration and integral surface triangulation. A particle-based approach for the generation and rendering of stream surfaces was proposed by Schafhitzel in [145].

The methods proposed by Schafhitzel [145] and Garth et al. [48] are also the only approaches describing the surface extraction in a time-dependent context for path surfaces. The generalization from stream surfaces to path surfaces is rather straightforward because only the kind of integration at the advancing surface front has to be replaced.

The only approach so far to address the real-time requirement was proposed by Funck et al. [181]. It combines the streak surface integration with a smoke metaphor, leading to cancelation effects of problematic surface parts: parts of the streak surface where an adaptive refinement is necessary are rendered less opaquely. In this way, smoke like structures are obtained by a streak surface integration without any adaptive refinement. On the other hand, the value of this approach for visual flow exploration is limited because it cannot guarantee to find all relevant flow structures, as fine structures can only be revealed if the initial tessellation of the mesh already respects these subtleties. Thus, while this approach gives interesting smoke-like structures, it is unable to produce fully adaptive, opaque streak surfaces.

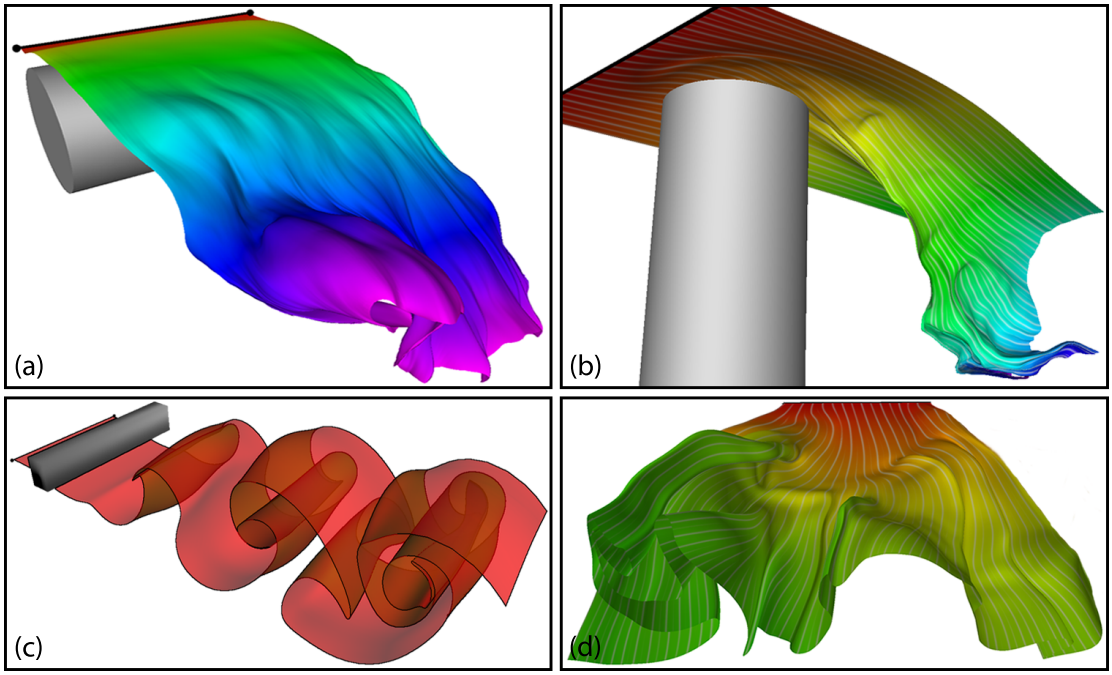


Figure 6.1: *Our method generates adaptively refined integral surfaces in 3D flows on the GPU. The shown surfaces were generated and rendered in less than 50ms. Figures (a-c) show streak surfaces in unsteady flows. Figure d) shows a stream surface.*

6.2 Contribution

In this chapter, we present the first real-time approach for adaptive streak surface integration and high-quality rendering. We achieve this by using particle-based approaches in which either the surface is represented as a set of surface patches that can be handled independent of each other (see Figure 6.1 (a)), or a closed surface triangulation is

computed from the given particle set (Figure 6.1 (b-d)). For both approaches we have developed methods for interactive surface refinement and coarsening based on local surface properties.

While the former approach is elegant in its simplicity, it requires redundant particle computations and lacks flexibility in the rendering process. Even though we use an advanced rendering method similar to high-quality point-splatting [15], rendering artifacts at patch boundaries can not be avoided entirely. The second approach, on the other hand, yields a closed surface representation providing a variety of rendering options, but it can result in deformed triangulations and rendering artifacts thereof.

This chapter contains the following specific contributions:

- A patch-based scheme for the adaptive generation of streak surfaces and a high-quality patch-based surface rendering technique.
- A particle-based adaptive refinement/coarsening scheme for streak surface generation and a novel method to construct a closed triangular streak surface from a set of particles.

The remainder of this chapter is organized as follows. An introduction to streak surfaces is given in Section 6.3. Section 6.4 presents a novel technique to construct and render a patch-based streak surface representation. In Section 6.5 we describe the particle-based technique for streak surface generation in which local connectivity information is used to assure a uniform sample density along the surface and to build a surface triangulation. In Section 6.6 we evaluate the performance of our approaches, and we discuss their advantages and limitations. We conclude this chapter with an outline of future research in the field.

6.3 Streak Surfaces

Streak surfaces are defined by repeatedly setting out particles on a line-shaped seeding structure over a certain time interval. The collection of all these particles at a certain time denotes the streak surface. Technically, a streak surface can be obtained in the following way for a 3D time-dependent flow field $\mathbf{v}(\mathbf{x},t)$: the seeding structure is considered to be a polyline consisting of the points $\mathbf{s}_0,\ldots,\mathbf{s}_n$. At the time $t_i = t_0 + i\,\Delta t$ we start a path line integration of the particle $\mathbf{x}_{i,j}$ from the seeding point $\mathbf{s}_j$ and observe its behavior over t:

$$\mathbf{x}_{i,j}(t) = \mathbf{x}_{i,j}(t_i) + \int_{t_i}^{t} \mathbf{v}(\mathbf{x}_{i,j}(s),s)\,ds \tag{6.1}$$

with $\mathbf{x}_{i,j}(t_i) = \mathbf{s}_j$, $i = 0,...,m$ and $j = 0,...,n$. For $t \geq t_m = t_0 + m\,\Delta t$, the streak surface can be considered as a rectangular vertex array $(\mathbf{x}_{i,j}(t))$. We call a column $(\mathbf{x}_{i,0},...,\mathbf{x}_{i,n})$ a time line, while a row $(\mathbf{x}_{0,j},...,\mathbf{x}_{m,j})$ is a streak line. The vertices are the surface points from which a closed surface representation has to be built.

During the integration, the distance between both adjacent time lines and streak lines may vary at any location of the surface. Thus, after every integration step the surface has to be checked everywhere for adaptive refinement or coarsening. This means that, based on an appropriate refinement/coarsening criterion, new particles have to be seeded between adjacent points along a particular time or streak line, or adjacent points have to be merged. This process is computationally very complex because streak surfaces appear to have a rather large distortion after their seeding. An increase of the surface area by a factor of 100 or more is not unusual, leading to a high number of refinement steps. It is worth noting that in an interactive application, the adaptive refinement/coarsening has to be monitored and carried out at any time simultaneously with real-time performance.

6.4 Patch-based Streak Surface Generation

By using a patch-based approach, the streak surface generation and rendering process is split into a set of independent operations on each patch. These operations can then be executed in parallel, and all the patches can be rendered independent of each other. The computation of adjacency information between surface points, as it is required for the computation of a surface triangulation, can be avoided.

6.4.1 Patch Generation and Refinement

As described in Section 6.3, a streak surface can be constructed by repeatedly releasing particles from a line-shaped seeding structure over a certain time interval and by connecting these particles to form a closed surface. All particles $(\mathbf{x}_{i,0},...,\mathbf{x}_{i,n})$ released at time $t_i = t_0 + i\,\Delta t$ reside on one advancing front. We call this front the time line tl_i.

A new advancing front (tl_i) is released in the form of n quadrilateral patches $\mathbf{p}_{i,v}$, with $v = 0,...,n-1$. Each patch consists of four vertices $(\mathbf{s}_v, \mathbf{s}_{v+1}, \mathbf{x}_{i,v}, \mathbf{x}_{i,v+1})$, which are duplicated and stored separately for each patch. The patch vertices are then advected through the flow as described before, and the shape changes a patch undergoes due to the particles' movement are used to steer the refinement process.

The refinement of surface patches is performed for each patch separately with respect to an area-based criterion. Specifically, we set a threshold to Ξ^2, where Ξ is the

distance between two adjacent points uniformly distributed along the seeding structure. If, at any time, the area of a patch is greater than $\alpha\Xi^2$, where α is a real number larger than 1 controlling the subdivision strength, the patch is subdivided into two quadrilaterals. This is performed by splitting the patch along its longest edge and the edge opposite to it. The two new patches and their vertices are stored separately, and the refined patch is removed (see Figure 6.2).

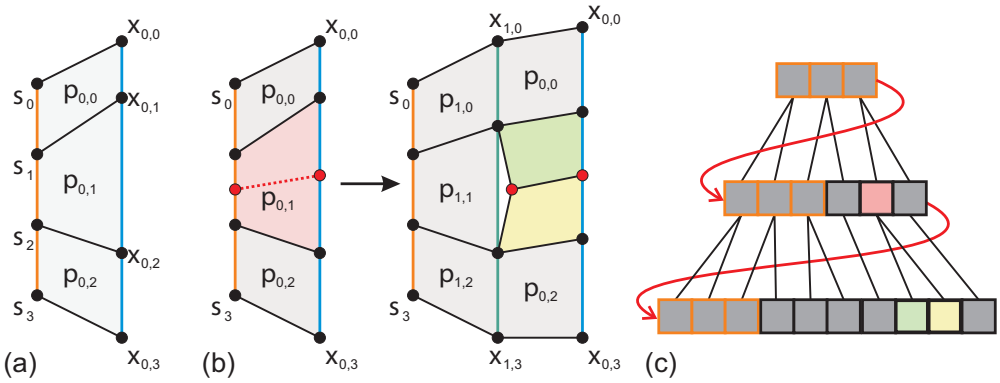


Figure 6.2: *(a) A patch-based streak surface representation after the first time line has been released. (b) Left: Patch $p_{0,1}$ meets the refinement criterion and is split into two patches. Right: The surface patches after the second integration step. The generation of new surface points due to the splitting operation has lead to a hole in the surface representation. (c) The corresponding layout of the linear memory segments storing the surface patches in each time step.*

6.4.2 GPU Implementation

Shader model 4.0 compliant GPUs provide possibilities to efficiently perform the patch-based streak surface generation: we employ a geometry shader to manipulate a primitive stream by appending or removing primitives, and the stream output stage is used to direct the resulting stream to intermediate buffers in GPU memory. Since buffer resources cannot be bound as pipeline input and stream output target simultaneously, we use two instances and toggle between them in a ping-pong fashion (see Section 4.5.2).

Each surface patch is represented by its four vertices, a scalar value counting the number of integration steps, and a counter indicating its refinement depth. On the GPU, for every patch this information is stored as one vertex (i.e., contiguous data block) in a vertex buffer. Since current GPUs cannot change the size of a resource residing in GPU memory dynamically, two buffers that are large enough to store the entire adaptively refined streak surface have to be allocated before the surface construction begins. By letting the user select the number n of patches that are released in each time step, a maximum refinement depth d and the maximum number of inte-

gration steps m a patch performs until it is removed, each buffer must be able to store $n \times 2^d \times (m-d+1)$ patches.

The streak surface construction starts by storing n zero area patch primitives $\mathbf{p}_j^0$, with $j = 0,\ldots,n-1$, at the beginning of the vertex buffer employed in the first advection pass. In the following we assume n to be an even number. These elements are used in every time step to repeatedly release a new patch front into the flow. The respective vertices of patch $\mathbf{p}_j^0$ are $(\mathbf{s}_j, \mathbf{s}_{j+1}, \mathbf{s}_j, \mathbf{s}_{j+1})$. In each integration step, all buffer elements are passed to the geometry shader and processed as follows: For each of the first $n/2$ elements $\mathbf{p}_j^0$ with $j = 0,\ldots,n/2-1$ the shader writes the two zero area patches $\mathbf{p}_{2\times j}^0$ and $\mathbf{p}_{(2\times j)+1}^0$ to the output buffer. Access to the vertices of these patches is achieved by binding the input stream buffer as shader resource. Since these n patches are always written first, they remain at the beginning of the buffer. For each of the remaining $n/2$ elements the shader appends two patch elements to the buffer, which represent the currently released patch front. Each of these patches is then expanded by integrating its last two vertices to new positions.

For the remaining buffer elements, which contain patches that were released into the flow at previous time steps, the refinement criterion is evaluated before the integration is performed. If no refinement is necessary, the geometry shader advects the patch vertices, increments the integration step counter and appends the patch element to the output stream. Otherwise, the geometry shader splits the element as described, advects the four original as well as the two new vertices, and appends the two new patch primitives to the output stream. The refinement counters of the new primitives are set to the counters of the refined patch and incremented by one. Figure 6.2 (c) illustrates the growth of the vertex array buffer due to the seeding and refinement of surface patches.

6.4.3 Patch-based Streak Surface Rendering

The patch-based surface representation can be rendered directly by sending the vertex array buffer through the graphics pipeline and rasterizing the patches separately. However, since T-vertices are introduced by the particular refinement strategy, holes in the surface representation can occur. To cover these holes, we adopt a rendering technique that was introduced by Botsch and co-workers in the context of point splatting [15]. Figure 6.4 shows an adaptively refined patch-based streak surface ($\alpha = 1.2$), which was rendered using simple point rendering of the patch centroids (left) and the patch-based splatting approach (right).

A two pass rendering approach is performed before deferred per-pixel surface lighting is computed. Therefore, all patches in the vertex buffer are rendered twice. In each

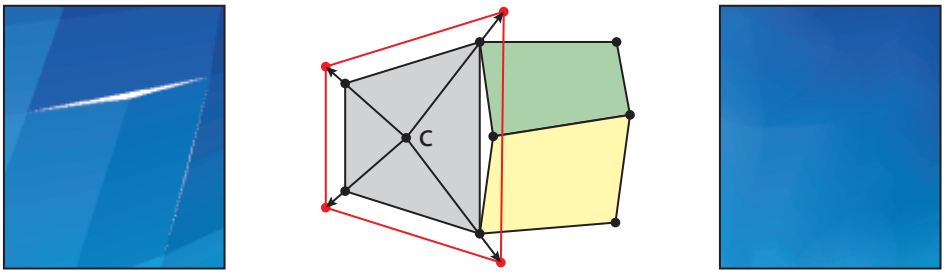


Figure 6.3: *Left: Separate triangulation and rendering of each patch leads to holes in the streak surface. Middle: Holes are covered by rendering enlarged patches. Right: Rendering enlarged patches in a way similar to high-quality point splatting yields a closed and smooth surface.*

pass a geometry shader enlarges every patch by changing its vertices $\mathbf{px}_k$ ($k = 0,\ldots,3$) to $\mathbf{px}_k + \beta||\mathbf{px}_k - \mathbf{c}||$. Here, $\mathbf{c}$ is the patch centroid and β is a user defined scaling factor. As shown in Figure 6.3, patches are then split into the four triangles spanned by their centroid and the patch vertices before they are rendered.
In the first rendering pass, commonly referred to as *visibility pass*, a depth imprint of the enlarged surface patches closest to the viewer is generated. In the second pass, also known as *attribute pass*, the patch-based surface representation is rendered again using a biased depth test on the generated depth imprint. In this way, only patch samples close to the first rendered surface survive. In a pixel shader, the patch attributes like color and normal are weighted by a Gaussian kernel centered at the patch centroid, and these contributions are finally accumulated via additive blending and normalization. In this way, a smooth transition of patch attributes is obtained in regions where multiple enlarged patches overlap.

Due to the bending of streak surfaces, it can happen that surface samples having a large geodesic distance from each other become close to each other and fall into the same pixel. In this case, the biased depth test might let both samples pass and accumulate in the pixel buffer. To avoid this, we assign two additional parameter values to each patch. The first value indicates the position of a patch in the ordered set of all possible patches along the seeding structure. Starting with the initial patches $\mathbf{p}_{i,v}$, which are assigned the positions $v \times 2^d$, in every refinement step the first new patch keeps the position of the refined patch and the second patch adds 2^{d-k} to this position. Here, k is the current refinement level. The index i of each patch is assigned as the second parameter value. In the visibility pass, these values are rendered into a separate render target, and they are then used in the attribute pass to discard those fragments that are close to the rendered surface samples but have parameter values that differ more than a given threshold.

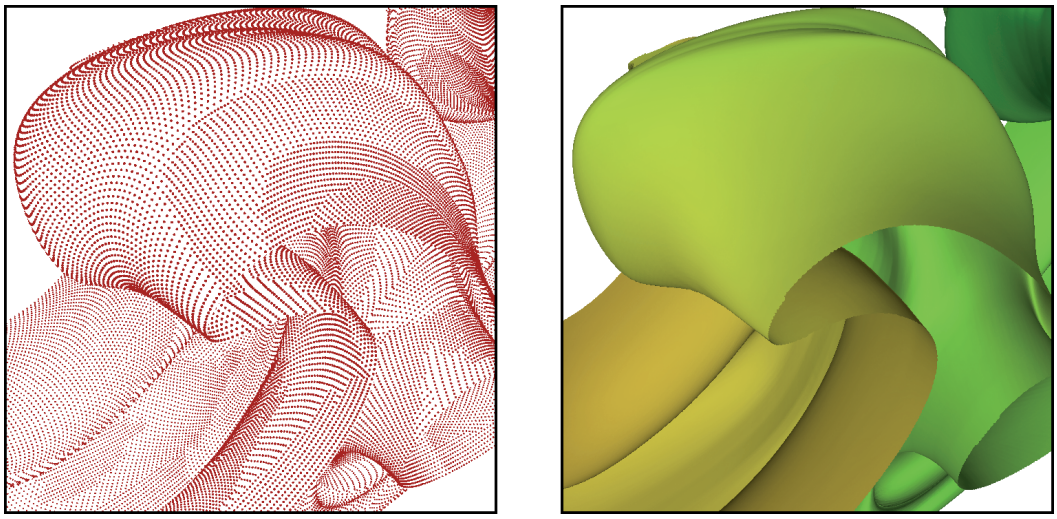


Figure 6.4: *Left: Rendering of the patch centroids of a patch-based streak surface. Right: The same streak surface rendered via quad-splatting.*

6.5 Mesh-based Streak Surface Generation

Patch-based streak surface generation entirely avoids to store and update any connectivity between the patches. On the other hand, because every patch stores its own set of vertices even though they might be shared between patches, a considerable amount of memory is wasted and numerical integration of the same vertex is performed up to four times. To overcome this overhead we propose a novel GPU approach to construct an adaptive triangular streak surface representation from the set of seeded particles.

Similar to the data layout that was used in the patch-based approach, all particles seeded into the flow are stored in a linear vertex array. Each particle $\mathbf{x}_{i,j}$ released from the seeding structure is assigned an index $id_{i,j} = j \times 2^d$, where d is the maximum refinement depth. The particle set belonging to a particular time line tl_i is stored in a contiguous block b_i in this buffer. The blocks are ordered such that block b_{i-1} follows block b_i, with block b_0 being the last in the buffer.

In every advection step the particles are processed in the order of their occurrence in the buffer, and they are written to the output buffer in the same order. If a new particle is generated due to the splitting of an existing particle, it is placed directly behind this particle in the output buffer. If a particle is removed, it is simply not written into this buffer. On the GPU this is realized by executing a geometry shader with a variable primitive output of 0-2 elements for each incoming primitive. In the same way as described in the previous section, the maximum buffer size has to be computed up front depending on the number of particles per time line, the maximum refinement

depth, and the maximum number of integration steps. Then, two ping-pong buffers of this size have to be allocated.

Analogous to the patch-based approach, a streak surface is constructed by repeatedly releasing time lines at a fixed frequency into the flow. And whenever a new time line is released from the seeding structure, we do not perform adaptive refinement/coarsening in the first advection step, but use the second possible geometry shader output to preserve the seed particles at the start of the vertex buffer.

6.5.1 Particle Refinement

Our method for generating an adaptive streak surface triangulation from a given set of subsequently released time lines can be separated into three passes:

- *Time line refinement*: Every time line is refined/coarsened based on local criteria like stretching, compression, and line curvature, as well as a global criterion taking into account the change in surface area.
- *Connectivity update*: The connectivity between particles on adjacent time lines is established.
- *Streak line refinement*: The connectivity information is used to compute local streak line properties, which are considered to steer the refinement of streak lines.

Time line refinement

Time line refinement adapts the particle density along each time line prior to the particle integration. The refinement/coarsening criteria we apply have been adopted from previous work in the field. The first criterion considers the flow divergence at a particle position as introduced in [65]. Let $\Phi(\mathbf{a},\mathbf{b})$ be the distance between particles $\mathbf{a}$ and $\mathbf{b}$, and Ξ the initial distance between two adjacent seed points, then the particle $\mathbf{x}_{i,j}$ spawns a new particle between $\mathbf{x}_{i,j}$ and $\mathbf{x}_{i,j+1}$—we call this operation particle splitting—if

$$\Phi(\mathbf{x}_{i,j},\mathbf{x}_{i,j+1}) > \alpha\,\Xi. \tag{6.2}$$

Similar to [50], the second criterion considers the approximate local curvature along a time line. Let $\Theta(\mathbf{u},\mathbf{v},\mathbf{w})$ be defined as

$$\Theta(\mathbf{u},\mathbf{v},\mathbf{w}) = \frac{\left(\frac{\mathbf{u}-\mathbf{v}}{||\mathbf{u}-\mathbf{v}||}\cdot\frac{\mathbf{w}-\mathbf{v}}{||\mathbf{w}-\mathbf{v}||}\right)+1}{2}, \tag{6.3}$$

where $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are the positions of three particles. A particle $\mathbf{x}_{i,j}$ is split if

$$\Theta(\mathbf{x}_{i,j-1}, \mathbf{x}_{i,j}, \mathbf{x}_{i,j+1}) + \Theta(\mathbf{x}_{i,j+2}, \mathbf{x}_{i,j+1}, \mathbf{x}_{i,j}) > \beta. \tag{6.4}$$

In this way, the deviation of the time line from a straight line is approximated and used to steer the local time line refinement. Figure 6.5 sketches the first pass of the mesh-based streak surface construction approach.

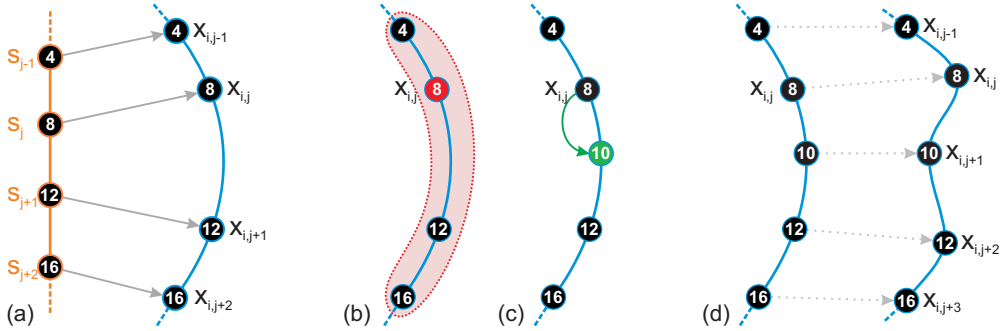


Figure 6.5: *(a) In each time step a new time line is released from the seeding line. Node values show particle ids. (b) Prior to integration, each particle $\mathbf{x}_{i,j}$ evaluates refinement criteria based on its local neighborhood (red). (c) $\mathbf{x}_{i,j}$ satisfies a refinement criterion and performs a particle split. (d) The resulting time line before (left) and after (right) the subsequent integration step.*

Particle splitting is performed by fitting a cubic polynomial $p(t)$ through $\mathbf{x}_{i,j-1}$, $\mathbf{x}_{i,j}$, $\mathbf{x}_{i,j+1}$ and $\mathbf{x}_{i,j+2}$, and by evaluating $p(t)$ at $t = \frac{1}{2}$:

$$p(1/2) = -\frac{1}{16}(\mathbf{x}_{i,j-1} + \mathbf{x}_{i,j+1}) + \frac{9}{16}(\mathbf{x}_{i,j} + \mathbf{x}_{i,j+2}). \tag{6.5}$$

Based on the indices $id_{i,j}$ of the initially seeded particles $\mathbf{x}_{i,j}$, every new particle on a time line gets assigned its index in the ordered set of all possible particles along this line as described in the previous section for surfaces patches. We will subsequently call these indices the particle ids. Figure 6.6 illustrates the changes in the particle layout on a time line due to refinement and coarsening events over three integration steps.

To prevent the streak surface from unlimited stretching, we adapt a criterion that was proposed for stream surfaces in [65]. We compare the current distance between two particles to their distance in the last time step in relation to the distance a particle has moved due to the integration. Let $\Psi(\mathbf{a}, \mathbf{b}, t)$ be the distance between particles $\mathbf{a}$ and $\mathbf{b}$ at time t, and $\mathbf{p}_{i,j,t}$ the position of particle $\mathbf{x}_{i,j}$ at time t. We mark an edge as invalid, meaning that it will not be refined any further, if the following expression evaluates to true:

$$\Psi(\mathbf{x}_{i,j}, \mathbf{x}_{i,j+1}, t) - \Psi(\mathbf{x}_{i,j}, \mathbf{x}_{i,j+1}, t-1) > \gamma\, \Phi(\mathbf{p}_{i,j,t}, \mathbf{p}_{i,j,t-1}) \tag{6.6}$$

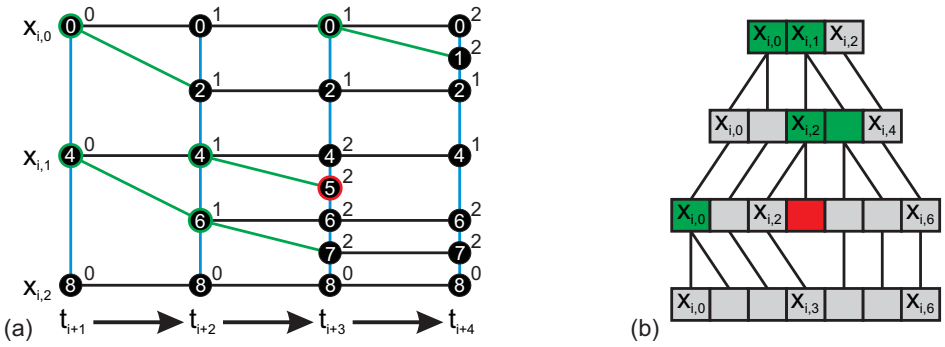


Figure 6.6: *(a) Evolution of a time line over three integration steps. Green nodes indicate refinement and red nodes coarsening events. Numbers next to the nodes indicate the refinement level. (b) Changes in the linear memory segment b_i due to vertex refinement/coarsening.*

If an edge has been classified as invalid or cannot be refined any further, it is not considered in the triangulation of the streak surface described below. In this way, the surface is cut in regions where it stretches too much, e.g., if it evolves around obstacles in the flow as demonstrated in Figure 6.7.

Finally, in addition to inserting new particles we remove a particle $\mathbf{x}_{i,j}$ if:

$$\begin{aligned}&(\Phi(\mathbf{x}_{i,j},\mathbf{x}_{i,j-1})+\Phi(\mathbf{x}_{i,j},\mathbf{x}_{i,j+1})<\delta\Xi)\ \wedge\\&(\Theta(\mathbf{x}_{i,j-1},\mathbf{x}_{i,j},\mathbf{x}_{i,j+1})+\Theta(\mathbf{x}_{i,j},\mathbf{x}_{i,j+1},\mathbf{x}_{i,j+2})<\zeta).\end{aligned}\tag{6.7}$$

Due to this coarsening we avoid vertex clustering in regions of high convergence, and we prevent the generated triangles from becoming too small.

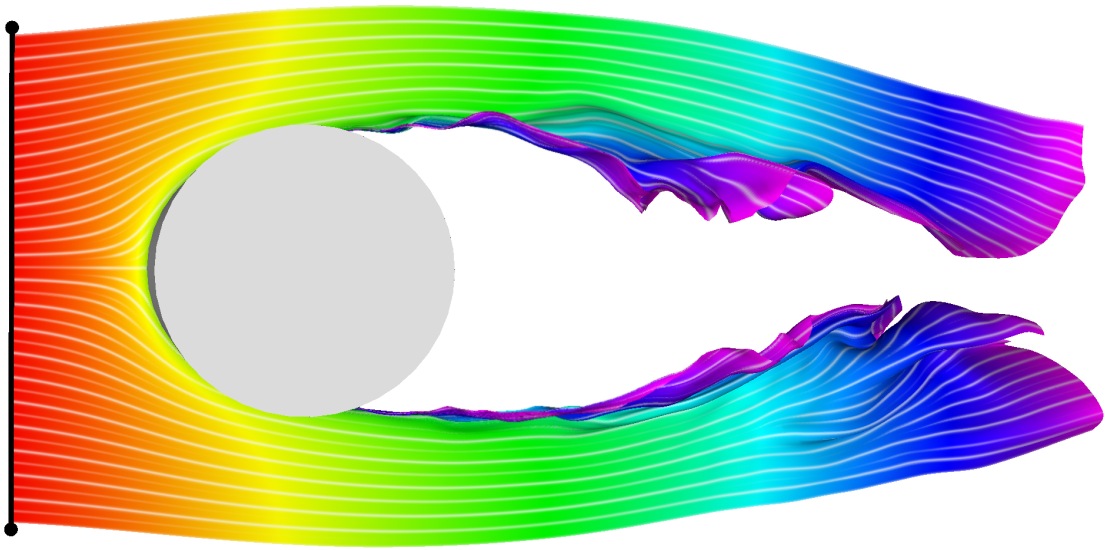


Figure 6.7: *Application of criterion* (6.6) *prevents a streak surface from unlimited stretching by cutting edges if no additional refinement can be performed.*

Connectivity update

Due to time line refinement and coarsening the connectivity between particles on adjacent time lines has to be computed in each integration step. Therefore, every particle on time line tl_i searches for the particle on tl_{i+1} and the one on tl_{i-1} having the id closest to its own one on the respective time line. We will call these two particles the predecessor and the successor of a particle. In particular, for a particle $\mathbf{x}_{i,j}$ we select the successor $\mathbf{x}_{i+1,succ}$ with the closest id $\leq$ the particle's id and the predecessor $\mathbf{x}_{i-1,pred}$ with the closest id $\geq$ the particle's id (see Figure 6.8 (a)). Once the predecessor and the successor have been determined, references to them are stored as offsets to the absolute position of the particle in the vertex buffer, and they are used as described below to build a closed surface representation.

