Particle Tracing Methods for Visualization and Computer Graphics

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

Vorgelegt von

Tobias Schafhitzel

aus Nürtingen

Hauptberichter:	Prof. Dr. D. Weiskopf
Mitberichter:	Prof. Dr. T. Ertl
Tag der mündlichen Prüfung:	16. Dezember 2008

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2008

Für Sonja

CONTENTS

List of Abbreviations and Acronyms **7**

Abstract and Chapter Summaries **9**

Zusammenfassung **17**

Acknowledgments **19**

Introduction **21**

I Graphics and Visualization Techniques **23**

1 Principles of Computer Graphics **25**
1.1 Visualization Pipeline 25
1.2 Rendering Pipeline 25
1.2.1 The Programmable Rendering Pipeline 26
1.2.2 General-Purpose Computation on the GPU 28
1.3 Grid Types 29
1.4 Volume Visualization 30
1.4.1 Direct Volume Visualization 30
1.4.2 Indirect Volume Visualization 33
1.5 Test Hardware Configuration 34

2 Mathematical Basics **35**
2.1 Numerical Derivatives 35
2.1.1 Gradient Computation 35
2.1.2 Second Order Derivatives 36
2.2 Surface Curvature 36
2.3 Computing Eigenvalues and Eigenvectors 37
2.4 Numerical Integration of ODEs 38
2.5 Interpolation 39

3 Overview of Flow Visualization **41**
3.1 Data Acquisition 41
3.1.1 Flow Measurements 41
3.1.2 Flow Simulations 42
3.2 Navier-Stokes Equations 42
3.3 Experimental Flow Visualization 44

II Particle Tracing for Direct Vector Field Visualization 47

4 Basics of Texture-based Flow Visualization 49
4.1 Line Integral Convolution 50
4.2 Semi-Lagrangian Texture Advection 51
4.2.1 Basic Algorithm 51
4.2.2 Visual Mapping 51
4.3 LIC on Curved Surfaces 52
4.3.1 Basic Idea 52
4.3.2 Algorithm 53

5 3D Texture-based Visualization 57
5.1 Semi-Lagrange Advection in 3D 57
5.1.1 3D Texture Advection on the GPU 57
5.1.2 Rendering 60
5.2 Illumination and Visual Perception 61
5.2.1 Gradient-Based Illumination 62
5.2.2 Line-Based Illumination 63
5.2.3 Perception-Oriented Illumination 66
5.2.4 Performance Measurements 67
5.3 Texture Advection on 2D Slices 69
5.3.1 Multiple Views and 3D Texture Advection 70
5.3.2 Slice-Based 2D Multi-Field Visualization 71
5.3.3 Interactively Linking 2D and 3D Texture Advection 74
5.4 Concluding Discussion 77

6 Flow Visualization on Curved Surfaces 79
6.1 Stream Surfaces and Path Surfaces 79
6.1.1 Previous Work 80
6.1.2 Definition of Stream Surfaces and Path Surfaces 82
6.1.3 Surface Generation 82
6.1.4 Point-based Rendering of the Surface 88
6.1.5 LIC on Stream Surfaces 89
6.1.6 Performance Measurements 92
6.2 Visualizing Multi-Modality Data using Surface LIC 93
6.2.1 Combining Functional and Anatomical Data 94
6.2.2 Multi-Volume Rendering 95
6.2.3 Data Acquisition 97
6.2.4 Results 98
6.3 Line Integral Convolution on 3D Vortex Structures 100
6.3.1 Vortex Visualization Process 101
6.3.2 Animated LIC 101
6.3.3 Particle Lighting 102

III Particle Tracing for Feature Detection in Fluid Flows 105

7 Methods for Computing Flow Features 107
7.1 Decomposing the Velocity Gradient Tensor . . . 107
7.2 Vortex Core Detection . . . 108
7.2.1 Q Criterion . . . 108
7.2.2 λ_2 Criterion . . . 108
7.2.3 Other Criteria . . . 109
7.3 Vortex Core Lines . . . 110
7.3.1 Formal Definition . . . 110
7.3.2 Predictor-Corrector Method . . . 111
7.4 Shear Layers . . . 112
7.5 Critical Points . . . 113

8 Feature Detection and Visualization 115
8.1 Feature-based 3D Texture Advection . . . 115
8.1.1 Previous Strategies . . . 116
8.1.2 Feature-based 3D Texture Advection . . . 116
8.1.3 Texture-based vs. Geometry-based Visualization . . . 118
8.1.4 Examples . . . 120
8.2 Topology-Preserving Vortex Core Line Detection . . . 121
8.2.1 Preliminary Steps . . . 122
8.2.2 Vortex Core Line Detection . . . 122
8.2.3 Dealing with Vortex Core Line Splits . . . 124
8.2.4 Finding an appropriate λ_2 Isovalue . . . 126
8.2.5 Vortex Core Line Visualization . . . 128
8.3 Visualizing Shear Stress . . . 130
8.3.1 Shear Layers vs. Vortices . . . 131
8.3.2 Introducing Shear Sheets . . . 132
8.3.3 Visualization . . . 135

9 Feature Tracking 139
9.1 Particle-based Feature Tracking . . . 139
9.1.1 Sampling the Extremum Line . . . 140
9.1.2 Predictor-Corrector Tracking Algorithm . . . 140
9.1.3 Graphical Representation of Tracked Features . . . 141
9.2 Temporal Development of Vortices . . . 143
9.2.1 Vortex Dynamics . . . 143
9.2.2 Taking into Account Vortex Dynamics . . . 144
9.2.3 Visualization . . . 145
9.3 Temporal Development of Shear Layers . . . 147
9.3.1 Shear Layers in Vortex Dynamics . . . 148
9.3.2 Visualization of Shear Layers in Time-dependent Flow . . . 154

IV Light Tracing **157**

10 Linear Light Tracing **159**
10.1 Raytracing Multifield Data 159
10.1.1 Octree Texture 160
10.1.2 Adaptive Page Table 162
10.1.3 Results and Discussion 164
10.2 Atmospheric Scattering 165
10.2.1 Equations of Atmospheric Scattering 166
10.2.2 Real-Time Atmospheric Scattering 168
10.2.3 Planetary Terrain Rendering 173
10.2.4 Performance Measurements and Examples 174

11 Nonlinear Light Tracing **177**
11.1 Nonlinear Ray Tracing on the GPU 177
11.1.1 Algorithm 177
11.1.2 Applications 178
11.2 Generation of Nonlinear Panorama Maps 178
11.2.1 Requirements for Rendering Panorama Maps 178
11.2.2 Modeling Ray Deflection 179
11.2.3 Generating Nonlinear Panorama Maps 182

12 Conclusion **185**

Bibliography **193**

List of Abbreviations and Acronyms

2D	two-dimensional
3D	three-dimensional
API	Application Program Interface
CPU	Central Processing Unit
CFD	Computational Fluid Dynamics
CT	computer tomography
d.h.	das heißt
DNS	Direct Numerical Simulation
Dr. rer. nat.	Doctor rerum naturalium
e.g.	exempli gratia
et al.	et alii, et aliae, et alia
etc.	et cetera (and so forth)
fps	frames per second
GPU	Graphics Processing Unit
GUI	Graphical User Interface
i.e.	id est
KB	kilobytes
LES	Large Eddy Simulation
LIC	Line Integral Convolution
MB	megabyte
MRI	Magnetic Resonance Imaging
ODE	Ordinary Differential Equation
O.K.	“oll korrect”
pixel	picture element
PIV	Particle Image Velocimetry
Prof. Dr.	Professor Doctor
RANS	Reynolds-Averaged Navier-Stokes
RGB	red, green, and blue
RGBA	red, green, blue, and alpha
SIMD	Single Instruction, Multiple Data
texel	texture element
voxel	volume element
z.B.	zum Beispiel

ABSTRACT AND CHAPTER SUMMARIES

Abstract

This thesis discusses the broad variety of particle tracing algorithms with focus on flow visualization. Starting with a general overview of the basics of visualization and computer graphics, mathematics, and fluid dynamics, a number of methods using particle tracing for flow visualization and computer graphics are proposed. The first part of this thesis considers mostly texture-based techniques that are implemented on the graphics processing unit (GPU) in order to provide an interactive dense representation of 3D flow fields. This part considers particle tracing methods that can be applied on general vector fields and includes texture based visualization in volumes as well as on surfaces. Furthermore, it is described how particle tracing can be used for extracting flow structures, like path surfaces, of the given vector field.

The second part of this thesis considers particle tracing on derived vector fields for flow visualization. Therefore, first a feature extraction criterion is applied on a fluid flow field. In most cases this results in a scalar field serving as base for the particle tracing methods. Here, it is shown how higher order derivatives of scalar fields can be used to extract flow features like 1D vortex core lines or 2D shear sheets. The extracted structures are further processed in terms of feature tracking.

The third part generalizes particle tracing for arbitrary applications in visualization and computer graphics. Here, the particles' path either might be defined by the perspective of the human eye or by a force field that influences the particles' motion by considering second order ordinary differential equations.

All three parts clarify the importance of particle tracing methods for a wide range of applications in flow visualization and computer graphics by various examples. Furthermore, it is shown how the flexibility of this method strongly depends on the underlying vector field, and how those vector fields can be generated in order to solve problems that go beyond traditional particle tracing in fluid flow fields.

Chapter Summaries

Introduction

Here, the topic of the thesis is introduced: particle tracing. The application of particle tracing is motivated in terms of examples. The focus is on flow visualization and, therefore, this topic is also emphasized in the introduction. Furthermore, the organization of this work is discussed, i.e., the classification of the single methods in parts and chapters is described as well as the relation between the different parts of this thesis. The introduction finally leads to the message of this thesis, that says that particle tracing is crucial for various algorithms for visualization and computer graphics, whereby its flexible application is strongly dependent on the flexibility of the usage of arbitrary

vector fields.

Part 1: Graphics and Visualization Techniques

This first of four parts encloses the fundamentals in computer graphics, mathematics and flow visualization. It includes Chapters 2 to 4.

Chapter 1: Principles of Computer Graphics

All the visualization and computer graphics algorithms proposed in this thesis base on a number of methods and principles that are introduced in this chapter. The global view of the visualization process–the visualization pipeline–is discussed as well as the global view on the rendering process using programmable graphics hardware, expressed by the rendering pipeline. This discussion gives more general information about current graphics hardware with focus on the entities that are most important for this thesis. Furthermore, the most common grid types are introduced, their pros and cons are discussed, and it is shown which grid types are used in the following algorithms. The chapter ends with a detailed discussion of the volume rendering integral that needs to be solved during direct volume rendering. This discussion also includes a comparison of slice-based volume rendering and ray casting because both approaches are used in this work. Finally, a brief introduction to indirect volume rendering is given, since it is also used in some algorithms in terms of Marching Cubes.

Chapter 2: Mathematical Basics

The mathematical basics are restricted to those topics that play a general role for the proposed methods. Since most of the algorithms deal with scalar and vector fields, the computation of first and second order derivatives is explained as well as the computation of the first and second principal direction. Particularly the computation of eigenvalues and eigenvectors play a crucial role in this thesis and, therefore, their computation is shown according to Cardano's formula.

The numerical integration of ordinary differential equations (ODEs) is also described in this chapter because its need for evaluating the ODE for Lagrangian particle tracing. Basically two methods are used: the first order Euler scheme and the fourth order Runge-Kutta method. Last but not least, an introduction to linear interpolation in 1D, 2D, and 3D structured grids is given as well as to the barycentric interpolation for triangles.

Chapter 3: Overview of Flow Visualization

According to the visualization pipeline, data acquisition builds the first stage in the visualization process. This stage is explained in more detail with respect to flow visualization, i.e., the acquisition of flow data out of flow measurements and simulations. Three different simulation methods are presented, which are used to create the input data for all simulated data sets in this thesis. Actually, the commonness between the different methods is that they all solve the Navier-Stokes equations, which describe the conservation of mass and momentum and are discussed in more detail as well. The last section in this chapter consists of the discussion of methods that are adopted from

experimental flow visualization, like streak lines, path lines, and time lines. For each line type, it is explained how they are created in a real environment and how they find usage in applications for computer-aided flow visualization.

Part 2: Particle Tracing for Direct Vector Field Visualization

This part contains different methods for the rendering arbitrary vector fields by applying texture-based techniques. This part contains Chapters 4 to 6.

Chapter 4: Basics of Texture-based Flow Visualization

Since the basics of texture-based flow visualization are discussed in more detail, this chapter has been sourced out to the second part of this thesis that is about particle tracing for direct vector field visualization. The discussion starts with a general overview of recent texture-based techniques for interactive flow visualization and gives an introduction to the most common texture-based technique: the line integral convolution (LIC) method. First the algorithm is discussed for a simple 2D scenario, then it is reformulated to 2D texture advection, where the LIC integral is evaluated in a step-by-step manner with a foregoing tune-in phase. Finally, is is shown how LIC can be used for curved surfaces. Both texture advection and LIC on curved surfaces are important methods for the following algorithms.

Chapter 5: 3D Texture-based Visualization

This chapter deals with 3D texture advection on the GPU. It is shown how semi-Lagrangian texture advection can be mapped efficiently to textures and shaders in order to obtain an interactive dense representation of a 3D flow field. For this purpose, the concept of logical and physical memory is introduced, where the logical memory represents the 3D volume and the physical memory its respective mapping to the 2D textures. Since the access on 2D instead of 3D textures is much faster, the gain in performance is immense. Another detail of this implementation consists of the reordering of slices: according to axis-aligned direct volume rendering, the slices need to be reordered once the angle between viewer and slice normals exceeds a threshold in order to avoid visible artifacts. This also influences the advection process because the data is also stored axis-dependent and, therefore, a more detailed consideration is required.

Once the first visual representation is achieved by direct volume rendering, it can be further improved by illumination. In fact, three different illumination approaches are applied to 3D texture advection:

1. Gradient-based illumination. A real-time gradient computation is performed and the gradients serve as normals for Phong illumination.