Finding the two particular neighbors requires every particle to search the vertex buffer to the left and to the right of it, with the search radius depending on the number of particles on time lines tl_{i-1}, tl_i and tl_{i+1}. We will describe in Section 6.5.3 how to determine these numbers in a very efficient way on the GPU.

Streak line refinement

In this pass, a complete time line is added to or removed from the streak surface. The criterion to steer the refinement/coarsening is based on the maximum Euclidean distance between neighboring time lines.

A new time line is inserted between tl_i and tl_{i+1} if the maximum of the shortest distances between particles on tl_i and the time line tl_{i+1} exceeds a user defined threshold. An existing time line is removed if the maximum of the shortest distances to both adjacent time lines falls below a given threshold. Unfortunately, since we do not know the exact time line between the given vertices, computing the shortest distance from a particle to this line is not possible in general. Therefore, we proceed as follows: Since $\mathbf{x}_{i+1,succ}$ is the closest existing control point on tl_{i+1} with $id_{i+1,succ} \leq id_{i,j}$ and its adjacent particle $\mathbf{x}_{i+1,succ+1}$ has a larger particle id, we first interpolate an intermediate position $\mathbf{z}$ on the line segment spanned by $\mathbf{x}_{i+1,succ}$ and $\mathbf{x}_{i+1,succ+1}$ as follows:

$$\mathbf{z} = \mathbf{x}_{i+1,succ} + a(\mathbf{x}_{i+1,succ+1} - \mathbf{x}_{i+1,succ}), \quad a = \frac{id_{i,j} - id_{i+1,succ}}{id_{i+1,succ+1} - id_{i+1,succ}}. \tag{6.8}$$

We then compute the Euclidian distance between $\mathbf{x}_{i,j}$ and $\mathbf{z}$, and we use this distance as the shortest distance of $\mathbf{x}_{i,j}$ to the time line tl_{i+1}. The distance to the preceding time line tl_{i-1} is determined analogously (see Figure 6.8 (b)).

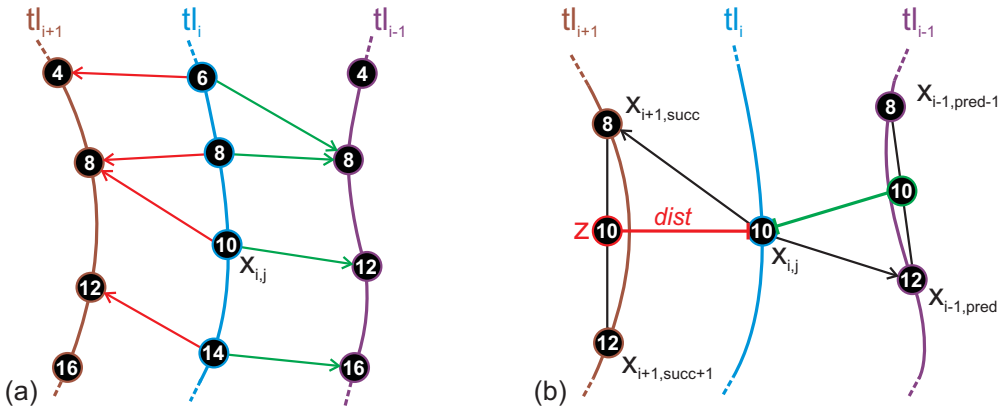


Figure 6.8: *(a) Each particle on time line tl_i selects its successor (red arrows) and predecessor (green arrows) on adjacent time lines based on the closest matching particle id. (b) The distance estimate of a particle $\mathbf{x}_{i,j}$ to its adjacent time line tl_{i+1} is based on an intermediate particle* **z**, *exhibiting the same particle id.*

A new particle front that is added due to streak line refinement contains the same number of particles as the time line triggering the refinement event. As shown in Figure 6.9, the new front is stored as a contiguous block in the vertex buffer right before this time line. Particle positions and normal values are linearly interpolated between $x_{i,j}$ and intermediate values on tl_{i+1} as described in Equation (6.8).

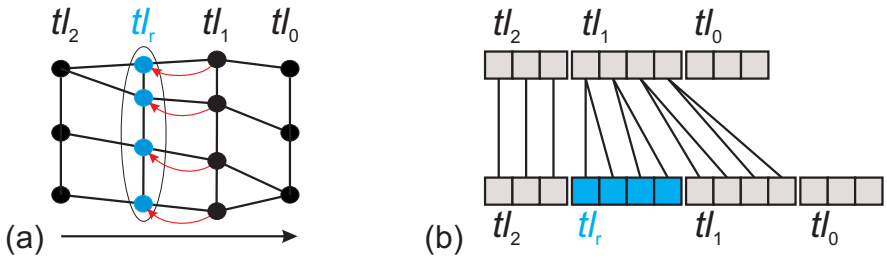


Figure 6.9: *Streak line refinement: (a) The time line tl_1 satisfies the refinement criterion and spawns the new time line tl_r. (b) illustrates the corresponding changes in the vertex array buffer.*

6.5.2 Streak Surface Triangulation and Rendering

To render the surface as a watertight triangle mesh a final pass is executed. Prior to triangulation, a geometry shader validates the connectivity and updates the neighborhood for all particles residing on time lines whose adjacent time lines have been removed due to streak line refinement.

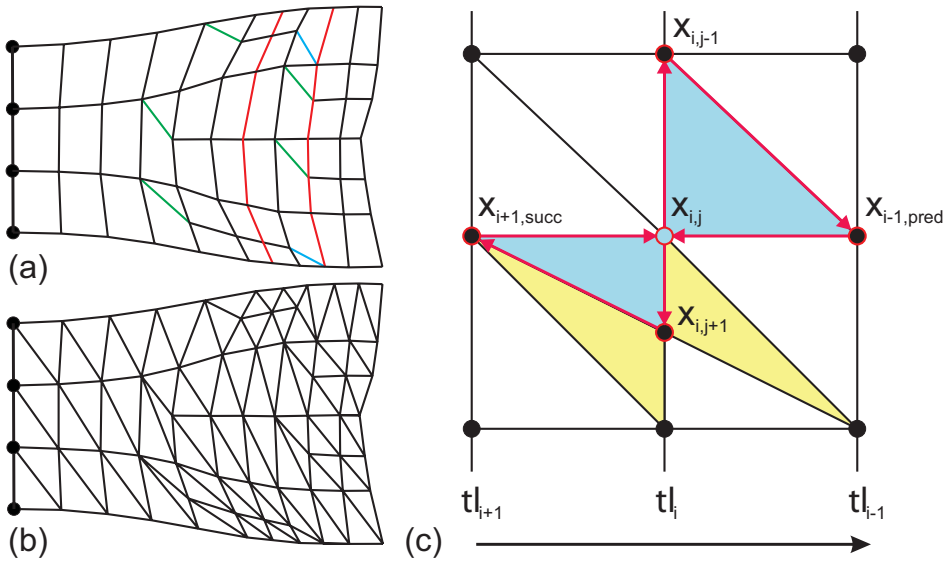


Figure 6.10: *Streak surface triangulation: (a) Vertex connectivity and refinement events: Green edges indicate vertex splitting, blue edges indicate vertex merging, and red edges indicate streak line refinement. The resulting triangulation is shown in (b). In (c) the two triangles generated by the vertex* $\mathbf{x}_{i,j}$ *are colored blue. Yellow triangles are generated by vertex* $\mathbf{x}_{i,j+1}$.

A closed surface representation is generated by using the particle connectivity to compute a triangulation of adjacent time lines. For each particle that is sent to the rendering pipeline the geometry shader creates two triangles and appends them to the output stream. The first triangle is spanned by the vertex $\mathbf{x}_{i,j}$, its local right neighbor $\mathbf{x}_{i,j+1}$, and its successor on the time line tl_{i+1}. The second triangle consists of the vertex $\mathbf{x}_{i,j}$, its local left neighbor $\mathbf{x}_{i,j-1}$, and its predecessor on the time line tl_{i-1}. Since this process is performed for every vertex, a watertight surface is generated. Figure 6.10 illustrates this triangulation process.

Triangles containing an edge that was marked invalid due to the criterion in Equation 6.6 are excluded from the output stream. Note that particles on the surface border ($i = 0 \vee i = n \vee j = 0 \vee j = m$) contain at least one invalid neighbor, such that the corresponding triangle is also excluded from the stream output.

Once the triangulation has been generated it can be rendered directly using various rendering styles. Since the tuples $(i, id_{i,j})$ that are stored for each particle correspond to a surface parametrization, they can be used to texture the streak surface. In Figure 6.11(a,c,e) this parametrization was used to color the surface with streak lines. Image (b) additionally emphasizes time lines. In images (d) and (f) depth peeling [40] was applied to create a semi transparent visualization of the streak surface (in combination with image based edge detection to amplify sharp features on the streak surface).

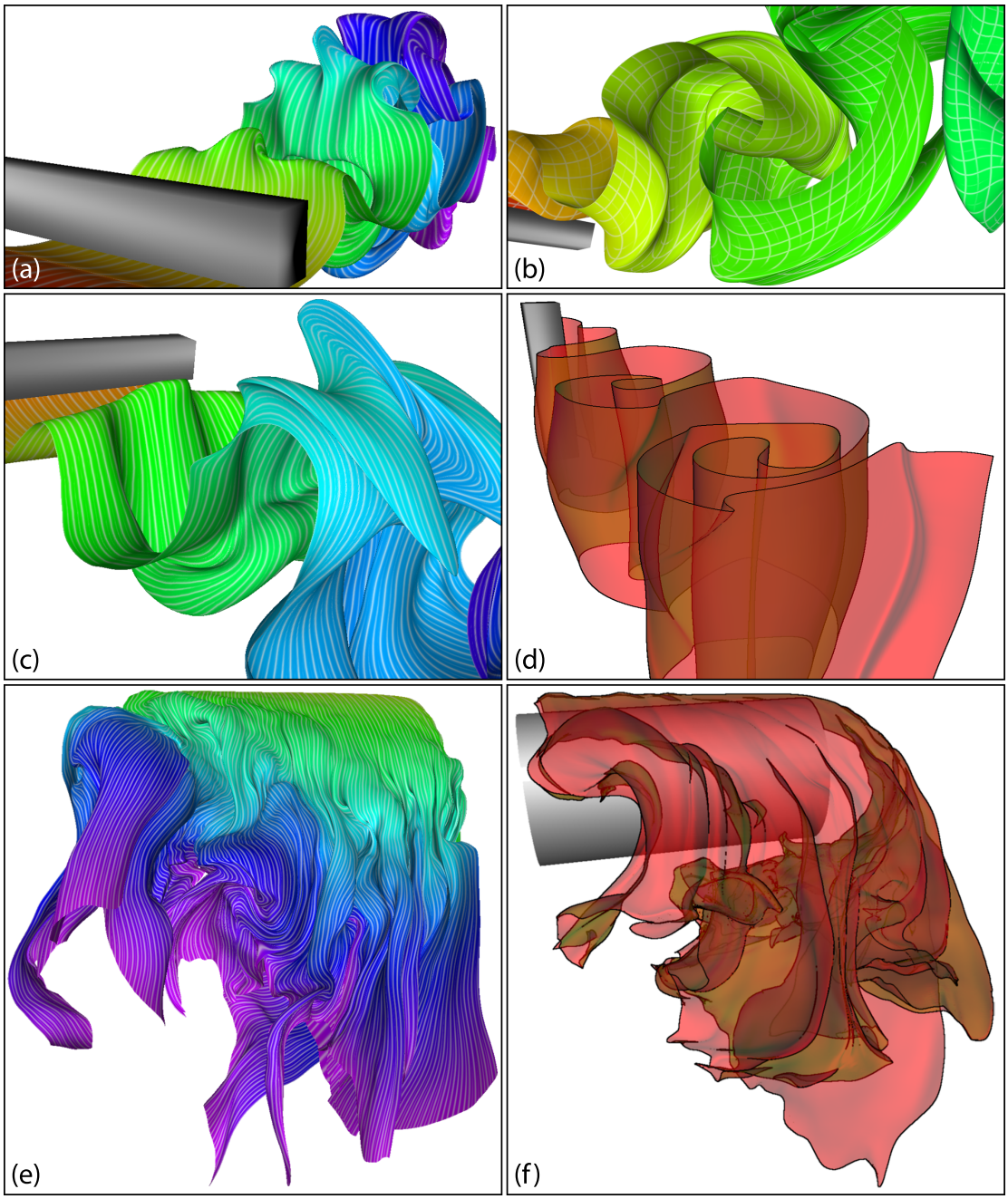


Figure 6.11: *Mesh-based streak surface visualization: Images (a–d) show streak surfaces in the square cylinder data set. Images (e–f) depict streak surfaces in the large eddy data set. The surface parametrization, consisting of time line and particle ids, is used to color the surfaces with stream lines (a,c,e). Image (b) additionally emphasizes time lines. Images (d) and (f) depict semi transparent streak surfaces. Here, depth peeling was employed to extract multiple self-occluding layers and image-based edge detection was used to amplify sharp surface features.*

6.5.3 GPU Implementation

During mesh-based streak surface generation, analogously to the particles, each time line gets assigned a unique id and a counter indicating its refinement depth. For a time line released at time $t_i = t_0 + i\,\Delta t$ the id is set to 0 and incremented by 2^d in each time step. New time lines that are added due to a refinement event adapt this key in the same way as it was described for particles before. This key is then used by the particles on each time line to index into a 1D array that stores time line specific information. This array has as many entries as there can be time lines, and it is realized as 2D texture on the GPU to avoid texture resolution limits. Figure 6.12 shows the content of this array for a set of time lines before (a) and after one integration step (b).

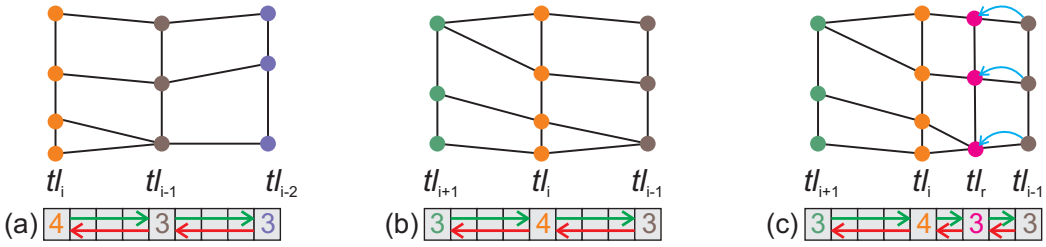


Figure 6.12: *Three time lines of nine possible time lines exist. The number of vertices on each time line is stored in corresponding entries in a 1D array. Red/green arrows indicate the offsets every time line stores to its neighbors in the array. (a) Array indices before and (b) after one integration step. (c) Offsets to adjacent time lines change due to streak line refinement.*

Furthermore, each particle carries two additional offsets, which are used in combination with the time line id to determine the id of adjacent time lines. These offsets are initialized with 2^d and changed accordingly whenever streak line refinement adds/removes an adjacent time line. Figure 6.12 (c) depicts the change of offsets due to the refinement of time line tl_{i-1}.

Parallel to the stream output buffer update during time line refinement (as described in Section 6.5.1), we bind a texture target to the rendering pipeline and rasterize each particle as a point primitive into the texel indexed by the respective time line id (this concept was introduced in Section 4.5.2). By using additive blending, the number of particles residing on each time line is obtained and can be accessed by the particles during the connectivity update and streak line refinement passes.

In the connectivity update pass, every particle writes to a second array its absolute position in the vertex array buffer in the same way. By using a maximum blend operator, the second array contains for each time line the absolute vertex buffer position of the last particle on the respective time line. These values are needed in the streak line

refinement pass to append all particles on a new time line as contiguous block to the vertex array buffer. Additionally, for each particle its distance to neighboring time lines is computed during the connectivity update, and the maximum distance to each adjacent time line is stored separately in an additional texture target. These values are then used during streak line refinement to evaluate the refinement criterion.

To find successors for particles on tl_i, the connectivity pass has to search in an interval containing as much elements as there are on tl_i and tl_{i+1} because the absolute position of a particle in its respective memory block b_i is not yet known. The tuple of time line and particle ids forms a strictly monotonic increasing key over the whole vertex array buffer that is used in a binary search in the interval to the left of a particle to find its successor. The predecessor is determined analogously.

In the streak line refinement pass, new time lines are appended as contiguous blocks to the vertex array buffer. Each particle on a time line tl_i that triggered a streak line refinement decides on the basis of its absolute position in the memory block b_i whether it should contribute two particles to the new time line or account for two particles of tl_i.

During both refinement passes, we do not remove neighboring particles/time lines at once. If multiple adjacent particles satisfy the coarsening criterion in the time line refinement pass, we remove only every second particle. The decision which particle will be removed is based on a modulo criterion applied to the tuple of particle id and depth counter. Analogously we do not remove adjacent time lines at once during the streak line refinement pass. Since the time line refinement technique is akin to the construction of a binary tree for each initially seeded particle and streak line refinement akin to spanning a binary tree of time lines between successively released particle fronts, this coarsening constraint corresponds to a decomposition of respective trees in reverse construction order. Initially released particles (and therefore time lines) are excluded from coarsening events to keep a minimum candidate set for future refinement (as both refinement strategies are restricted by the maximum refinement depth).

6.6 Results and Discussion

Performance tests were carried out on a 2.66 GHz Core 2 Duo processor, equipped with a NVIDIA GTX280 graphics card with 1024 MB local video memory. Results were rendered to a 2560×1600 viewport. In all of our experiments an explicit fourth-order Runge-Kutta scheme at single floating point precision was used for numerical particle integration. Detailed timings for interactive streak surface construction and rendering are given in the following.

6.6.1 Performance

Representative timings in milliseconds (ms) for integration, adaptive refinement and rendering using the patch-based approach are listed in Table 6.1. Values in the first three columns show the number of patches *n*, the maximum particle lifetime *m*, and the refinement depth *d*. The values in column labeled *Pts* show the average number of surface patches. Column *Int* contains timings for integration and refinement, *Vis* for the rendering of the resulting surface, and column *Ttl* the total amount of time required for the construction of the adaptively refined streak surface and subsequent rendering. Let us note that some of the presented settings require buffers larger than the available GPU memory. In these cases, we used static buffer sizes independent of the chosen parameters and list only timings in our performance measures, where no buffer overflow occurred.

n	m	d	Pts	Int	Vis	Ttl
50	500	4	40k	1.3	5.0	7.5
50	500	8	55k	1.8	6.6	9.9
100	1000	4	128k	3.6	5.4	10.5
100	1000	8	167k	4.7	7.0	13.5
200	1000	4	365k	9.4	9.9	20.6
200	1000	8	545k	13.9	14.6	29.9
400	1000	4	1.28M	28.5	30.0	59.7
400	1000	8	2.08M	48.8	49.8	99.8

Table 6.1: *Performance statistics for the patch-based streak surface generation and rendering. Timing statistics in milliseconds are listed in columns 5-7. Even for more than one million surface patches the streak surface construction and rendering took less than 60 milliseconds.*

Timing statistics for the mesh-based streak surface generation and rendering are given in Table 6.2. The maximum depth for both refinement strategies were equally set to *d*. Values in the column labeled *Pts* contain the number of surface particles, column *Int* and *Con* show the times that were required for particle integration including time line refinement and the connectivity update, respectively. Column *Slr* gives timings for streak line refinement and column *Vis* gives the time required for surface triangulation and rendering. Finally, column *Ttl* shows the total time required for the construction and rendering of the adaptively refined triangular mesh.

6.6.2 Quality Comparison

To compare the visual quality, we have used both approaches to generate the same streak surfaces at comparable sample densities. As shown in Figure 6.13, the patch-based approach suffers from artifacts that are common to point-splatting approaches.

n	m	d	Pts	Int	Con	Slr	Vis	Ttl
30	500	4	49k	2.4	1.1	0.9	2.2	8.1
30	500	8	64k	3.1	1.4	1.0	2.8	9.5
50	500	4	116k	5.2	2.3	1.8	4.6	14.8
50	500	8	188k	8.0	3.8	2.5	7.3	22.3
100	1000	4	295k	12.0	5.8	3.8	11.2	34.2
100	1000	8	351k	14.1	7.1	4.4	13.3	39.8
200	1000	4	952k	36.7	20.3	11.3	35.5	105.0
200	1000	8	1.18M	46.0	25.7	14.1	44.7	132.2

Table 6.2: *Performance statistics for the mesh-based streak surface generation and rendering. Columns 5-9 present timings in milliseconds. The construction and rendering of a mesh-based streak surface consisting of more than 350K particles took less than 40 milliseconds.*

In particular, the patch alignment in regions of high curvature tends to produce rather rough surface structures. While increasing the patch areas can cure those artifacts, it tampers with the actual extracted streak surface and requires to increase the bias of the attribute pass. This, however, in turn leads to the accumulation of incoherent surface parts. In addition, blending of overlapping patch attributes tends to blur high frequent surface features. The mesh based approach, on the other hand, avoids all these problems and delivers a closed surface representation that can be rendered using standard polygon rasterization. Sharp features and high frequent geometric details are preserved and the interpolation of vertex normals results in a smooth illumination.

To achieve comparable quality, the patch-based approach requires a significantly

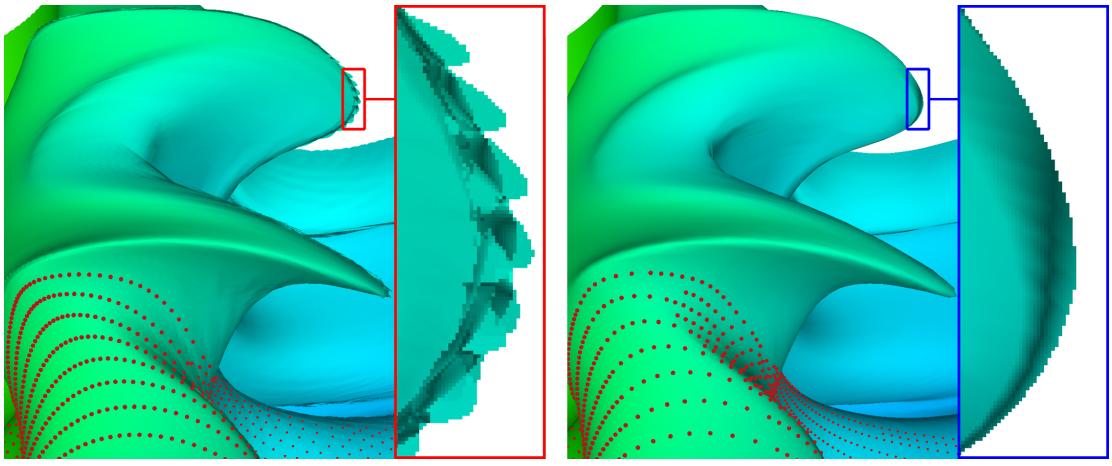


Figure 6.13: *This image shows the same streak surface that was generated using the patch-based (left) and the mesh-based (right) approaches at comparable sample density. While patch-based splatting results in artifacts and blurring at fine surface details and silhouettes, the mesh-based approach yields a high-quality surface representation.*

higher sampling density. The following plot shows the sample density of both approaches, extracting streak surfaces at comparable visual quality.

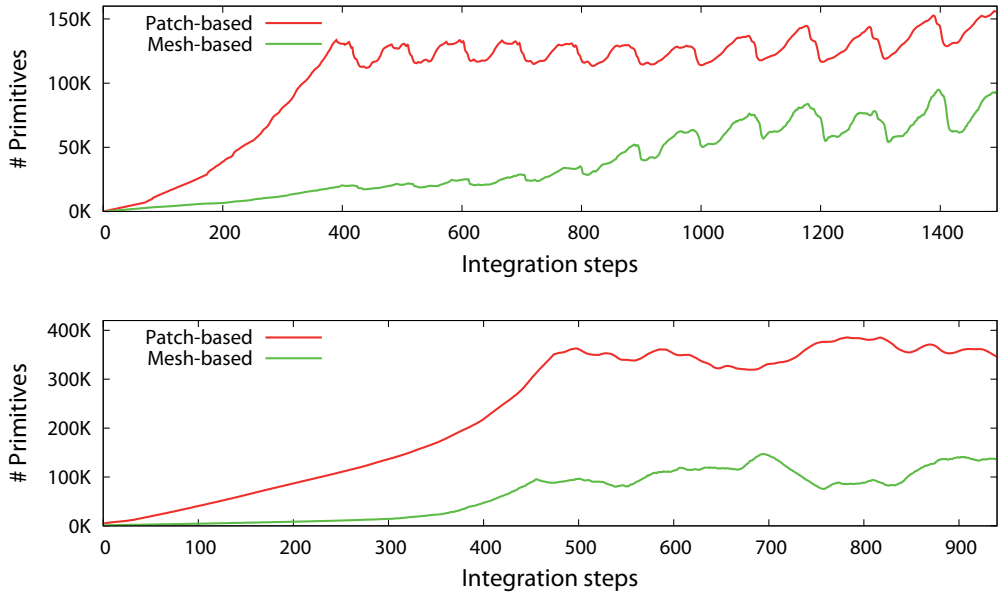


Figure 6.14: *The plots show the sample density of both approaches during streak surfaces generation at comparable visual quality. Top: Statistics for the square cylinder data set. Bottom: Statistics for the large eddy simulation (round cylinder) data set.*

6.7 Summary

In this chapter, we have presented two real-time techniques for constructing and rendering adaptively refined streak surfaces on the GPU. The patch-based approach performs particle integration and adaptive refinement in one step. In the proposed setup we tried to minimize additional complexity regarding the refinement criterion, integration expense and the maximum output performed by the geometry shader, resulting in real time performance even for huge amounts of patches traced in parallel. We also presented visualization methods for this representation by adapting point-splatting techniques to render the loose patch set as closed surface.

The mesh-based approach addresses the increased integration expense by introducing connectivity information between the surface samples. This does not only remove redundant particle integration but also allows the application of more sophisticated adaption criteria as well as coarsening the particle set during surface construction. On that account, the mesh-based approach delivers visually comparable streak surfaces to

the patch-based approach with a much smaller set of surface samples. Furthermore, the closed surface representation can be rendered outright and a multitude of rendering styles can be applied efficiently.

We are aware of the fact that the current triangulation can lead to distorted triangles in highly diverging flow regions or areas of high shear strain between adjacent time lines. Thus, we will investigate alternative triangulation methods in the near future.

Chapter 7

Interactive Separating Streak Surfaces

Streak surfaces are among the most important features to support 3D unsteady flow exploration, but they are also among the computationally most demanding. Furthermore, to enable a feature driven analysis of the flow, one is mainly interested in streak surfaces that show separation profiles and, thus, detect unstable manifolds in the flow. The computation of such separation surfaces requires to place seeding structures at the separation locations and to let the structures move correspondingly to these locations in the unsteady flow. Since only little knowledge exists about the time evolution of separating streak surfaces, at this time, an automated exploration of 3D unsteady flows using such surfaces is not feasible. Therefore, in this chapter we present an interactive approach for the visual analysis of separating streak surfaces. Our method draws upon recent work on the extraction of Lagrangian coherent structures (LCS) and the real-time visualization of streak surfaces on the GPU. We propose an interactive technique for computing ridges in the finite-time Lyapunov exponent (FTLE) field at each time step, and we use these ridges as seeding structures to track streak surfaces in the time-varying flow. By showing separation surfaces in combination with particle trajectories, and by letting the user interactively change seeding parameters such as particle density and position, visually guided exploration of separation profiles in 3D is provided. To the best of our knowledge, this is the first time that the reconstruction and display of semantic separable surfaces in 3D unsteady flows can be performed at interactive rates, giving rise to new possibilities for gaining insight into complex 3D flow phenomena.

7.1 Introduction

For the visual analysis of flow data, feature extraction methods are a well-established class of techniques because the extraction of features offers insight into different flow

phenomena while reducing the amount of data to be processed. Another important and well-established class of visualization algorithms are real-time interactive exploration approaches, such as interactively seeding and tracking particles, characteristic lines, or integral surfaces. The increasing amount and complexity of flow data brings limitations to both classes of techniques: the features themselves may become so complex that their visual representation becomes challenging. On the other hand, interactive exploration techniques suffer from the danger that important phenomena are missed because certain areas are not explored. A solution for this is a combination of feature extraction and interactive exploration: either the complex features are extracted and visualized by appropriate real-time flow exploration tools, or the seeding in interactive flow exploration is controlled by a feature extraction approach. In this chapter, we propose such a combination for particular features (LCS, i.e. ridges of FTLE fields) and interactive exploration tools (generalized streak surfaces).

Ridges of finite-time Lyapunov exponent fields are well-established features for computing separating structures in time-dependent flows (see Section 2.5 for a thorough introduction to this subject). While their definition is well-understood, for 3D time-dependent flows their visualization is complicated because the ridges of interest are 3D hypervolumes in the 4D space-time domain, i.e., surface structures changing their shapes and appearance over time. Because of this, existing algorithms in 3D carefully focus on particular times and locations to show ridge surfaces, making a systematic exploration of all FTLE ridges in 3D time-dependent flows a time-consuming process.

Streak surface extraction is a prominent tool for interactive flow exploration. Since for every location in the space-time domain there is a one-parametric family of streak lines passing through, the sheer amount of existing streak lines (and therefore streak surfaces as well) leave the chance of missing interesting and important streak surfaces.

The approach presented in the following aims to overcome the drawbacks of both FTLE ridges and interactive streak surface exploration. It is justified by the following observation: FTLE ridges are approximately material structures [56, 153] and can therefore be interpreted as generalized streak surfaces.

We use this for the following algorithm: given a 3D unsteady flow field, we interactively place and move a planar seeding structure $\mathbf{s}$ (usually a rectangle) in the flow domain at a certain time t. Note that moving the seeding structure $\mathbf{s}$ is possible both in space and time. We consider the restriction of the FTLE field on $\mathbf{s}$ (i.e., a 2D field), either by computing the FTLE values on $\mathbf{s}$ in-turn or by computing the entire FTLE field (i.e., a 4D scalar field) in a preprocess and resampling the values onto $\mathbf{s}$ via interpolation. We then extract ridge structures in the FTLE field on $\mathbf{s}$ in real-time, and

employ them as seeding structures for a streak surface integration. Since the ridges on **s** change their shape by moving **s** in space and/or time, the surfaces generated this way are generalized streak surfaces (an extension of the concept of generalized streak lines [187]). The streak surfaces are shown only for the integration time which was used for computing the FTLE, since only for this integration time a separation was detected. As our choice of seeding locations for streak surface integration aims to uncover separation structures, we will refer to them as separating streak surfaces in the following.

Our method exploits the fact that 2D FTLE ridges (LCS) in 3-space are advected in a similar way to streak surfaces seeded at 1D FTLE ridges on a 2D manifold in 3-space. This statement is based on two facts: firstly, the 1D FTLE ridges on the seeding plane are approximately on 2D ridges in 3-space as long as the seeding plane is approximately perpendicular to the flow (this was exploited in [47], where ridges on cutting planes are considered instead of 2D ridges). Secondly, FTLE ridges are approximately material structures and do in fact converge to exact material structures if the integration time goes to infinity [153].

In [139] and [105] this temporal coherence of LCS was exploited to efficiently compute time series of FTLE ridges via simultaneous advection of a sampling grid and incremental 1D ridge tracking, respectively. Since a finite integration time is used in our work, the generalized streak surfaces we extract do not coincide with 2D FTLE ridges in general. However, we will demonstrate in this work that these surfaces resemble the 2D ridges at high fidelity and can be computed very efficiently. Furthermore, since generalized streak surfaces move according to the flow they provide a more intuitive flow exploration metaphor than 2D FTLE ridges. Notably, no visual information will be generated in regions where many 2D ridges exist but for none of them an approximating streak surface has its origin on the selected seeding structure. This allows using our approach as an effective technique to focus on particular flow structures in space and time.

7.2 Contribution

In this chapter, we present the first approach to construct separating streak surfaces in 3D unsteady flows at interactive rates. This enables visually guided 3D flow exploration based on the concept of LCS. Our approach distinguishes from previous approaches in that it avoids computing LCS in 3D. Instead, the computation is restricted to a 2D manifold and streak surfaces are constructed at significantly less computational effort. All processing stages of the proposed algorithm are realized on the GPU, including FTLE

computation, ridge extraction, streak surface reconstruction, and surface rendering. The specific contributions of our work are:

- A navigation tool that allows placing a 2D sampling grid in space-time and computing FTLE values on it in an interactive way.
- A new ridge extraction method that is specifically tailored to the GPU and produces ridges well-suited as seeding structures.
- An extension to the patch based streak surface technique presented in Chapter 6 to reconstruct generalized streak surfaces emanating from 1D FTLE ridges.

The remainder of this chapter is organized as follows: After reviewing previous work that is related to ours, Section 7.4 is dedicated to the spatial selection of separating streak surfaces based on the FTLE. In Section 7.5 we introduce our new ridge extraction algorithm. The reconstruction of streak surfaces from extracted 1D FTLE ridges is described in Section 7.6. Section 7.7 presents a detailed analysis of our approach with respect to performance and quality. We conclude this chapter with a brief summary and an outline of future research in the field.

7.3 Related Work

Our approach is based on a number of established techniques in visualization, namely FTLE extraction, ridge extraction, streak surface integration, and interactive flow exploration. A thorough overview of feature extraction techniques in flow visualization and geometric flow visualization techniques can be found in [130] and [115], respectively.

FTLE / LCS

Lagrangian coherent structures (LCS) as ridges of FTLE fields were introduced by Haller [56, 58] and experienced an intensive research since then [100, 57, 154, 185]. Shadden [153] has shown that ridges of FTLE are approximate material structures, i.e., they converge to material structures for increasing integration times. This fact was used in [140] to extract topology-like structures and in [105] and [137, 47] to accelerate the FTLE computation in 2D and 3D flows.

In the visualization community, different approaches have been proposed to increase performance, accuracy and usefulness of FTLE as a visualization tool. For example, volume rendering and slicing techniques were used for a visual analysis of 3D FTLE

fields in [49, 47], [138] proposed to extract LCS as filtered height ridges and compared LCS- and topology-based flow visualizations. LCS extraction in 2D FTLE fields was discussed in [137], and [24, 47] considered the FTLE to control the visualization of particles to show divergent regions in 3D flows. However, none of them is designed for a real-time exploration of the separating structures in 3D space and time.

Ridge Extraction

To extract ridge structures, a variety of different approaches has been proposed in the literature. We mention local conditions by relaxing conditions of extremal structures [37, 104], topological/watershed approaches [141], definitions based on extremal curvature structures, adaptive methods [137], or particle based methods [77]. [126, 151] focus on the extraction of ridge surfaces in 3D fields. To the best of our knowledge, none of these approaches aims at a real-time extraction of ridge structures in time-varying fields.

7.4 FTLE

Our approach for visualizing separating streak surfaces is based on seeding particles along Lagrangian coherent structures in a 3D unsteady flow field. Since LCS are formed by ridges in the finite-time Lyapunov exponent field, the FTLE first has to be computed before meaningful seeding structures can be found. The FTLE is a scalar quantity that measures the stretching induced by the flow. Let $\phi_{t_0}^{t_0+\Delta t}(x)$ denote the flow map that defines the mapping of particles at position $\mathbf{x}$ in space and t_0 in time via path line integration over the time interval $t_0 + \Delta t$. According to [56] the FTLE is then defined as

$$\sigma_{t_0}^{\Delta t}(\mathbf{x}) = \frac{1}{|\Delta t|} \ln \sqrt{\lambda_{max} \left((\nabla \phi_{t_0}^{t_0+\Delta t}(\mathbf{x}))^{\mathrm{T}} \nabla \phi_{t_0}^{t_0+\Delta t}(\mathbf{x}) \right)},$$

where λ_{max} is the largest eigenvalue of the right Cauchy-Green deformation tensor $(\nabla \phi_{t_0}^{t_0+\Delta t}(\mathbf{x}))^{\mathrm{T}} \nabla \phi_{t_0}^{t_0+\Delta t}(\mathbf{x})$ of the flow map. For a thorough introduction to the concepts of the FTLE and a mathematical derivation, we refer the reader to Section 2.5.3.

In the following, we will assume that the flow map—and the FTLE derived thereof—is computed by sampling particles on a planar seeding structure $\mathbf{s}$, which is discretized by a uniform 2D sampling grid. For estimating the flow deformation in the vicinity of one of these particles, however, we consider additional particles that are seeded within an ε-region around it. The value this variable takes can be adjusted by the user, mak-

ing the FTLE computation independent of the chosen grid spacing. However, following [73], we have chosen ε according to the grid spacing on the planar sampling grid in all of our examples.

The FTLE computation, and thus the following ridge extraction, is restricted to a sub-domain of the 3D flow domain. The user is provided a navigation tool to place $\mathbf{s}$ in the 3D field. Both the size and the resolution of the sampling grid can be set by the user. At the center of each grid cell the FTLE value is computed as described above. Specifically, if a center is at position (x,y,z), the trajectories of 6 particles seeded at positions $(x\pm\varepsilon,y\pm\varepsilon,z\pm\varepsilon)$ are traced and the deformation gradient is computed from the particle destinations. Particle tracing and FTLE computation is entirely performed on the GPU, and the resulting values are written into a 2D texture.

It should be noted that the FTLE computation we perform can generate less reliable results, since the particles can separate significantly during path line integration. Even though there exist approaches to overcome this problem, e.g. by a FTLE redefinition to local criteria on the center trajectory [73] or renormalization of the particle neighborhood [9], we have not yet integrated these approaches into our method.

Figure 7.1 shows two snapshots of an exploration session in which FTLE values have been computed on different sampling grids. In both cases the computation was performed at a grid size of 256×256 with an integration time of 0.15 s (100 Runge-Kutta 4th order integration steps, requiring 20 time steps of the unsteady flow field). Since the computation is performed on the GPU, interactive update rates of less than 150 ms are achieved as long as all time steps necessary to calculate the FTLE at a given point in time can be stored in the GPU memory.

This indicates that often it is not necessary to pre-compute a time-resolved 3D FTLE field sequence prior to the flow exploration. The values can be updated in-turn once the user moves $\mathbf{s}$ or changes any of the parameters the FTLE depends upon, like the start time t_0, the integration time Δt, the spatial sampling distance ε, or the size and resolution of the sampling grid. However, in scenarios where the time-resolved flow field sequence does not fit into GPU memory (i.e. all time-steps needed to calculate the FTLE at a given point in time), pre-computing the time-dependent FTLE should be preferred. In this case the pre-computed FTLE values can simply be interpolated at the grid cell centers. Note that in this case the FTLE parameters are fixed and, therefore, ε might significantly differ from the grid spacing on $\mathbf{s}$.

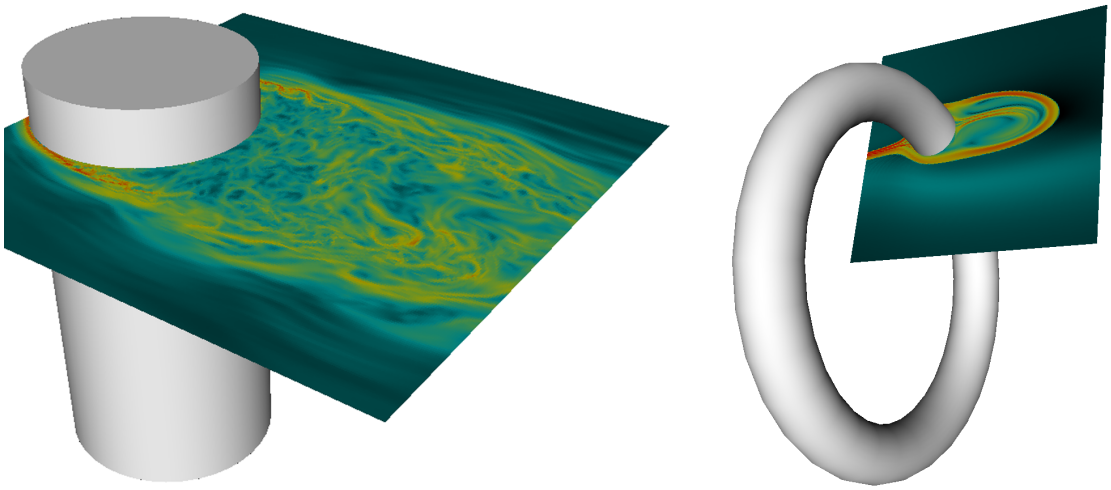


Figure 7.1: *Two FTLE fields on a planar probe at grid size* 256×256*. The parallelized FTLE computation on the GPU yields interactive update rates (less than* 150 *ms for the given probe texture resolution).*

7.5 FTLE Ridge Extraction

In the following we describe our novel extraction technique for 1D ridges in 2D FTLE fields. Ridge extraction techniques—also in the context of LCS extraction—have been studied extensively over the last years. For an introduction to this field and more information on related work, we refer the reader to Section 2.5.

Our ridge extraction technique builds upon the concepts of *height ridges* and *watersheds*. The definition of height ridges involves point-wise evaluations of algebraic equations based on geometric ridge properties, which are expressed via first-order derivatives and derivatives into the main curvature directions, i.e., the (transversal) ridge directions [37]. Let $f(\mathbf{p})$ denote the FTLE value at a point $\mathbf{p}$ on $\mathbf{s}$, and let $\mathbf{H}$ and $\mathbf{g}$ denote the Hessian matrix and the gradient of f, respectively. According to [126] the height ridges are a subset of the zero-contour of $\det(\mathbf{Hg}, \mathbf{g}) = 0$, which can be extracted in 2D using the marching squares algorithm.

In general, *unfiltered* height ridges do not provide suitable seeding regions for streak surfaces. Even though height ridges cannot really branch as shown in [151], they tend to appear as branched structures at larger scales. Highly branched and fragmented structures, however, result in many separate and even non-manifold surface parts. From a visualization point of view the streak surfaces constructed from such ridges do not allow any intuitive interpretation due to their complex topology and visual clutter thereof. Figure 7.2 shows a set of unfiltered FTLE height ridges (left) and compares them to the

ridges we are interested in (right). Even though height ridges can be post-processed to eliminate undesirable ridge parts, and thus to yield simpler seeding structures [137], it seems to be difficult to implement this process efficiently on the GPU.

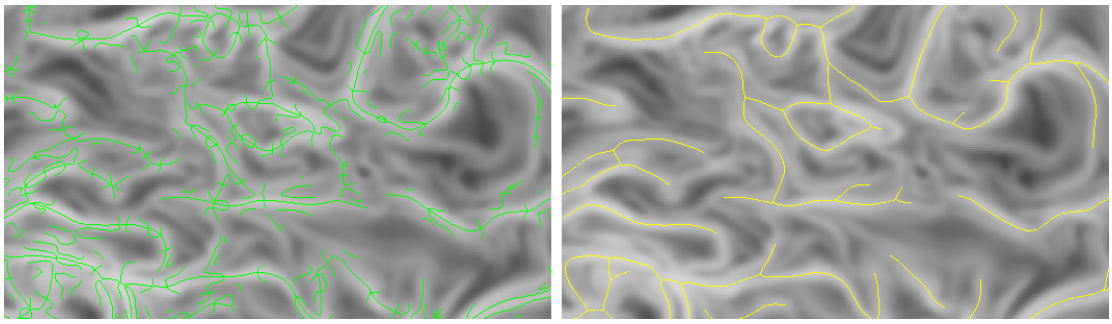


Figure 7.2: *Left: unfiltered height ridges; Right: ridges extracted by our approach. Grey scale values in the background correspond to FTLE values extracted at the sampling grid resolution. Let us note, that ridge extraction was performed at a larger scale space level.*

Watersheds [134] are another popular approach for ridge extraction in 2D. It is based on the topology of the underlying 2D scalar field and aims at extracting slopelines separating hills and basins. In this definition a ridge is considered as a slopeline going from one maximum to another maximum through a single saddle point. Even though the topology of watershed ridges is often much simpler than that of height ridges, they nevertheless fail to focus on the main axis of hills of the height field. Using a general watershed approach can also lead to rather cumbersome special cases in which significant ridges are missed because they do not separate different minima correctly.

To overcome the limitations of height ridges and watersheds we introduce a novel ridge extraction algorithm. Generally speaking, a ridge is a graph $G = (V, E)$ consisting of a set V of vertices and a set E of edges. Vertices $\mathbf{v} \in V$ can be end points ($deg(\mathbf{v}) = 1$), line points ($deg(\mathbf{v}) = 2$) or crossings ($deg(\mathbf{v}) > 2$). With respect to this definition, for our purpose the specific goals are a) to minimize the number of crossings per ridge, and thus to avoid non-manifold surfaces, and b) to maximize the ridge length, i.e. to connect as many vertices as possible, and thus to prevent the streak surfaces from falling into many parts.

The basic idea underlying our algorithm is to separate the extraction of the ridge topology from the computation of the exact ridge locations, similar to the concept proposed in [146]. Starting with the FTLE field in a 2D texture in GPU memory, the texture is first filtered via a gaussian kernel of size 5×5 to smooth high-frequency FTLE regions. The amount of smoothing operations depend on a user selected scale space level (typically, $5 - 10$ smoothing iterations are performed). Then, the texture is processed

to classify the FTLE values and build threshold regions. These regions are successively thinned to compute a pixel-accurate ridge skeleton, from which ridge line segments are extracted at sub-pixel accuracy. The different steps of this algorithm are illustrated in Figure 7.3.

The result of our technique are continuous ridges with a simple topology. They consist of points that are local maxima into the direction of the local ridge normals, similar to the concept of watersheds. These ridges are returned as a common graph structure G with uniform vertex spacing.

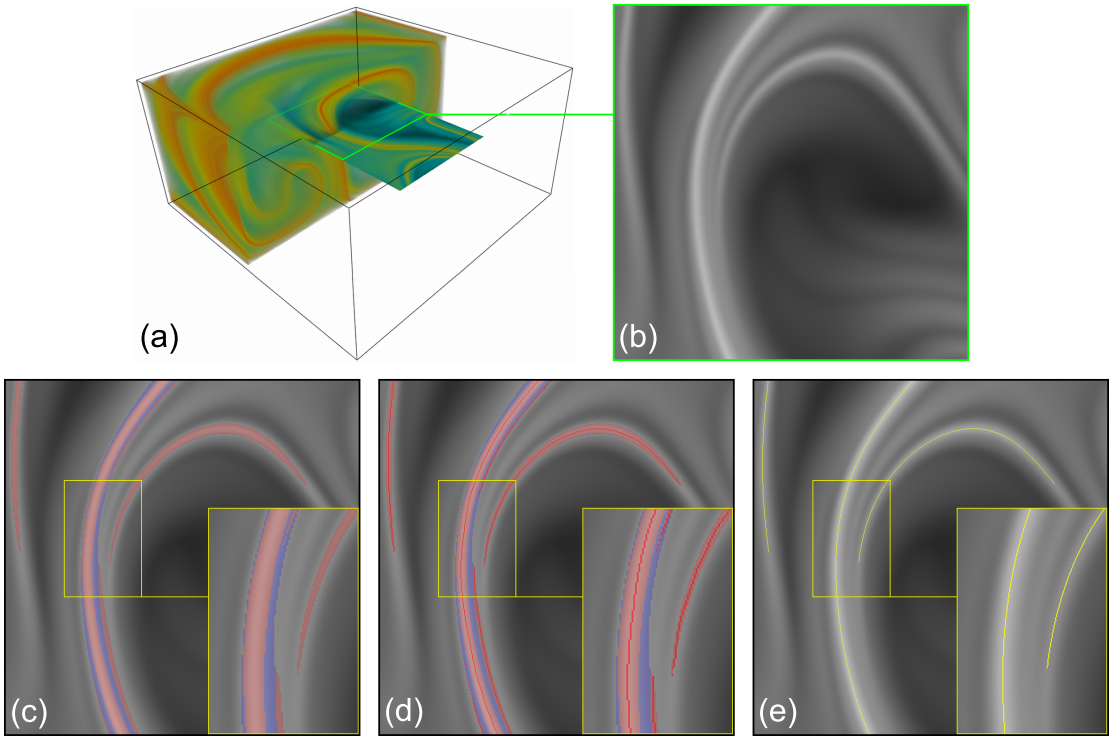


Figure 7.3: *Steps of the ridge extraction algorithm. (a) A planar probe positioned in the flow domain, and the corresponding color coded FTLE scalar field. Image (b) shows a cutout of the FTLE on the planar probe. (c) Threshold regions. (d) Thinning yields the pixel-accurate ridge skeleton. (e) Extracted ridge line segments at sub-pixel accuracy.*

7.5.1 Ridge Topology

The extraction of the ridge topology is performed by first classifying the discrete set of FTLE values on the sampling grid based on the height and the local curvature of this field, and then by shrinking the resulting regions towards the ridge skeletons. If a sufficient symmetry of the hills in the height field along their main axis can be assumed, the skeleton will roughly coincide with valid ridge locations.

Classification

Performing the classification of FTLE values based on a height threshold is not sufficient in general, since ridges can be of different heights. This classification also fails to segment regions of nearby ridges that are separated by valleys of insufficient depth. To solve this problem, we introduce an additional threshold that is used to separate convex and non-convex FTLE regions.

Let $f_{\mathbf{ww}}$ be the second order directional derivative of the gradient $f(\mathbf{p})$ into direction $\mathbf{w}$ at a fixed position $\mathbf{p}$. Moreover let λ_1, λ_2 (with $\lambda_1 \leq \lambda_2$) be the eigenvalues of $\mathbf{H}$ at $\mathbf{p}$. The point $\mathbf{p}$ is a convex point if every second order directional derivative is non-positive: $\forall \mathbf{w} \neq 0 : f_{\mathbf{ww}} \leq 0$. Since $\lambda_1 \leq f_{\mathbf{ww}} \leq \lambda_2$ holds for any arbitrary $|\mathbf{w}| = 1$, $\mathbf{p}$ is convex if and only if $\mathbf{H}$ is negative semi-definite. This results in the following condition to be fulfilled by every ridge point:

$$\lambda_2 \leq 0$$

Applying this criterion in a pixel shader to every sample on the seeding structure $\mathbf{s}$ gives the desired classification into points belonging to convex regions and points belonging to non-convex regions. For this we calculate the Hessian pixel-wise using discrete filters on the smoothed FTLE field f on $\mathbf{s}$ and store the classification in an additional 2D texture at the sampling resolution on $\mathbf{s}$.

The used criterion, on the other hand, is rather sensitive against small but random fluctuations and, thus, leads to a rather noisy classification in approximately planar regions. This misclassification, which results in unfeasible skeletons, is resolved by allowing small positive values of λ_2, i.e., a curvature threshold $\kappa > 0$. Combined with a height threshold h to exclude ridges at locations where the FTLE value is too low we arrive at the condition

$$\lambda_2 \leq \kappa \quad \wedge \quad f(\mathbf{p}) \geq h \, . \tag{7.1}$$

We are aware that in the context of height ridges thresholding λ_1 would have been a more natural choice, since their definition does not restrict λ_2 which corresponds to the actual ridge direction. Practice however has shown, that in contrast to a λ_1-test the application of the more restrictive λ_2-test reasonably reduces branching and therefore ridge complexity.

Figure 7.4 shows FTLE classifications using the different criteria with varying threshold values. As can be seen, vastly different results are obtained, ranging from rather fuzzy to well-defined and smooth threshold regions. According to our experience, choosing κ one or two orders smaller than the largest occurring curvatures on $\mathbf{s}$ pro-

vides the best results, i.e. $\kappa \in [0.01; 0.1] \cdot \max_{\mathbf{s}}\{|\lambda_1|, |\lambda_2|\}$. For the minimal ridge height h, reasonable values are between 50% and 80% of the maximum FTLE value.

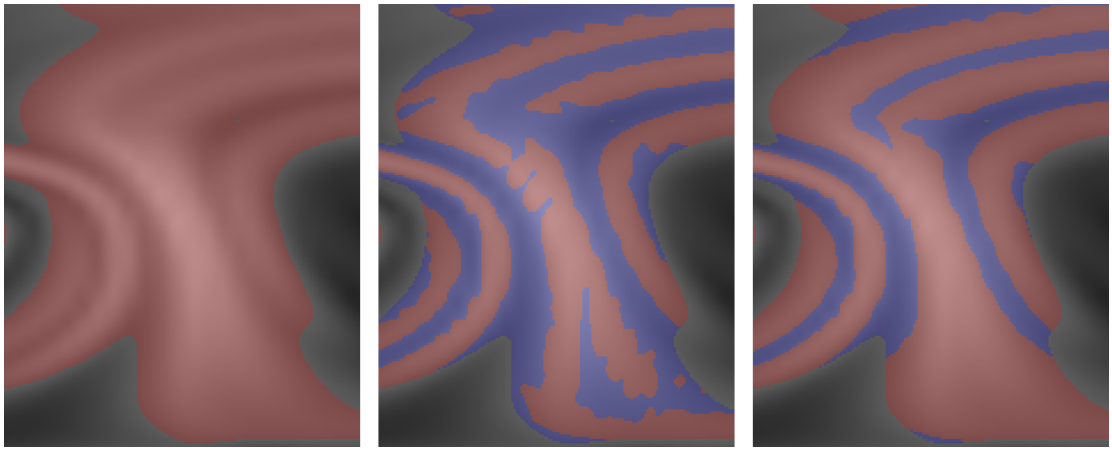


Figure 7.4: *FTLE classification into convex (red) and non-convex (blue) regions, using height threshold (left) , height and curvature threshold with $\kappa = 0$ (middle) and with $\kappa = 10^{-3}$ (right).*

Skeletonization via Curve-Thinning

Applying the convexity test (7.1) to the FTLE field results in a binary threshold image. We assume that pixels labeled 1 passed the test, while all others are labeled 0. We are now seeking for the topological skeletons of those regions consisting of pixels that passed the test, i.e., the skeletons of the convex regions.

To compute these skeletons efficiently on the GPU we employ a 2D version of the region thinning algorithm proposed by [122]. The algorithm is very robust against noise at the region boundaries, and since it performs purely local computations at every pixel it can be parallelized effectively. Furthermore, it directly generates the inner skeleton of a region, meaning that the algorithm avoids branches touching the region contour. At every pixel the algorithm considers the 4-neighborhood to classify this pixel, i.e., a pixel is classified as N- (or W-, E-, S-) border-pixel if it has value 1 and its neighbor in the respective direction has value 0:

	N	
W	$\bullet$	E
	S	

In every iteration, at every pixel four sub-iterations are performed to remove certain border pixels. The first sub-iteration NW removes N- and W- border-pixels that match at least one of the following three adjacency templates:

$$\begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline x & 1 & x \\ \hline x & 1 & x \\ \hline \end{array} \quad \vee \quad \begin{array}{|c|c|c|}\hline 0 & x & x \\ \hline 0 & 1 & 1 \\ \hline 0 & x & x \\ \hline \end{array} \quad \vee \quad \begin{array}{|c|c|c|}\hline 0 & 0 & \cdot \\ \hline 0 & 1 & 1 \\ \hline \cdot & 1 & \cdot \\ \hline \end{array}$$

Here '0' and '1' mark pixels that have to be exactly matched. Of the neighbors marked 'x', per template at least one has to be '1' while those marked with '$\cdot$' are irrelevant for the evaluation of the respective template. Upon finishing this sub-iteration, the algorithm proceeds with sub-iterations SE, NE, and SW in exactly this order. The templates for these sub-iterations are derived by rotating the templates used in the first sub-iteration accordingly. The thinning process is performed in as many iterations as are required until no more pixels are removed from the input image (typically a maximum of 20 iterations is sufficient).

7.5.2 Sub-pixel Ridge Refinement

Given the set of skeleton pixels that is output by the thinning algorithm, we construct a graph representation of the skeletons by connecting neighboring pixels. For every skeleton pixel with at least one neighbor a vertex at its center is created. Two vertices are connected via an edge if they belong to horizontally or vertically adjacent pixels (N,W,S, or E template positions) or if they belong to diagonally adjacent pixels which do not share a common neighbor. Let us note that this implies $deg(\mathbf{v}) \leq 4$ for all ridge vertices $\mathbf{v}$.

The graph is stored on the GPU as a linear array of vertex primitives, each carrying a pixel coordinate $\mathbf{p_v}$ in the sampling grid $\mathbf{s}$ and an adjacency list $U_\mathbf{v} \subset V$ with $1 \leq |U_\mathbf{v}| \leq 4$ implemented as pointers (indices) to up to 4 neighbors. The array is created by invoking a geometry shader for every texel in the 2D texture storing the skeleton classification. Using the stream output functionality of current GPUs, we can generate primitives solely for the pixels who are part of the skeleton. To establish the connectivity between these vertex primitives, in a second rendering pass, we scatter their array indices back into an index texture of the same dimensions as the initial texture. In a third pass every vertex finally determines the connectivity information U by a lookup into the index texture.

The ridge graph is then refined iteratively at sub-pixel precision. Underlying the refinement process is the condition that every ridge vertex $\mathbf{v}$ should be lying on a maximum of the image function f into the direction of the local ridge normal $\mathbf{n_v}$. Consequently, the ridge vertices have to be moved upwards the FTLE field until they reach

such a maximum. Since moving vertices along the image gradient $\mathbf{g}$ would cause the ridge graph G to be heavily distorted or even collapse at the absolute maxima of f, we restrict the movement to the $\mathbf{n_v}$-direction by projecting $\mathbf{g}$ onto $\mathbf{n_v}$. This also ensures the convergence of the refinement process under the assumption of a reasonable initial guess produced by the skeletonization. To avoid degenerate cases and to additionally obtain evenly spaced vertices, we incorporate some smoothing into each refinement step by interpolating vertex positions along ridge lines. Specifically, the position $\mathbf{p_v}$ of a vertex $\mathbf{v}$ is updated according to

$$\mathbf{p_v}' = \begin{cases} \mathbf{p_v} + \delta\ \mathbf{r_v} & \text{, if } |U_\mathbf{v}| = 1 \\ (1-\sigma)\ \mathbf{p_v} + \frac{\sigma}{|U_\mathbf{v}|}\left(\sum\limits_{u \in U_\mathbf{v}} \mathbf{p_u}\right) + \delta\ \mathbf{r_v} & \text{, else} \end{cases} \tag{7.2}$$

with

$$\mathbf{r_v} = \begin{cases} \langle (\mathbf{p_{u_1}} - \mathbf{p_{u_2}})^\perp, \mathbf{g} \rangle \cdot (\mathbf{p_{u_1}} - \mathbf{p_{u_2}})^\perp & \text{, if } U_\mathbf{v} = \{\mathbf{u_1}, \mathbf{u_2}\} \\ \sum\limits_{\mathbf{u} \in U_\mathbf{v}} \langle (\mathbf{p_u} - \mathbf{p_v})^\perp, \mathbf{g} \rangle \cdot (\mathbf{p_u} - \mathbf{p_v})^\perp & \text{, else} \end{cases}$$

Here, δ is the step-size along the gradient, σ is the amount of line smoothing, and $\mathbf{w}^\perp$ denotes a unit-length vector perpendicular to $\mathbf{w}$. In order to allow for the procedure to converge, we choose the largest δ for which the step size $|\delta\ \mathbf{r_v}|$ is smaller than one pixel. σ was set to 0.25 during all our experiments.

We perform a fixed number of iteration steps (typically 50), which are computed for every vertex in parallel on the GPU. In a final post-process the graph G is modified by removing vertices that have moved more than a user-specified distance threshold d_{max} during the refinement (by default we set d_{max} to the length of 5 pixels). In this way, we eliminate skeletons that were too far from ridges after the initial skeletonization. Therefore, each vertex $\mathbf{v}$ gets assigned an additional attribute $d_\mathbf{v}$, which stores the distance a vertex has been moved. Similar to $\mathbf{p_v}$, $d_\mathbf{v}$ is smoothed along ridge lines to prevent oscillation artifacts caused by thresholding:

$$d'_\mathbf{v} = \begin{cases} (1-\omega)\ d_\mathbf{v} + \frac{\omega}{|U_\mathbf{v}|}(\sum\limits_{u \in U_\mathbf{v}} d_\mathbf{u}) + |\mathbf{p_v}' - \mathbf{p_v}| & \text{, if } |U_\mathbf{v}| \leq 2 \\ d_\mathbf{v} + |\mathbf{p_v}' - \mathbf{p_v}| & \text{, else} \end{cases} \tag{7.3}$$

Compared to σ, ω should be chosen significantly smaller. For instance, $\omega = 0.05$ was used throughout all of our experiments. Vertices with $d_v > d_{max}$ are marked invalid. Finally this mark is propagated through G in k_{cut} iterations, marking all vertices invalid from which an invalid vertex can be reached in k_{cut} steps. k_{cut} was set to 5 throughout all

experiments. In Figure 7.5 extracted ridges before (left) and after (right) the sub-pixel refinement stage are shown.