2. Line-based illumination. This model avoids the expensive computation of gradients and instead uses the tangent vector as input. This is quite cheap because the tangents are already given by the entries of the vector field.

3. Perception-oriented illumination. NPR-techniques like cool-warm shading or halos are applied for emphasizing the structure of the particles inside the flow.

The individual techniques are finally compared in tables showing their performance measurements. The conclusion of this section is that 3D texture advection is a powerful method for interactively rendering 3D, even unsteady, flow fields.

In the second section of this chapter, 3D texture advection is embedded into a multiple views application in order to overcome the issue of occlusion. An additional slice-by-slice representation adopted from the traditional display of medical data is used for enable the user more insight into the 3D flow data. In order to support user interaction, the user is able to select isoslabs (intervals of isovalues) by marking spatial regions within the 2D representation. Masking is performed by mapping the marked regions to a criterion, like velocity magnitude or λ_2 vortex strength. The interaction is tightly linked to a 3D representation by 3D texture advection, where the unselected regions are faded out and the isoslabs are visualized. User interaction is further supported by an appropriate color coding, which is used to map the selected isovalues of the respective regions to the 2D flow representation. For this purpose, the individual channels of the HSV color space are used to combine the advected density values, the strength of the underling criterion, and the selected and unselected areas in a single representation.

Chapter 6: Flow Visualization on Curved Surfaces

This chapter discusses texture-based visualization on curved surfaces. First the creation of path surfaces is described by GPU-based particle tracing. It is shown how the GPU can be used to deal with diverging surfaces, i.e., how particle insertion and removal can be realized on the GPU. For this purpose, appropriate data structures are introduced and the usage of a binary tree is described that allows a reorganization of the particles without losing their neighbor information. This is crucial because the neighbor information is needed for the point-based rendering that is applied to obtain a closed surface representation out of a number of points (the integrated particles). For rendering the method of pointset surfaces (PSS) is used. In order to apply LIC for curved surfaces, PSS rendering is extended in a way it projects the vector field and the positions onto the image plane; this data serves as input for the curved surface LIC. Storing the projected velocity vectors for each integration step allows a texture-based rendering of path lines that are mapped onto the path surface, which can be used to clarify the reason why a path surface becomes wider or tighter in specific regions.

The concept of surface LIC is also adopted to surfaces that have been obtained by rendering isosurfaces of CT or MRI data. Although the issue to solve is another, the surface LIC algorithm is the same. In detail, the second section of this chapter considers the blending of multiple volumes, i.e., of brain activation data, which is rendered by direct volume rendering, and an enclosing MRI isosurface. The question is how to select the opacity of the enclosing isosurface that the interior–the brain activation data–is occluded as little as possible while the quality of the isosurface's shape doesn't suffer too much. This issue is solved by applying a LIC computation on the first principal directions of the isosurfaces curvature. This results in fine line-like structures on the isosurface, so that the isosurface's transparency can be chosen very high without losing any details.

In a third application, LIC is applied on extracted vortex regions, which are rendered as isosurfaces of λ_2 vortex strength. Since the flow close to those regions is

supposed to be tangential to this isosurface, surface LIC is well-suited for rendering of particles' trajectories. Once the particles are animated, a vortex's direction of rotation is perceptible at a glance. The perception is further improved by applying Phong illumination and cool-warm shading on the LIC representation.

Part 3: Particle Tracing for Feature Detection in Fluid Flows

The methods presented in this part are restricted to flow fields that result of flow simulations or measurements. The fluid flows serve as input for various flow feature detection mechanisms. This part includes the following Chapters 7 to 9.

Chapter 7: Methods for Computing Flow Features

This chapter describes methods from fluid dynamics that are used for the extraction of flow features. It starts with the decomposition of the velocity gradient tensor in its symmetric and antisymmetric parts. Furthermore, an overview of recent vortex core detection methods is given, whereby the Q criterion and the λ_2 criterion are discussed in more detail. This is followed by an in-depth discussion on vortex core lines, where their formal definition according to 1D height ridges is described. For their computation, a predictor-corrector algorithm is discussed. The next section introduces two criteria for computing regions of high shear stress. Finally, a short explanation of critical points is given.

Chapter 8: Feature Detection and Visualization

In this chapter, the detection and visualization of flow features is discussed. Beginning with the introduction of an importance function, it is shown how it can be used for masking purposes using 3D texture advection. The importance function is related to a feature function that is previously computed and stored in a 3D scalar field. The reliability of feature-based 3D texture advection is shown in terms of various examples.

The second section proposes a topology-preserving vortex core line detection method. As the name already implies, it considers, in contrast to previous methods, a vortex's topology. In detail, if a vortex has any bifurcations, they are regarded for the vortex core line computation as well. The algorithm bases on the isoskeletons of λ_2 isosurfaces in order to capture the vortices' topology. In a first step, the vortex core line is detected without considering bifurcations. Second, it is shown how bifurcations can be considered by exploiting the topological information delivered by the isoskeleton. Since the results achieved with this approach strongly depend on the selected isovalue of the respective λ_2 isosurfaces, a semi-automatic algorithm for the detection of an appropriate isovalue is presented. The section concludes with a comparison of the new method to previous vortex core line detection approaches.

This is followed by the third section, where shear stress is considered. Here, an appropriate visualization of this quantity is proposed–the shear sheets–which are defined as surfaces of maximal shear. The computation generalizes 1D height ridges to 2D. Shear sheets are computed according to λ_2 vortex core lines by a predictor-corrector method. However, due to the higher dimensionality of shear sheets, the predictor-corrector method needs to be modified, which is shown in terms of the new algorithm.

Finally, point-based rendering of shear sheets is discussed, where illuminated surfels are drawn to represent a smooth surface.

Chapter 9: Feature Tracking

This chapter describes the tracking of flow features. The basic algorithm performs a sampling of an extremum line of either a vortex or a shear layer criterion, which results in a number of particles. In the next step, the particles are integrated along the flow field in order to test for their intersection with a flow feature of the next time step. This chapter further discusses the graphical representation of tracked structures.

The second section extends vortex tracking by the consideration of vortex dynamics. The result is an algorithm that is able to find for a selected vortex structure its time of strongest perturbation, i.e., the time another vortex strongest impact the selected vortex. This enhances the investigation of turbulent flow because more feedback is given on the creation and the temporal development of vortex structures.

The third section gives a detailed introduction to the temporal creation of shear layers and their role in the vortex creation process. It is shown why shear layers are indispensable for feature tracking and for a correct understanding of a flow. This section concludes with the combined tracking of vortices and shear layers, taking into account vortex $\leftrightarrow$ shear layer transitions.

Part 4: Light Tracing

Particle tracing is applied for the tracking of photons. The trajectories are either given by the perspective of the human eye or by force fields. This part includes Chapters 10 and 11.

Chapter 10: Linear Light Tracing

This chapter describes particle tracing in terms of tracking photons on their way to the human eye. The first case discusses GPU-based ray casting on AMR grids. For this purpose, two different kinds of data structures are discussed, which are both used to hold the AMR data while providing the possibility of fast texture accesses. Both methods, the octree texture and the adaptive page table, are discussed in detail and compared against each other.

This is followed by an example from computer graphics. Here, atmospheric light scattering is computed using ray casting. In contrast to the example above, there exists no grid containing any scalar values—here, a continuous scattering function is evaluated. The section starts with a short introduction to Rayleigh and Mie scattering and the basic equations. Then the generation of the scattering texture is explained. The discussion on the correct access of this texture shows how easy this method also can be applied for rendering terrain instead of plain spheres.

Chapter 11: Nonlinear Light Tracing

This chapter discusses nonlinear ray tracing. In contrast to linear ray tracing, the perspective is deformed by bending the view rays. The view rays, considered as trajectories of photons, are influenced by a force field. First the basic algorithm of GPU-based

nonlinear ray tracing is described, followed by an example where nonlinear panorama maps are generated. Here, the force field is given by the gravity of the terrain and leads the view rays to magnify tight valleys or give the possibility to look around a corner. In fact, two different approaches are discussed: the normal map approach, which is faster, requires less memory, but unfortunately is also less flexible; the volumetric approach, which makes use of 3D textures but gives the user more flexibility by manipulation. Both approaches are compared for their advantages and disadvantages.

Conclusion

This chapter concludes this work. The wide range of applications of particle tracing is summarized in a flow diagram.

ZUSAMMENFASSUNG

Diese Doktorarbeit behandelt die verschiedenen Anwendungsmöglichkeiten der Partikelverfolgung. Den Schwerpunkt bildet hierbei der Einsatz von Algorithmen für die Partikelverfolgung in der Strömungsvisualisierung. Die Arbeit beginnt mit einer Einführung in die Grundlagen verschiedener Themenbereiche wie Visualisierung, Computergraphik, Mathematik und Strömungsmechanik. Auf diesen Techniken aufbauend werden in den folgenden Kapiteln mehrere Methoden zur Verwendung der Partikelverfolgung im Bereich der Visualisierung und der Computergraphik vorgestellt. Der erste Teil dieser Arbeit befasst sich dabei vor allem mit texturbasierten Techniken, welche, um eine interaktive Dichterepräsentation dreidimensionaler Vektorfelder zu gewährleisten, auf Grafikhardware (GPU = Graphics Processing Unit) umgesetzt wurden. Dabei zielen die in diesem Teil der Arbeit vorgestellten Methoden auf allgemeine Vektorfelder ab, welche mit Hilfe texturbasierter Volumen- und Oberflächendarstellungen repräsentiert werden sollen. Desweiteren wird beschrieben, wie diese Methoden dazu verwendet werden können, Strukturen aus Strömungsfeldern zu extrahieren, wie z.B. zeitabhängige Strömungsflächen.

Der zweite Teil dieser Arbeit behandelt die Partikelverfolgung in abgeleiteten Strömungsfeldern, wie sie in der Strömungsvisualisierung zu finden sind. Dabei werden zuerst mittels Methoden der Strömungsmechanik sogenannte Strömungsmerkmale aus dem Strömungsfeld extrahiert, welche dann als Ausgangsbasis für die Verwendung der Partikelverfolgung herangezogen werden. Die Datenrepräsentation der Strömungsmerkmale erfolgt in Skalarfeldern, wobei deren Ableitungen höherer Ordnung wiederum zur Berechnung von charakteristischen Strukturen, wie z.B. 1D Wirbelkernen oder 2D Scherschichtstrukturen verwendet werden. Die extrahierten Strukturen werden außerdem auf ihre zeitliche Veränderungen untersucht, indem sie entlang der Zeit verfolgt werden.

Im dritten Teil wird die Anwendung der Partikelverfolgung in verschiedenen Bereichen der Visualisierung und der Computergraphik diskutiert. Hierfür werden die Partikelbahnen entweder über die Perspektive des menschlichen Auges oder über Kraftfelder beschrieben, welche die Partikelbewegung durch Differentialgleichungen zweiter Ordnung beeinflussen.

Jeder dieser drei Teile der Arbeit versucht die wichtige Rolle der Partikelverfolgung in einem breiten Spektrum von Anwendungen in der Visualisierung und der Computergraphik anhand verschiedener Beispiele zu verdeutlichen. Desweiteren wird dadurch die Flexibilität dieser Methode durch Verwendung verschiedener Vektorfelder verdeutlicht und gezeigt, wie diese generiert werden, um auch Probleme jenseits der klassischen Partikelverfolgung in Strömungsfeldern zu lösen.

ACKNOWLEDGMENTS

I am grateful to my advisors Thomas Ertl, who gave me the chance to spend the last years in research, and Daniel Weiskopf for all the fruitful and sometimes exhausting (this is well-meant) discussions about any kind of theoretical and technical problems. Thanks for your enduring willingness for collaborating with me, even when you were located on the other side of the Atlantic.

Many thanks to Ralf Botchen for all the discussions on and off the job, for the motivation we gave each other, and all the coffee breaks (which were actually only taken for brainstorming). Furthermore, I would like to thank you for being my climbing partner for the last years. Further thanks go to Friedemann Rößler— who was an irreplaceable part of our "coffee team"—for the extensive discussions on curvature and all the brainstorming sessions at Columbus after work. I also want to thank my long-term room mate Eduardo Tejada for all the constructive mathematical discussions, for all the hard-working nights we spent at the office for our common projects, for teaching me the role of a skewered cow for a "real barbecue", and for sharing all the other culinary highlights with me. Many thanks to my short-term room mates Simon Stegmaier, who introduced me to climbing, and Markus Üffinger for the endless discussions about the conservation of mass and momentum. Not to forget Joachim Vollrath for his encouraging engagement during our search for vortex core lines, and Martin Falk for all the rays we bent together.

Thanks to Kudret Baysal and Ulrich Rist from the Institute for Gasdynamics and Aerodynamics for all the advice about fluid dynamics and the collaboration we had all the time. I am further thankful to João P. Gois and Antonio Castello from Brasil for the detailed discussions on mathematical formalism and Morse theory.

I express my gratitude to all the students who supported me to achieve my goals (in alphabetical order and without the persons who became my colleagues in the meantime): Tjark Bringewat, Thomas Derr, Markus Hocke, Pablo Nigorra, Jörg Oberfell, Clemens Spenrath, and Mikael Vaaraniemi.

I want to thank all my colleagues for the nice time I had at the Visualization and Interactive Systems Institute. In particular, these are (without the persons already mentioned above): Sven Bachthaler, Wolfgang Bayerlein, Katrin Bidmon, Harald Bosch, Marianne Castro, Carsten Darchsbacher, Joachim Diepstraten, Mike Eissele, Shymaa Elleithy, Thomas Engelhardt, Steffen Frey, Mark Giereth, Frank Grave, Rul Gunzenhäuser, Sebastian Grottel, Gunter Heidemann, Julian Heinrich, Marcel Hlawatch, Benjamin Höferlin, Andreas Hub, Steffen Koch, Sebastian Klenk, Steffen Koch, Hermann Kreppein, Andreas Langjahr, Dietmar Lippold, Julia Möhrmann, Christoph Müller, Thomas Müller, Christian Pagot, Guido Reina, Matthias Ressel, Martin Rotard, Filip Sadlo, Harald Sanftmann, Thomas Schlegel, Martin Schmid, Bernhard Schmitz, Waltraud Schweikhardt, Christiane Taras and Michael Woerner.

Finally, I am grateful to my girlfriend Sonja for all the energy and support she gave me, even in hard times, for always believing in me, and for sharing and fighting all the

sorrows I had. You were always be there for me. I also want to thank my and her family for always giving me good advice, for sharing my happiness and my doubts, and for helping me not to give up but rather to continue my work. Thanks for your backup.

Tobias Schafhitzel

INTRODUCTION

The motion of particles plays a crucial role in visualization and computer graphics. Particularly in flow visualization, massless particles are used to explore the behavior of a flow. The application of particles reaches from the moment a flow is measured or simulated to its final visualization. Considering flow measurements like PIV (particle image velocimetry), particles are seeded inside a flow in order to record their motion. Then a vector field is created exploiting the gathered information, which is used in turn for visualizing particles. This makes particles indispensable for fluid mechanics as well as for flow visualization. From the visualization's point of view, particles provide a good way to support the perception of steady or unsteady 2D and 3D flow fields. For this purpose, various techniques have been developed for emphasizing a particle's path of motion, like stream or path lines, streak lines, and time lines.

In this thesis, it will be shown how these basic techniques can be used for several methods in visualization and computer graphics that go beyond the conventional application of particle tracing inside flow fields. This work is separated into four main parts: Part I introduces in basic techniques of visualization and computer graphics, mathematics, and fluid dynamics that are relevant for the proposed methods. Part II discusses the application of particle tracing in terms of real-time texture-based flow visualization. Here, classical particle tracing is performed using stream lines, path lines, and streak lines. Furthermore, the class of path lines is extended to path surfaces, i.e., a 2D representation inside a 3D domain. This part deals with general vector fields, regardless of whether they are fluid fields or vector fields derived from any computations. Part III becomes more specific in terms of the underlying vector field. Here, only fluid fields are investigated in order to compute flow features. Flow features play an important role for the extended exploration of flow fields, particularly of 3D flow fields. In 3D, flow features can either serve as masking functions, i.e., particle tracing in terms of stream lines, path lines, or streak lines is performed in the same manner as described in Part II of this thesis with additional masking enabled. Then only areas of interest are rendered. But flow features–which are mostly derived directly from the fluid field–can also serve as potentials for other vector fields, which are usually the first or second order derivatives of the feature field and where also particle tracing can be performed. These vector fields can in turn be treated like the vector fields in Part II, where stream lines or any other flow structures are computed. Because the computation works on derived vector fields, the resulting structures mostly have a certain physical meaning, like vortex core lines, which use λ_2 as input, or shear sheets, which use a measure of shear stress as input. This part demonstrates the flexibility of particle tracing and the dependence on the vector field used for integration. Part IV further generalizes the application of particle tracing. Here, the particles' trajectories consist of either a linear or nonlinear representation of the human's eye perspective.

In general, the motion of massless particles can be expressed by the ordinary differential equation for Lagrangian particle tracing. This equation reflects the dependency

of the new position of a particle on the underlying vector field and, therefore, builds the base of this work. All the parts of this work make use of this equation; in most cases only the underlying vector field is exchanged. We distinguish between three kinds of vector fields:

1. Fluid fields, i.e., vector fields derived from flow simulations or measurements. Usually, this kind of vector field is directly used for particle tracing in order to emphasize the trajectories inside the flow.

2. Derived vector fields. In this case, usually the vector field is derived from a potential that is the result of a computation on the input fluid field or, which is also shown in this work, of the derivatives of an arbitrary scalar data set. This class of vector fields is used for creating geometrical representations like curvature lines, extremum lines, and extremum surfaces which are, in the case of fluid fields, only indirect linked to the original vector field.

3. Force fields. Although this kind of vector field influences the motion of a massless particle, it has only indirect impact on its velocity and direction. However, by solving second order ordinary differential equations and with the knowledge of the initial velocity, particles can also be traced along that kind of fields.

Now we can see that the range of applications of particle tracing is very broad; the only dependency is of the underlying vector field and its reasonableness—and this thesis discusses how those vector fields can be derived and how particle tracing can be applied on them.

This work investigates the statement that a wide range of problems in visualization and computer graphics, which apparently have no commonness, can be still reduced to a common denominator: the motion of massless particles. It does not matter if a particle is actually a fluid particle or a photon (a light particle)—it is just a particle and actually makes use of only two ordinary differential equations (not in general, but in this thesis): the change of the particle's location in dependance of a direction vector and the change of the direction vector in dependance of a force function.

Part I

Graphics and Visualization Techniques

CHAPTER

1 PRINCIPLES OF COMPUTER GRAPHICS

All the methods discussed in this thesis rely on a number of techniques, basically from visualization and computer graphics, mathematics, and fluid dynamics. Since this thesis has a strong focus on visualization and computer graphics, the most relevant basics of this area are discussed in this chapter. Starting with the visualization and rendering pipelines, which represent the visualization and rendering process on an abstract level, this chapter then goes in more detail: the grid types which are used in this work are introduced and compared against each other. Then the volume integral and indirect volume visualization are introduced, since they build the base of various techniques discussed in this work.

1.1 Visualization Pipeline

The visualization pipeline in Figure 1.1 describes the complete visualization process from data generation to the final visualization. To the beginning, data is acquired from an arbitrary data source. In this thesis, most data is acquired from flow simulation and flow measurements. Usually this data, called raw data, is not appropriate for visualization, since it might contain data which is not interesting for the visualization process or the data is not in the desired format. Therefore, unnecessary data need be filtered out first in order to obtain only the data that is desired for visualization. Beside data segmentation and clipping, this stage also contains interpolation and smoothing algorithms as well as an optional resampling of the data.

In the next step, the data is mapped to graphical primitives. In this thesis, vector fields and scalar fields are used, which are mapped to graphical representations like volumes (Chapter 5) and isosurfaces (Chapter 8), respectively. Additionally, the attributes of the graphical primitives can be used for representing further information, e.g., the strength of a vortex can be mapped to the color of its isosurface (Chapter 9).

The last step renders the graphical primitives in form of geometry or volumes. Techniques like shading, lighting or additional shadows support the perception of the scene. Shading is a common method nowadays and, therefore, is applied to almost all visualizations in this thesis.

1.2 Rendering Pipeline

The rendering pipeline represents the processing of information on the graphics hardware. Outgoing from a CPU-based application, the vertices are passed through the rendering pipeline shown in Figure 1.2 in order to achieve a graphical representation.

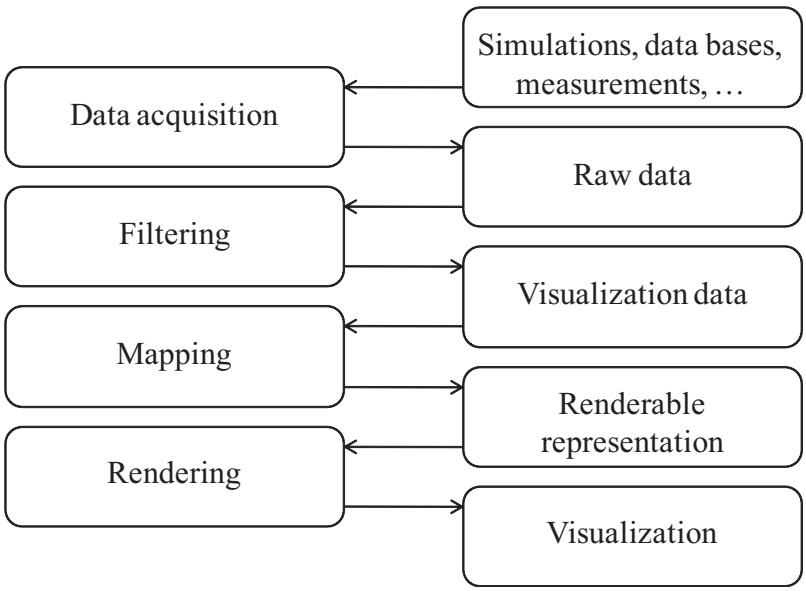


Figure 1.1: The visualization pipeline: starting with the data acquisition from simulations, measurements or data bases, the data undergoes four processing stages until the final representation is achieved–the visualized image.

The usual way of a vertex sent from the CPU to the graphics processing unit (GPU) consists of transformations and different tests performed on the GPU. First, the light computation as well as the definition of the texture coordinates is applied on the vertex given in wold coordinates. Then the vertex is transformed into eye coordinates, i.e., in a coordinate system defined by the position of the camera in world coordinates, which is also used for culling. In the next step, the vertex is projected onto the view plane, where it is represented by its clip coordinates. Here, clipping and the removal of hidden surfaces is performed. Finally, perspective division is applied to obtain normalized device coordinates which are transformed to window coordinates (viewport transformation). After the transformations the rasterization is performed, i.e., the graphical primitives are converted in a raster format in window coordinates. During this stage vertex attributes are interpolated, per-fragment processing and texturing as well as hidden surface removal are performed.

1.2.1 The Programmable Rendering Pipeline

As long as the individual stages are not influenced by the user we are talking about the *fixed function* pipeline. However, modern graphics hardware allows the user to manipulate certain parts of the rendering pipeline, namely three stages in recent hardware displayed as gray boxes in Figure 1.2. Here we are talking about the modern *programmable* pipeline. All the three stages can be programmed by shader code, which is actually assembler code. Since writing code in assembler is very awkward, in the last years several high-level shader programming languages have been developed.

For the following discussion, we differentiate between the two most important graphics APIs, OpenGL and DirectX. While OpenGL is an extensible, open standard that provides more experimental features, DirectX is developed and supported only by

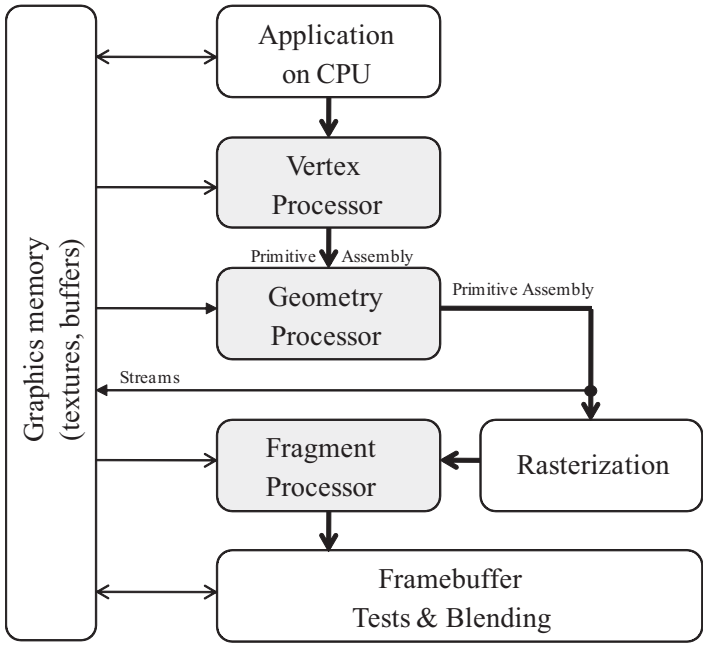


Figure 1.2: The simplified rendering pipeline [31]. The gray boxes represent the programmable parts of the recent rendering pipeline. The application on the CPU sends vertices and attributes to the graphics card, where the information is passed through the different stages denoted by the boxes above. Note that all programmable units are optional and can be replaced by the fixed function pipeline.

one vendor (Microsoft) and provides in many cases the more efficient implementations. The reason is the commercial use of DirectX, since it builds the standard graphics API for computer games and, therefore, the gaming industry is quite interested in the further development of DirectX. Also both APIs possess their own high-level shader languages: GLSL (GL hading language) [140] for OpenGL and HLSL (high-level shading language) for DirectX. An alternative can be found in NVIDIA's CG (C for graphics) [108; 44], which supports both APIs. All three shader languages have a C-like syntax.

The first programmable unit of the rendering pipeline consists of the vertex processor. The input data consists of single vertices and their respective attributes like texture coordinates and color. Note that no information about the graphical primitive is available in the vertex processor. The output data is the projected vertex with its new texture coordinates. These texture coordinates can also be used to transfer vertex dependent data between the different programmable units. Furthermore, in the vertex shader (the implementation), the access to the constant registers is permitted. Additionally to the vertex position and the texture coordinates, color and fog are written to the output. Actually, this provides the user the possibility to write any data into the output register that should be interpolated for the following pixel shader.

The vertex processor is followed by the geometry processor. In contrast to the vertex processor, the geometry processor works on graphical primitives and, therefore, the connection information must be available additionally to the vertex position. For this purpose, the primitive assembly is performed before the geometry shader which

is in charge for the generation of new graphical primitives. For example, this stage allows the creation of line segments out of a single vertex and, therefore, can be used to significantly reduce the transfer load between CPU and GPU. Usually, the type of graphical primitive for the input and the output data is defined externally by the application on the CPU. It also should be mentioned that the result of the geometry shader either can be transferred to the rasterization or, alternatively, can be written to a stream that is processed a second time (in a second render pass). If a prompt visualization is desired, the geometry is passed to the rasterization. Again, first primitive assembly has to be performed.

After rasterization the fragment processor enables the access to the resulting fragments (pixel with additional information). Here, shading can be performed, e.g., per pixel lighting can be implemented, or the textures can be mapped onto objects, or any other image-based computation can take place. Actually, the result is always a fragment color and an optional depth value, except for the case when the fragment is discarded. The values of these registers are finally passed to the framebuffer tests (depth test, stencil test, alpha test, alpha blending) and are, after successfully passing these tests, drawn to the framebuffer.