The result of the ridge extraction stage is the array of ridge vertices containing both the invalid and the valid ridge vertices $V^+ \subseteq V$. From these vertices the set of valid edges $E^+ \subseteq V^+ \times V^+$ is derived.

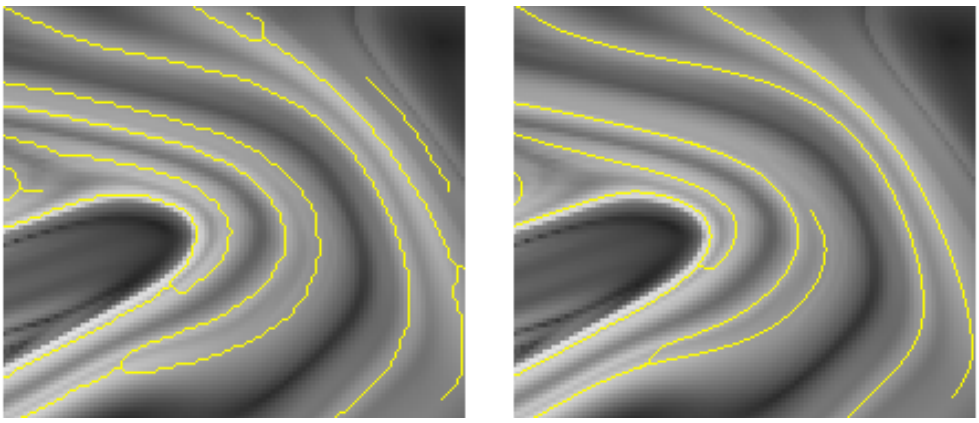


Figure 7.5: *Ridges extracted with our approach. Left: Ridges obtained by connecting adjacent vertices. Right: Sub-pixel precise ridges after the refinement and post-processing stage.*

7.6 Separating Streak-Surface Visualization

Our ridge extraction technique yields a set of ridge structures for a given point in time. These structures are then used as seeding curves for streak surfaces. Since the seeding structures change over time, the surfaces generated this way are generalized streak surfaces.

The ridges are provided as a set of uniformly distributed line segments E^+ consisting of a discrete set of control vertices V^+. To construct separation surfaces, we repeatedly release particles from the set of seed points V^+ into the flow and compute their trajectories in the 3D unsteady flow. All particles released at a given point in time are then integrated and rendered. Since the FTLE values have been computed by integrating particles over a specific time interval, the life time of the particles seeded at the FTLE ridges is restricted to the same interval.

Figures 7.6 and 7.11 (b) show separating streak surfaces that were visualized by rendering the set of particles as individual spherical point sprites. As proposed by Sigg et al. [157], an analytic ray/sphere intersection is performed in the pixel shader stage to determine correct depth values on a per fragment basis. Numerical particle integration on the GPU is performed as described in Section 4.5.2.

Figure 7.6: *Particle based surface visualization. Red particles correspond to points on the separating surface. Green particles serve as context information. They represent points on time surfaces, which are released from the planar probe at a fixed frequency.*

Red particles in these images correspond to control points residing on a separating streak surface. Green particles represent control points of time surfaces. Those time surfaces are aligned parallel to the planar probe and released into the flow at a fixed frequency to serve as additional context information, emphasizing the separating nature of the extracted streak surfaces. Approximating the surface through a set of individual samples allows us to use large sets of particles at real-time performance. However, as particles start to diverge, missing connectivity between surface samples and the omission of an adaptive refinement make it difficult to identify the separating surface.

In Chapter 6 we presented two different approaches for the visualization of closed streak surfaces with the focus on the efficient construction of such surfaces on the GPU. The first approach represents the surface as a set of separate quad-patches, which deform under the influence of the flow. Each patch is traced separately through the flow, and it is adaptively refined into a set of sub-patches if the stretching becomes too severe. The refinement process introduces new vertices that are not shared by adjacent patches, and, thus, successive integration can lead to holes in the surface representation. The second approach generates a closed surface by repeatedly releasing time lines from a single static seeding structure and triangulating adjacent (adaptively refined) time lines.

Unfortunately, the mesh-based approach can not easily be applied in the current scenario since the seeding curve changes permanently. This makes it difficult to establish particle connectivity and to construct a consistent surface triangulation. Especially

since the topology of the seeding curves changes from time step to time step, we would first have to determine matching curve segments in successive time steps to build a triangulation. Finding these matchings is a rather time consuming task and can not efficiently be mapped to the GPU. For this reason we adopt a variant of the patch-based approach presented in Section 6.4.

Interactive unsteady flow exploration using the planar probe metaphor is usually initiated by placing the seeding structure $\mathbf{s}$ at a fixed location in space, but letting it move in time to depict the development of separating surfaces in the evolving flow domain. As the seeding ridge structures change while moving the planar probe in time, we extend the patch based technique in the following way: In each time step t_i and for every edge $e \in E^+$ we construct a zero area quad, and we release two control vertices of each patch into the flow. Before releasing the remaining two vertices in time step t_{i+1}, the ridge line segments E^+ extracted in t_i are traced along the gradient of the 2D FTLE field at t_{i+1} as described in section 7.5.2. Thus, the vertices are moved according to the movement of the ridge structure from one time step to the next. In this way we employ the temporal coherence of FTLE ridges to find for each ridge vertex a corresponding vertex in the next time step. The remaining two patch vertices are then released into the flow from the new positions on the seeding plane. Figure 7.7 sketches the construction of surface patches for a seeding structure that moves over time.

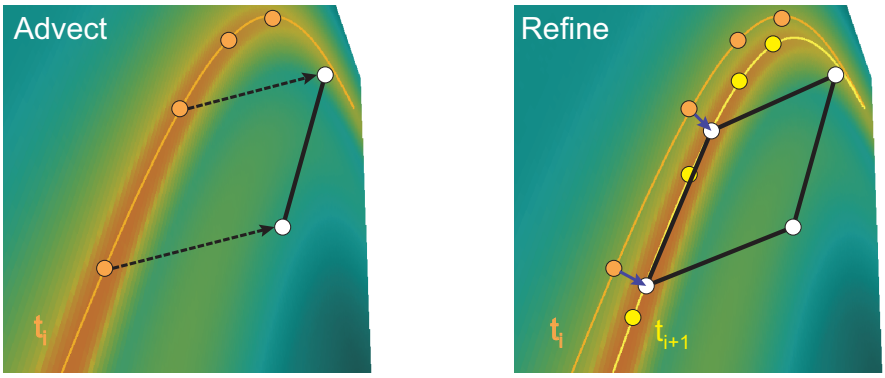


Figure 7.7: *Patch-based surface construction. Before releasing the second pair of vertices at time t_{i+1}, the line segment of the corresponding ridge structure extracted at time t_i is traced along the FTLE gradient field $\mathbf{g}_{i+1}$. The four white vertices (right) depict the control points of the resulting quadrilateral.*

As the adjusted edges E_i^+ (extracted at t_i) do not exactly match the edges E_{i+1}^+, subsequent integration can lead to holes in the surface representation. This is fixed to a certain degree by the proposed point splatting approach, which renders a slightly enlarged footprint to smear out holes between adjacent patches. Adaptive patch re-

finement is performed as described in Section 6.4. Figures 7.8 (a), 7.9, and 7.11 (a) show separating streak surfaces that have been constructed and visualized using our approach. In Figures 7.8 (b, c) and 7.10 depth peeling was applied to create a semi transparent visualization of the separating streak surfaces.

7.7 Results and Discussion

To validate the effectiveness of the proposed techniques, we have conducted a number of experiments on different data sets given on cartesian 3D grids. Performance statistics were measured on a 2.83 GHz Core 2 Quad processor, equipped with a NVIDIA Quadro FX5800 with 4 GB local video memory. Results were rendered into a viewport at FullHD resolution (1920×1080). The following data sets were used:

- *3D double gyre:* A 3D extension of the synthetic, periodic 2D double gyre [153], sampled at a spatial resolution of $256 \times 128 \times 256$ and a temporal resolution of 10 for one period:

$$\begin{aligned}\mathbf{gyre}(x,y,z,t) \; = (&-\pi A \cdot \sin(\pi f(x,t+5z)) \cdot cos(\pi y), \\ &\pi A \cdot \cos(\pi f(x,t+5z)) \cdot sin(\pi y) \cdot \frac{\mathrm{d}f}{\mathrm{d}x},\ 0) \text{ with}\end{aligned}$$

$f(x,t) = a(t)x^2 + b(t)\,x,\ a(t) = \varepsilon sin(\omega t),\ b(t) = 1 - 2\varepsilon sin(\omega t),$
$A = 0.1,\ \varepsilon = 0.25,\ \omega = \frac{2\pi}{10}$ and $(x,y,z,t) \in [0;2] \times [0;1] \times [0;2] \times [0;10)$

- *Square cylinder:* A 3D DNS simulation of the flow around a square cylinder between parallel walls [143]. The vector field was resampled onto a uniform grid at resolution $192 \times 64 \times 48$. 102 time-steps were used. The scalar FTLE fields were pre-computed at fourfold the spatial and eightfold the temporal resolution.

- *Flow around a cylinder*: A large eddy simulation of an incompressible unsteady turbulent flow around a wall-mounted cylinder [44]. 22 time-steps were simulated. The size of the data grid is $256 \times 128 \times 128$. Pre-computed FTLE fields were generated at twice the spatial, and fourfold the temporal resolution.

- *LBM Flow:* A GPU-based Lattice-Boltzmann simulation of the flow around a donut-shaped obstacle. The spatial resolution of the simulation domain is $128 \times 64 \times 64$. The FTLE fields on the 2D sampling grid were computed on the fly during flow exploration. In every time step, 10 vector fields were used to compute the FTLE values. The interactive simulation created these data sets in advance.

7.7.1 Visual Exploration

3D double gyre: Placing the seeding plane parallel to the (x,y) plane essentially means to compute the FTLE ridges on the classical 2D double gyre. Our extraction shows that the obtained ridges agree with expected ridges known from the literature, with the main difference that our extraction works in real-time. Seeding streak surfaces from the ridges confirms that the ridges are approximate material structures: the generalized streak surfaces and the ridges show a good coincidence as depicted in Figure 7.8 (c).

Placing the seeding plane parallel to the (x,z) plane reveals approximate sine shaped ridge structures (see Figure 7.8 (a)). Note that this curve does not coincide with moving saddles of the vector field which is a well-known characteristic of the data set [153].

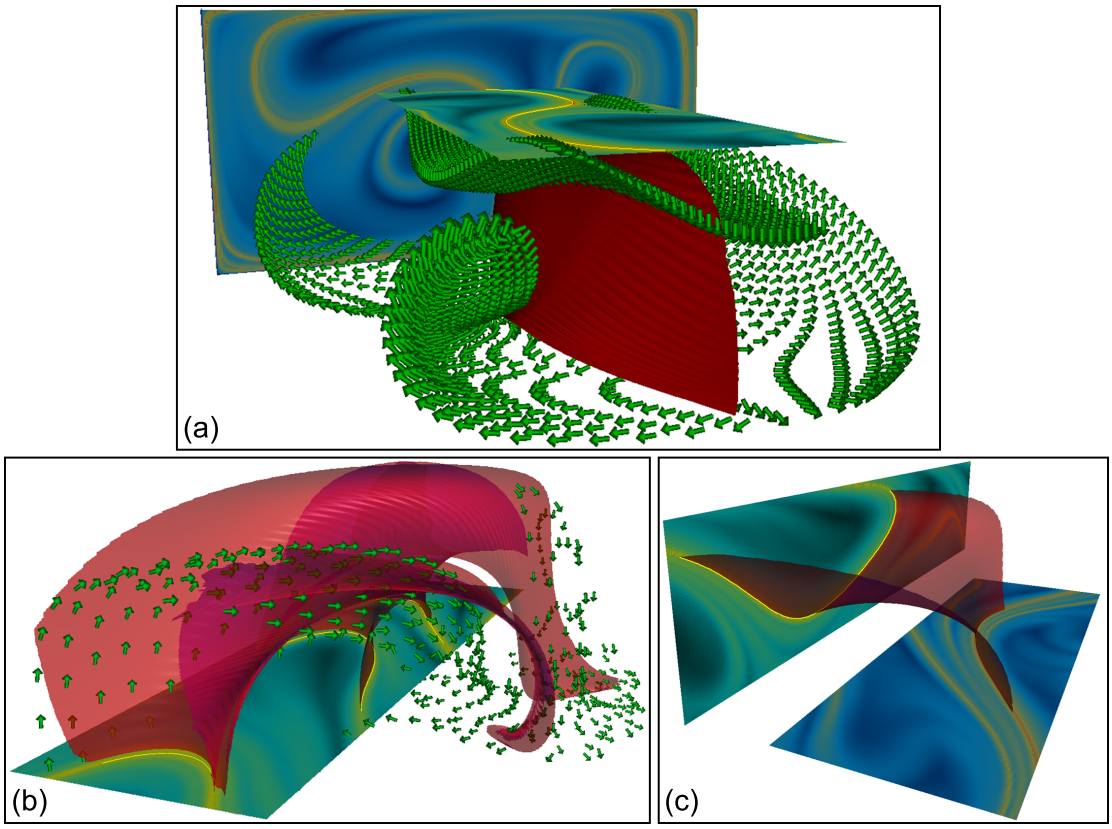


Figure 7.8: *Separating streak surfaces in the double gyre data set. Green arrows show the velocity direction on time surfaces that are additionally released from the planar probe. In image (b), individual surface layers are extracted via depth peeling. Image (c) depicts the correspondence between separating streak surfaces and FTLE ridges. Besides the seeding plane, a second plane is placed such that it intersects the surface and FTLE values are visualized on it. Here, the separating surface stays on the 2D FTLE ridges.*

Square cylinder: Seeding from a plane in front of the cylinder with a distance according to the integration time to compute the FTLE field reveals one distinct streak surface separating the flow passing above and below the cylinder (see Figure 7.9 (b)). Placing the seeding plane behind the cylinder perpendicular to the main flow direction shows periodically appearing and disappearing streak surfaces which alternate in moving upward and downward. This confirms the appearance of the well-known von Karman vortex street behind the cylinder. In order to show that the streak surfaces are indeed separating structures, we release time surfaces from the seeding plane at times when streak surfaces are released. The time surfaces get advected and clearly get distorted mostly around the intersections with the streak surfaces. As can be seen in Figures 7.6 and 7.9 (a), this shows the separating structure of our streak surfaces.

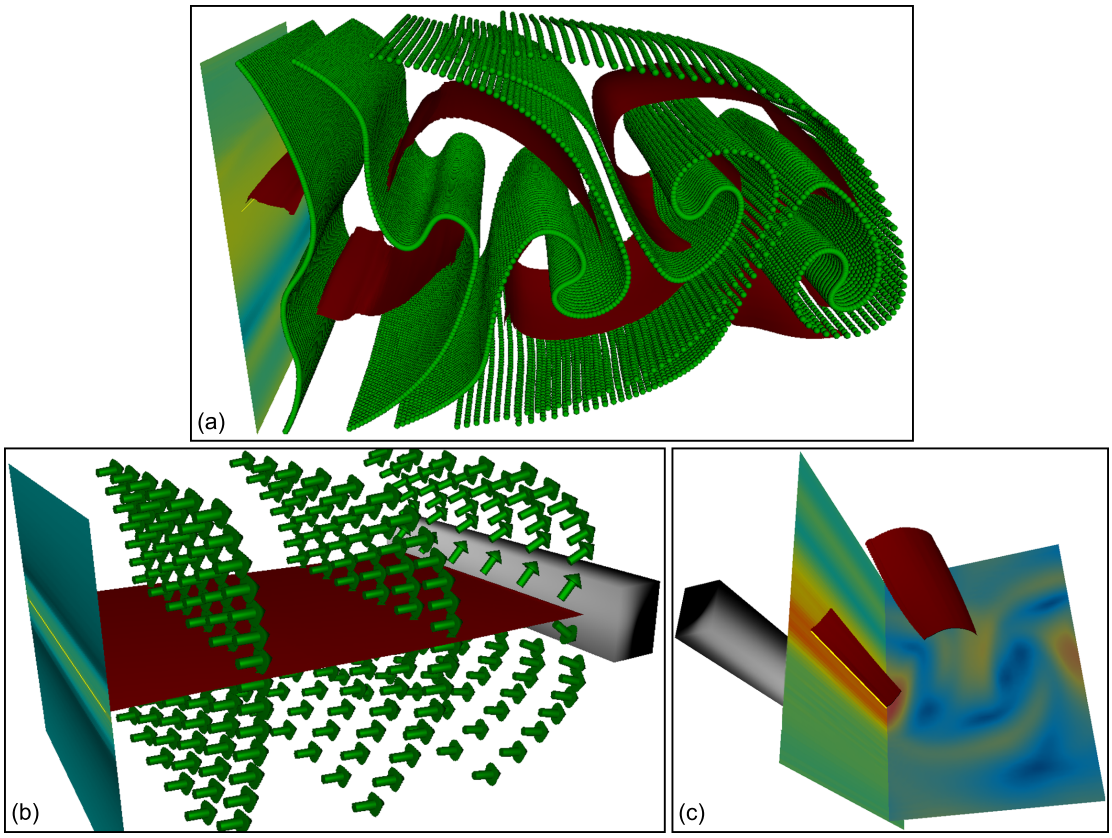


Figure 7.9: *Separating streak surfaces in the square cylinder data set, visualized using the patch-based approach. Separating surfaces in (a) reveal the well-known von Karman vortex street behind the cylinder. Image (b) depicts a single surface separating the flow passing above and below the cylinder. The correspondence between separating streak surfaces and FTLE ridges is shown in (c). In general, as the integration time used to compute the FTLE is finite, surfaces keep staying in regions of high FTLE but do not stay on the 2D FTLE ridges any more.*

LBM Flow: A tube shaped generalized streak surface is revealed by placing the seeding plane in front of the torus, separating the flow passing through the hole and around the toroidal obstacle. Moving **s** closer towards the obstacle creates two surfaces parallel to the first one, indicating the occurrence of a separation behind the torus.

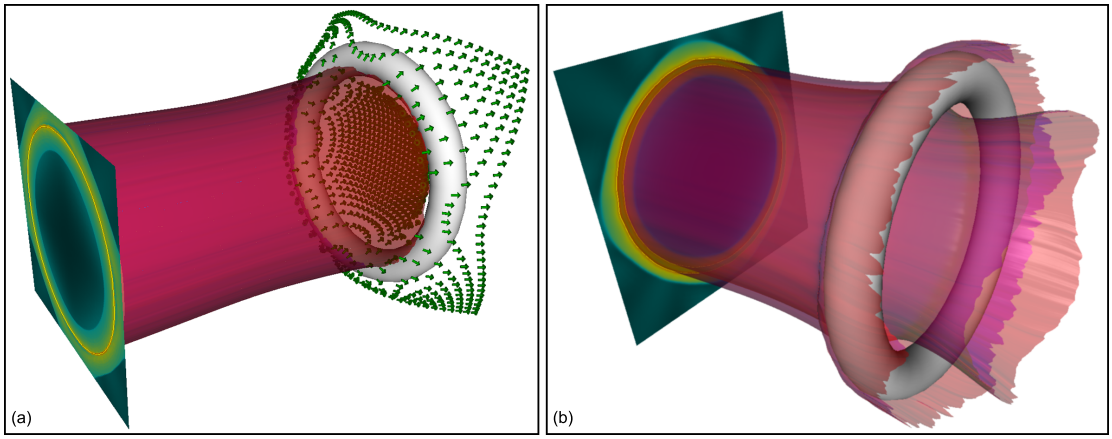


Figure 7.10: *Separating streak surfaces in the flow around a torus. Depth peeling was applied to extract multiple surface layers.*

Flow around a cylinder: Two streak surfaces enclosing the cylindrical obstacle are revealed by placing the seeding probe perpendicular to the inflow in front of the object. The extracted surfaces emanating from 1D FTLE ridges on the planar probe closely resemble the 2D FTLE ridges obtained by incremental ridge tracking in a similar data set [139].

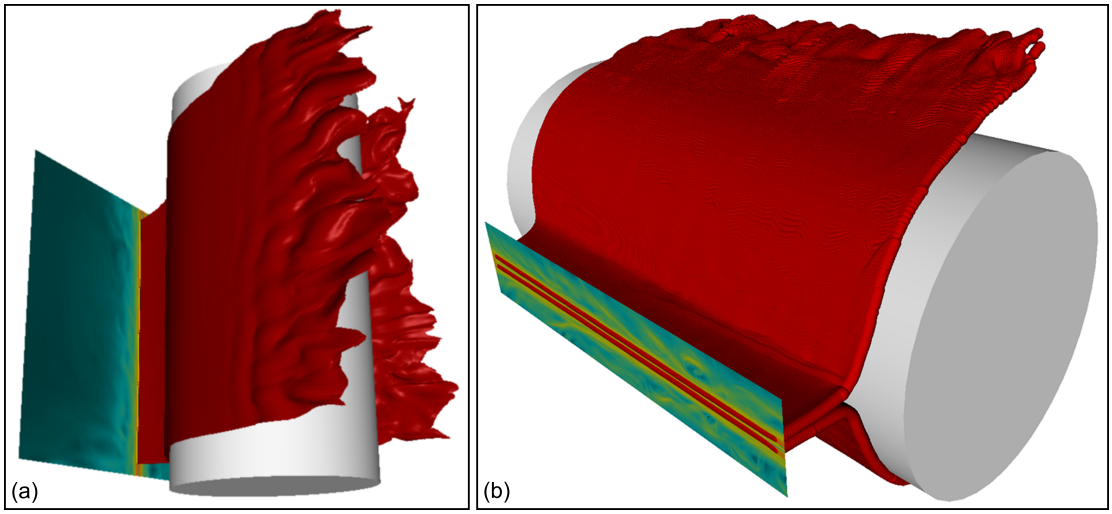


Figure 7.11: *Placing the planar probe perpendicular to the inflow in front of the cylinder reveals two separating surfaces enclosing the object. In (a), surfaces are visualized with the patch-based approach. In (b), the particle-based approach was employed.*

7.7.2 Performance

We applied an explicit fourth-order Runge-Kutta scheme at single floating point precision for numerical particle integration during FTLE (pre-)computation as well as for the integration of streak surfaces. Performance measures for FTLE pre-computation are presented in Table 7.1. Representative timings in hours (h) are given in column *Time* for varying temporal (*Timesteps*) and spatial (*SpatialRes*) FTLE resolutions. Column *Integration* contains the integration time Δt and the amount of integration steps.

Timesteps	SpatialRes	Integration	Time
80	$256 \times 256 \times 128$	8s in 50 steps	0.8h
576	$384 \times 128 \times 96$	10s in 100 steps	5.0h
576	$768 \times 256 \times 192$	10s in 100 steps	43h

Table 7.1: *Performance statistics for GPU-based FTLE computation.*

Timings for the ridge extraction on the planar probe as well as for the streak surface generation and visualization are given in Table 7.2. We extract seeding structures (FTLE calculations and ridge extraction) at a fixed temporal frequency (*Seed Interval*) on the planar probe **s** with varying texture resolutions (*Sampling texture resolution*). In cases where the FTLE was calculated interactively, columns *FTLE Setup* and *FTLE Time* show the used parameters and the respective calculation time, whereas resampling the precalculated FTLE using trilinear interpolation comes at negligible cost. Timings for FTLE thresholding, curve thinning and ridge refinement are summarized in column *Ridge extraction*. Column *#Quads+Particles* shows the average amount of primitives employed to visualize the separating streak surfaces and additional context information, column *Adv+Vis* the time spent for respective particle integration and rendering. Column *FPS* contains the average achieved frame rate during the interactive flow exploration sessions.

Seed Interval	Sampling texture resolution	FTLE Setup	FTLE Time	Ridge extraction	# Quads + Particles	Adv + Vis per Frame	FPS
25ms	250×250	-	-	6.9ms	108k	13.2ms ●	32.0
50ms	500×500	-	-	15.2ms	639k	80.0ms □	7.3
50ms	400×800	-	-	25.7ms	100k	10.7ms ●	24.4
100ms	250×250	10s in 50 steps	58.6ms	10.5ms	57k	6.7ms ●	18.1
100ms	250×250	15s in 100 steps	120ms	9.2ms	42k	11ms □	3.3
100ms	400×800	10s in 50 steps	277ms	23.0ms	200k	16.7ms ●	1.6

Table 7.2: *Performance statistics for the extraction and visualization of separating streak surfaces. The surfaces were visualized using either the particle based approach (entries marked ●), or the patch-based approach (marked □) including adaptive surface refinement.*

7.7.3 Limitations

For the visual exploration of turbulent flows the proposed technique seems problematic. As can be seen in Figure 7.12, when placing **s** in turbulent flow regions the FTLE exhibits rather fuzzy ridge structures undergoing frequent topology changes. Hence, many small, disconnected, and strongly moving surface parts will be generated, leading to visual clutter. Reducing the FTLE integration time Δt as proposed by [140] to simplify the "Lagrangian skeleton" can only be done to a certain extent, as the surface integration time is restricted to Δt.

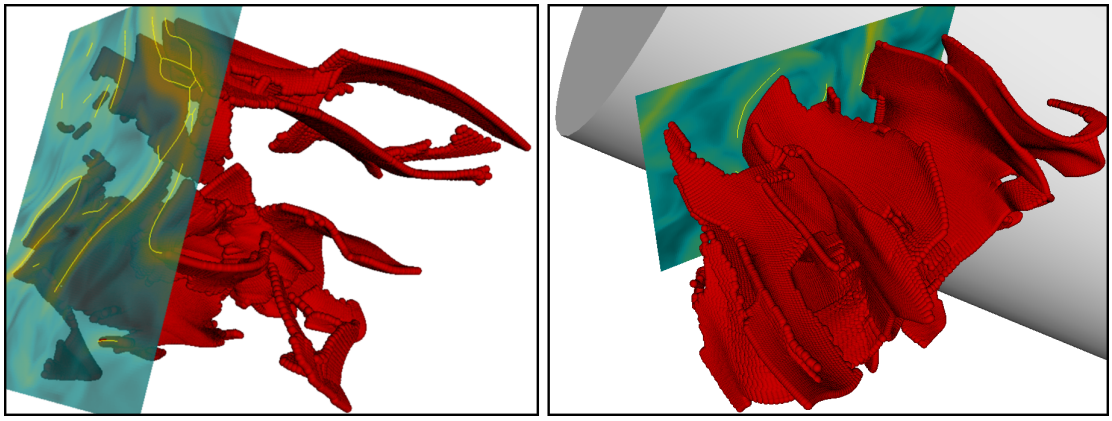


Figure 7.12: *Placing the seeding probe in turbulent regions. Frequent movements and topology changes of ridges result in highly fragmented surface parts and visual clutter thereof.*

The application of the curvature criterion (see Eq. 7.1) followed by skeletonization especially aims to simplify the extracted ridges. It is clear, on the other hand, that this can change the ridge topology, e.g. by removing non-shallow saddles, or lead to slightly misplaced ridges at crossings. Therefore, care has to be taken to not "misinterpret" the resulting ridges.

7.8 Summary

In this chapter we have presented a real-time technique for the extraction of 1D FTLE ridges on a 2D planar seeding structure in 3D unsteady flows. Since we employ ridges as seeding structures for a generalized streak surface integration, we focused on the extraction of a subset of all valid ridges. Whereas we aimed to a) obtain predominant features, i.e., long continuous ridge lines and b) to remove unwanted ridges such as discontinuous structures and crossings to avoid visual clutter while rendering the separating streak surface.

The GPU-based framework allows users to experience a visually guided exploration of semantic separating surfaces by moving the probe in space and/or time and changing parameters steering the ridge extraction and streak surface construction process interactively. To the best of our knowledge, this is the first time that the reconstruction and display of semantic separable surfaces in 3D unsteady flows can be performed at interactive rates, giving rise to new possibilities for gaining insight into complex 3D flow phenomena.

In fact, the interactive treatment of LCS allows insight not only into the locations of the separating structures but also into their temporal evolution including changing shapes, appearance and disappearance. Moreover, regions of interest can be determined interactively by moving around the seeding plane. This way, a fast visual impression of the ”big picture” of the flow as well an in-depth analysis of relevant parts (both in space and time) becomes possible.

In the future, we will pursue research into the following two directions: Firstly, we will perform a detailed analysis of the similarities and differences between separating streak surfaces and 2D LCS in 3D flows. This includes the variation of parameters such as the integration time for computing the FTLE field and the comparison of unstable manifolds in both scenarios. Secondly, adaptive meshing techniques for constructing high-quality polygonal generalized streak surfaces will be examined. In this respect it will be worthwhile to investigate ridge extraction techniques that are specifically tailored to the intended application and can monitor topological changes and degeneracies.

Chapter 8

Flow On Surfaces

In this chapter, we present new approaches to realize interactive geometry- and texture-based visualization techniques for surface flow. Such flow fields either live on a surface or can be re-sampled onto it from a surrounding 3D lattice. To achieve real-time performance, we introduce the *Orthogonal Fragment Buffer* (OFB), a sample-based surface representation which is independent of the original surface resolution and representation. We will discuss how previously introduced visualization techniques (see Chapter 4) can be adapted to this GPU-friendly data structure to effectively reveal surface flow. Furthermore, we will use the OFB to store additional surface attributes and employ them to create a view-independent dense flow representation on the basis of line integral convolution (LIC). Additionally, we will show how artistic approaches such as surface coloring can be applied to visualize surface flow via color advection along particle trajectories on the object. The quality and performance of the presented approaches is validated using real simulation data projected onto arbitrary clip geometry positioned in 3D unsteady flow, curvature fields on surfaces, and synthetic vector fields designed on surfaces.

8.1 Introduction and Related Work

To utilize particle-based GPU techniques in a broader range of practical applications, there is a dire need to extend these techniques towards the visualization of unstructured grids. In principle, it is clear how to perform particle tracing in unstructured grids composed of n-simplices like triangles and tetrahedra [75, 161, 120, 79, 148]. However, GPU-based particle tracing in such data sets performs at least an order of magnitude slower—in comparison to structured grids—due to the following reasons: Additional arithmetic and memory access operations occur for cell search and exact point location.

Pointer indirection is not directly available on the GPU, so additional index structures have to be stored. The traversal of such data sets leads to dependent fetches (introducing memory latencies) and highly incoherent memory access (in turn failing to employ local caching mechanisms). Furthermore and most importantly, particle integration in such data sets imposes an unbalanced load onto the parallel execution units on the the GPU. Shader units on GPUs are organized as multiple SIMD groups. Within one SIMD group (or *warp*), all the processing elements run in lockstep. While dynamic branching within a set of elements (executing the same shader kernel on different input data) is allowed, kernel execution on all elements is not terminated till all elements in the warp exit the code fragment. Thus, while some units might have to integrate over many elements in one particle advection step, others only have to consider a single element but are stalled until the whole warp terminates.

In this chapter, we will alleviate above problems for flow on arbitrary (unstructured) surfaces. We introduce the OFB as a new data structure that stores surface samples at a nearly uniform distribution over the surface, and is specifically designed to support efficient random read/write access to these samples. The data access operations have a complexity that is logarithmic in the depth complexity of the surface. Thus, compared to data access operations in unstructured grids (such as triangle meshes) or tree data structures like octrees, data-dependent memory access patterns are greatly reduced. Furthermore, the data layout of our surface representation maintains spatial sample coherence and, thus, exhibits very good spatial access locality. In addition, since the OFB adheres to a uniform resampling of the original surface, particle tracing can be written as a balanced stream program that effectively exploits computational and bandwidth capacities of recent GPUs. Due to these reasons, OFB-based particle tracing allows us to track millions of particles along velocity fields on surfaces interactively. Because of the intermediate surface representation we choose, our method can not only be used for the visualization of flow on polygonal surfaces but also for any arbitrary type of surface that can be sampled.

Next to geometry-based flow visualization techniques, texture-based approaches such as line integral convolution [28, 161] (LIC) are a well known class of algorithms to reveal directional information in velocity fields. Since these techniques generate a single, dense representation for the whole flow field, in 3D (unsteady) flows, they are often restricted to arbitrary clip geometry positioned in the flow domain. While a LIC representation can be pre-computed over the whole flow domain and resampled on the respective surface during visualization, this approach is hardly suitable for an interactive exploration of unsteady flow due to intense mermory access and numerical

operations during data generation. Image-based techniques [175, 98, 183] achieve interactivity by restricting the computation to the visible part of the surface. However, these approaches tend to be prone to deliver artifacts for flow fields living on a surface due to the following fact. Typically, line integral convolution works by smearing a random intensity (or color) distribution along particle trajectories, resulting in a high correlation among points on the surface residing on one characteristic trajectory. Image-based approaches perform the line integration on the surface projected into screen space and, thus, trajectories cannot be calculated correctly at silhouette boundaries or along edges obstructing parts of the surface under the current view. By using our view-independent data structure for particle tracing during LIC calculation this problem can be solved. Furthermore, by calculating LIC for every sample in the OFB and encoding it directly in the data structure, a view-independent flow representation can be obtained interactively.

8.2 Contribution

The primary focus of this chapter is the development of an efficient method for particle tracing on arbitrary (unstructured) surfaces. To achieve this, we introduce a spatial data structure that stores a resampled version of the surface—the Orthogonal Fragment Buffer (OFB). The OFB is conceptually similar to the Layered Depth Cube (LDC) introduced by Lischinski and Rappoport [106], which itself builds on Layered Depth Images (LDI) [155]. While an LDI captures all depth layers of an object in the order they are seen from one particular direction, an LDC captures these layers from three mutually orthogonal directions, thus representing a surface point up to three times in the data structure. In our approach, the sampling is also performed along sets of parallel rays emanating from mutually orthogonal uniform 2D grids. Along these rays, however, only surface points with an angle less or equal to 45 degrees between the surface normal at this point and the ray direction are considered. In this way, redundant sampling of the same point is avoided, and a quasi-uniform sampling with a maximum distance foreshortening of $\frac{1}{\sqrt{3}}$ is generated. Hence, our data structure can be seen as a redundance-free LDC.

Since the sampling can be performed by coordinate projections into uniform 2D grids, the OFB can be seen as a hashing of surface points using the projections as hash functions. Due to the underlying regular grid structure, this hashing maintains sample coherence so that the OFB exhibits very good spatial access locality. However, since the hashing maps multiple samples onto the same grid cell, it is not perfect. Specif-

ically, it produces up to d collisions per sample, with d being the depth complexity of the surface. Since the samples falling into the same cell can be sorted with respect to their distance to the sampling grid, the computational complexity of finding an entry in the hash table is $\mathcal{O}(log_2(d))$. On the other hand, if the samples falling into the same cell are not sorted, data-dependent memory access patterns—and memory latencies thereof—can be avoided entirely. Therefore, depending on the efficiency of data dependent memory access operations on the underlying hardware architecture, either a sorted or an unsorted OFB can be chosen flexibly.

Next to the OFB's advantages for GPU-based particle tracing discussed above, it is also especially suited for graphics hardware due to the following reasons: If the underlying surface is given as a triangular mesh, OFB contruction can exploit advanced features of current GPUs—in combination with their rasterization capabilities—to create even high-resolution surface representation for objects of reasonable depth complexity within the fraction of a second. Furthermore, the OFB data interface exhibits not only fast read but also efficient write access. Hence, it can be employed to access and update surface attributes in real-time. This makes it not only possible to resample a 3D unsteady flow field onto arbitrary clip geometry interactively, but also to enrich the appearance of an object during rendering by encoding the extracted flow representation directly in the OFB data structure. We will exploit this fact not only to store a texture-based flow visualization in the OFB, but also to improve the display of results extracted with geometry-based techniques. Rendering geometric primitives on top of an object often results in their (partial) obstruction by the underlying surface. E.g., unless the surface is flat and oriented perpendicular to the view direction, virtual geometry [55, 90] will intersect the object due to the screen-aligned nature of point sprites. Characteristic lines, approximated by a discrete set of linear line segments, also suffer from this problem. To solve these issues, we will present techniques that directly encode the extracted geometric flow representation in the OFB surface representation. Furthermore, we will present an approach for particle-based color advection along the surface and will exploit it to create various (artistic) flow representations.

Finally, let us note that surfaces do not need a parametrization (which is typical, e.g., for surface attributes stored in 2D textures) to access information stored in an OFB. This makes it even possible to employ our data structure to visualize flow on dynamic surfaces changing their shape over time.

However, we should point out that our method is subject to typical limitations of resampling approaches, such as the loss of detail caused by an under-sampling of the surface or the blurring of sharp features like edges due to the regular sampling pattern.

The rest of this chapter is organized as follows. In the next Section, we introduce the OFB, describe its internal structure and show how GPU hardware can be exploited to construct it interactively. Section 8.4 discusses particle tracing on the sample-based data structure. Section 8.5 describes how the rendering quality of geometry-based flow visualization techniques can be improved by encoding the extracted representations in the OFB. In Section 8.5.1, we demonstrate how particle tracing can be employed to construct surface-aligned (oriented) sprite patches. Section 8.6 introduces a view-independent texture-based flow representation on the basis of the OFB. Finally, we conclude this chapter with a summary of and discussion about limitations of our work.

8.3 The Orthogonal Fragment Buffer

In an OFB, a surface is stored as a set of sampled surface points. Sampling is performed along three mutually orthogonal *sampling directions* by projecting the surface orthographically along these directions into correspondingly aligned *sampling planes*. Every plane is discretized by a sampling grid, and each grid cell stores the distance to the sampling plane of the closest surface point projecting onto the cell center. In addition to this distance, a vector field is stored. The vector field is either constructed by sampling the velocity field of a surrounding 3D (unsteady) flow during OFB construction, or it is given by a (normalized) vector field defined at the surface vertices. While resampling a vector field into an OFB, we project it into the local surface tangent plane (as we want to visualize flow along the surface). The tangent plane is computed from the interpolated surface normal, resulting in a smooth variation of the plane across the surface. Optionally, further space for additional attributes is reserved in the OFB, which can be filled by a flow visualization technique.

Since along one sampling direction up to d surface points can fall into the same cell (where d is the depth complexity of the surface), up to d distances might have to be stored for each direction. Distances are sorted such that the ith distance is the distance of the ith closest point to the sampling plane. Every surface point is projected only once into the sampling plane with the smallest angle between the surface normal at that point and the sampling direction of the respective grid. In this way, redundant sampling of the same point into multiple grids is avoided, and a nearly uniform sampling with a minimum and maximum sampling frequency of $1/(\sqrt{3}\cdot s)$ and $1/s$, respectively, across the surface is generated (where s is the size of an OFB cell). Figure 8.1 illustrates the sampling strategy used to generate an OFB.

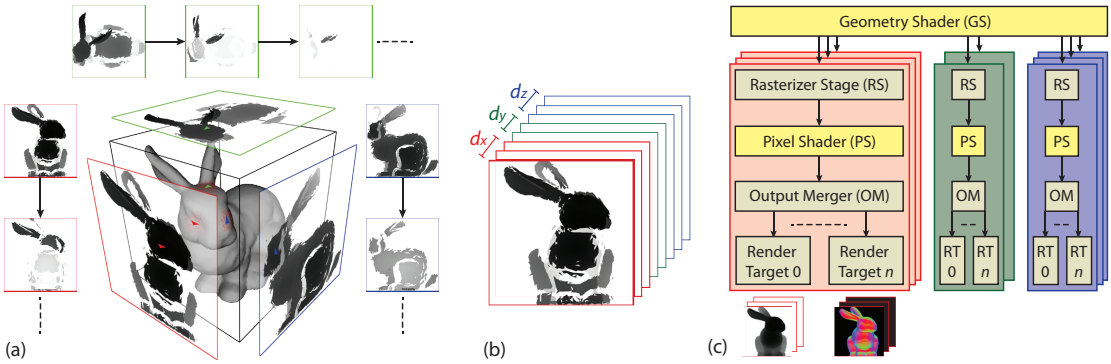


Figure 8.1: *OFB construction: (a) Surface points are projected along one of three mutually orthogonal sampling directions. Surface points falling into the same grid cell are stored in multiple sampling grids. (b) An OFB stores all sampling grids of the three sampling directions in a single texture resource. (c) GPU pipeline setup: The geometry shader selects the target sampling grid stack on a per-triangle basis and then rasterizes a triangle into all corresponding slices. Stencil testing routes the surface sample into the next unoccupied sub-sample. Multiple OFBs can be bound to the output merger stage to capture all surface attributes at once.*

8.3.1 OFB Construction

Sampling the surface along a particular direction can be performed in many different ways, e.g. by using ray-casting or rasterization. In this section, we demonstrate how the sampling can be performed by rasterization on the GPU. In particular, we employ stencil routing [132, 118] in combination with a novel geometry shader algorithm to direct sampled surface points into the respective sampling grids.

For single pass OFB construction on the GPU our method utilizes the geometry shader and the *k*-buffer introduced by Myers and Bavoil [118]. A *k*-buffer is a render target, i.e., a texture map that can keep the contributions of up to *k* fragments per pixel instead of just one as in single-sample rendering. When rendering to a so-called multisampled texture target with multisampled antialiasing being disabled, an incoming fragment is spread to all *k* multisamples of the respective destination pixel in the *k*-buffer. Since for each multisample a separate stencil mask is tested, stencil routing as proposed by Purcell et al. [132] can be used to direct an incoming fragment to a specific multisample. Stencil routing works by initializing the stencil mask of the *i*-th multisample with $i+1$ (value 1 is used to detect an overflow), and by letting a fragment pass the stencil test if the stencil mask is equal to 2. The stencil fail and pass operations are set to “decrementing”, such that a stencil mask of 2 is consecutively obtained at all multisamples.

Via stencil routing up to k (=8 on our target graphics hardware) surface points seen under a pixel can be rendered simultaneously into one texel of a multisampled render target. Since multiple render targets can be used and because an 8 bit stencil buffer is supported, surfaces with a depth complexity of up to 254 can be sampled in a single rendering pass. An OFB finally consists of one *Texture2DArray* containing three stacks of multisampled texture slices, where each slice stores distances of surface samples to the respective sampling planes. By using this approach, an OFB can be built at extreme resolution within a fraction of a second for surfaces of reasonable depth complexity.

To efficiently sample the surface along three mutually orthogonal directions, we exploit the capability of the geometry shader to direct its output to multiple render pipelines, each having its own depth, stencil and multiple color buffers. Every surface triangle is projected and rasterized only into the most appropriate sampling grid depending on its normal. Assuming the surface being represented as a triangle mesh with depth complexities d_x, d_y, d_z along the x, y, z-coordinate axes, the following GPU setup is used to construct the OFB:

- **Pipeline Setup:** $T = \lceil \frac{d_x}{k} \rceil + \lceil \frac{d_y}{k} \rceil + \lceil \frac{d_z}{k} \rceil$ render pipelines are bound to the output merger stage. To each of these pipelines k-times multisampled render targets are attached. Contiguous sets of $\lceil \frac{d_x}{k} \rceil$, $\lceil \frac{d_y}{k} \rceil$, and $\lceil \frac{d_z}{k} \rceil$ pipelines are used to perform the sampling along the x-, y-, and z-direction, respectively. In each set, all sub-samples in the $\lceil \frac{d_i}{k} \rceil$ $(i = x \vee y \vee z)$ multisampled 2D texture slices residing on the same pixel raster position get assigned a unique (increasing) stencil value > 1.

- **Geometry Shader Setup:** For every triangle, its face normal is computed and the triangle is directed into those pipelines that belong to the sampling direction with the smallest angle to the normal. Before triangles are sent to a pipeline, they are transformed according to the respective sampling direction, i.e. they are projected into a 2D sampling plane aligned perpendicular to this direction.

- **Pixel Shader Setup:** The pixel shader outputs the fragments' depth as well as additionally queried attributes (e.g. the velocity field) into the multisampled render targets. With stencil testing activated and configured as described above, the output merger then stores the attributes in the next unoccupied sub-sample of the output buffers and decrements the stencil bits of all sub-samples in the respective pixel.

To store surface point positions and corresponding attributes (e.g. vector field samples), multiple OFBs are used. In DirectX 10, up to 8 buffers—each with 4 32bit chan-

nels at most—can be bound as output targets to a pixel shader, enabling to capture up to 128 bytes of surface attribute data at once. Each OFB is initialized at startup or when the surface geometry is changed. The position OFB stores for every sample its distance to the corresponding sampling plane. It should finally be noted that the OFB samples can be sorted with respect to their distance to the sampling plane [118]. Especially for objects having large depth complexity this can significantly improve the complexity of the OFB read-operation, from linear to logarithmic in the surface's depth complexity.

8.3.2 OFB Point Location

The OFB interface provides the following method for locating a surface point in the data structure. The method takes as input a 3D position (x,y,z) in normalized object coordinates and tests whether a corresponding sample is stored in the OFB. To find this sample, the point coordinate is projected into the three OFB sampling planes. This generates for the respective sampling grids a 2D integer coordinate (u,v) and a distance d of the point to the sampling plane. If the sampling directions are aligned with the three coordinate axes the projection reduces to a component selection, i.e., in the z-direction $(u,v) = (\lfloor x \cdot S \rfloor, \lfloor y \cdot S \rfloor)$ and $d = z$, where S is the OFB grid size. The distance d is now compared to all distances stored at index (u,v), and of all these values the index g of the slice containing the distance closest to d within the interval $[d-s, d+s]$ is kept (with s being the cell size in the sampling grid). If the point is associated with a surface normal the search can be restricted to the sampling grid whose sampling plane is most perpendicular to the respective normal. The method finally returns the index tuple $I = (u,v,g)$, which can then be used to read a velocity field vector from the OFB or to write an attribute into it.

8.3.3 OFB Rendering

Enhancing a surface with attributes (like color) stored in an OFB during rendering means to interpret the OFB as a texture consisting of several layers and fetching for every rendered surface point the color from this texture. This is realized by executing for every rendered surface point an OFB query as described above. The color at this sample is then read and used to modulate the point's appearance. To support smooth color variations, the OFB interface provides distance-weighted color interpolation. If a surface is rendered at a resolution that is higher than the resolution of the OFB, for every surface point an OFB query is issued. In contrast to finding the closest sample, however, all samples within a radius of $\sqrt{3}$ times the size of a cell in the OFB grid are

determined under all three projections. In each projection we also inspect the distance values in all grid cells adjacent to the cell (u,v). The color of a surface point is then computed from these samples by means of inverse distance weighting.

By using this interpolation scheme we can also generate an OFB mipmap hierarchy to resolve minification issues. Therefore, multiple OFBs at ever decreasing resolution are constructed by subsequently reducing the resolution of the sampling grids about a factor of 2 in every dimension. Starting with the initial OFB at the finest resolution, the color of a sample at subsequent levels is computed by distance-based interpolation, with the color samples being fetched from the next finer level. In this way a stack of OFBs is generated, from which the appropriate resolution can be chosen during rendering.

8.4 Particle Tracing on Surfaces

To trace a particle on the surface, we compute its trajectory according to the ordinary differential equation given in (2.1). In principle, it is clear how to perform particle tracing on polygonal surfaces consisting of triangles [75, 161, 120, 79]. Particles are traced from edge to edge by projecting the vector field onto the triangle plane and performing the particle integration in this plane. Although this approach can be realized in a straightforward way on the CPU, it imposes severe limitations on the number of particles that can be moved at interactive rates. Specifically, our tests have shown that not more than 10K particles per second can be integrated in one step on a triangle surface of reasonable resolution. In contrast, as we will show in the remainder of this chapter, the proposed particle tracing algorithm can trace millions of particles in high-resolution OFBs.

The GPU implementation of particle tracing on a triangle mesh, on the other hand, yields a highly non-uniform load in the parallel shader units performing the particle integration. While for a given (global) integration step size some units have to integrate over many triangles in one integration step, only one triangle might be considered by other units. On recent GPUs, this results in execution stalls and, thus, in a significant loss of performance. This limitation can be avoided by tracing particles on the sample-based surface representation stored in an OFB. Since the OFB represents the surface at a nearly uniform resolution on a regular sampling grid, every unit performs a similar amount of memory accesses and numerical operations.

Particle trajectories along the OFB surface representation are calculated on the basis of the classical Euler integration scheme with a fixed step size Δs. In each advection step, the OFB sample closest to the current particle position is located as described in

Section 8.3.2. To prevent a particle from leaving the sampled surface representation, its position is set to the position of this sample. The index of this sample is used to read the respective vector field sample, along which the particle is then moved to its new position. A step along the surface is performed by first projecting the current vector sample into the three sampling planes. This gives for each plane a 2D vector (t_u, t_v) in this plane. By using these vectors and the projections of the particle position into the respective grids, we can now determine the grid cells in each grid into which the particle might be entering when making a step that is equal to the cell size. In all of these cells we determine the sample closest to the new particle position, and we set the new particle position to the position of this sample. Overall, given the index tuple I of a particle $\mathbf{x}$ in the OFB, in every iteration the following steps are performed:

- **Vector lookup**: The vector sample $\mathbf{v}$ is read from the index I in the OFB.
- **Projection**: $\mathbf{v}$ is projected into the three sampling grids.
- **Integration**: Discrete cell traversal in the sampling grids along the projected vector sample and closest point location in the traversed cells yields the index I' of the OFB sample closest to $\mathbf{x}$.
- **Update**: I is set to I' and the new particle position is determined by an inverse transformation of (the grid coordinates of and the distance value stored at) I' into object-space coordinates.

Let us note that it is clear that the accuracy of the proposed particle tracing method is limited due to the sample-based surface representation that is used. Since the particle positions are restricted to the OFB samples, they will in general not accurately follow the characteristic lines in a given vector field. However, due to the extreme OFB resolution that can be used interactively, the trajectories reconstructed by means of our method match those extracted with a triangle-based approach at high fidelity. Furthermore, the described setup assumed a normalized velocity field on the surface and, thus, performed only one (uniform) integration step per advection update. If no normalized velocity field is used, different particles require varying amounts of integration steps (as the step size is restricted to the size of the OFB sampling grid cell s). However due to the uniform sampling approach, the point location requires less effort and exhibits very good spatial access locality. Furthermore, the difference in performed integration steps varies by far less than compared to unstructured grids.

8.5 Geometry-based Surface Flow Visualization

By adapting the particle tracing techniques introduced in Chapter 4 accordingly, a multitude of different rendering modalities can be applied to visualize flow on surfaces. Some exemplary results are shown in Figure 8.2.

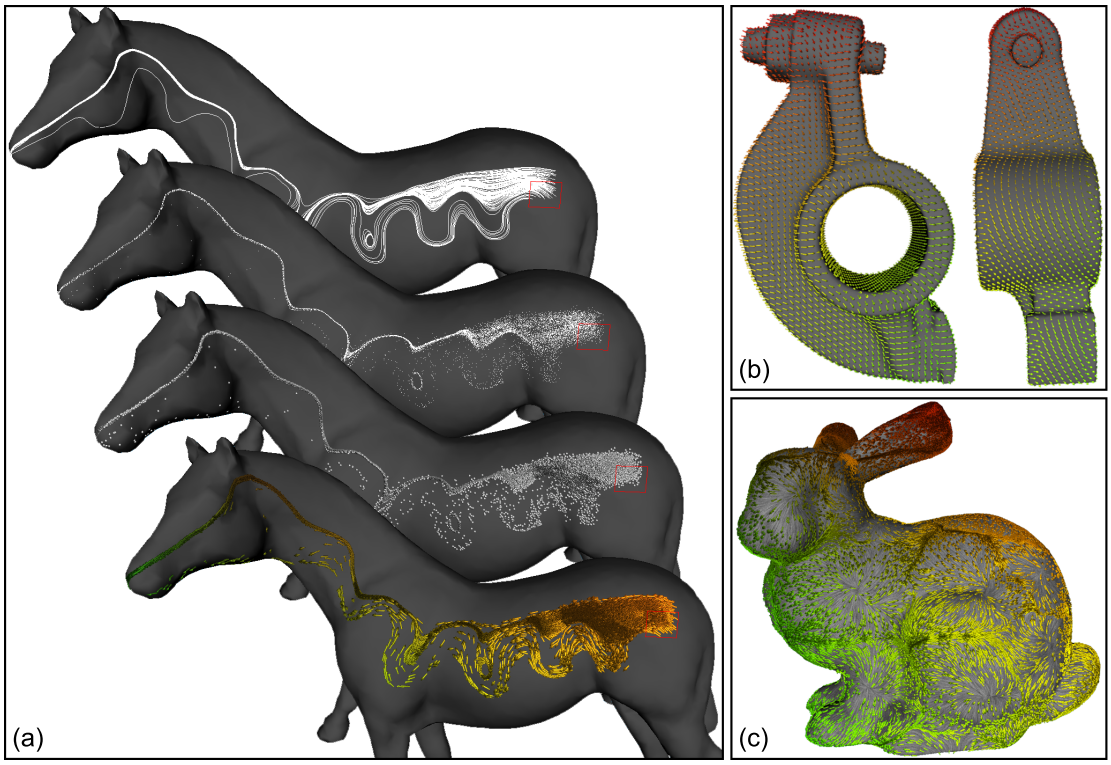


Figure 8.2: *Geometry-based surface flow visualization: Image (a) depicts different rendering modalities for particle trajectories. From top to bottom: Streamlines, single points, sprites, oriented sprites. In (b) a surface curvature field is visualized by uniformly distributed, oriented particle sprites. Image (c) depicts surface flow in a synthetic velocity vector field.*

8.5.1 OFB Surface Coloring

Next to the OFBs storing the resampled surface position, arbitrary surface attributes (besides the velocity vector field) can be stored in the data structure. By introducing an additional color OFB, we can encode the geometry-based flow visualization directly in the OFB. While particles travel along the surface, they can write attributes back into the OFB, and we exploit this fact to realize advanced (or more artistic) visualization modalities, as well as to solve problems inherent to surface flow visualization on the basis of (discretized) geometric primitives.

Spherical Particles

Instead of writing a single value into (one pixel at) the respective OFB position at index I, it has proven worthwhile to transfer color through footprints of enlarged extent. In the most simple case, a spherical volume centered at a particle's position is used to transfer color to the OFB. In this case, OFB samples closer to the particle position $\mathbf{x}$ than a selected sphere radius r get inked by a color $\mathbf{c}_x$ (specified on a per-particle basis). The color of a sample at position $\mathbf{p}$ is updated according to

$$\mathbf{c}_p = \text{lerp}(\mathbf{c}_p, \mathbf{c}_x, g), \tag{8.1}$$

where the spherical brush shape g evaluates to

$$g = \begin{cases} 0, & \text{if } \|\mathbf{x}-\mathbf{p}\| > r \\ f(\|\mathbf{x}-\mathbf{p}\|), & \text{else.} \end{cases} \tag{8.2}$$

Here f is a user-defined falloff function which is used to simulate smooth color fading with increasing distance to the center point.

It is clear that when a particle uses a volumetric brush to transfer color to the OFB, not every sample in the data structure should be tested for inclusion in the brush volume. Thus, a method for reducing the number of potential candidates to be tested is required. Therefore, we exploit the fact that the OFB structure was built by sampling the surface along three mutually orthogonal directions. As a consequence, of all samples only those have to be tested whose projections along these directions fall into the regions covered by the projected bounding box of the spherical brush volume. We use a geometry shader to efficiently determine all potential candidates per particle, and then perform the candidate tests in parallel in a pixel shader kernel.

Each particle is sent as single vertex to the GPU and passed to the geometry shader. The geometry shader spawns three quadrilaterals from this vertex, each of which is aligned to one of the three sampling directions and rendered into all slices of the corresponding OFB sampling grids. The size of these quadrilateral is chosen according to the current extent of the spherical color footprint. For every generated fragment, a pixel shader queries the corresponding sample in the OFB sampling grid slice and computes the distance of this sample to the brush center. Whenever a sample is closer to the center than the brush extent, the shader evaluates equation 8.1 and writes the color into the OFB. This allows us to apply advanced rendering techniques even while thousands of particles move along the surface in parallel and transfer color to the OFB surface representation.

By resetting the color OFB in every frame, the obtained results correspond to spherical particle sprites without the problem of an obstruction of a sprite's quadrilateral by the surface geometry. This issue can otherwise only be solved by aligning sprite geometry along the underlying surface. By chaining multiple particles together we can realize advanced shapes which would be difficult to realize otherwise. E.g., in Figure 8.3 (a) at randomly selected sample points along the surface a sequence of particles was seeded consecutively. The later a particle was released, the smaller is the assigned extent of its color footprint. By moving all particles along the vector field direction, the impression of moving particles with a tail is simulated.

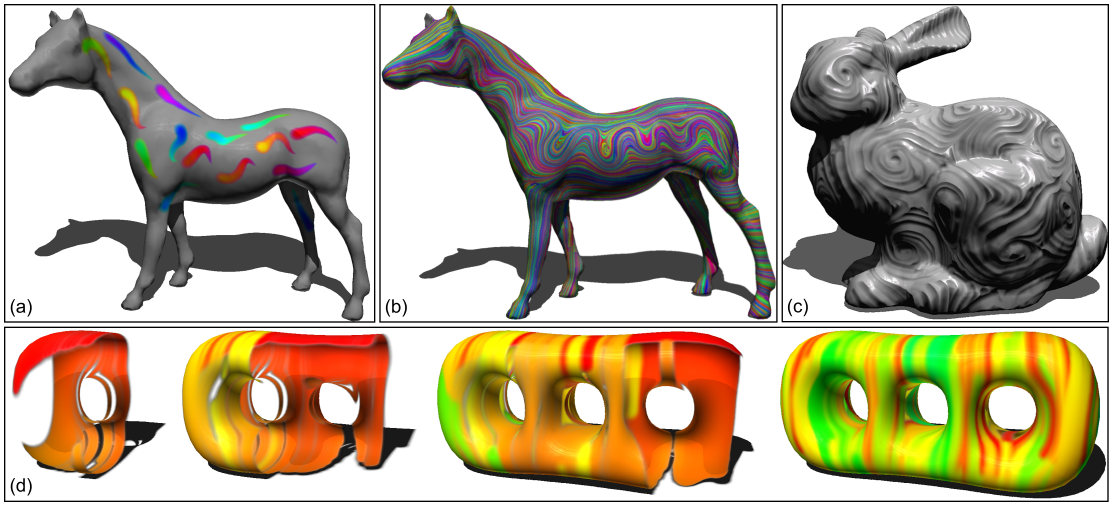


Figure 8.3: *Attribute advection in the OFB. Image (a) depicts advanced particle shapes realized by chains of particles released consecutively from random sample positions. In (b) particles travel along the surface and transfer color at a fixed temporal frequency into the OFB. In image (c) surface normals are modulated along stream line trajectories. In (d), surface flow is visualized by color advection along the velocity vector field on a transparent object.*

By retaining the change in OFB color over time, characteristic trajectories can be visualized efficiently through color advection along the surface. Stream lines in a vector field designed on a polygonal surface are shown in Figure 8.3(b). In this example, 10K particles were simultaneously traced on the surface, each of them spreading a spherical color footprint to the surface. Despite the large amount of particles used, particle advection and coloring was performed at 80 fps on an NVIDIA 8800GTX GPU. Let us note that the storage of intermediate positions along the trajectory in additional resources—as described in Section 4.7—becomes superfluous, as the characteristic lines are directly encoded in the color OFB. Extending color transfer by opacity or adding additional surface attributes (such as normal perturbations) makes it possible to realize even

more artistic flow visualization. E.g., in Figure 8.3 (d) the surface flow was visualized by revealing an initially transparent surface through color advection along the velocity vector field, and in 8.3 (c) surface normals were perturbed along trajectories in a surface flow field. Let us note that these examples do not contribute to a better understanding of the underlying flow phenomena. These techniques were developed in the course of this dissertation with respect to particle-based creation of artistic content, and published in [26].

Surface-aligned Point Sprites

As a spherical volume brush model considers the Euclidean distance to the particle position, surface points having a geodesic distance to the center that is larger than r may also be colored. Furthermore, shapes of more complex particle glyphs (such as oriented virtual geometry like arrows) can hardly be realized by spherical particle brushes. To overcome these limitations, we will introduce an alternative approach in the following.

For flow on surfaces, the proxy geometry of an (oriented) point sprite should ideally be modeled as a deformable sheet that wraps around the surface. We call this approach a surface sprite, and we model it by a polygonal mesh consisting of surface samples connected via edges. Such a mesh is constructed with the help of particle tracing along the surface. To avoid confusion, in the following we will call particles that are advected along the surface flow as part of a geometry-based flow representation *flow-particles*, while particles used to construct the surface aligned mesh will be denoted *mesh-particles*. Mapping the mesh onto a surface is done by tracing out a set of *mesh-particles* from a *flow-particle* position, which essentially corresponds to finding a local parametrization of the surface area surrounding this point.