As already shown in Figure 1.2, each of the three programmable units have access to the graphics memory. This makes texture fetches possible in all shader programs. Similar to the geometry processor, the framebuffer can also be bound to the graphics memory in order to write into it. This can be done by render targets, which consist of textures of a certain format and are used to save the contents of the framebuffer to textures. This mechanism is often used for multi-pass computations that need the result of a foregoing pass for further computations. GPGPU computations are an example for that and are explained in the next section. Although rendering into a 2D texture is quite intuitive by using a 2D render target, rendering into a 3D texture, though, is more complex and is only supported by the newest generation of graphics hardware in combination with the newest graphics APIs (supported by both DirectX10 and OpenGL 2.0).

Another issue consists of the similar read and write access to render targets. This is the case when the data of a previous rendering pass, which is stored in a 2D render target, should be serve as input for the following render pass whereby the result should be written at the same position inside the 2D texture. Since this action is either not possible (DirectX) or not specified (OpenGL), the well-established workaround of enabling a so-called ping-pong rendering is quite common. Here, an additional copy of the 2D render target is created, which is bound alternately with the second 2D render target to the rendering device. The idea is to use the one 2D texture as input while the other is used for writing the result. Then the textures are swaped for the next render pass; the texture previously serving for writing now provides the input data while the other texture is used for writing. This mechanism has been also used in this thesis, e.g., for 3D texture advection (Section 5.1).

1.2.2 General-Purpose Computation on the GPU

In contrast to CPUs, the GPU is well-suited for performing applications in parallel. In detail, though CPUs can also run programs with different threads, the number of threads is usually tightly linked to the number of processors of the respective machine

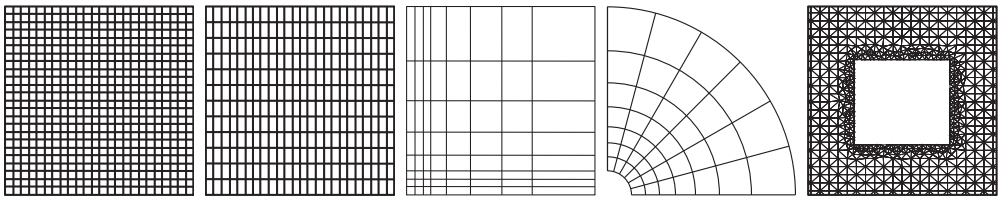


Figure 1.3: Different grid types: cartesian grid; uniform or regular grid; rectilinear grid; curvilinear grid; unstructured grid.

(nowadays quadcore processors are the state of the art). The GPU works, though it is more restrictive, very parallel in the fragment processing stage. This makes the GPU also interesting for general-purpose computations (GPGPU) [93; 80]. Since in shader model 3.0 also branching and loops are supported, what makes their usage for GPGPU even more attractive.

The realization of a GPGPU program is simple: the fragment shader represents the algorithm that should be performed. Thereby, each fragment represents one "object". For all fragments the same fragment shader is performed in parallel. The results are written in write-only textures attached to the framebuffer. These texture are either used for further processing on the GPU or read back to system memory.

Recently, the compute unified device architecture (CUDA) [32] has been developed for integrating GPGPU code into C programs. Although CUDA has not been used in this thesis, some GPGPU algorithms are presented in Section 6.1.

1.3 Grid Types

Since the methods described in this thesis mostly work on discrete data, the scalar fields and vector fields are represented by grids. In practice, there exist a large number of grid types, mostly variations of the basic grid types shown in Figure 1.3. Starting from the left to the right, two general types of grids are shown: structured (1–4) and unstructured (5) grids. The difference between both types lies in the connectivity, where only for structured grids the connectivity is given implicitly. Therefore, the topology of unstructured grids is irregular. Which grid type is actually used in practice depends on the aim of the respective simulation or measurement.

Cartesian grids are the most appropriate grids for graphics processing, since data access is very fast and easy. Due to the fact that the distance between the grid points is constant, most algorithms become simpler. The computation of numerical derivatives (Section 2.1) is quite fast, as well as the data interpolation (Section 2.5). The mapping to world space for this grid type is given by the indices (i, j, k); the grid cells appear as cubes in 3D.

Uniform grids are similar to cartesian grids, except for the fact that the distance between the grid points is not uniform in all directions. Here, the distances are only required to be uniform along the respective direction. This leads the shape of a grid cell to appear as a cuboid. Similar to cartesian grids, the mapping to world space is simple and is given by $(i \cdot \Delta x, j \cdot \Delta y, k \cdot \Delta z)$, where Δx, Δy, and Δz are the grid point distances in the respective directions. This mapping also allows a fast computation of numerical derivatives.

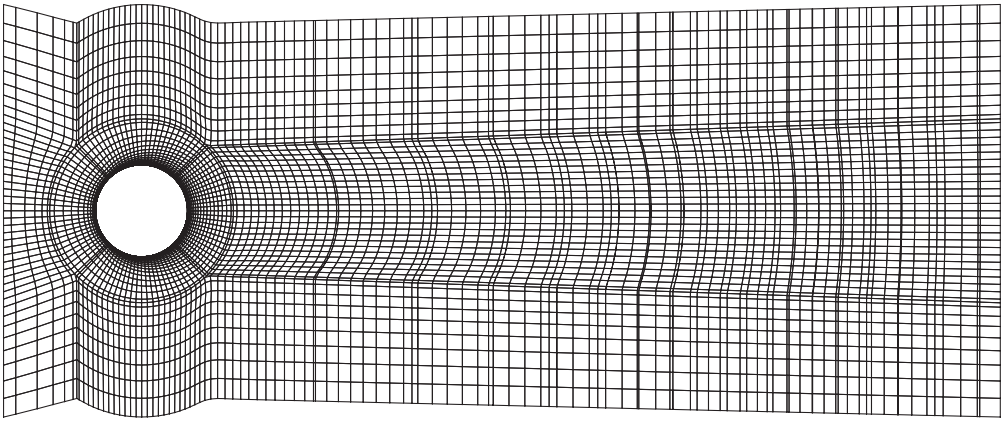


Figure 1.4: Curvilinear grid around a cylinder to provide an optimal sampling of the data set.

More flexibility in terms of spatial sampling is given by rectilinear grids. In practice, rectilinear grids are often used to enable adaptive sampling, i.e., a higher sampling in regions of interest, whereas the remaining regions are sampled with a lower resolution. In fluid mechanics, often the regions close to boundaries and vortex detachment regions are given in a high resolution. Although this new structure does not influence the simplicity of storing the data, the mapping to world coordinates becomes more complex and requires a mapping function stored in an array $(x[i], y[j], z[k])$.

Although rectilinear grids provide the flexibility of adaptive grid modeling, they are rather unsuitable for sampling bent objects, e.g. a cylinder. Figure 1.4 shows an example of a curvilinear grid. As already mentioned above, rectilinear grids are well-suited for the use close to boundary regions. For the cylinder, however, it makes more sense to use a grid that aligns to the curved shape. The reason is simple: using curvilinear grids instead of rectilinear grids obviously reduces the number of sample points. The mapping of curvilinear grids to world space becomes further complex: the indices i, j, and k used for describing the positions according to the different directions are not required to lie in this direction anymore. Therefore, the mapping requires all indices for input $(x[i, j, k], y[i, j, k], z[i, j, k])$.

In contrast to the grid types discussed previously, unstructured grids can be used for an efficient sampling of a complex object. However, this grid type has an irregular topology, i.e., no implicit connectivity information is given. This makes it less efficient to store the data. As it will be shown later in Section 2.5, interpolation within an unstructured grid is more expensive than for the other grid types.

In this thesis, only cartesian and regular grids are used, since they can be directly mapped to the GPU as textures.

1.4 Volume Visualization

1.4.1 Direct Volume Visualization

In Chapters 4 and 10, direct volume visualization is applied for the visualization of 3D scalar fields. This section discusses the basics of direct volume rendering. The goal

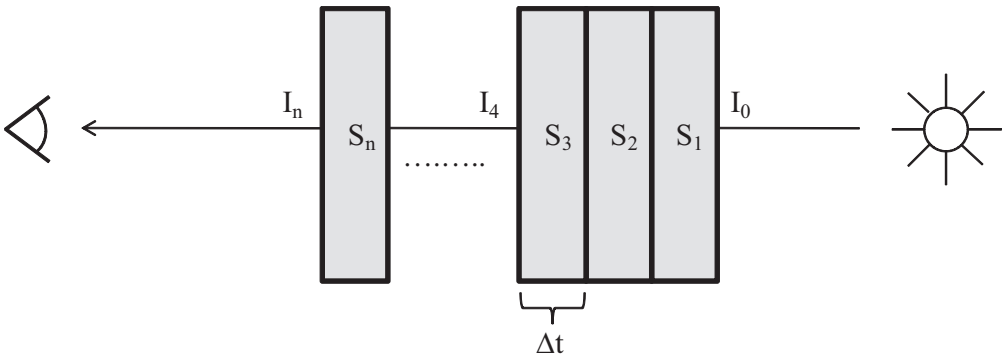


Figure 1.5: Light penetrating the slabs S_1 to S_n. At the first slab, the full light intensity I_0 arrives. After each slab, this intensity is more attenuated and, therefore, the light reaches the observer with intensity I_n.

of this method consists of a representation of non-opaque material, where three effects should be considered: emission, absorbtion, and scattering. The first effect is a result of chemical reactions or excitation on an atomic level, the latter effects both can be observed in clouds: a cloud appears dark when the light is absorbed and bright when the light is scattered. A more detailed discussion on light scattering can be found in Section 10.2.

The basic idea of direct volume rendering is quite simple. The question is: "how much light would arrive from a light source I_0 after penetrating an object of thickness Δt? ". The following discussion answers that question and describes how the volume rendering integral can be derived [109]. Please note that only the absorbtion is regarded in this model.

Formulating the question above in an equation would result in the Lambert-Beer law, which determines concentrations with spectroscopic methods

$$I = I_0 \cdot e^{-\epsilon_\lambda c \Delta t} ,$$

where c denotes the concentration of the penetrated substance and ϵ_λ the extinction coefficient, which depends on the wavelength. Assuming a volume consisting of n slabs S_n as illustrated in Figure 1.5 with the respective concentrations c_i and the extinction coefficients ϵ_{λ_i}. Then, if there is no energy loss outside the volume, the light intensity leaving the first slab S_1 and reaching the second slab S_2 is defined by

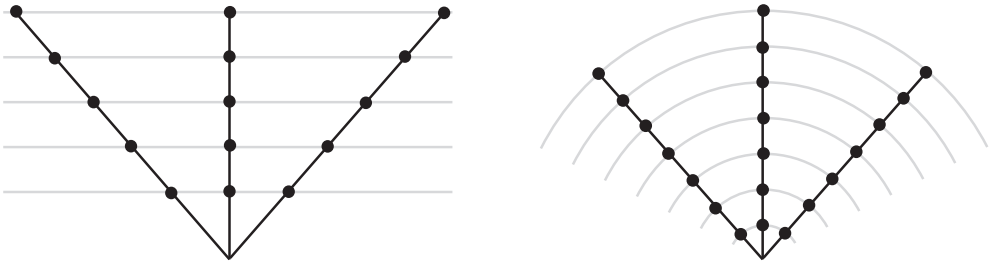
$$I_1 = I_0 \cdot e^{-\epsilon_{\lambda_1} c_1 \Delta t} .$$

Applying this equation is a recursive manner, the light reaching slab S_3 then is defined by

$$\begin{aligned} I_2 &= I_1 \cdot e^{-\epsilon_{\lambda_2} c_2 \Delta t} \\ &= \left(I_0 \cdot e^{-\epsilon_{\lambda_1} c_1 \Delta t} \right) \cdot e^{-\epsilon_{\lambda_2} c_2 \Delta t} \\ &= I_0 \cdot e^{-(\epsilon_{\lambda_1} c_1 + \epsilon_{\lambda_1} c_1) \Delta t} , \end{aligned}$$

and in general for n slabs by

$$I = I_n = I_0 \cdot e^{-\sum_{i=1}^{n} \epsilon_{\lambda_i} c_i \Delta t} .$$



(a) (b)

Figure 1.6: Slice-based volume rendering (a) vs. ray casting (b). Although the slice-based approach is faster in most cases, its sampling rate is not constant. Ray casting maintains a constant sampling rate, but the evaluation of the blending function often is more expensive.

In the continuous case, i.e., when the slice thickness $\Delta t \to 0$ and we integrate over all slices $T = n \cdot \Delta t$, the light intensity reaching the observer's eye is given by

$$I = I_0 \cdot e^{\int_0^T \epsilon_\lambda(t)c(t)dt} ,$$

with the continuous functions of concentrations $c(t)$ and the extinction coefficients $\epsilon_\lambda(t)$. For volume rendering a 3D scalar field is given, where the scalar values should describe the extinction. Therefore, the ray parametrization t is replaced by a position on the ray $\mathbf{x}(t)$. The extinction coefficients and concentrations are merged to a global extinction τ. This results in the following equation, which already could be used to generate X-ray like visualizations

$$I = I_0 \cdot e^{-\int_0^T \tau(s(\mathbf{x}(t)))dt} . \tag{1.1}$$

The function $s(\mathbf{x}(t))$ returns the scalar value of the given scalar field at a certain position $\mathbf{x}(t)$. Unfortunately, Eq. 1.1 only considers the light which was attenuated while penetrating the volume; no colors are considered so far. Keep in mind that only a scalar field is given as input, i.e., the color mapping has to be defined by a *transfer function*. This transfer function can also be used to map the extinction τ to a user-defined value. Introducing a scalar value dependent color value $\mathbf{c}(s(\mathbf{x}(t)))$ at a specific position, then its contribution is attenuated to its way to the observer. When the light passing the complete volume is additionally considered, then the resulting color is determined by integrating over all color values along the ray by

$$\mathbf{C} = I_0 \cdot e^{-\int_0^T \tau(s(\mathbf{x}(t)))dt} + \int_0^T \mathbf{c}(s(\mathbf{x}(t'))) \cdot e^{-\int_{t'}^T \tau(s(\mathbf{x}(t)))dt} dt' , \tag{1.2}$$

where the second term attenuates the color intensity emitted at $\mathbf{x}(t')$ to the observer.