Our method is similar in spirit to the patchinos and the exponential maps, which were introduced for decal painting by Pedersen [124] and Schmidt et al. [149], respectively. The patchinos, however, require a global parametrization of the base mesh. In contrast to exponential maps, on the other hand, we trace geodesics on the surface—or more precisely on a sampled version of the surface—instead of developing the surface to the tangent plane around a center point. Thus, the local parametrization we construct is completely independent of the underlying mesh resolution.

To align a point sprite on the surface, we first construct a local coordinate frame on the basis of the flow field. It is built from the velocity vector at the respective flow-particle position (the tangent $\mathbf{v}$), the surface normal $\mathbf{n}$, and the vector perpendicular to both (the bi-normal $\mathbf{b} = \mathbf{v} \times \mathbf{n}$). Constructing a local parametrization now starts by seeding two mesh-particles at the flow-particle position and tracing them along the

surface. One of them is traced in the direction of the tangent vector and the other one is traced in the opposite direction. Since along these traces the tangent varies according to the change in the surface normal (see Figure 8.4), it is corrected in every step i as

$$\mathbf{v}_i = \|\mathbf{n}_i - (\mathbf{n}_i \cdot \mathbf{b}_{i-1})\mathbf{b}_{i-1}\| \times \mathbf{b}_{i-1}.$$

In this way the trace wraps around the surface even in regions with high curvature.

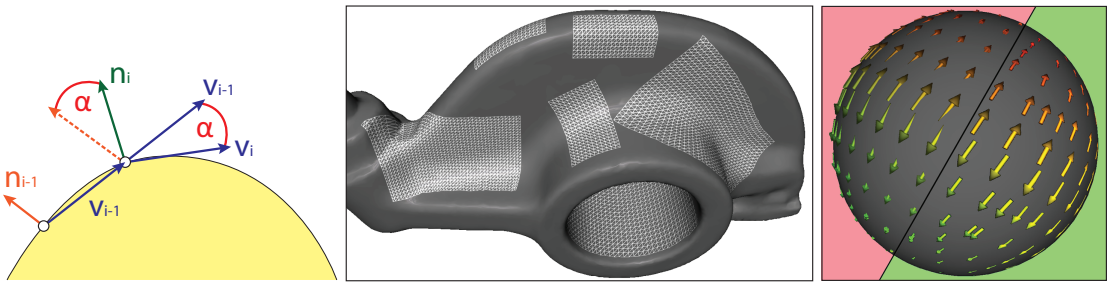


Figure 8.4: *Left: The direction vector* $\mathbf{v}$ *is rotated by* α *degrees about the bi-normal, where* α *is the angle between the current normal* $\mathbf{n}_i$ *and the normal* $\mathbf{n}_{i-1}$ *at the previous mesh-particle position in the plane perpendicular to the bi-normal. Middle: Surface-aligned point sprite meshes rendered as wire-frame. Here, rather large sprites were used to demonstrate the folding of the brush meshes along the surface. Right: The advantages of surface-aligned sprites (green) to screen-aligned oriented point sprites (red) are shown. As can be seen, the (partial) obstruction of point sprites due to surface intersection can be solved.*

After n steps, a polyline consisting of $(2 \cdot n)$ line segments is generated. From every mesh-particle on this line two new traces are started; one into the direction of the bi-normal and another one into the opposite direction. After m traces a 2D grid consisting of $(2 \cdot n + 1) \cdot (2 \cdot m + 1)$ particles has been laid out on the surface around the center point. We employ the geometry shader in combination with the stream output stage to construct the surface sprite meshes. During rendering, adjacent particles in the grid are finally connected with the help of one static index buffer (used by all surface sprites) to form a triangle mesh, yielding the local parametrization used to map colors of a surface sprite to the object. The triangle mesh can be texture mapped by specifying texture coordinates at the mesh-particles used to construct the mesh. The texture color can be transferred to the OFB by rendering every mesh triangle into the sampling grid slices corresponding to the sampling plane with the smallest area foreshortening, and writing the color of every fragment into the OFB as described in Section 8.3.2. Alternatively, the surfaces-aligned sprites can be rendered directly into the frame buffer. In Figure 8.4 (right), the advantage of surface-aligned sprites in contrast to screen aligned point sprites is shown.

8.6 Texture-based Surface Flow Visualization

Line integral convolution [28, 161] is the most popular texture-based flow visualization technique used to create a dense representation conveying directional information about the underlying velocity vector field. This technique is based on particle trajectories (namely path lines), yet, instead of extracting separate trajectories and visualizing them through geometric primitives, LIC works by smearing a random noise intensity distribution along the characteristic lines. This results in a high intensity correlation along characteristic lines and high noise frequencies perpendicular to each characteristic line. LIC has been used in a number of approaches to interactively visualize vector fields given on a surface [175, 98, 183]. These approaches, however, differ significantly from ours in that they operate on the visible surface points in image-space. If these techniques are used to visualize flow fields living on a surface, they result in artifacts as they cannot determine the points along a trajectory that are not visible under the current view. If, on the other hand, particle tracing on the view-independent OFB surface representation is performed, this problem can be solved and, thus, frame-to-frame coherence in animated visualizations is assured.

Imagine a random noise intensity distribution over the whole flow domain, i.e. at each point on a given object for surface flow or at each sample position in the flow domain for clip surfaces in 3D flow. LIC now determines the intensity at a every sample $\mathbf{x}_0$ in space and time t_0 by stepping along the characteristic line—in both directions—passing through that point and accumulating intensity values weighted by a filter kernel. Mathematically, this can be posed as the convolution of a color function C and a convolution kernel k along the characteristic lines:

$$Color = \frac{\int_{-L}^{L} C(\mathbf{x}(t,t_0,\mathbf{x}_0)) \cdot k(t)\,dt}{\int_{-L}^{L} k(t)\,dt}\,, \tag{8.3}$$

where $[-L,L]$ defines the support of the convolution kernel, and $k(t)$ is the filter kernel. Generally, a symmetric filter, e.g. box or tent function, is used. In our setting, a line integral convolution is computed at every OFB sample and the output values are written back into the OFB. LIC calculation is performed in the pixel shader stage, and the necessary samples are created by rendering the surface mesh in the same way as for OFB construction (see Section 8.3.1). The convolution is then realized by spawning at every sample two particles, of which one is moved for some distance along its trajectory and the other one moves the same distance in reversed time. While moving along the surface, the particles read values from the random intensity/color distribution at the

current position and weight this color with the kernel function. Combining the accumulated color values from both particles yields the output value. Let us note that if LIC is extracted in 3D and only its computation and subsequent visualization is restricted to a clip surface, then particle tracing is performed in the 3D unsteady flow field (and therefore no velocity vector field OFB is needed). In Figure 8.5 (a–c) LIC for synthetic velocity fields living on a surface is shown, and Figure 8.5 (d) depicts LIC in a 3D flow field restricted to clip geometry.

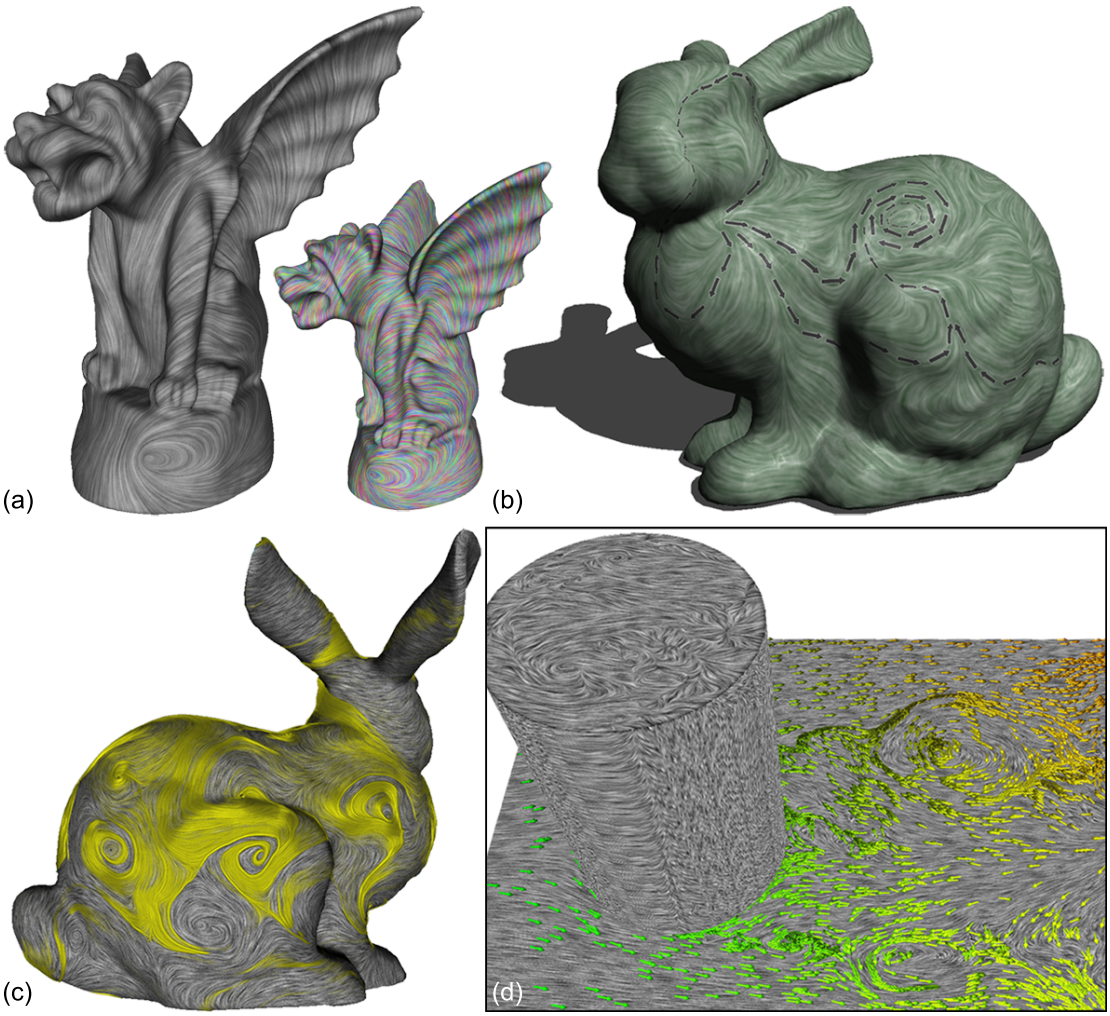


Figure 8.5: *Images (a–c) depict LIC extracted from synthetic velocity fields defined on a surface. In (a) a comparison between the application of a random intensity distribution (left) to a color distribution (right) during LIC extraction is shown. (b) Surface sprites reveal additional information about the velocity direction and magnitude. In (c) oriented point sprites of elongated cylindrical shape were rendered on top of the surface. Image (e) depicts 3D LIC in unsteady flow restricted to a 2D clip surface.*

To create the texture based flow representation in Figure 8.5 (a) an OFB with a sampling grid resolution of $1K \times 1K$, containing more than three million surface samples was employed. During LIC computation, at each sample 20 integration steps were performed to collect intensity values along the trajectory, whereas the whole visualization took less than 270 ms including LIC extraction, color transfer to the OFB and subsequent rendering. Thus, the performance even allows to animate surface LIC by computing the convolution in every frame along stream lines in unsteady flow.

8.7 Summary

In this chapter we have presented particle-based techniques for the visualization of flow on surfaces. These techniques employ a new sample-based data structure to trace even millions of particles along arbitrary surfaces at interactive rates. We have shown how previously introduced geometry-based approaches can directly be employed by adapting the underlying particle integration algorithm to the sample-based data structure. Furthermore, we have shown how the OFB can be employed to solve common rendering issues inherent to geometry-based surface flow visualization techniques. Moreover, we have used the OFB to interactively extract a view-independent, texture-based flow representation on the basis of line integral convolution.

Let us note that the proposed methods may deliver erroneous results because of the sample-based nature of our data structure. However, due to the extreme OFB resampling resolution that can be used, we could not assess any difference in the resulting renderings in comparison to results obtained by particle tracing on a triangular surface representation. Yet, the proposed techniques run at interactive rates and can provide rapid visual feedback. Thus, they allow for an effective visual exploration of surface flow or to restrict the visualization of 3D unsteady flow to arbitrarily shaped clip surfaces.

Finally, let us note that the approaches presented in this chapter were developed as part of a new surface coloring technique and, thus, not all topics of the publication were covered. For interested readers, we refer to [26], where even more surface coloring techniques are presented and a detailed description of various filtering approaches improving the quality of renderings obtained from the OFB is given.

Chapter 9

Particle-based Volume Editing

So far, we have discussed GPU-based particle techniques in the context of interactive flow visualization. In addition, particle-based techniques often play a fundamental role in other scientific visualization or computer graphics related areas. In the following we will present an example how GPU accelerated particle tracing—or the massive parallel processing power of recent GPUs in general—can be employed in the field of scientific volume rendering to augment data sets in real-time. In (human) medicine, volume rendering has become an integral technique for diagnostics, fundamental research or even the education of prospective physicians. Especially in peer group consultation or round table discussions, there is a dire need for intuitive metaphors to communicate insight gained from such data sets. In this chapter, we address this issue by introducing basic methodology for interactive GPU-based volume editing and enhancement. Here, we aim at developing a framework exhibiting similar functionality to current image processing tools to support scientists to communicate findings and to ease processing work inherent to such data (like classification and segmentation). We present fast techniques to modify the appearance and structure of volumetric scalar fields given on cartesian grids. Similar to 2D circular brushes as used in surface painting we present 3D spherical brushes for intuitive coloring of particular structures in such fields. This paint metaphor is extended to allow the user to change the data itself, and the use of this functionality for interactive structure isolation, hole filling, and artefact removal is demonstrated. Building on previous work in the field, we introduce high-resolution selection volumes—which can be seen as a resolution-based focus+context metaphor—and we utilize such volumes for interactive volume editing at sub-voxel accuracy. We introduce an approach based on particle tracing to place internal annotations on extracted iso-surfaces, and we extend this techniques to realize surface aligned cutaway-views that can effectively reveal internal surface structures.

9.1 Introduction and Related Work

Interactive visual exploration of volumetric scalar fields is required in many different areas ranging from medicine and engineering to physics and biology. To support the exploration task, volume rendering techniques have been developed to a high degree of sophistication over the last decade. Today, direct volume rendering of data sets as large as 512^3 and beyond is possible at fully interactive rates on commodity desktop systems, and especially due to the rapid advancements in graphics hardware technology, these capabilities are continually increasing.

Volume rendering is a powerful means for visualizing 3D scalar fields, and especially if used in combination with semi-automatic transfer functions and different rendering styles does it allow for an effective visual communication of complex structures in such fields and relationships between them. To improve the analysis process in practical applications, however, it is often desired to not only render the data but also to interactively edit this data. Examples thereof include the manual classification and segmentation of structures, the removal of structures to uncover regions of interest and, thus, to isolate important parts of the data, or the coloring of parts to emphasize relevant structures and to give extra information about them. Such mechanisms can help to effectively reveal and communicate the relevant information in 3D scalar fields and to create images that are easy to understand even by an unexperienced user.

Today, the core functionality that is required to support the aforementioned mechanisms is available on recent GPUs. Specifically, it is now possible to directly write into 3D textures on the GPU, and to efficiently apply local operations on the data stored in these textures, such as filtering or gradient computation. Thus, the time is ripe for opening a new area in volume visualization, which is concerned with the development of techniques for interactive volume editing. One of the research challenges here is to develop novel algorithms that are tailored to the specific GPU functionality, and which can directly be incorporated into interactive volume rendering tools to enable immediate visual feedback.

The approaches presented in the following were motivated by a number of different *volume illustration techniques* that have been proposed over the last decade. Many of these techniques have been integrated into GPU-based volume rendering systems to achieve interactive user-control. Interrante et al. [67, 66] used curvature-directed strokes and dense sets of integral curves to convey surface shape. A general volume illustration rendering pipeline to enhance important features and regions was proposed by Ebert and Rheingans [38]. Viola et al. [180] suggested importance driven volume ren-

dering to highlight interesting structures in volume data based on user-selected object importance. Different rendering styles including point stippling [109, 93], temporal domain enhancement [110], 2D texture synthesis on cross-sections of a volumetric model [121], and volumetric halos to improve depth perception of 3D structures [18] have been used to enhance the expressiveness of volume visualizations. A new approach that uses the shape of the object to be illustrated to control its rendering styles, and which also allows to adapt the objects shape to a given curve skeleton, was presented in [30].

Especially if used in combination with focus+context techniques to combine multiple aspects of the data into a single visual event [179, 63, 16, 91, 17], illustrative volume rendering has been shown to be very effective in communicating the essential information in complex volumetric data sets. An interactive system providing a toolbox of automatic illustration methods as well as focus+context mechanisms to enable selective exploration of volume data was presented by Bruckner et al. [19]. In particular, they introduced external, screen-space aligned annotations to add extra information about particular structures and selection volumes to locally modulate the appearance of a volume. Our work builds on these mechanisms and extends them towards a more general use for volume illustration.

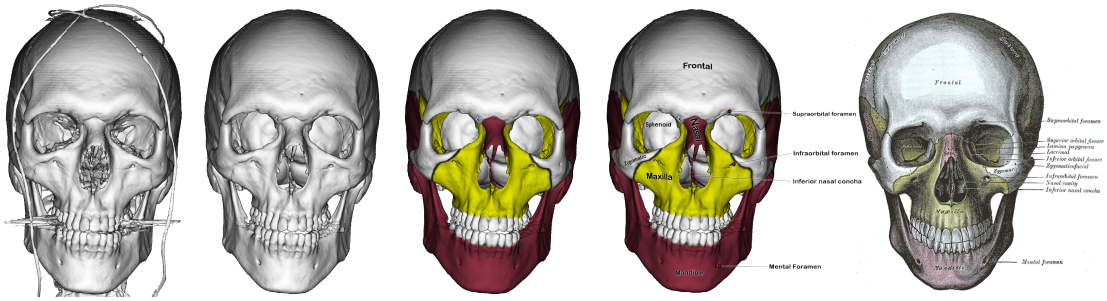


Figure 9.1: *A volume editing session. From left to right: an iso-surface in the initial data set, structures are removed, surface color is applied, annotations are added. The rightmost image is taken from the classical anatomy book "Gray's Anatomy" by Henry Gray [53] for comparison.*

9.2 Contribution

The primary focus of this chapter is the development of fast and flexible methods for user-guided volume editing, such as coloring, erasing, pasting, segmentation, and annotation. Our goal is to realize a volume processing tool exhibiting similar functionality to current image processing tools, which allow the user to interactively perform

a multitude of image adjustments and enhancements. To achieve interactivity, all of the algorithms proposed in the following run entirely on the GPU, and they have been integrated into a GPU-based volume ray-caster to provide immediate visual feedback. We introduce some novel ways to leverage advanced GPU functionality like geometry shaders and the possibility to directly render into 3D textures, and we effectively exploit computational and bandwidth capacities on recent GPUs. Therefore, all of the editing operations demonstrated throughout this chapter were executed at frame rates of 50 fps and higher. Thus, a framework for visibility-guided interactive volume editing is presented.

Some of the editing techniques we introduce can effectively be used for volume illustration, where the basic goal is to enhance the perception of structures in the data and the relationships between them by emphasizing important features. In particular, we extend the work on direct volume illustration by Bruckner and Gröller [19], in that we provide a technique based on particle-tracing in a gradient field to annotate structures in a volume data set.

We make the following specific contributions:

- We present an efficient GPU realization of the volume painting method proposed by Bruckner and Gröller [19], and we demonstrate the use of this method for interactive volume coloring as well as structure elimination and enhancement. This method was used in Figures 9.1 to color an iso-surface, to erase parts of it and to add additional structures to it.

- We extend the idea of selection volumes and present a volume editing technique that is independent of the volume resolution. It edits on a high-resolution selection volume and can, therefore, be used to apply editing effects at sub-voxel accuracy. Figure 9.4 (c) demonstrates editing effects on an iso-surface in the initial volume and a high-resolution selection volume.

- We introduce *surface particles* to compute a local iso-surface parametrization. By using such particles, 2D textures can be mapped onto an iso-surface. This allows to generate internal annotations that are aligned with an iso-surface, and which can effectively be used to give additional information about areal structures visible in the current view. Figure 9.7 depicts two classified iso-surfaces which are enhanced by surface-aligned annotations.

- Building on the concept of surface particles, we present surface-aligned "see-through" textures to generate windowed cutaways on iso-surfaces in 3D scalar

fields. By using such textures, occlusions can effectively be reduced and important internal parts of a volume can be exposed. This method was used in Figure 9.8 to interactively generate cutaway views in the respective data sets.

The remainder of this chapter is organized as follows. Our proposed volume editing technique and its efficient realization on recent GPUs is presented in Section 9.3. In Section 9.4 we present high-resolution selection volumes and demonstrate their use for sub-voxel accurate volume editing. Section 9.5 introduces particle tracing along isosurfaces with respect to a user defined force and the gradient field of the underlying scalar volume to create structure-aligned textures for volume augmentation and annotation. We conclude this chapter with a discussion of the advantages and limitations of our work.

9.3 Volume Editing

The specification of appearance properties of volume data is typically performed via color transfer functions. Based on the seminal work by Kindlemann and Durkin [76] on the design of feature-specific transfer functions that can be derived automatically from a data classification using first- and higher-order statistics, such approaches have now been developed to a high degree of sophistication. Nevertheless, automated classification of volumes remains a challenging task, and semi-automatic techniques which allow the user to interactively guide the classification process often result in a more accurate assignment of appearance properties. Examples thereof include the user-guided selection of seed voxels to initialize automated region-growing [87] or more sophisticated segmentation algorithms like the random walker [52], the dual-domain approach of Kniss et al. [82, 81], or the machine-learning approach by Tzeng et al. [168], where a transfer function is iteratively refined from user-defined segmentations in 2D volume slices.

To support semi-automatic classification and segmentation of 3D volume data we now describe an interactive technique for voxel coloring. This technique works in the 3D domain, and it thus allows the user to consider the 3D shape of the structures to be colored as well as the spatial relationships between them. Figure 9.2 (c) demonstrates the application of this approach for the classification of a human skull. The proposed technique has been integrated into a GPU-based volume ray-caster, enabling the user to obtain immediate visual feedback about the result of the issued operations. Later in the text we show how to overcome the restriction of volume coloring to the initial volume resolution by exploiting selection volumes for coloring at sub-voxel accuracy.

9.3.1 3D Texture Painting

Initially, a 3D scalar field of size T_x, T_y, T_z is loaded into a 3D texture—the source texture—on the GPU. Scalar values are mapped to color and opacity via a selected transfer function. If the user only wants to paint on an iso-surface in the scalar field, a one component 3D texture is used instead of a RGBA texture. Coloring always works on an additional 3D texture—the color texture—on the GPU, into which the user paints with the selected color. In iso-surface coloring this texture is initialized with a constant material color, otherwise it is initialized with the source color values. Working on such a copy allows for a special paint mode in which the paint operation resets the color by copying respective values from the source texture. In iso-surface painting, colors are reset by zeroing.

The 3D color texture is rendered using texture-based volume ray-casting [92], i.e., by sampling the texture along the rays of sight and by blending color and opacity contributions according to the selected blend equation. In iso-surface rendering, sampling is performed in the scalar source texture. Once the iso-surface is hit along a ray, the surface normal at this position is fetched from a pre-computed gradient volume and a local lighting model is evaluated. In this model, the color at the sample position in the color texture is used as material color.

Upon initialization, the user starts painting the volume with a virtual brush. To position the brush in 3D space we either use a simple mouse-based interface or a six degree-of-freedom input device, i.e., a PHANToM Desktop Device Premium 1 from Sensable Technologies. This also allows us to give haptic feedback to the user, e.g., while painting on an iso-surface we use the force feedback to indicate whether the brush touches the surface. To detect a contact between a surface and the brush we simply test the brush center point for being in close proximity to the surface, i.e., by sampling the volume at this point and testing whether the value is closer to the iso-value than a given tolerance. If this is the case, force feedback along the inverse gradient direction at this point is issued.

In our work we use a spherical volume brush for painting, which means that voxels closer to the brush center point than the selected sphere radius are painted with the current paint color. To manipulate the color of a voxel at position $\mathbf{q}$, indicated by Color_q, we use the paint equation proposed in [159, 59]:

$$\text{Color}_q = \text{lerp}(\text{Color}_b \ \mathbf{OP} \ \text{Color}_q, \text{Color}_q, g), \tag{9.1}$$

The brush shape g is set such that a spherical color falloff with increasing distance

to the brush center point is simulated:

$$g = \begin{cases} 0, & \text{if } |\mathbf{q}-\mathbf{p}| > r \\ f(|\mathbf{q}-\mathbf{p}|) & \text{else.} \end{cases} \tag{9.2}$$

Here, **OP** is one of a number of operations like REPLACE, ADD, or BLEND, which can be selected to modulate the initial volume color, $\mathbf{p}$ is the position of the center point, Color_b is the brush color, and r is the support of a user-defined falloff function f, which is used to simulate smooth color fading.

When using a volume brush to color a volume data set, the color of every voxel contained in the brush volume has to be updated according to the selected color modulation function. In principle, the color update can be performed on the CPU, requiring the modulated texture to be reloaded onto the GPU. Even if it is possible to only replace those parts of the GPU texture that were affected by the coloring operation, this strategy still results in significant bandwidth requirements due to frequent data uploads to the GPU in the course of painting. To overcome this limitation, we propose a novel technique—similar to the approach presented in Section 8.5.1—that runs entirely on the GPU and minimizes CPU-GPU data transfer.

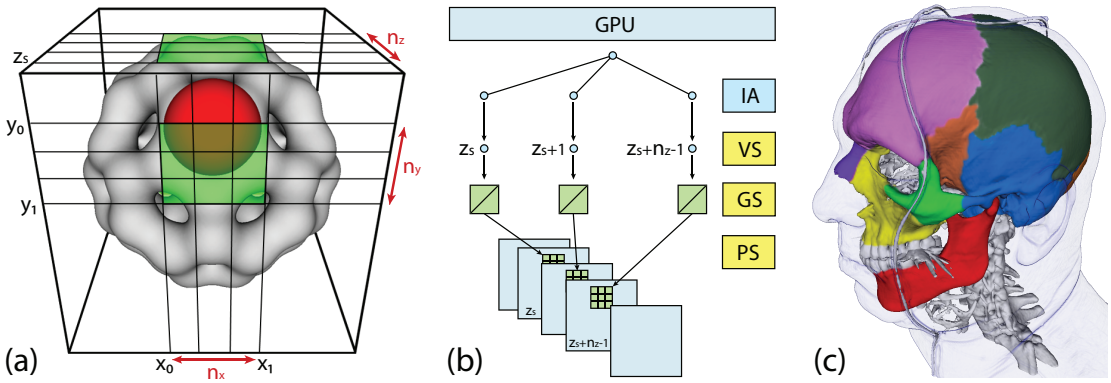


Figure 9.2: *Volume editing with a spherical volume brush. Image (a) depicts a spherical brush (red) positioned in the volume domain, and (b) illustrates the pipeline setup for painting into a 3D texture on the GPU. A single vertex is issued by the application program, and it is duplicated by the input assembler. In the geometry shader, every point is amplified to one quadrilateral, which in turn is sent to the rasterizer. The rasterizer uses slice IDs to route generated fragments into corresponding 3D texture slices. In the pixel shader the fragments are colored with respect to the selected modulation function. In (c) an iso-surface classified with our method is shown.*

9.3.2 GPU Implementation

To efficiently update 3D texture elements that are affected by a coloring operation, we exploit novel features of current Direct3D 10 class graphics hardware. Specifically, we use the geometry shader to create geometry on the GPU, we employ new functionality to update slices of a 3D texture directly on the GPU, and we utilize instanced render calls to reduce the number of calls that have to be issued from the application program. In Figure 9.2 an overview of the pipeline setup for rendering into a 3D texture is shown.

Before the painting process is started, the user selects the specific brush parameters including the cutoff radius r used in Equation 9.2. From this radius the extend of the brush bounding box in local texture coordinate space is computed, yielding the size $n_x \times n_y \times n_z$ of the sub-volume that is affected by the coloring operation. These values are computed on the CPU and sent to the GPU as constant shader variables. To compute the position of the brush center point $\mathbf{p}_c$ in local texture coordinates in the range [0,1], we either use the coordinate returned by the 3D input device, or, in iso-surface painting, it can also be determined from the z-buffer depth value in the pixel under the mouse cursor.

The application program then renders into a viewport of size T_x, T_y. A single vertex—with a coordinate equal to $\mathbf{p}_c$ scaled by T_x, T_y, T_z—is sent to the GPU, where it is rendered as instanced geometry with instance count n_z. This causes the GPU to generate a stream of n_z vertices, all of which carry the position $\mathbf{p}$ and an *instance ID* in the range $[0, n_z - 1]$. These vertices are passed through the vertex shader to the geometry shader, which, for each incoming vertex, spawns a quadrilateral centered at $\mathbf{p}_x, \mathbf{p}_y$ and covering $n_x \times n_y$ pixels. The ID of the 3D texture slice into which this quadrilateral is to be rendered is computed as

$$SID = \mathbf{p}_z - \frac{n_z}{2} + IID, \tag{9.3}$$

where *IID* is the instance ID of every vertex. This slice ID is used by the rasterizer to direct the fragment into the corresponding z-slice of the 3D texture. In the pixel shader, for every fragment its distance to the brush center is computed and Equations 9.1 and 9.2 are evaluated. Updated color values are then written into the respective position of the 3D color texture slice, and the updated texture can immediately be used in the rendering pass.

9.3.3 Structure Removal and Enhancement

The method proposed in the previous section can efficiently be used to paint color into a volume. Moreover, it provides a means to interactively erase parts from the

volume and to add new structures to it. Erasing is performed by painting voxels with zero opacity, thus making structures completely transparent. Even though the erasing operation is conceptually simple, it does provide a very powerful means to interactively create cutaway views. In particular it can be used when traditional volume cutaway techniques have difficulties, e.g., when occluded and occluding structures are close together and have similar material properties. Figure 9.3 (d) shows such a case and a cutaway view that was generated by our method. Without using a data segmentation or a highly detailed clip geometry that can accurately separate structures from each other, in such scenarios the automated generation of a cutaway view remains a challenging task.

Modification of structure is realized by a slight change of the color modulation function. Instead of replacing or modulating the colors stored in the 3D color texture, a density offset is painted into the scalar source volume. By adding (or subtracting) offsets of different strength, size and shape, a number of editing effects can be achieved (see Figures 9.3 (a–c)).