Since the evaluation of the volume rendering integral in Eq. 1.1 is very expensive, in practice it is usually approximated by either slice-based volume rendering [18; 33], ray casting [76; 100; 101], cell projection [159] or splatting [199]. Because in this thesis only the first two methods are used, this discussion does not consider cell projection and splatting. Both methods discretely sampling the light ray and, therefore,

the quality depends on the sampling rate. Slice-based volume rendering uses, as the name already implies, semi-transparent slices for sampling a volume. If the number of slices goes to infinity, the approximation converges. The slices are either drawn from back to front and are blended by $\mathbf{C}^{\mathrm{out}} = \alpha\mathbf{C} + (1-\alpha)\mathbf{C}^{\mathrm{in}}$ or front to back with $\mathbf{C}^{\mathrm{out}} = \mathbf{C}^{\mathrm{in}} + (1-\alpha^{\mathrm{in}})\alpha\mathbf{C}$ and $\alpha^{\mathrm{out}} = \alpha^{\mathrm{in}} + (1-\alpha^{\mathrm{in}})\alpha$. Front to back blending is computationally more expensive because a separate computation of the color and the alpha value is required. Due to perspective projection, the sampling distance is not constant. Figure 1.6 illustrates this issue.

In contrast to slice-based volume rendering, ray casting guarantees a constant sampling rate. Here, the rays are shot from the observer through the scene, which implies front to back blending. Both methods are quite fast, even though not very accurate [40] because they only approximate the result. The work by Engel et al. [40] shows a way how the quality of volume visualization can be improved by pre-integration.

In this thesis, slice-based volume rendering is used as well as ray casting. In Section 5.1.1 and Section 8.1, slice-based volume rendering has been applied for visualizing particles inside a 3D flow. Section 6.2 uses this approach for rendering anatomical data. Last but not least, in Section 10.2, ray casting is applied for drawing the Earth's atmosphere.

1.4.2 Indirect Volume Visualization

Some words should be spent on *indirect* volume visualization. Here, in contrast to direct volume visualization, an intermediate representation in form of isosurfaces is generated. Each isosurface displays the material inside a scalar field which is similar to the selected isovalue. In 3D, this results in a surface representation, whereas direct volume visualization draws the complete volume. Nevertheless, indirect volume visualization is commonly used in the field of medical visualization in order to emphasize specific structures, like bones or soft tissue (see Section 6.2) as well as in flow visualization in order to extract vortex or shear layer structures (see Chapter 8).

The most popular algorithm for indirect volume visualization is the marching cubes (MC) algorithm [107]. The algorithm proceeds in a cell-by-cell manner, i.e., the intersection of the isosurface is determined for a cell before the algorithm is marching to the next cell. For each grid point of the cell, the value 1 is assigned if the corresponding scalar value is equal or higher than the current isovalue and 0 if it is below. For each cube this results in $2^8 = 256$ possible states. The MC algorithm is used for mapping each state to a graphical representation, i.e., a geometry that represents the isosurface within the grid cell. Usually, the eight-bit code that holds the information at the grid points is used for a lookup in order to obtain the respective graphical representation.

In Section 6.2 and Section 8.3, the isosurfaces are generated according to the MC algorithm. The λ_2 isosurfaces in Section 6.3 are determined by ray casting while the λ_2 isosurfaces in Section 8.2 are created by a topological correct MC implementation by Lewiner et al. [103].

1.5 Test Hardware Configuration

All the implementations discussed in this thesis are based on C++ and DirectX 9.0 or OpenGL, and were tested on an AMD Athlon 64 X2 Dual 4400+ (2.21 GHz) CPU (2 GB of RAM) Windows XP machine with one or both of the following PCI Express GPUs: ATI Radeon X1900 (512 MB) and NVIDIA GeForce 8800 GTX (756 MB). All shader programs and effect files are formulated in high-level shading language (HLSL,GLSL,CG) to achieve a code that is easy to read and maintain.

CHAPTER

2 MATHEMATICAL BASICS

This chapter introduces fundamental methods from mathematics which are used in this thesis. This includes the computation of first and second order numerical derivatives. The computation of surface curvature is described in Section 2.2 and is used later in Section 6.2. The determination of eigenvalues and eigenvectors (Section 2.3) plays an important role in the computation of flow features, which are discussed in Chapter 7 in this work. Since the focus of this work lies on the ordinary differential equation for Lagrangian particle tracing, the numerical integration of it is described in Section 2.4. Finally, in Section 2.5, trilinear and barycentric interpolation are introduced.

2.1 Numerical Derivatives

Since in this thesis only cartesian grids are used, only the method of finite differences is discussed for computing numerical derivatives. According to the Taylor expansion around a point x, the value of a function f can be obtained by

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \dots . \tag{2.1}$$

By rearranging Eq. 2.1, the derivatives of f can be computed according to forward/backward differences

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) , \tag{2.2}$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h) , \tag{2.3}$$

or central differences

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) . \tag{2.4}$$

The errors $\mathcal{O}(h)$ and $\mathcal{O}(h^2)$ result from the higher order terms of Eq. 2.1.

2.1.1 Gradient Computation

A gradient ∇f of a function f points in direction where the function value changes most strongly. In visualization gradients of 3D scalar fields are commonly used as

normal vectors of isosurfaces or for computing interval lines in terms of scalar field topology. A gradient in 3D is defined by

$$\nabla f = \begin{pmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{pmatrix}, \tag{2.5}$$

where the vector components are the first order derivative in the x, y and z-direction and can be solved numerically using finite differences.

2.1.2 Second Order Derivatives

Let $\boldsymbol{v}$ be a 3D vector field with $\boldsymbol{v}(\boldsymbol{x}) = (v_x, v_y, v_z)^T$. Then its derivative is defined by a 3×3 tensor, namely the Jacobian

$$J = \begin{pmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{pmatrix}. \tag{2.6}$$

If $\boldsymbol{v}$ has a potential f, i.e., $\boldsymbol{v} = \nabla f$, then the second order derivative of f is defined by its Hessian

$$\mathcal{H}f = \begin{pmatrix} \frac{\partial f}{\partial^2 x} & \frac{\partial f}{\partial x \partial y} & \frac{\partial f}{\partial x \partial z} \\ \frac{\partial f}{\partial y \partial x} & \frac{\partial f}{\partial^2 y} & \frac{\partial f}{\partial y \partial z} \\ \frac{\partial f}{\partial z \partial x} & \frac{\partial f}{\partial z \partial y} & \frac{\partial f}{\partial^2 z} \end{pmatrix}. \tag{2.7}$$

For both the Jacobian and the Hessian, the directional derivative along $\boldsymbol{x} = (x, y, z)^T$ can be obtained by $\boldsymbol{x}' = J \cdot \boldsymbol{x}$ and $\boldsymbol{x}' = \mathcal{H}f \cdot \boldsymbol{x}$, respectively. In contrast to the Jacobian, the Hessian is always symmetric and, therefore, has three real eigenvalues.

2.2 Surface Curvature

The first principal (curvature) direction is defined as the direction along the highest curvature on a surface. The corresponding curvature strength is called first principal curvature. The second principal direction indicates the direction to the flattest area. By construction, both principal directions are perpendicular to the surface normal. The two principal directions and the corresponding curvatures can be obtained by computing the eigenvalues of the second fundamental form and their eigenvectors [86; 114].

In the following, we assume that the surface is either defined by its normal, i.e., only the normal vector is known for the time being, or the surface is already defined by its tangential vectors. In both cases, first, an orthogonal frame $(\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3)$ is constructed (see Figure 2.1). One basis vector is defined $\boldsymbol{e}_3 = \boldsymbol{n}$, where $\boldsymbol{n}$ is the normal vector of the surface. If only the normal is known so far, the basis vector $\boldsymbol{e}_1$ is chosen within the plane that is perpendicular to $\boldsymbol{e}_3$; the direction within that plane can be chosen arbitrarily. The remaining basis vector is computed as $\boldsymbol{e}_2 = \boldsymbol{e}_1 \times \boldsymbol{e}_3$, and also lies in the plane that is perpendicular to $\boldsymbol{e}_3$. These steps can be skipped in the case the

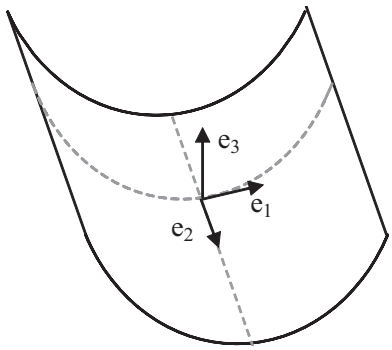


Figure 2.1: Orthogonal frame for the computation of the surface curvature. Here, $\boldsymbol{e}_1$ stands for the first principal direction, $\boldsymbol{e}_2$ for the second principal direction, and $\boldsymbol{e}_3$ for the surface normal.

tangential is already known. Then, the second fundamental form can be formulated as the matrix

$$A = \begin{bmatrix} \widetilde{\omega}_1^{13} & \widetilde{\omega}_1^{23} \\ \widetilde{\omega}_2^{13} & \widetilde{\omega}_2^{23} \end{bmatrix}, \tag{2.8}$$

where $\widetilde{\omega}_j^{i3}$ represents the deflection in the direction of $\boldsymbol{e}_i$ when you move along $\boldsymbol{e}_j$. These values are formally known as *twists* if $i \neq j$ and can be obtained by the dot product of $\boldsymbol{e}_i$ and the derivative of the gradient in $\boldsymbol{e}_j$ direction:

$$\widetilde{\omega}_j^{i3} = \boldsymbol{e}_i \cdot \frac{\partial \boldsymbol{e}_3}{\partial \boldsymbol{e}_j}$$

Note that the twist terms $\widetilde{\omega}_2^{13}$ and $\widetilde{\omega}_1^{23}$ need to be equal. In order to compute the eigenvalues of A, the matrix is first rotated about $\boldsymbol{e}_3$ until the twist terms disappear. This is done by diagonalizing A to obtain D_A:

$$A = SD_AS^{-1} = \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix} \begin{bmatrix} \kappa_1 & 0 \\ 0 & \kappa_2 \end{bmatrix} \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix}^{-1} \tag{2.9}$$

where $(u_i, v_i)^T$ denote the eigenvectors and κ_i with $i \in [1,2]$ stands for the first and the second eigenvalues of A, respectively. For a more detailed description on the eigenvalue problem see Section 2.3. Finally, the principal directions are given by $\boldsymbol{x}_i = u_i\boldsymbol{e}_1 + v_i\boldsymbol{e}_2$ with $|\boldsymbol{e}_1| = |\boldsymbol{e}_2| = 1$.

2.3 Computing Eigenvalues and Eigenvectors

This section describes the computation of the eigenvalues and eigenvectors of a 3×3 matrix. An eigenvector $\boldsymbol{x}$ of a matrix A is defined by $A \cdot \boldsymbol{x} = \lambda\boldsymbol{x}$, where λ is the corresponding eigenvalue. Therefore, the characteristic polynomial needs to be solved

$$\chi = \det|A - \lambda\mathbf{1}| = 0\,. \tag{2.10}$$

Considering a 3×3 matrix, the discriminant is a cubic equation in form of $x^3 + ax^2 + bx + c = 0$, where the coefficients are directly obtained by χ. One way to solve this

equation is to apply Cardano's formula. Here, the cubic equation is substituted by $x = t - a/3$ to get the equation $t^3 + pt + q = 0$ with

$$p = b - \frac{a^2}{3} \quad \text{and} \quad q = \frac{2a^3 - 9ab}{27} \; .$$

The discriminant $D = (q/2)^2 + (p/3)^3$ decides whether the cubic equation has real or complex solutions. All solutions are given by $t = u + v$ with

$$u = \sqrt[3]{-\frac{q}{2} + \sqrt{D}} \quad \text{and} \quad v = \sqrt[3]{-\frac{q}{2} - \sqrt{D}} \; .$$

The solutions are shown for t; the solution for x can be found by re-substitution.

$D > 0$:

$$t_1 = u + v \quad \text{and} \quad t_{2,3} = -\frac{u+v}{2} \pm \frac{u-v}{2} i\sqrt{3} \tag{2.11}$$

$D = 0$:

$$t_1 = 2u = \frac{3q}{p} \quad \text{and} \quad t_{2,3} = -u = -\frac{3q}{2p} \tag{2.12}$$

$D < 0$:

$$t_1 = -\sqrt{\frac{4}{3}p} \cdot \cos\left(\frac{1}{3}\arccos\left(-\frac{q}{2} \cdot \sqrt{-\frac{27}{p^3}}\right)\right) \tag{2.13}$$

$$t_2 = -\sqrt{\frac{4}{3}p} \cdot \cos\left(\frac{1}{3}\arccos\left(-\frac{q}{2} \cdot \sqrt{-\frac{27}{p^3}}\right) + \frac{\pi}{3}\right) \tag{2.14}$$

$$t_3 = -\sqrt{\frac{4}{3}p} \cdot \cos\left(\frac{1}{3}\arccos\left(-\frac{q}{2} \cdot \sqrt{-\frac{27}{p^3}}\right) - \frac{\pi}{3}\right) \tag{2.15}$$

Finally, the corresponding eigenvectors are found by substituting λ of Eq. (2.10) by the eigenvalues. Then only the resulting linear equations need to be solved.

2.4 Numerical Integration of ODEs

Particle tracing is the focal point of this thesis and, therefore, the ordinary differential equation (ODE) for Lagrangian particle tracing is solved more than one time in the following chapters, the two most common numerical integration schemes are briefly discussed in this section.