When erasing or changing density values, surface normals have to be updated if iso-surface are rendered. This is accomplished by a) finding all voxels in the pre-computed normal map that are contained in the brush volume, b) re-computing the normals using central differences in the source volume, and c) writing updated normals into the normal map. Steps a) and c) are performed in exactly the same way as described for volume coloring, with the only difference that the brush volume has to be enlarged by one voxel in each dimension to capture all affected voxels.

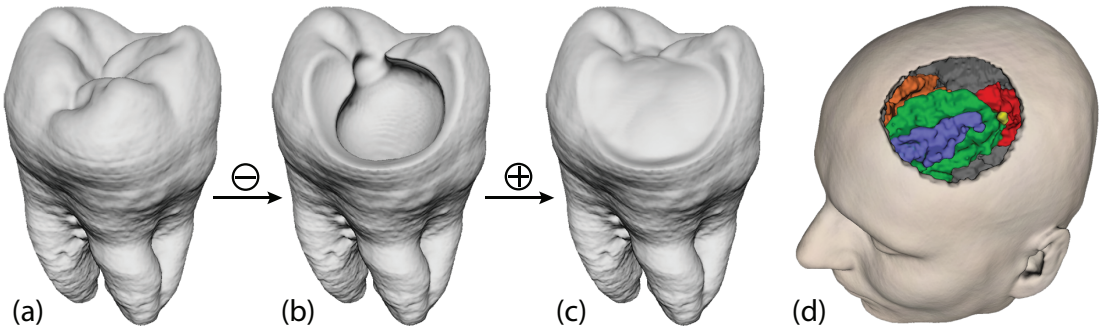


Figure 9.3: *Structural editing. Images (a–c) show (from left to right) the original data set, interior regions excavated by density reduction, the filled hole by adding structure. In (d) parts of a bone iso-surface in a MRI data set were erased manually to reveal interior brain structures.*

9.4 Selection Volumes

The volume coloring method as described so far restricts the accuracy of the coloring process to the resolution of the given volume data set. This allows one to assign voxel properties on a per-voxel basis, but the method is not capable of assigning such properties at sub-voxel accuracy. On the other hand, in particular if color painting is used to manually segment objects in the data, sub-voxel accuracy is required to determine correct segment boundaries. Similar to surface-based segmentation methods, where the mesh is not constrained to lie on voxel boundaries, our goal is to provide a much higher spatial resolution in regions where the user expects voxel-based classification to fail.

For this purpose we use selection volumes as introduced by Gröller [19], who stated that "A selection volume specifies a particular structure of interest in a corresponding data volume. It stores real values in the range [0,1] where zero means not selected and one means fully selected". A selection volume has the same spatial resolution as the original volume and its voxel values are used to modulate the initial data values. To make selection volumes applicable for data segmentation, we extend them in several ways: Firstly, in addition to extent and position the user can select the resolution of the selection volume. Secondly, the selection volume is "filled" with data values by resampling the source texture. It can thus be seen as an upsampled version of a sub-volume, and it is accompanied by a color volume of equal resolution to support voxel editing. Thirdly, the GPU volume ray-caster, which is used to render the original volume and the selection volume in combination, is adapted appropriately. This means, that the ray-caster not only finds the intersection points between the rays and the selection volume but also adapts the step size within this volume to its resolution. In iso-surface rendering, a uniform step size is used to avoid cracks at selection volume boundaries.

In Figure 9.4, we illustrate the use of selection volumes for sub-voxel classification, segmentation, and modeling. The leftmost image shows two voxel-sized structures that have been segmented manually in a selection volume. Due to the increased resolution of this volume, object boundaries can be resolved at very high accuracy. In the middle images, structures in the interior of a volume were classified by using a particular color transfer function. In the right image a high-resolution selection volume was used to obtain smooth structure boundaries. The rightmost image shows the effect of iso-surface enhancement in a high-resolution selection volume and the low-resolution base volume. Text was painted onto an iso-surface by manually adding density offsets into the respective source textures.

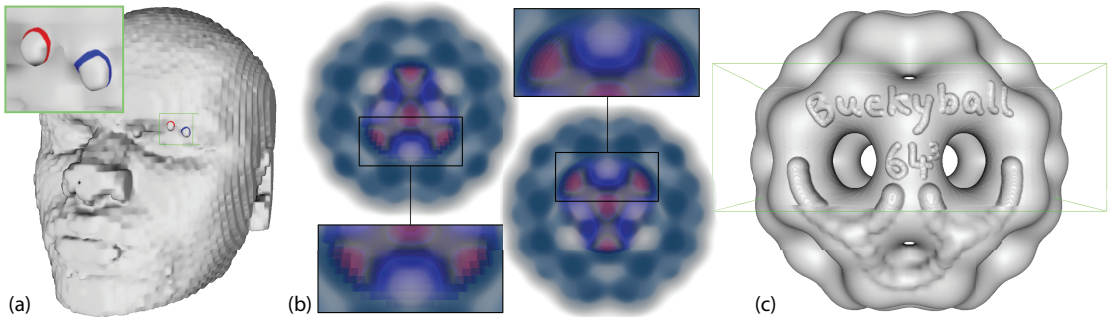


Figure 9.4: *The use of selection volumes is demonstrated: (a) Two small features are segmented at sub-voxel accuracy. (b) A sub-volume at the initial (left) and a much higher resolution (right) is rendered with a different transfer function (center region) than the initial volume. (c) Editing effects on an iso-surface in the initial volume and a high-resolution selection volume.*

9.4.1 Upsampling

To build a selection volume two different strategies are pursued. For direct volume rendering, voxel colors are trilinearly interpolated in the initial color texture. For iso-surface rendering, a piecewise quadratic tensor product spline is used for resampling the source texture (see Figure 9.5 (a)). This results in a C^1-continuous quasi-interpolant exhibiting a smooth gradient field.

Denoting initial samples with v_i in voxel coordinates (i.e., ranging from 0 to $N-1$ for N voxels), additional samples at positions $x \in [i-0.5, i+0.5[$ are computed in two steps. First, intermediate values $A := 0.5\,(v_{i-1}, v_i)$ and $B := 0.5\,(v_i, v_{i+1})$ are computed. Then, a quadratic Bézier-spline with the control polygon A, v_i, B is constructed using the DeCasteljau algorithm. Thus, at x the associated index i has to be computed first by rounding to the next integer, i.e.,$i := \lfloor x+0.5 \rfloor$. The parameter p_i at which to evaluate the spline is then given as $p_i(x) := 0.5 + x - i$. Observing that the interpolation to compute A is collinear with the interpolation between A and v_i (and analogously for B and v_{i+1}), only two linearly interpolated fetches are necessary. These fetches can be performed by the GPU as $A' := \mathrm{lerp}\,(v_{i-1}, v_i, 0.5 + 0.5 \cdot p)$ and $B' := \mathrm{lerp}\,(v_i, v_{i+1}, 0.5 \cdot p)$, where $\mathrm{lerp}(a,b,c) := a + c \cdot (b-a)$. Finally, the second stage of the DeCasteljau algorithm to yield the final value $v_{res} := \mathrm{lerp}\,(A', B', p)$ is computed in a pixel shader.

Since the interpolated nodes lie halfway between the samples of the initial volume, we introduce a transition region that is half a voxel wide (with respect to the initial grid). In this region, trilinear interpolation in the source texture is performed to guarantee C^0 continuity between the selection volume and the source volume. In the interior, the selection volume is built by tri-quadratic quasi-interpolation in the source texture,

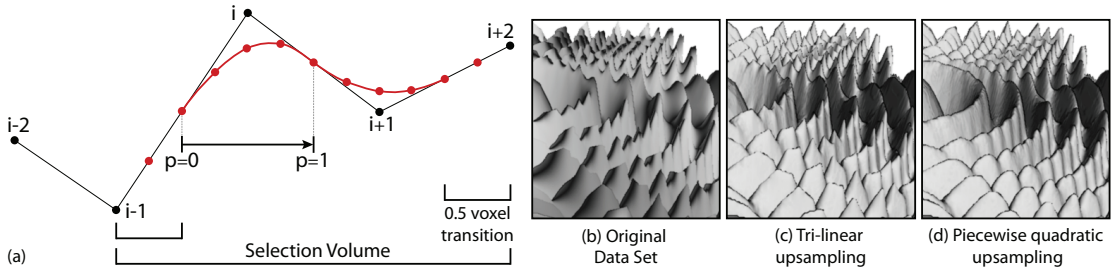


Figure 9.5: *Illustration (a) depicts the piecewise quadratic spline used for upsampling. Images (b–d) show different resampling results. In (b) the original (high-frequency) Marschner-Lobb [111] data set is shown. Ideally, concentric waves should be visible. However, due to filtering errors, these waves are distorted to a certain extent. Image (c) depicts a selection volume resampled with a trilinear filter kernel and (d) our method, respectively.*

and a smooth normal map is computed on-the-fly from this volume. Figure 9.4 (d) demonstrates the fine editing details that can be achieved by applying the operations described so far on a high-resolution selection volume.

In general, selection volumes can be used to add fine structures or color details to a 3D volume or an iso-surface in it. Selection volumes can thus be used to directly paint additional text on a surface, which provides a general means for adding surface-aligned annotations. However, as writing text on a curved surface in 3D is rather cumbersome, we propose an alternative GPU method to automatically align 2D textures containing text or other annotations on an iso-surface. For a good description of the process to be used to automatically place screen-space annotations we refer the reader to [19].

9.5 Surface Particles

We start our description by introducing GPU surface particles, which are used to map a 2D grid consisting of vertices and edges between them onto an iso-surface, i.e., to find a local surface parametrization. Our approach is similar in spirit to the one proposed by Ropinski et al. [135], but, in contrast, it is performed directly in 3D object space, and it operates entirely on the GPU. The 2D grid is rendered on top of the iso-surface as a textured polygon mesh. The texture contains the annotation to be used, for instance, a bit-mapped text or a pattern indicating a particular property.

A surface-particle can be thought of as a particle moving on the surface along regular patterns to approximate a local surface paramerization. The direction of the movement is given by an external direction field that is defined by the user when placing the annotation. In any case, to move a particle $\mathbf{x}$ on the surface we compute its trajectory

in a vector field $\mathbf{v}$, starting at an initial position $\mathbf{x}_0$ on the surface. This requires to solve the ordinary differential equation given in (2.1). It is clear, that in general the numerical integration brings away the particle from the surface. Even if the vector field is everywhere defined in the local surface tangent plane, a particle is moving away from the surface in non-planar regions. To avoid this behavior, after every integration step we trace the particle back onto the surface, resulting in the following steps that have to be performed:

- **Integration** From the previous particle position, $\mathbf{x}$, and the velocity at this position, $\mathbf{v}$, the new position $\mathbf{x}'$ is computed on the basis of Euler integration (see Eq. 2.3). In the very first iteration $\mathbf{v}$ is set to zero.

- **Backtracing** $\mathbf{x}'$ is corrected by tracing the particle back onto the selected iso-surface.

- **Vector lookup** The velocity vector $\mathbf{v}$ at position $\mathbf{x}'$ is determined. This can be as simple as a texture lookup into a 3D vector field, or a 2D vector field if a surface parametrization exists, or it can be a more complex computation such as a curvature estimation.

While it is clear how to perform particle integration and vector lookup, the method to trace particles back to the surface requires some further explanation. In principle, moving it back onto the surface would require to bend the line segment connecting the current and the fixed previous particle position around the surface, thereby constraining the bending to the plane defined by this line segment and the surface normal at the previous position. Since this approach requires some exhaustive computations, we approximate it by iteratively correcting the current position towards the surface, thereby assuming the surface to be locally flat. Figure 9.6 (a) illustrates this approximation for a particle that has left the surface after integration.

Back-tracing is performed by using the surface normal at the previous position, i.e., the gradient of the scalar field at this position, scaled by the difference between the scalar values at the previous and the current position. The direction of this vector determines whether the current position is inside the surface or outside. Note that using the normal at the current position is not feasible in general, since this point is not on the surface and the normal at this point may be affected by noise. Given this direction, the current particle is traced from the current position into this direction until the difference between the scalar values at the corrected position and the selected iso-value drops below a user-given tolerance. In this case we have reached the surface and terminate

the correction. If the particle crosses the iso-surface, which is indicated by increasing difference between the scalar value at the particle position and the iso-value, the step size is halved and the trace is restarted at the last position.

The accuracy of the proposed method depends on the local curvature of the iso-surface. The less planar the surface is, the higher can be the length distortion of a line segment connecting the previous and the current point. The reason therefore is, that we only consider the normal at the previous point to determine the direction into which the particle is corrected. This problem could be alleviated by also considering the curvature direction in the plane spanned by the previous surface normal and the advection direction, but as the step size we use for particle integration is typically small, i.e., in the order of the voxel size, in our experiments length distortions did not result in any noticeable artifacts.

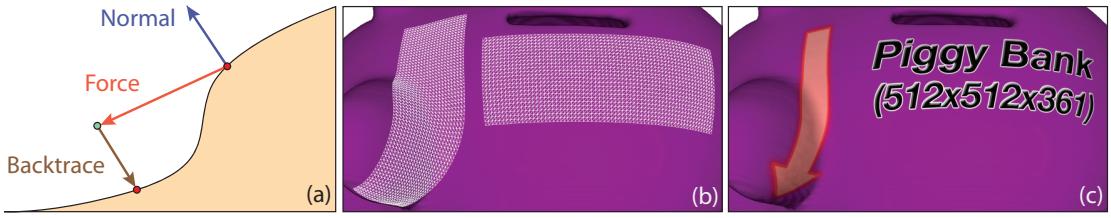


Figure 9.6: *In (a) one particle advection step is illustrated: Firstly, the particle is moved into the direction of the vector field (red) to an intermediate position (green). In the next step it is traced into the direction of the previous normal vector until it reaches the surface. Images (b) and (c) show surfaces aligned annotation in wireframe and textured with a bit-map image, respectively.*

9.5.1 Volume Annotations

Volume annotations in the form of arrows and labels have a long history in hand-made technical and medical illustrations. Textual annotations are typically used in two different ways. They are either placed directly on the surface of a structure—aligning their shape to the surface shape—or they are placed in screen-space close to the image of a structure, and they are then connected to the structure with a line. In general, the former method has the advantage that annotations remain fixed to a structure when the user interacts with the volume, while free-floating labels have to be rearranged in screen-space to avoid overlapping annotations, crossing of connecting lines, or placements too far away from the structure. Free-floating annotation, on the other hand, are advantageous for pointing to small structures which do not cover enough space on screen to allow the user to read the annotation on it. Therefore, our system supports both approaches to annotate volumes, and it thus allows the user to flexibly select the appropriate choice.

By using surface particles we can now construct a regular grid, which is aligned with an iso-surface and can be textured with an arbitrary annotation. As the process is performed entirely on the GPU, the user can interactively place high-resolution annotations in the volume. To start the process, the user first selects a texture, the annotation texture, which is to be used as annotation. Then, some additional information has to be specified:

- The position on the iso-surface where the annotation is to be centered.
- The orientation of the annotation.

To specify the annotation center point the user picks a point on the iso-surface. The orientation of the annotation texture is specified by picking a second point and by interpreting the vector from the first to the second point as the u-axis of the local (u,v) surface parametrization. In the following, we will call this vector the orientation vector. Given this information, a set of surface particles is traced to generate a grid that is aligned with the surface.

At first, two surface particles are spawned at the annotation center point. One of them is traced along the orientation vector, and the other one is traced into the inverse direction. Both particles are traced for a number of equidistant steps and with the help of a geometry shader and the stream output stage, their intermediate positions are streamed into a buffer residing in GPU memory. Both the number of steps and the step size in voxel units can be selected by the user.

At every particle position the direction vector moving the particle along the surface is computed from the direction vector at the previous position. Starting with the normalized projection of the orientation vector into the tangent plane at the annotation center point, at every upcoming position the same procedure is performed with the previous direction vector. That is, for a particle at position $\mathbf{x}_u$ we compute a tangent frame consisting of three mutually orthonormal vectors: $\mathbf{n}$, the surface normal, $\mathbf{v}$ the direction vector in the local tangent plane, and the binormal $\mathbf{b} = \mathbf{n} \times \mathbf{v}$. During particle integration, $\mathbf{v}_u$ is updated as follows:

$$\mathbf{v}_u = \frac{\mathbf{v}_{u-1} \times (\mathbf{n}_u \times \mathbf{v}_{u-1})}{\|\mathbf{v}_{u-1} \times (\mathbf{n}_u \times \mathbf{v}_{u-1})\|} \tag{9.4}$$

I.e., the force vector is updated by projecting it into the surface tangent plane at the new position. Surface normals are computed by trilinear interpolation of the gradients at adjacent voxel centers. Finally, the particle is advected using $\mathbf{v}$ and it is then traced back to the surface as described in the previous paragraph.

After the two surface particles that were released at the annotation center point have been traced for i steps, a number of $2i+1$ surface points are stored in a GPU render target. If these points are connected, they form a line on the surface, which is centered at the annotation center point and oriented along the annotation direction. To expand this "line" to a full 2D grid, at every point we trace two additional surface particles into direction **b** and into the inverse direction. Tracing these particles for j steps results in a set of $(2i+1)\cdot(2j+1)$ points, from which a regular triangular annotation grid is built (see Figures 9.6 (b, c)). All grid points are rendered into a vertex buffer, which is then used to render the grid using an appropriate index buffer residing in GPU memory. The grid is textured with the selected annotation texture, and it is rendered before ray-casting the volume to initialize the depth buffer. To avoid depth fighting between the iso-surface and the annotation grid, the grid is slightly shifted towards the viewer.

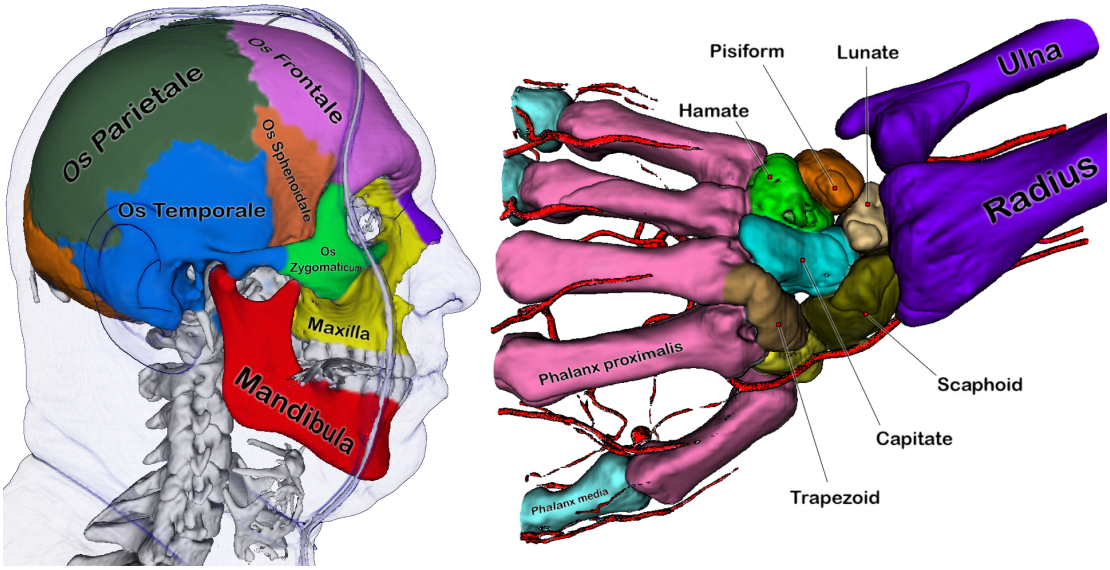


Figure 9.7: *Two annotated data sets are shown. Left: A focus+context visualization of the visible human head, colorized and annotated with the presented techniques is shown. Right: An annotated human hand. Here, next to internal surface aligned decals, external labels were used to annotate the data set.*

9.5.2 Windowed Cutaway Views

In this section, we show how to efficiently create a shape-aligned windowed cutaway on an iso-surface by exploiting an annotation grid as introduced before. In technical illustrations, cutaways are often used to reduce occlusions and expose important internal parts. There is a vast body of literature related to this issue that we will not attempt

to overview here, however, Diepstraten et al. [35] and Li et al. [102] discuss some of the mechanisms to automatically generate cutaway views and provide many useful references on this subject.

Starting with such a surface-aligned grid, we proceed in two stages. Firstly, we duplicate the mesh and displace the vertices of the copy along the inverse surface normal direction at the center vertex. The length of the displacement can be selected by the user to generate thin or thick cutaway sections. Secondly, both meshes are connected along their borders to build a closed mesh. This mesh is then used as a clip geometry as proposed by Weiskopf et al. [182], and it is directly incorporated into the texture-based volume ray-caster.

Prior to ray-casting, we render a layered depth-buffer of the mesh from the current view. During volume rendering, every ray first samples these buffers and then tests all samples along the ray for being inside or outside the mesh, i.e. by testing whether a sample is in-between a front and a back face of the cutaway mesh. Samples inside the mesh do not contribute to the final ray color, thus cutting away the volume contained in it. Figure 9.8 demonstrates the use of shape-aligned cutaways to expose internal parts of a volume.

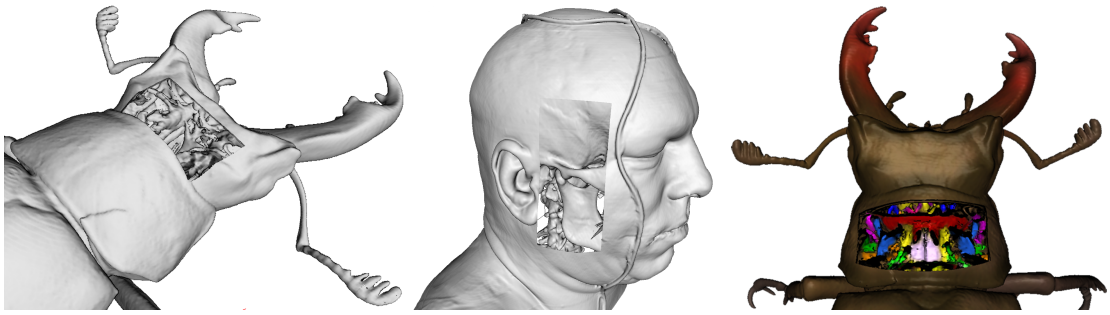


Figure 9.8: *Several windowed cutaway views are shown. Right: By integrating this metaphor into the paint environment, classification based on iso-surface coloring becomes possible even without erasing information in the underlying data set.*

9.6 Performance Analysis

Throughout this chapter we have shown a number of different effects that were generated by the proposed volume editing techniques. A typical use of these techniques is demonstrated in Figure 9.1, where a human skull data set was interactively processed and augmented to obtain an illustrative image as shown in “Gray’s Anatomy” [53]. In

the following, we investigate the performance of these techniques in more detail. Timings were performed on a 2.4 GHz Core 2 Duo processor and an NVIDIA 8800GTX graphics card with 768 MB local video memory. Image generation was done at 1280×1024 resolution. Regardless of this extreme resolution, for all models shown we achieve real-time performance with update rates of 50 fps and higher, including editing and rendering.

All brush-based editing effects like coloring, erasing, and adding, as well as resulting normal map updates, were executed in less than 3 ms up to a brush extend of 64^3 voxels. The times it takes to build a selection volume at different resolutions, i.e., from $(3 \times 2)^3$ to $(64 \times 8)^3$, is given in Table 9.1. As can be seen, even at a resolution as high as 128^3, GPU-based resampling is still capable of achieving interactive rates.

	Covered voxels				
Scaling	3^3	11^3	19^3	32^3	64^3
2	0.14	0.19	0.24	0.51	2.7
4	0.16	0.31	0.76	2.5	17.9
8	0.2	1.0	4.29	17.1	134.6

Table 9.1: *Timing statistics for tri-quadratic iso-surface and trilinear color resampling. All times are given in milliseconds.*

Finally, we measured the time it takes to construct a surface-aligned annotation grid by means of the method described in Section 9.5. Table 9.2 shows respective times for varying grid sizes. From these timings it can be concluded that the proposed method is fast enough to allow for interactive placements of annotation textures on high-resolution surface structures. In particular, since the rendering of these textures only consumes an insignificant amount of time, many of them can be used simultaneously on a single object.

Gridsize	11^2	21^2	41^2	81^2
Time (in ms)	1.6	2.0	3.6	14.7

Table 9.2: *Timings for the construction of surface-aligned annotation grids.*

9.7 Summary

In this chapter, we have presented a number of GPU-based techniques for interactive volume editing. By efficiently using novel functionality on recent GPUs, we have developed a technique for interactive volume painting. We have further shown that this

technique provides a powerful means to erase structures in a volume and, thus, to isolate features in it. In combination with high-resolution selection volumes these techniques can effectively be used for manual volume segmentation at sub-voxel accuracy. We have also introduced structure-aligned annotations on the basis of particle-tracing along iso-surfaces—with respect to the underlying scalar volume's gradient field—to supplement classical free-floating annotations that are placed in screen-space, and we have demonstrated how to utilize this approach to interactively create windowed cut-away views. In particular, as all of these operations are performed in the 3D domain, with immediate visual feedback provided, they are very intuitive to use and allow the user to quickly observe the relationships between relevant features in the data.

In the future we will further extend some of the proposed techniques: Firstly, we will develop semi-automatic volume segmentation techniques by combining manual segmentation as proposed with automatic techniques on the GPU (such as the random walker approach). We believe that such a combination can considerably improve the segmentation process, both with respect to accuracy and speed.

Chapter 10

Conclusion

This thesis presented techniques for the interactive visual exploration of time-resolved 3D unsteady flow velocity fields. Feedback from scientists in various fields has confirmed that the developed real-time exploration techniques are well-suited to gain insight into complex flow phenomena. An interactive exploration environment enables experts to incorporate their experience into the visual data analysis process and to exploit their perceptual and cognitive abilities to detect relevant features in the flow.

All approaches discussed in this work can be employed on consumer class hardware and are, thus, available to a wide range of users. As the size of 3D unsteady flow data sets usually exceeds the memory capacities of standard PCs, we have developed a multi-core approach to asynchronously manage the time steps needed during an interactive flow exploration session. By decoupling visualization from data handling, this concept does not only result in interactive frame rates but also allows the visualization of an unlimited amount of time steps. Since flow visualization techniques generally require the application of numerical operations to a large amount of individual samples in the data, we have presented parallelization strategies that effectively exploit the computational processing power of recent graphics processing units to achieve the feature extraction and subsequent visualization in real time.

We have shown how Lagrangian particle tracing can effectively be mapped onto the GPU to allow for the integration of a huge number of particles in parallel. We have presented various rendering modalities to encode additional flow quantities into the visual representation of each particle and have developed mechanisms to automatically restrict their display to important regions in the flow. This allows to reveal phenomena of interest, while at the same time preserving context information. Furthermore, we have employed the particle tracing paradigm to extract geometric flow representation

such as characteristic trajectories and adaptive integral surfaces interactively, and have presented a variety of rendering modalities including focus+context approaches to reduce the presented visual information to relevant features in the flow. We have extended the particle tracing paradigm to flow on arbitrary surfaces and have developed a variety of geometry- and texture-based visualization techniques for such flow fields.

Feature-based visualization techniques are well-suited to reduce the flow data to physically meaningful patterns. However, due to the intense pre-processing required by these techniques to achieve the data reduction, such approaches are generally not suited for an interactive exploration environment. Yet, we have shown how certain concepts from this class can efficiently be combined with geometry-based flow visualization techniques to effectively study large-scale transport behavior.

Moreover, we have discussed how the massive parallel processing power of modern GPUs can not only be exploited to explore large 3D data sets, but also to manipulate them interactively. This allows scientists to encode findings directly into the data set or a visual representation of it and, thus, to communicate the obtained insight intuitively.

10.1 Future Work

None of the presented techniques are inherently restricted to flow fields sampled onto uniform grids, however, we have only validated them in such data. We aim at extending our system to support unstructured time-resolved 3D unsteady flow fields, as such data sets are of practical importance. While the extension seems straightforward, as only the underlying data structure has to be exchanged and even GPU-based concepts for particle tracing in unstructured grids are available, all existing methods adhere to rather outdated graphics API standards and lack in performance. However, GPU manufacturer have noticed an increasing interest in the scientific community for their platform, and the capabilities of recent graphics hardware and related graphics APIs are evolving towards more generalized computing architectures. Thus, it is of interest to investigate how new GPU capabilities can be exploited to develop more efficient data structures and algorithms that allow for fast point location and interpolation in unstructured grids.

The current framework achieves interactivity due to the fact that all time steps needed by the visualization system at a given point in time reside in local video memory. As numerical capabilities continue to increase, so does the size of the data sets to be visualized. However, the typical amount of memory generally does not scale with the growth in processing power and, thus, it will be a challenging task to develop in-

teractive (distributed) visualization strategies that can cope with flow fields that exceed the locally available memory.

As the numerical processing power of GPUs still grows exponentially, it will also be of interest to adopt even more concepts from the class of feature-based visualization into interactive visualization techniques. As we have shown, such a combination is well-suited to support the user in finding relevant features in the flow.

Bibliography

[1] Comparison of ATI graphics processing units. Wikipedia: http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units/.

[2] Comparison of Nvidia graphics processing units. Wikipedia: http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units/.

[3] Microsoft DirectX Developer Center. Microsoft DirectX Developer Center: http://msdn.microsoft.com/de-de/directx/default(en-us).aspx.

[4] OpenGL Sprecification & Documentation. OpenGL: http://www.opengl.org/documentation/.

[5] 2006 IEEE Visualization Design Contest: See What's Shaking, August 2006. http://viscontest.sdsc.edu/2006/.

[6] K. Akeley. Reality engine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116, New York, NY, USA, 1993. ACM.

[7] E. Aurell, G. Boffetta, A. Crisanti, G. Paladin, and A. Vulpiani. Predictability in the large: an extension of the concept of Lyapunov exponent. Technical Report chao-dyn/9606014. TNT-96-SHPRE-5-PAP-V-4, Jun 1996.

[8] P. J. Basser. New histological and physiological stains derived from diffusion-tensor MR images. *Ann. N.Y. Acad. Sci. 820*, page 123138, 1997.

[9] G. Benettin, L. Galgani, A. Giorgilli, and J. M. Strelcyn. Lyapunov characteristic exponent for smooth dynamical systems and hamiltonian systems; a method for computing all of them. *Mechanica*, 15(1):9–20, 1980.

[10] J. Bloomenthal. Calculation of reference frames along a space curve. pages 567–571, 1990.