For a given ODE

$$\frac{d\mathbf{x}(t)}{dt} = \boldsymbol{v}(\mathbf{x}(t)) \; ,$$

the first, and computationally least expensive method consists of the Euler method

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\boldsymbol{v}(\mathbf{x}(t)) \; , \tag{2.16}$$

where the solution of $\mathbf{x}(t)$ is advanced to $\mathbf{x}(t+h)$. Although this method is very fast, it is much more inaccurate compared to the next method, the fourth-order Runge-Kutta formula

$$\begin{aligned}
\mathbf{k}_1 &= h\boldsymbol{v}(\mathbf{x}(t)) \,, \\
\mathbf{k}_2 &= h\boldsymbol{v}\left(\mathbf{x}(t) + \frac{k_1}{2}\right) \,, \\
\mathbf{k}_3 &= h\boldsymbol{v}\left(\mathbf{x}(t) + \frac{k_2}{2}\right) \,, \\
\mathbf{k}_4 &= h\boldsymbol{v}\left(\mathbf{x}(t) + k_3\right) \,, \\
\mathbf{x}(t+h) &= \mathbf{x}(t) + \frac{\mathbf{k}_1}{6} + \frac{\mathbf{k}_2}{3} + \frac{\mathbf{k}_3}{3} + \frac{\mathbf{k}_4}{6} + \mathcal{O}(h^5) \,.
\end{aligned} \tag{2.17}$$

In this thesis, all integrations on the GPU, like the one used for texture advection, are performed by the Euler method of Eq. 2.16. This is only due to performance purposes, since the Euler method requires much fewer instructions than an higher order integration scheme. For the CPU implementations, that are mostly used for precomputation, the more accurate Runge-Kutta method of Eq. 2.17 is applied.

2.5 Interpolation

Since most of the data used in this thesis is given in a discrete form, interpolation is necessary to obtain continuous functions, that are in turn required for a lot of computations. Despite the fact, that higher order interpolation schemes achieve more accurate results, in this thesis only linear interpolation methods are used. This is because linear interpolation is directly supported by the graphics hardware. Nevertheless, if desired, the interpolation can also be performed by a more accurate method. However, the interactivity would suffer in most cases.

Considering the 1D linear interpolation between two points x_a and x_b, the function value at x is described by linear interpolation

$$f(x) = \frac{x_b - x}{x_b - x_a} f(x_a) + \frac{x - x_a}{x_b - x_a} f(x_b) \,, \tag{2.18}$$

i.e., the longer the distance of the sample point x_b to the desired position x, the more x_a influences the function value at x. Keep in mind that Eq. 2.5 assumes the function to behave linear between both points. That leads to high inaccuracies, particularly if the distance between the sample points is too large. However, it delivers sufficient results if the underlying data set are given at a high sampling rate.

In 3D, trilinear interpolation must be performed instead of Eq. 2.5. The idea is quite similar: again the distance of the other sample points decides about the influence of a sample point. In 3D only the distances on 1D lines are replaces by 3D volumes spanned by the opposite sample points. Again the function value at a point $\mathbf{x} = (x, y, z)^T$ should be computed, where $\Delta x \times \Delta y \times \Delta z$ is the size of a single grid

cell. Then the function is trilinear interpolated by

$$\begin{aligned} f(\mathbf{x}) = (1-\alpha)(1-\beta)(1-\gamma) \quad & f(x,y,z) & + \\ (1-\alpha)(1-\beta)\gamma \quad & f(x,y,z+1) & + \\ (1-\alpha)\beta(1-\gamma) \quad & f(x,y+1,z) & + \\ (1-\alpha)\beta\gamma \quad & f(x,y+1,z+1) & + \\ \alpha(1-\beta)(1-\gamma) \quad & f(x+1,y,z) & + \\ \alpha(1-\beta)\gamma \quad & f(x+1,y,z+1) & + \\ \alpha\beta(1-\gamma) \quad & f(x+1,y+1,z) & + \\ \alpha\beta\gamma \quad & f(x+1,y+1,z+1)\ , & \end{aligned}$$

where $\alpha = \mathrm{frac}(x/\Delta x)$, $\beta = \mathrm{frac}(y/\Delta y)$, and $\gamma = \mathrm{frac}(z/\Delta z)$, and $\mathrm{frac}(a)$ denotes the fractional part of a.

Considering unstructured data like triangles or tetrahedra, barycentric interpolation is used instead. The basic idea is the same: subareas or subvolumes are used again for determining the weighting factors of each grid point. Given is a triangle with its vertices $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$. The function value at a point $\mathbf{x}$ within this triangle can be determined by barycentric interpolation

$$f(\mathbf{x}) = \frac{A(\mathbf{x},\mathbf{x}_2,\mathbf{x}_3)}{A(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3)} f(\mathbf{x}_1) + \frac{A(\mathbf{x},\mathbf{x}_1,\mathbf{x}_3)}{A(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3)} f(\mathbf{x}_2) + \frac{A(\mathbf{x},\mathbf{x}_1,\mathbf{x}_2)}{A(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3)} f(\mathbf{x}_3)\ , \qquad (2.19)$$

with $A(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3)$ denoting the area spanned by $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$. Furthermore, the coefficients must sum up to 1. Barycentric interpolation on triangle meshes is used in Section 6.3, where it is performed by the GPU.

CHAPTER

3 OVERVIEW OF FLOW VISUALIZATION

This chapter starts with the acquisition of flow data, which can result either from flow measurements or flow simulations. Since both measured and simulated data are used in this work, the particular methods are described in more detail. This is followed by the introduction of the Navier-Stokes equations, which describe the conservation of mass, momentum and energy in a flow field. Both the compressible and the incompressible cases are described in Section 3.2. The chapter concludes with a short discussion of experimental flow visualization.

3.1 Data Acquisition

According to the visualization pipeline in Section 1.1, the visualization process starts with the acquisition of data. In terms of flow visualization, the acquired data usually consists of multifield data representing quantities like pressure, temperature, and the velocity field. The data is usually the result of either physical flow measurements or flow simulations. In contrast to simulations, flow measurements mostly have a lack of resolution, i.e., the actual resolution of a data set is very limited, even if it was measured by recent hardware [147]. Furthermore, the measured data usually suffers from noise. On the other hand, flow simulations indeed provide an arbitrary resolution, but usually the results are achieved by numerical methods and, therefore, are not fail-safe.

3.1.1 Flow Measurements

The most common technique for measuring flow is particle image velocimetry (PIV). This technique follows the principle of photography, just in a more complex manner. The basic set-up consists of a camera and a laser. The laser acts as flashlight, whereby the light is scattered by particles moving along the flow. Nevertheless, the technique is to a high degree non-intrusive, i.e., the particles do (almost) not affect the fluid flow. The idea of PIV is to record at least two images at times t and $\mathrm{t} + \Delta\mathrm{t}$. Both images are divided into tiles which are considered as areas of uniform flow and, therefore, will contain only one velocity vector. Each interrogation area holds its own discretized function that is represented by pixel values. The next step is the computation of the displacement vector, i.e., the velocity vector, by cross-correlation of the two functions of each PIV image pair. The idea of cross-correlation is to compute the shift which is necessary to make the discrete function of the first PIV image identical to the discrete function of the second image. Extending the set-up with a second camera, it is possible

to obtain 3D vector fields by StereoPIV. However, as already mentioned above, the resulting resolution is rather bad for visualization purposes. The most recent method, the tomographic PIV method [147], enables a resolution up to $128 \times 128 \times 32$. In Chapters 8 and 9 such a tomographic PIV data set is used.

3.1.2 Flow Simulations

Numerical simulation methods basically solve the Navier-Stokes equations. In Section 3.2 the Navier-Stokes are discussed in more detail. In this thesis, the simulated data sets are the result of one of the following methods:

Direct Numerical Simulation: This technique considers no turbulence model at all. This makes the computation rather expensive, because the whole range of spatial and temporal variations needs to be resolved. Therefore, the resolution depends on the Reynolds number $\mathrm{Re} = (\boldsymbol{\nu}\mathrm{D})/\nu$, where $\boldsymbol{\nu}$ is the velocity, D the characteristic diameter, and ν viscosity. Although this method is the most accurate, it is the most computational expensive method. Due to its high memory requirements it is only barely used in practice, except for fundamental research. Figure 9.1 shows an example of a DNS data set.

Reynolds-averaged Navier-Stokes Equations: Since it is not possible to solve the Navier-Stokes equations for high Reynolds numbers, this model replaces the velocity and pressure by their separated components consisting of mean value and fluctuation. The resulting equations are called Reynolds-averaged Navier-Stokes equations, which are solved by using a turbulence model.

Large Eddy Simulation: This model is faster than DNS and more precise than RANS. In LES only the large scale motions of the flow are calculated, the effect on the sub-grid scales is modeled by using a sub-grid scale model (SGS). This is like solving the filtered Navier-Stokes equations with an additional SGS term, which is usually defined by the Smagorinsky model [163]. The unresolved turbulence scales are compensated by the addition of an "eddy viscosity" into the governing equations. An example of an LES-simulated data set can be found in Figure 4.3.

3.2 Navier-Stokes Equations

The Navier-Stokes equations are used for describing the motion of viscous fluid substances. All the equations in Chapter 7 are based on the derivatives of the incompressible Navier-Stokes equations. That makes this (simpler) case of particular interest.

The Navier-Stokes equations base on the conservation of mass, momentum and energy. The conservation law relates the rate of change of the amount of the considered property to externally determined effects. In the following, the *extensive properties* (like mass, momentum and energy) of a quantity of matter, called *control mass*, are considered inside a specific spatial area, the *control volume*.

Mass is neither created or destroyed and leads to the equation of mass conservation

$$\frac{\mathrm{dm}}{\mathrm{dt}} = 0 \,, \tag{3.1}$$

where m denotes the control mass. In contrast to the mass, the momentum can be influenced, namely by forces $\mathbf{f}$

$$\frac{d(m\boldsymbol{v})}{dt} = \sum \mathbf{f} \, . \tag{3.2}$$

The momentum is denoted by $m\boldsymbol{v}$, where $\boldsymbol{v}$ denotes the velocity.

Since it is difficult to track a specific amount of fluid in a fluid flow, it is common to investigate the flow within a given volume instead: the control volume. Therefore, the equations above are now reformulated taking into account a control volume instead a control mass. For this purpose, we are now introducing ϕ as a conserved *intensive property* that represents the conserved property per unit mass. For mass conservation $\phi = 1$, for momentum conservation $\phi = \boldsymbol{v}$. Intensive properties are, in contrast to extensive properties, independent of the amount of matter considered. The respective extensive property ϕ' can be expressed by

$$\phi' = \int_{V_{cm}} \rho\phi \, dV \, ,$$

where V_{cm} is the volume occupied by the control mass. The respective derivative is expressed by the control volume equation

$$\frac{\partial}{\partial t}\int_{V_{cm}} \rho\phi \, dV = \frac{\partial}{\partial t}\int_{V_{cv}} \rho\phi \, \partial V + \int_{S_{cv}} \rho\phi(\boldsymbol{v} - \boldsymbol{v}_b) \cdot \mathbf{n} \, dS \, . \tag{3.3}$$

where V_{cv} denotes the control volume, S_{cv} the surface of the control volume, ρ the fluid density, and $\mathbf{n}$ the normalized surface normal of S_{cv}. The velocity of the surface motion is given by $\boldsymbol{v}_b$, which is zero for a static control volume. In detail, Eq. (3.3) tells us that the rate of change of an amount of an extensive property within a control mass is similar to the rate of change of the amount of the same property inside a control volume plus the flow through the surface of the control volume. This part is also referred to as advected flow.

Inserting $\phi = 1$ in Eq. 3.3, the integral form of the conservation of mass can be obtained by

$$\frac{\partial}{\partial t}\int_V \rho dV + \int_S \rho\boldsymbol{v} \cdot \mathbf{n} \, dS = 0 \, ,$$

which can be transformed in a coordinate-free form by applying the Gauss theorem to the convective term and allowing the volume to become infinitesimally small

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\boldsymbol{v}) = 0 \, . \tag{3.4}$$

The same is done for the momentum. Inserting $\phi = \boldsymbol{v}$ in Eq. 3.3, the conservation of the momentum is achieved by

$$\frac{\partial}{\partial t}\int_V \rho\boldsymbol{v} \, dV + \int_S \rho\boldsymbol{v}\boldsymbol{v} \cdot \mathbf{n} \, dS = \sum \mathbf{f} \, . \tag{3.5}$$

The term on the right hand side needs to be reformulated to its intensive form. For this purpose, the forces need to be considered in more detail. Two kinds of forces act on

the control volume: the surface forces (pressure, normal, and shear stresses) and the body forces (gravity or centrifugal forces).

The easiest way to find a formulation for the surface forces is to consider them as Newtonian, i.e., stress versus rate of strain must be linear. For Newtonian fluids the stress tensor T is defined by

$$\mathsf{T} = -\left(p + \frac{2}{3}\nu\nabla\cdot\boldsymbol{v}\right)\mathrm{I} + 2\nu\mathsf{D}\ . \tag{3.6}$$

D defines the strain rate tensor that can be computed directly of the velocity gradient tensor by $\mathsf{D} = 1/2(\nabla\boldsymbol{v} + (\nabla\boldsymbol{v})^{\mathsf{T}})$. In the following chapters, D is replaced by S which is already occupied in this discussion. For a more detailed discussion on decomposing the velocity gradient tensor, it is referred to Section 7.1 in this work. Introducing the stress tensor T to Eq. 3.5 and denoting the body forces by $\mathbf{b}$, the integral form of the conservation of momentum is obtained by

$$\frac{\partial}{\partial t}\int_V \rho\boldsymbol{v}\,\mathrm{d}V + \int_S \rho\boldsymbol{v}\boldsymbol{v}\cdot\mathbf{n}\,\mathrm{d}S = \int_S \mathsf{T}\cdot\mathbf{n}\,\mathrm{d}S + \int_V \rho\mathbf{b}\,\mathrm{d}V\ ,$$

which can, again, be transformed to a coordinate-free form by applying the Gauss theorem to the convective and diffusive parts

$$\frac{\partial(\rho\boldsymbol{v})}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}\boldsymbol{v}) = \nabla\cdot\mathsf{T} + \rho\mathbf{b}\ . \tag{3.7}$$

Finally, in Eq. 3.4 and Eq. 3.7, we have obtained the Navier-Stokes equations for compressible flows.