[11] D. Blythe. The Direct3D 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[12] P. Bogacki and L. F. Shampine. A 3(2) pair of runge-kutta formulas. *Appl. Math. Lett.*, 2:331–325, 1989.

[13] E. Boring and A. Pang. Directional Flow Visualization of Vector Fields. *Proceedings of IEEE Transactions on Visualization and Computer Graphics 1996*, pages 389–392, 1996.

[14] A. I. Borisenko and I. E. Tarapov. Vector and tensor analysis with applications. *Dover*, pages 121–122, 1968.

[15] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics*, 0:17–141, 2005.

[16] S. Bruckner, S. Grimm, A. Kanitsar, and E. Gröller. Illustrative Context-Preserving Volume Rendering. In *EuroVis*, pages 69–76, 2005.

[17] S. Bruckner, S. Grimm, A. Kanitsar, and M. E. Gröller. Illustrative Context-Preserving Exploration of Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1559–1569, 2006.

[18] S. Bruckner and E. Gröller. Enhancing Depth-Perception with Flexible Volumetric Halos. *IEEE Transactions on Visualization and Computer Graphics*, 13(6), 2007.

[19] S. Bruckner and M. E. Gröller. VolumeShop: An Interactive System for Direct Volume Illustration. In *Proceedings of IEEE Visualization 2005*, pages 671–678, oct 2005.

[20] R. W. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy. Real-Time Out-of-Core Visualization of Particle Traces. In *IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics (PVG2001)*, pages 45–50, 2001.

[21] S. Bryson and C. Levit. The Virtual Windtunnel: An Environment for the Exploration of Three-dimensional Unsteady Flows. In *Proc. IEEE Vis*, pages 17–24, 1991.

[22] K. Bürger, F. Ferstl, H. Theisel, and R. Westermann. Interactive Streak Surface Visualization on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15:1259–1266, 2009.

[23] K. Bürger, S. Hertel, J. Krüger, and R. Westermann. GPU Rendering of Secondary Effects. In *Vision, Modeling and Visualization 2007*, 2007.

[24] K. Bürger, P. Kondratieva, J. Krüger, and R. Westermann. Importance-Driven Particle Techniques for Flow Visualization. In *Proceedings of IEEE VGTC Pacific Visualization Symposium 2008*, 2008.

[25] K. Bürger, J. Krüger, and R. Westermann. Direct Volume Editing. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2008)*, 14(6):1388–1395, November-December 2008.

[26] K. Bürger, J. Krüger, and R. Westermann. Sample-Based Surface Coloring. *IEEE Transactions on Visualization and Computer Graphics*, 99(PrePrints), 2009.

[27] K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, and R. Westermann. Interactive Visual Exploration of Unsteady 3D-Flows. In *Eurographics/IEEE VGTC Symposium on Visualization (EuroVis)*, 2007.

[28] B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. pages 263–270, 1993.

[29] N. L. R. Center. Image: Wake vortex study at wallops island, May.

[30] W. Chen, A. Lu, and D. S. Ebert. Shape-aware Volume Illustration. *Computer Graphics Forum (Proceedings of Eurographics 2007)*, 26(7):705–714, 2007.

[31] J. Clyne and J. Dennis. Interactive Direct Volume Rendering of Time-Varying Data. *In Proceedings of Data Visualization 99*, pages 109–120, 1999.

[32] S. D. Conte and C. de Boor. *Elementary Numerical Analysis*. McGraw-Hill, New York, NY, USA, 1980.

[33] M. Cox and D. Ellsworth. Application-controlled Demand Paging for Out-Of-Core Visualization. In *Proc. IEEE Vis*, pages 235–244, 1997.

[34] R. Crawfis, N. Max, B. Becker, and B. Cabral. Volume rendering of 3d scalar and vector fields at llnl. In *Supercomputing 93: Proceedings of the annual conference on Supercomputing*, pages 570–576, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, November 1993. IEEE Computer Society.

[35] J. Diepstraten, D. Weiskopf, and T. Ertl. Interactive Cutaway Illustrations. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3):523–532, 2003.

[36] H. Doleisch and H. Hauser. Smooth Brushing for Focus+Context Visualization of Simulation Data in 3D.

[37] D. Eberly, R. Gardner, B. Morse, S. Pizer, and C. Scharlach. Ridges for image analysis. *J. Math. Imaging Vis.*, 4(4):353–373, 1994.

[38] D. Ebert and P. Rheingans. Volume Illustration: Non-photorealistic rendering of volume models. In *IEEE Visualization 2000 (Conference Proceedings)*, pages 195–202, 2000.

[39] D. S. Ebert and P. Rheingans. Volume Illustration: Nonphotorealistic Rendering of Volume Models. *Proceedings of IEEE Transactions on Visualization and Computer Graphics 2001*, 7:253–264, 2001.

[40] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.

[41] C. L. Feffermann. Existence and smoothness of the Navier-Stokes equation., 2000. http://www.claymath.org/millennium/Navier-Stokes_Equations/navierstokes.pdf.

[42] F. Ferstl, K. Bürger, H. Theisel, and R. Westermann. Interactive Separating Streak Surfaces. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2010)*, 16(6):to appear, November-December 2010.

[43] L. Forssell and S. Cohen. Using Line Integral Convolution for Flow Visualization: Curvilinear Grids, Variable-Speed Animation, and Unsteady Flows. *IEEE TVCG*, 1(2):133–141, 1995.

[44] O. Frederich, E. Wassen, and F. Thiele. Flow Simulation around a Finite Cylinder on Massively Parallel Computer Architecture. In *International Conference on Parallel Computational Fluid Dynamics*, pages 85–93, 2005.

[45] T. Frühauf. Raycasting vector fields. *Proceedings of IEEE Transactions on Visualization and Computer Graphics 1996*, pages 115–120, 1996.

[46] A. L. Fuhrmann and E. Gröller. Real-Time Techniques for 3D Flow Visualization. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *VIS '98: Proceedings of the conference on Visualization '98*, pages 305–312, 1998.

[47] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen. Efficient Computation and Visualization of Coherent Structures in Fluid Flow Applications. *IEEE Transactions on Visualization and Computer Graphics*, 13:1464–1471, 2007.

[48] C. Garth, H. Krishnan, X. Tricoche, T. Bobach, and K. I. Joy. Generation of Accurate Integral Surfaces in Time-Dependent Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1404–1411, 2008.

[49] C. Garth, G. Li, X. Tricoche, C. Hansen, and H. Hagen. Visualization of Coherent Structures in Transient 2D Flows. In *Topology-Based Methods in Visualization II (Proceedings of TopoInVis 2007)*, pages 1–14, March 2009.

[50] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann. Surface Techniques for Vortex Visualization. In *Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 155–164, 2004.

[51] T. Glau. Exploring instationary fluid flows by interactive volume movies. *In Proceedings of Data Visualization 99*, pages 277–283, 1999.

[52] L. Grady, T. Schiwietz, S. Aharon, and R. Westermann. Random Walks for Interactive Organ Segmentation in Two and Three Dimensions: Implementation and Validation. In *MICCAI*, 2005.

[53] H. Gray. *Gray's anatomy*. Running Press, 1901.

[54] M. Griebel, T. Donrseifer, and T. Neunhöfer. *Numerical Simulation in Fluid Dynamics: A Practilca Introduction*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[55] S. Guthe, S. Gumhold, and W. Strasser. Interactive visualization of volumetric vector fields using texture based particles. In *Proceedings of WSCG*, volume 10, pages 33–41, 2002.

[56] G. Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Phys. D*, 149(4):248–277, 2001.

[57] G. Haller. Lagrangian coherent structures from approximate velocity data. *Physics of Fluids*, 14(6):1851–1861, 2002.

[58] G. Haller and G. Yuan. Lagrangian coherent structures and mixing in two-dimensional turbulence. *Phys. D*, 147(3-4):352–370, 2000.

[59] P. Hanrahan and P. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, 24(4):215–223, 1990.

[60] R. M. Haralick. Ridges and valleys on digital images. *Computer Vision, Graphics, and Image Processing*, 22(1):28–38, 1983.

[61] P. Hartman. *Ordinary Differential Equations*. 1973.

[62] K. M. Hasan, P. J. Basser, D. L. Parker, and A. L. Alex. Analytical computation of the eigenvalues and eigenvectors in dt-mri. *J. Magn. Reson*, pages 41–47, 2001.

[63] H. Hauser, L. Mroz, G. I. Bischi, and M. E. Gröller. Two-Level Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):242–252, 2001.

[64] J. Helman and L. Hesselink. Representation and Display of Vector Field Topology in Fluid Flow Data Sets. *IEEE Computer*, 22(8):27–36, 1989.

[65] J. P. M. Hultquist. Constructing stream surfaces in steady 3D vector fields. In *IEEE Transactions on Visualization and Computer Graphics*, pages 171–178, 1992.

[66] V. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In *ACM SIGGRAPH*, pages 109–116, 1997.

[67] V. Interrante, H. Fuchs, and S. Pizer. Illustrating transparent surfaces with curvature-directed strokes. In *IEEE Vis*, pages 211–218, 1996.

[68] J. D. Furst and Stephen M. Pizer. Marching ridges. In *In 2001 IASTED International Conference on Signal and Image Processing*, 2001.

[69] Jan, T. Weinkauf, and H.-C. Hege. Galilean invariant extraction and iconic representation of vortex core lines. In *EuroVis*, pages 151–160, 2005.

[70] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995.

[71] B. Jobard, G. Erlebacher, and M. Hussaini. Hardware-Accelerated Texture Advection For Unsteady Flow Visualization. In *Proc. IEEE Vis*, pages 155–162, 2001.

[72] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62.

[73] J. Kasten, C. Petz, I. Hotz, B. Noack, and H.-C. Hege. Localized finite-time lyapunov exponent for unsteady flow analysis. In M. Magnor, B. Rosenhahn, and H. Theisel, editors, *Vision Modeling and Visualization*, volume 1, pages 265–274. Universität Magdeburg, Inst. f. Simulation u. Graph., 2009.

[74] R. M. Kelso, T. T. Lim, and A. E. Perry. An experimental study of round jets in cross-flow. *Journal of Fluid Mechanics*, 306(-1):111–144, 1996.

[75] D. N. Kenwright and D. A. Lane. Optimization of Time-Dependent Particle Tracing Using Tetrahedral Decomposition. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 321, 1995.

[76] G. Kindlmann and J. W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86, 1998.

[77] G. L. Kindlmann, R. S. J. Estépar, S. M. Smith, and C.-F. Westin. Sampling and visualizing creases with scale-space particles. *IEEE Trans. Visualization and Computer Graphics*, 15(6):1415–1424, Nov/Dec 2009.

[78] G. L. Kindlmann, X. Tricoche, and C.-F. Westin. Anisotropy creases delineate white matter structure in diffusion tensor mri. In *MICCAI (1)*, pages 126–133, 2006.

[79] P. Kipfer, F. Reck, and G. Greiner. Local exact particle tracing on unstructured grids. *Computer Graphics Forum*, 22(2):133–142, 2003.

[80] R. M. Kirby, H. Marmanis, and D. H. Laidlaw. Visualizing Multivalued Data from 2D Incompressible Flows Using Concepts from Painting. *Proceedings of IEEE Transactions on Visualization and Computer Graphics 1999*, pages 333–340, 1999.

[81] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[82] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey, and C. Hansen. Interactive Texture-Based Volume Rendering for Large Data Sets. *IEEE Computer Graphics and Applications*, 21(4):52–61, 2001.

[83] L. Kobbelt, S. Campagna, J. Vorsatz, and H.-P. Seidel. Interactive multi-resolution modeling on arbitrary meshes. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 105–114, New York, NY, USA, 1998. ACM.

[84] P. Kondratieva. *Real-Time Approaches for Model-Based Reconstruction and Visualization of Flow Fields*. PhD thesis, Technische Universität München, 2008.

[85] P. Kondratieva, K. Bürger, J. Georgii, and R. Westermann. Real-Time Approaches for Model-Based PIV and Visual Fluid Analysis. 106/2009:257–267, 2009.

[86] M. Kraus and K. Bürger. Interpolating and Downsampling RGBA Volume Data. In *Proceedings of Vision, Modeling, and Visualization 2008*, 2008.

[87] K. Kreeger and A. Kaufman. Interactive volume segmentation with the PAVLOV architecture. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pages 61–68, 1999.

[88] J. Krüger. *GI-Edition Lecture Notes in Informatics (LNI)*, chapter A GPU Framework for Interactive Simulation and Rendering of Fluid Effects. GI, 2007.

[89] J. Krüger, K. Bürger, and R. Westermann. Interactive Screen-Space Accurate Photon Tracing on GPUs. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, pages 319–329, June 2006.

[90] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A particle system for interactive visualization of 3D flows. *IEEE TVCG*, 11(5):744–756, 2005.

[91] J. Krüger, J. Schneider, and R. Westermann. ClearView: An Interactive Context Preserving Hotspot Visualization Technique. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006)*, 12(5), September-October 2006.

[92] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[93] J. Krüger and R. Westermann. Efficient Stipple Rendering. In *Proceedings of IADIS Computer Graphics and Visualization*, 2007.

[94] Y. P. L. Barreira. *Lyapunov Exponents and Smooth Ergodic Theory*. 2002.

[95] D. Lane. Visualizing Time-Varying Phenomena in Numerical Simulations of Unsteady Flows. In *34th Aerospace Science Meeting & Exhibit*, 1996.

[96] R. Laramee, G. Erlebacher, D. Weiskopf, C. Garth, X. Tricoche, T. Weinkauf, H. Theisel, F. Post, B. Vrolijk, H. Hauser, and H. Doleisch. Texture and Feature-Based Flow Visualization. In *Tutorial #2, IEEE Vis*. 2006.

[97] R. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. Post, and D. Weiskopf. The State of the Art in Flow Visualization: Dense and Texture-Based Techniques. *Computer Graphics Forum*, 23(2):203–221, 2004.

[98] R. S. Laramee, B. Jobard, and H. Hauser. Image space based visualization of unsteady flow on surfaces. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 131–138, 2003.

[99] Z. L. P. F. Laramee R., Hauser H. Topology-based flow visualization, the state of the art. In H. H. Hauser H. and T. H., editors, *In Topology-Based Methods in Visualization: Proc. of the 1st TopoInVisWorkshop (TopoInVis 2005)*, pages 1–20, 2007.

[100] F. Lekien, C. Coulliette, A. J. Mariano, E. H. Ryan, L. K. Shay, G. Haller, and J. Marsden. Pollution release tied to invariant manifolds: A case study for the coast of florida. *Phys. D*, 210(1), 2005.

[101] G.-S. Li, X. Tricoche, and C. Hansen. GPUFLIC: Interactive and Accurate Dense Visualization of Unsteady Flows. In *Proc. EuroVis*, pages 29–33, 2006.

[102] W. Li, L. Ritter, M. Agrawala, B. Curless, and D. Salesin. Interactive cutaway illustrations of complex 3D models. *ACM Trans. Graph.*, 26(3):31–40, 2007.

[103] A. Liapunov. Stability of Motion. 1966.

[104] T. Lindeberg. Edge detection and ridge detection with automatic scale selection. *Int. J. Comput. Vision*, 30(2):117–156, 1998.

[105] D. Lipinski and K. Mohseni. A ridge tracking algorithm and error estimate for efficient computation of lagrangian coherent structures. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 20(1):017504, 2010.

[106] D. Lischinski and A. Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In *Proceedings, Ninth Eurographics Workshop on Rendering*, pages 301–314, 1998.

[107] H. Löffelmann and M. E. Gröller. Enhancing the visualization of characteristic structures in dynamical systems. In *Proceedings of the 9th Eurographics Workshop on Visualization in Scientific Computing*, pages 35–46, 1998.

[108] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[109] A. Lu, C. Morris, D. Ebert, P. Rheingans, and C. Hansen. Non-photorealistic volume rendering using stippling techniques. In *IEEE Vis*, pages 211–218, 2002.

[110] E. Lum and K. Ma. Hardware-Accelerated Parallel Non-Photorealistic Volume Rendering. In *International Symposium on Non-photorealistic Rendering and Animation (NPAR)*, June 2002.

[111] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 100–107, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[112] O. Mattausch, T. Theußl, H. Hauser, and M. E. Gröller. Strategies for Interactive Exploration of 3D Flow Using Evenly-Spaced Illuminated Streamlines. In K. Joy, editor, *Proceedings of Spring Conference on Computer Graphics*, pages 213–222. SCCG, apr 2003.

[113] N. Max, R. Crawfis, and C. Grant. Visualizing 3D Velocity Fields Near Contour Surfaces. In *IEEE Visualization 94*, pages 248–255, 1994.

[114] B. H. McCormick. Visualization in scientific computing. *SIGBIO Newsl.*, 10(1):15–21, 1988.

[115] T. McLoughlin, R. Laramee, R. Peikert, F. Post, and M. Chen. Over Two Decades of Integration-Based, Geometric Flow Visualization. In *Computer Graphics Forum, (to appear)*, 2010.

[116] F. Meyer. Topographic distance and watershed lines. *Signal Process.*, 38(1):113–125, 1994.

[117] G. E. Moore. Cramming more components onto integrated circuits. pages 56–59, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[118] K. Myers and L. Bavoil. Stencil routed A-Buffer. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 21, 2007.

[119] R. W. D. Nickalls. A New Approach to Solving the Cubic: Cardan's Solution Revealed. *The Mathematical Gazette*, 77(480):354–359, 1993.

[120] G. M. Nielson and I.-H. Jung. Tools for Computing Tangent Curves for Linearly Varying Vector Fields over Tetrahedral Domains. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):360–372, 1999.

[121] S. Owada, F. Nielsen, M. Okabe, and T. Igarashi. Volumetric illustration: designing 3D models with internal textures. *ACM Trans. Graph.*, 23(3):322–328, 2004.

[122] K. Palgyi and A. Kuba. A parallel 3d 12-subiteration thinning algorithm. *Graphical Models and Image Processing*, 61(4):199 – 221, 1999.

[123] S. Park, B. Budge, L. Linsen, B. Hamann, and K. Joy. Dense Geometric Flow Visualization. In *Proc. EuroVis*, pages 21–28, 2005.

[124] H. K. Pedersen. A framework for interactive texturing on curved surfaces. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 295–302, 1996.

[125] R. Peikert and M. Roth. The "parallel vectors" operator: a vector field visualization primitive. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 263–270, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[126] R. Peikert and F. Sadlo. Height Ridge Computation and Filtering for Visualization. In *Proceedings of IEEE VGTC Pacific Visualization Symposium 2008*, pages 119–126, 2008.

[127] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[128] D. Pnueli and C. Gutfinger. Fluid mechanics. New York, USA, 1992. Cambridge University Press.

[129] F. Post, R. Laramee, B. Vrolijk, H. Hauser, and H. Doleisch. Feature Extraction and Visualization of Flow Fields. In *Proc. Eurographics*, pages 69–100, 2002.

[130] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):775–792, 2003.

[131] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes: The art of scientific computing*, chapter Eigensystems. Cambridge University Press, 3 edition, 2007.

[132] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50, 2003.

[133] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl. Interactive Exploration of Volume Line Integral Convolution Based on 3D Texture Mapping. In *Proc. IEEE Visualization*, 1999.

[134] M. Roerdink. The watershed transform: definitions, algorithms, and parallellization strategies. *Fundamenta Informaticae*, 41(1):187–228, 2000.

[135] T. Ropinski, J.-S. Prani, J. Roters, and K. H. Hinrichs. Internal Labels as Shape Cues for Medical Illustration. In *Proceedings of the 12th International Fall Workshop on Vision, Modeling, and Visualization (VMV07)*, pages 203–212, 2007.

[136] A. Sadarjoen, T. van Walsum, A. Hin, and F. Post. Particle Tracing Algorithms for 3D Curvilinear Grids. In *IEEE Scientific Visualization, Overviews, Methodologies, and Techniques*, pages 311–335, 1994.

[137] F. Sadlo and R. Peikert. Efficient visualization of lagrangian coherent structures by filtered amr ridge extraction. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1456–1463, 2007.

[138] F. Sadlo and R. Peikert. Visualizing lagrangian coherent structures and comparison to vector field topology. In *Topology-Based Methods in Visualization II (Proceedings of TopoInVis 2007)*, pages 15–30, March 2009.

[139] F. Sadlo, A. Rigazzi, and R. Peikert. Time-Dependent Visualization of Lagrangian Coherent Structures by Grid Advection. In *Proceedings of TopoInVis 2009 (to appear)*. Springer, 2009.

[140] F. Sadlo and D. Weiskopf. Time-Dependent 2D Vector Field Topology: An Approach Inspired by Lagrangian Coherent Structures. *Computer Graphics Forum*, 29(1):88–100, 2010.

[141] J. Sahner, T. Weinkauf, N. Teuber, and H.-C. Hege. Vortex and Strain Skeletons in Eulerian and Lagrangian Frames. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):980–990, September - October 2007.

[142] W. T. S. G. Salzbrunn T., Jänicke H. The state of the art in flow visualization: Partition-based techniques. In T. H. Hauser H., Straburger S., editor, *In SimVis (2008)*, pages 75–92, 2008.

[143] S.Camarri, M. Salvetti, M. Buffoni, and A.Iollo. Simulation of the three-dimensional flow around a square cylinder between parallel walls at moderate Reynolds numbers. In *Proceedings of XVII Congresso di Meccanica Teorica ed Applicata*, 2005.

[144] T. Schafhitzel. Image: 2D line integral convolution. http://www.vis.uni-stuttgart.de/ger/research/proj/spp1147/lic/.

[145] T. Schafhitzel, E. Tejada, D. Weiskopf, and T. Ertl. Point-based Stream Surfaces and Path Surfaces. In *Proceedings of Graphics Interface 2007*, pages 289–296, 2007.

[146] T. Schafhitzel, J. E. Vollrath, J. P. Gois, D. Weiskopf, A. Castelo, and T. Ertl. Topology-preserving lambda$_2$-based vortex core line detection for flow visualization. *Computer Graphics Forum*, 27(3):1023–1030, 2008.

[147] G. Scheuermann, T. Bobach, H. H. K. Mahrous, B. Hamann, K. Joy, and W. Kollmann. A Tetrahedra-based Stream Surface Algorithm. pages 151–158, 2001.

[148] M. Schirski, C. Bischof, and T. Kuhlen. Interactive Particle Tracing on Tetrahedral Grids Using the GPU. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 153–160, 2006.

[149] R. Schmidt, C. Grimm, and B. Wyvill. Interactive decal compositing with discrete exponential maps. *ACM Transactions on Graphics*, 25(3):605–613, 2006.

[150] D. Schneider, A. Wiebel, and G. Scheuermann. Smooth Stream Surfaces of Fourth Order Precision. In *Eurographics/IEEE VGTC Symposium on Visualization (EuroVis)*, pages 871–878, 2009.

[151] T. Schultz, H. Theisel, and H.-P. Seidel. Crease Surfaces: From Theory to Extraction and Application to Diffusion Tensor MRI. *IEEE Transactions on Visualization and Computer Graphics*, 16:109–119, 2010.

[152] S. Shadden. Lagrangian Coherent Structures: Analysis of time-dependent synamical systems using finite-time Lyapunov exponents, 2005. http://www.cds.caltech.edu/ shawn/LCS-tutorial/overview.html.

[153] S. C. Shadden, F. Lekien, and J. E. Marsden. Definition and properties of lagrangian coherent structures from finite-time lyapunov exponents in two-dimensional aperiodic flows. *Physica D: Nonlinear Phenomena*, 212(7):271–304, 2005.

[154] S. C. Shadden, F. Lekien, J. D. Paduan, F. P. Chavez, and J. E. Marsden. The correlation between surface drifters and coherent structures based on high-frequency radar data in monterey bay. *Deep Sea Research Part II: Topical Studies in Oceanography*, 56(3-5):161 – 172, 2009. AOSN II: The Science and Technology of an Autonomous Ocean Sampling Network.

[155] J. Shade, S. Gortler, L. wei He, and R. Szeliski. Layered depth images. In *Proceedings of ACM SIGGRAPH 98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, 1998.

[156] H.-W. Shen and D. Kao. UFLIC: A Line Integral Convolution Algorithm for Visualizing Unsteady Flows. In *Proceedings IEEE Visualization 97*, pages 317–323, 1997.

[157] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-Based Ray Casting of Quadratic Surfaces. In *Proceedings of the Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pages 59–65, 2006.

[158] J. Smagorinsky. General Circulation Experiments with the Primitive Equations. *Monthly Weather Review*, 91:99–+, 1963.

[159] A. R. Smith. Paint. Technical Memo 7, Computer Graphics Lab, New York Institute of Technology, July 1978.

[160] D. Stalling. *Fast Texture-Based Algorithms for Vector Field Visualization*. PhD thesis, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1998.

[161] D. Stalling and H.-C. Hege. Fast and Resolution-Independent Line Integral Convolution. In *Proceedings of ACM SIGGRAPH 95: Proceedings of the 22th annual conference on Computer graphics and interactive techniques*, pages 249–256, aug 1995.

[162] C. Stoll, S. Gumhold, H. peter Seidel, and M. Planck. Visualization with stylized line primitives. In *In Proceedings of IEEE visualization 2005*, pages 695–702, 2005.

[163] H. Theisel, J. Sahner, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Extraction of Parallel Vector Surfaces in 3D Time-Dependent Fields and Applications to Vortex Core Line Tracking. In *Proc. IEEE Vis*, pages 631–638, 2005.

[164] H. Theisel and H.-P. Seidel. Feature flow fields. In H. Bonneau, Hahmann, editor, *In Data Visualization 2003: Proc. of the 5th Joint EUROGRAPHICS IEEE TCVG Symp. on Visualization (VisSym)*, pages 141–148, 2003.

[165] H. Theisel, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Saddle connectors - an approach to visualizing the topological skeleton of complex 3d vector fields. In G. Turk, J. J. van Wijk, and R. Moorhead, editors, *Proc. IEEE Visualization 2003*, pages 225–232, Seattle, U.S.A., October 2003.

[166] H. Theisel, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Topological methods for 2D timedependent vector fields based on stream lines and path lines. In *IEEE Transactions on Visualization and Computer Graphics*, volume 11, pages 383–394, 2005.

[167] G. Turk and D. Banks. Image-Guided Streamline Placement. In *Proc. Computer Graphics and Interactive Techniques*, pages 453–460, 1996.

[168] F.-Y. Tzeng, E. B. Lum, and K.-L. Ma. An intelligent system approach to higher-dimensional classification of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):273–284, 2005.

[169] S. Ueng, K. Sikorski, and K. Ma. Efficient streamline, streamribbon, and streamtube constructions on unstructured grids. In *Transactions on Visualization and Computer Graphics*, pages 2:100–110, 1996.

[170] T. van Walsum. *Selective Visualization on Curvilinear Grids*. PhD thesis, Delft University of Technology, The Netherlands, 1995.

[171] J. van Wijk. Image Based Flow Visualization for Curved Surfaces. In *IEEE Vis*, pages 123–130, 2003.

[172] J. J. van Wijk. Spot noise-texture synthesis for data visualization. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 309–318, 1991.

[173] J. J. van Wijk. Flow visualization with surface particles. *IEEE Computer Graphics and Applications*, 13(4):18–24, jul 1993.

[174] J. J. van Wijk. Implicit Stream Surfaces. In *IEEE Transactions on Visualization and Computer Graphics*, pages 245–252, 1993.

[175] J. J. van Wijk. Image based flow visualization. In *Proceedings of ACM SIGGRAPH 2002: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 745–754, 2002.

[176] R. Verstappen and A. Veldman. Spectro-consistent discretization of Navier-Stokes: a challenge to RANS and LES. *Journal of Engineering Mathematics*, 34(1):163–179, 1998.

[177] J. Villasenor and A. Vincent. An algorithm for space recognition and time tracking of vorticity tubes in turbulence. *CVGIP: Image Underst.*, 55(1):27–35, 1992.

[178] L. Vincent and P. Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13:583–598, 1991.

[179] I. Viola, E. Gröller, K. Bühler, M. Hadwiger, B. Preim, D. Ebert, M. C. Sousa, and D. Stredney. Illustrative Visualization. IEEE Visualization Tutorial on Illustrative Visualization, 2005.

[180] I. Viola, A. Kanitsar, and E. Gröller. Importance-driven Volume Rendering. In *IEEE Visualization 2004 (Conference Proceedings)*, pages 139–145, 2004.

[181] W. von Funck, T. Weinkauf, H. Theisel, and H.-P. Seidel. Smoke Surfaces: An Interactive Flow Visualization Technique Inspired by Real-World Flow Experiments. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1396–1403, 2008.

[182] D. Weiskopf, K. Engel, and T. Ertl. Volume clipping via per-fragment operations in texture-based volume visualization. In *IEEE Vis*, pages 93–100, 2002.

[183] D. Weiskopf and T. Ertl. A Hybrid Physical/Device-Space Approach for Spatio-Temporally Coherent Interactive Texture Advection on Curved Surfaces. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 263–270, 2004.

[184] D. Weiskopf, T. Schafhitzel, and T. Ertl. Real-Time Advection and Volumetric Illumination for the Visualization of 3D Unsteady Flow. In *Proc. EuroVis*, pages 13–20, 2005.

[185] M. Weldon, T. Peacock, G. B. Jacobs, M. Helu, and G. Haller. Experimental and numerical investigation of the kinematic theory of unsteady separation. *Journal of Fluid Mechanics*, 611:1–11, 2008.

[186] A. Wiebel and G. Scheuermann. Eyelet Particle Tracing – Steady Visualization of Unsteady Flow. In *IEEE Vis*, pages 607–614, 2005.

[187] A. Wiebel, D. Schneider, H. Jaenicke, X. Tricoche, and G. Scheuermann. Generalized Streak Lines: Analysis and Visualization of Boundary Induced Vortices. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1735–1742, 2007.

[188] S. Wiggins. *Chaotic transport in dynamical systems*. 1992.

[189] M. Wilczek. Image: Volume rendering of the absolute value of vorticity in fully developed turbulence. http://pauli.uni-muenster.de/tp/menu/forschen/ag-friedrich/mitarbeiter/wilczek-michael/video-gallery.html.

[190] D. D. Y. Levy and A. Seginer. Graphical visualization of vortical flows by means of helicity. *AIAA Journal*, 28(8), 1990.

[191] X. Yuan and B. Chen. Illustrating Surfaces in Volume. In *Proceedings of VisSym'04, Joint IEEE/EG Symposium on Visualization (Konstanz, Germany, May 19–21, 2004)*, pages 9–16, 337, 2004.

[192] M. Zöckler, D. Stalling, and H.-C. Hege. Interactive visualization of 3d-vector fields using illuminated stream lines. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 107–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.