However, as already mentioned above, most vector fields are solutions of the incompressible Navier-Stokes equations, which are indeed simpler because density is assumed to be constant. The general definition says that a flow can be regarded as incompressible if its Mach number, i.e., the ratio between the velocity and the speed of sound, is smaller than 0.3. In the incompressible case, the mass conservation as well as the conservation of momentum are simplified to

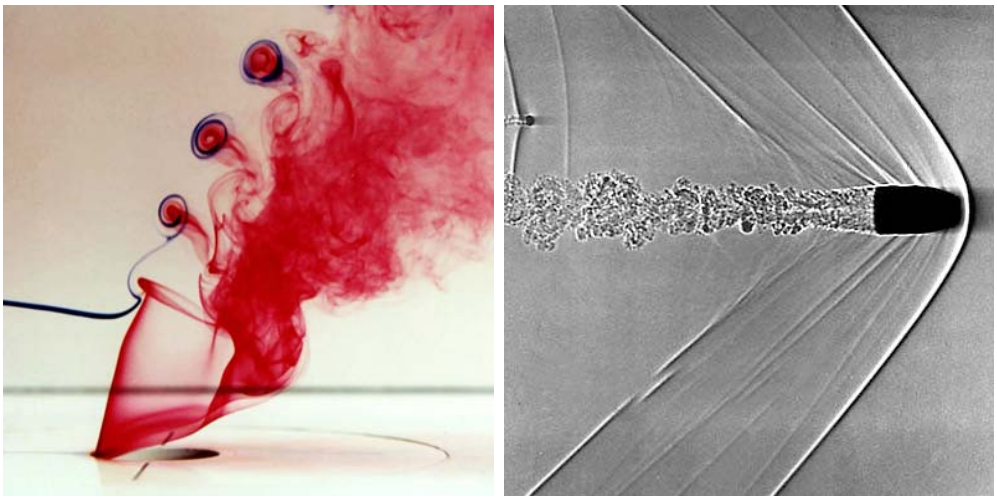
$$\nabla\cdot\boldsymbol{v} = 0\ , \tag{3.8}$$

$$\frac{\partial\boldsymbol{v}}{\partial t} + (\nabla\cdot\boldsymbol{v})\boldsymbol{v} = \nu\nabla^2\boldsymbol{v} - \frac{1}{\rho}\nabla p + \mathbf{b}\ . \tag{3.9}$$

The equations above are also the base for extracting flow features, like vortices or shear layers, as described in Chapter 7 and Chapter 8. For a more detailed derivation of the Navier-Stokes equations it is referred to [184] and [45].

3.3 Experimental Flow Visualization

This section introduces some intrusive and non-intrusive techniques for physical flow visualization which are adopted for real-time techniques like LIC, texture advection or extremum lines in Parts II and III in this thesis. Intrusive techniques are techniques where the fluid flow might be disturbed. This can either be the result of adding material or energy to the flow. In the following, some of those physically-based visualization methods are discussed with respect for their application in real-time (computer-aided) flow visualization. All the forthcoming methods are standard methods in the computer-aided flow visualization.



(a) (b)

Figure 3.1: Experimental flow visualization: (a) jet in a cross-flow [79], visualized with dye in water (courtesy of T.T.Lim, National University of Singapore), (b) shadowgraph photography showing a supersonic bullet flying between a small, short duration (courtesy of Andrew Davidhazy, Rochester Institute of Technology).

Streak Lines: Streak lines are the result of inserting continuously dye or smoke into the flow. The insertion takes place at a fixed position. Figure 3.1 (a) shows an example where dye is inserted into water. Mathematically, a streak line can be expressed by $(d\mathbf{x}/dt) = \mathbf{v}(\mathbf{x}, t)$ with the additional condition that each particle has passed the point $\mathbf{x}_0$. This can be done by requiring $\mathbf{x}(t = \tau) = \mathbf{x}_0$ with $0 \leqslant \tau \leqslant t_c$. The parameter t_c defines a time of interest. As showed in Section 5.1, texture advection can be used to draw streak lines.

Path Lines: A path line can be generated by inserting reflecting particles and exposing a photographic plate for several seconds. The result is a line which has been started at a specific position and follows the flow to the respective time. Formally, again a particle's path is described by the ODE above with a slightly modified additional condition $\mathbf{x}(t_0) = \mathbf{x}_0$, which says that the particle is started at a specific position $\mathbf{x}_0$.

If steady flow is considered, the above ODE can be reformulated to $d\mathbf{x}/ds = \mathbf{v}(\mathbf{x})$ with the curve parametrization s. This shows that the stream line is parallel to the velocity vector at each position on the curve. The extention of stream lines and path lines to surfaces is discussed in Section 6.1.

Time Lines: Time lines can be obtained by installing electrodes in the fluid. In water, for example, small bubbles are released by the electrodes which are traveling with the flow. Time lines result by connecting n particles at different position $\mathbf{x}_i(t_0)$ to a seed line. Then a time line at time t_j is defined by the set of particles with $\mathbf{x}_i(t_j)$.

Examples of non-intrusive techniques are "shadowgraphs" and "Schlieren photography". Both methods are quite similar, whereby shadowgraphs are the simpler

method. In contrast to intrusive methods no material which might influence the flow is added. Shadowgraphs are the result of considering the difference of light refraction, i.e., if the light penetrates regions of different density, it results in a different refraction and, therefore, this also influences the shadows which are generated due to the light refraction. If a photographic plate is placed behind a scene, areas of different refraction can be recorded by sending parallel light through the scene to the plate. Figure 3.1 (b) shows an example of a shadowgraph. However, this method is usually weak of contrast. The Schlieren photography tries to solve this issue by extending the experimental set-up with a lens and a knife-edge. The idea is to focus the light after it is refracted and to place a knife-edge at the focal point. The light which is refracted towards the knife is blocked, the remaining light is recorded by the camera. This results in a higher contrast compared to the shadowgraphs method.

Part II

Particle Tracing for Direct Vector Field Visualization

CHAPTER

4 BASICS OF TEXTURE-BASED FLOW VISUALIZATION

This chapter introduces the basic algorithms for texture based vector field visualization. Early texture based techniques are spot noise [174] and line integral convolution (LIC) [19]. Section 4.1 discusses LIC in more detail since it builds the base of most recent texture based approaches. One of the most related approaches makes use of texture advection [110], which can be extended to 2D Lagrangian-Eulerian Advection (LEA) [71] or 2D Image Based Flow Visualization (IBFV) [177]. Most of the work discussed in this part of the thesis depend on LEA and IBFV and, therefore, in Section 4.2 both methods are described in more detail. One reason for recent advances in texture-based flow visualization is the increasing performance and functionality of GPUs, which can be used to improve the speed of 2D flow visualization [69; 193; 177; 104].

Texture-based methods are also applied for the visualization of 3D flow [169; 193; 191; 195; 196; 42]. A comprehensive overview of recent texture based techniques is given in 2D and 3D is given in the survey articles [91; 90]. The implementation of a fast 3D texture advection according to [195; 196; 90] is discussed in Section 4.2. In the context of 3D LIC, dye visualization can be used to highlight features [156; 186]. A more accurate dye advection model using level-set methods can be found in [189]. Botchen et al. used noise advection and dye advection in order to represent flow uncertainties [13; 14].

Especially 3D flow visualization is subject to perceptual issues, which can be addressed by a combination of interactive clipping and user intervention [134]. Alternatively, 3D LIC volumes can be effectively represented by selectively emphasizing important regions of the flow, enhancing depth perception, and improving orientation perception [65]. Perception of 3D flow can also be enhanced by shading according to limb darkening via transfer functions [58], by interactively changing the rendering style [158], or by volume rendering of implicit flow volumes [204]. In Section 5.1, perception of 3D semi-Lagrangian texture advection is improved by applying several illumination models [195; 196]. Recently, Falk and Weiskopf proposed a new illumination model for emphasizing 3D LIC structures [42].

Another extension of 2D texture-based visualization consists of the display of vector fields on curved surfaces [92; 178; 192; 4]. In Section 6.1, it is shown how LIC for curved surfaces [192] can also be used to emphasize path lines on path surfaces [150], while in Section 6.2 and 6.3 the visualization of brain curvature [149] and the flow around vortices [153] is discussed.

4.1 Line Integral Convolution

LIC—line integral convolution is the name of the most popular texture-based technique for visualizing arbitrary vector fields and was proposed by Cabral and Leedom [19] in 1993. Actually, this method computes particle traces, namely path lines, but in contrast to conventional particle tracing methods a noise texture is convolved in a way the stream lines are perceptible without using any additional geometry.

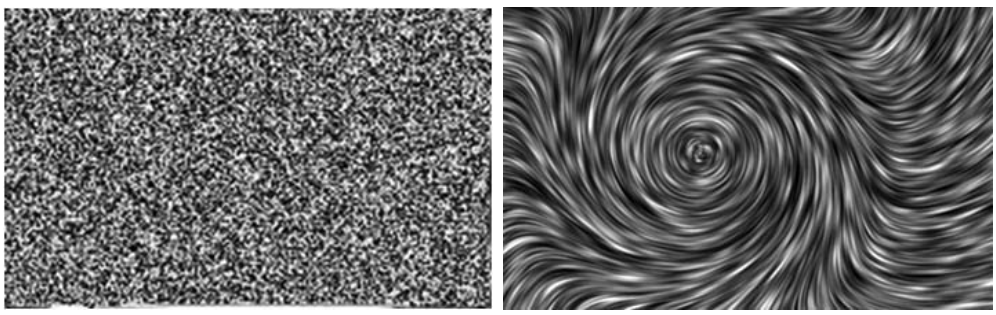
For the sake of simplicity, in the following a 2D planar domain is considered. The idea is to work with different frequencies in order to extract path lines of a vector field. For this purpose the following ingredients are required:

- An input vector field $\boldsymbol{v}$
- An input noise texture covering the entire domain $T(x, y)$
- An appropriate convolution kernel $k(\tau - \tau_0)$
- The evaluation of the Lagrangian particle tracing ODE resulting in a path line $\phi(\tau - \tau_0)$

The parameter τ denotes the time and x, y the 2D coordinates of the input texture. Usually, the input texture consists of a stationary white noise, i.e., a texture holding randomly generated grey values. Let $\phi_0(t)$ denote a path line that contains a specific position (x_0, y_0) at time τ_0. Then the pixel's intensity at this position is computed according to

$$I(x_0, y_0) = \int_{-L}^{L} k(\tau - \tau_0) \cdot T(\phi_0(\tau - \tau_0)) d\tau, \tag{4.1}$$

where $[-L, L]$ defines the support of the convolution kernel. In the most cases $k(t)$ is a symmetric filter, e.g., a box or tent function. Evaluating Eq. (4.1) results in low frequencies along the respective path lines, but maintains high noise frequencies perpendicular to those lines, which leads to line-like visual pattern easily perceived by the viewer. Figure 4.1 shows an example: on the left hand side the noise texture $T(x, y)$ is shown; on the right hand side the resulting LIC image. LIC builds the base of various other texture-based techniques, most of them are improving the indeed bad performance the LIC integral evaluation.



(a) (b)

Figure 4.1: Line Integral Convolution.

4.2 Semi-Lagrangian Texture Advection

In 2002, Jobard et al. [70] proposed a texture-based method for visualizing vector fields, called Lagrangian-Eulerian advection. Actually, this method bases on the texture advection approach by Max and Becker [110], that moves textures along a given vector field. The Lagrangian-Eulerian advection benefits, in contrast to LIC, from the exploitation of graphics hardware. This section gives an overview over the mechanism of semi-Lagrangian texture advection according to the descriptions in [195; 196].

4.2.1 Basic Algorithm

As the name of the method already implies, the advection is considered from both a Lagrangian and an Eulerian point of view. From an Eulerian point of view, particles lose their individuality and are represented by their property values (such as color or gray-scale values), which are stored in a property field $\rho(\mathbf{x}, t)$. The parameter $\mathbf{x}$ denotes position and t denotes time. This property field is typically given on a uniform grid. The evolution of the property field is governed by the convection equation which states, similar to the Eq. (3.4), the conservation of the property values

$$\frac{\partial \rho(\mathbf{x}, t)}{\partial t} + \boldsymbol{\nu}(\mathbf{x}, t) \cdot \nabla \rho(\mathbf{x}, t) = 0 \quad , \tag{4.2}$$

where $\boldsymbol{\nu}$ is the input vector field. Then the advection is performed along pathlines, by solving the ordinary differential equation for Lagrangian particle tracing

$$\frac{d\mathbf{x}(t)}{dt} = \boldsymbol{\nu}(\mathbf{x}(t), t) \quad . \tag{4.3}$$

Backward explicit integration is employed to compute particle positions at a previous time step, $\mathbf{x}(t - \Delta t)$. For example, first-order Euler integration yields

$$\mathbf{x}(t - \Delta t) = \mathbf{x}(t) - \Delta t \boldsymbol{\nu}(\mathbf{x}(t), t) \quad .$$

Starting from the current time step t, an integration backwards in time returns the position along the pathline at the previous time step. The property field is evaluated at this previous position to access the value that is transported to the current position, leading to a backward advection scheme,

$$\rho(\mathbf{x}(t), t) = \rho(\mathbf{x}(t - \Delta t), t - \Delta t) \quad , \tag{4.4}$$

in the general case. This advection is suitable for steady as well as for unsteady flow because the time dependency t of the vector field is taken into account.

4.2.2 Visual Mapping

Similar to LIC, property values are represented by a noise texture; usually filtered noise is taken in order to avoid aliasing artifacts. This noise texture serves as *injection texture*, i.e., it injects new property values for advection. For visual mapping the basic

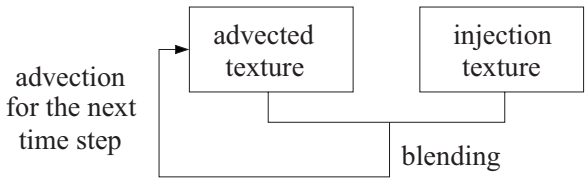


Figure 4.2: Basic structure of the injection mechanism of IBFV.

idea of 2D IBFV (Image-Based Flow Visualization) [177] can be used. IBFV introduces new property values at each time step, described by the injection texture I. The structure of the injection mechanism of 2D IBFV is illustrated in Figure 4.2.

The original scheme of 2D IBFV proposes to apply an affine combination of advected property values and the injection texture

$$\rho(\mathbf{x}, t) = (1 - \alpha)\rho(\mathbf{x}(t - \Delta t), t - \Delta t) + \alpha I(\mathbf{x}, t) \quad ,$$

where $\rho(\mathbf{x}, t)$ represents the texture holding the advected property values, called *property texture*. Weiskopf and Ertl [191] generalized the the original compositing schemes for 2D IBFV and 3D IBFV [169] to allow for a unified description of both noise and dye advection. Here, the restriction to an affine combination of the advected value and of the newly injected value is replaced by a generic linear combination of both. Furthermore, the generalization makes it possible to advect and blend several materials independently. The extended blending equation according to [191] is given by

$$\rho(\mathbf{x}, t) = W(\mathbf{x}, t) \circ \rho(\mathbf{x}(t - \Delta t), t - \Delta t) + V(\mathbf{x}, t) \circ I(\mathbf{x}, t) \quad , \tag{4.5}$$

with two, possibly space-variant and time-dependent, weights W and V that need not add up to one. The symbol "$\circ$" denotes a component-wise multiplication of two vector quantities. The different components of each texel in the property field describe the density of different materials. Continuous blending of injected "particles" leads to streakline-like visual structures. Figure 4.3 shows an example of 2D semi-Lagrangian texture advection. Three materials are used: red and green streaklines are drawn as a result of noise advection; dye is drawn in yellow. Semi-Lagrangian texture advection is well-suited for being implemented on the GPU. A more detailed description can be found in Section 5.1.

4.3 LIC on Curved Surfaces

Another modification of the original LIC algorithm (Section 4.1) consists of the hybrid method by Weiskopf and Ertl [192]. Here, the LIC integral is evaluated in both object and image space in order to achieve line-like visual patterns on curved surfaces. This algorithm can be considered as G-buffer algorithm [146] because it relies on image-space information to perform particle tracing and convolution. It is important to distinguish between the dimensionality of the domain and the dimensionality of the attached data. An image-space or G-buffer method always uses a 2D domain (the image plane), but the attached data (i.e., the G-buffer attributes) can be of other dimensionality.

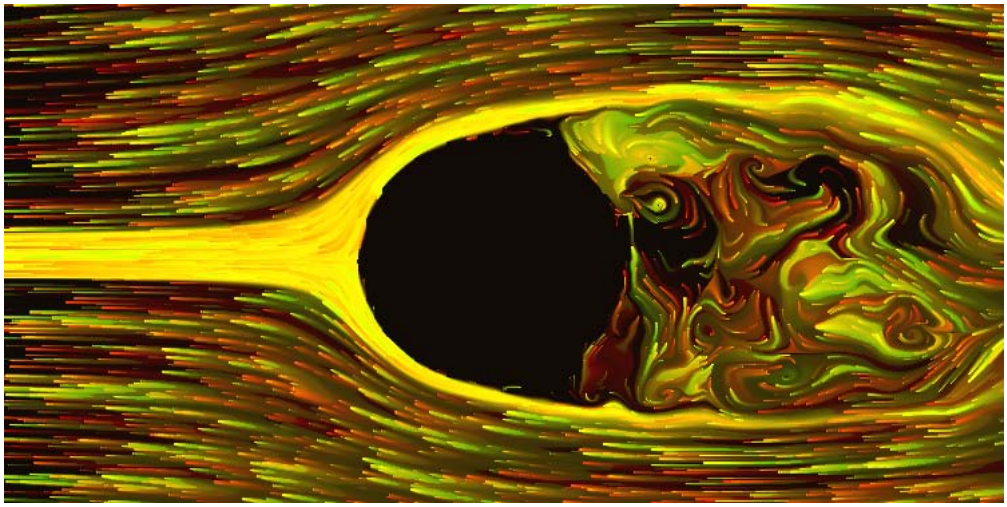


Figure 4.3: 2D semi-Lagrangian texture advection with three different materials. For noise advection a red and a green material is used with different blending values. Additionally, dye advection is performed (yellow dye). This data set shows the wake behind a cylinder at a very high Reynolds number (Re=200.000). In order to emphasize the vortex structures, a normalized vector field is used.

4.3.1 Basic Idea

The hybrid object/image space method needs the following G-buffer attributes: (1) the 3D position of the respective pixel in object space and (2) the 3D vector field in object space. The only other data that is used for LIC is an input noise. This noise is modeled as a 3D solid texture to ensure temporal coherence under camera motion. According to Weiskopf et al. [192], the LIC texture I of Eq. (4.1) is extended by taking into account contributions of object and image space separately:

$$I(x_I^0, y_I^0) = \int k(\tau - \tau_0) T(\mathbf{r}_O(\tau - \tau_0; x_I^0, y_I^0))\, d\tau \,, \tag{4.6}$$

where the subscript $_I$ denotes parameters given in image space, the subscript $_O$ denotes parameters given in object space and $\mathbf{r}_O$ represents positions along a pathline. The pathline is determined by the initial image-space position (x_I^0, y_I^0), which has a corresponding initial 3D object-space position on the surface at initial time τ_0.

The basic idea of surface LIC is to split the computation in two stages: (1) the projection of the curved surface and its corresponding vector field onto the image plane—the *projection stage*; (2) the evaluation of Eq. (4.6) on the image plane—the *LIC stage*. This approach has the following important advantages: it does not require a parametrization of the surface; it evaluates the LIC integral only for visible surface elements; the LIC computation can be performed on a planar 2D domain; and a mapping to an efficient GPU algorithm is possible.

4.3.2 Algorithm

For the following discussion two types of space are introduced: the normalized device space (D-space), serving as representation of the image space; the P-space as the re-

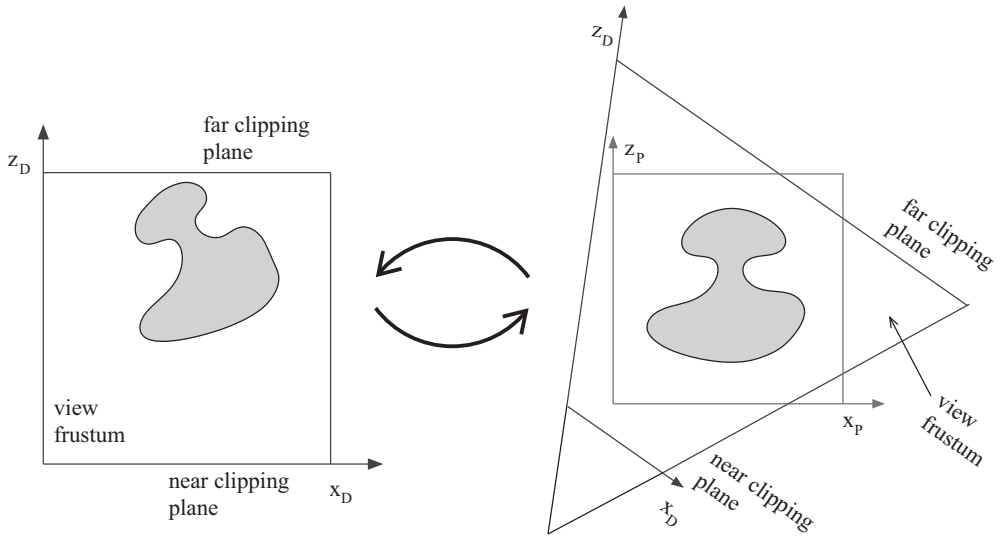


Figure 4.4: Coupled D-space (left) and P-space (right) representations of the same scene. © 2004 Weiskopf and Ertl [192].

spective representation in the physical space, i.e., in which the geometry is defined. D-space is defined on $[0,1]^3$, with two coordinates describing the image plane and the third coordinate representing depth with respect to the image plane. For example, when a 3D object is visualized on the screen, the device space can be seen as the surface of the screen with an additional depth information per pixel. An object can be transformed from P-space to D-space by applying a projection onto the image plane.

Let us consider a particle path in P-space, again determined by the ordinary differential equation for Lagrangian particle tracing

$$\frac{d\mathbf{r}(t)}{dt} = \boldsymbol{v}(\mathbf{r}(t), t) \; , \tag{4.7}$$

where $\boldsymbol{v}(\mathbf{r}, t)$ denotes the vector field and $\mathbf{r}(t)$ denotes the position of the particle at time t. The idea of the algorithm is to apply the LIC computation Eq. (4.1) on a per-pixel basis with respect to the image plane (D-space). To exploit the advantages of combined P-space and D-space representations, the particle pathes are simultaneously computed in both domains as shown in Figure 4.4. On the left side, the object is projected in D-space. If we consider D-space as normalized cube, the object is seen from above. The boundaries along the depth z_D are defined by the near and the far clipping plane. In P-space, z_D represents the distance from the viewer to the object.

To solve Eq. (4.7), an explicit numerical integration, such as a first-order explicit Euler scheme, works with P-space coordinates $\mathbf{r}_P \equiv \mathbf{r}$ and the original tangential vectors $\boldsymbol{v}_P \equiv \boldsymbol{v}$. After each integration step, the corresponding position in D-space is computed. An important point is that the vector field is not given on a P-space but on a D-space domain. This implies that we have different representations for the vector

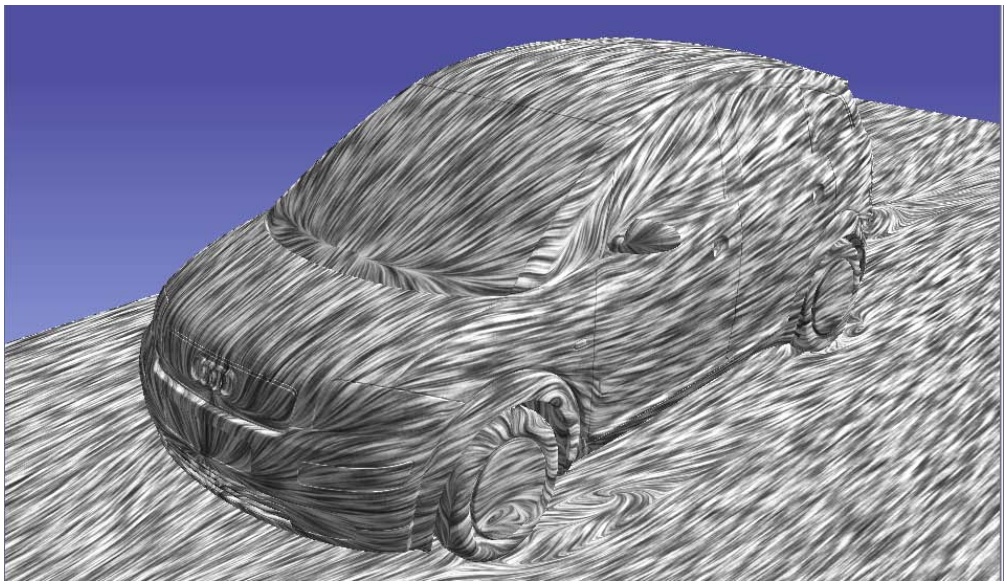


Figure 4.5: Visualization of a flow simulation around an Audi A2 using the hybrid LIC approach by Weiskopf and Ertl [192].

field and the associated point on the surface. The modified particle-tracing equation then is:

$$\frac{d\mathbf{r}_P(t)}{dt} = \mathbf{v}_P(\mathbf{r}_D(t), t) \,, \tag{4.8}$$

where $\mathbf{r}_D$ and $\mathbf{r}_P$ represent the same position with respect to D-space and P-space, respectively.

This modified integration process is efficient because the 3D representations of the quantities are reduced to a 2D D-space representation whenever possible. Since the geometry on which the LIC should be applied is usually opaque, only the closest surface layer needs to be considered. Here, the depth component can be indirectly computed because the depth values of the surface is known.

The first stage of the surface LIC algorithm, the projection stage, is responsible for creating two image-plane aligned 2D textures and initializes them with the starting positions $\mathbf{r}_P$ and the vector field $\mathbf{v}_P$. This is done by rendering the surface geometry. In detail, the geometrical primitives are projected onto the image plane and sorted in depth order with z test enabled. The positions are written in object coordinates. The vector field is either fetched from a 3D texture containing $\mathbf{v}_P$ or directly from the texture coordinates attached to the primitives' vertices (in the case only a surface flow vector field is given). If the velocity vectors are not tangential to the surface, they are projected onto it. Keep in mind that non-tangential vectors physically would lead to a detachment of particles, that in turn results in short line structures when LIC is applied.

In the second part, the LIC stage, Eq. (4.8) is solved by iterating over integration steps. This stage is performed in the 2D x–y sub-domain of D-space, and it successively updates the coordinates $\mathbf{r}_P$ and $\mathbf{r}_D$ along the particle traces, while simultaneously accumulating contributions to the convolution integral.

Figure 4.5 shows an example of a surface LIC computed with the method by

Weiskopf and Ertl [192]. Here, the vector field is attached to the triangles of the scene's geometry. All vectors are tangential to the respective surface. In order to obtain long line-like structures the vector field has been normalized before the LIC computation was applied.

CHAPTER

5 3D TEXTURE-BASED VISUALIZATION

In this chapter, the extension of 2D semi-Lagrangian texture advection to 3D is discussed, that results in streaklines within a 3D domain when particle tracing is applied. Furthermore, though only examples of fluid vector fields are shown, 3D texture advection is appropriate for the visualization of arbitrary vector fields, like gradients or other derived vector fields. The following discussion proposes a number of newly developed techniques that concentrates on three aspects: a fast GPU-based implementation that allows the interactive use of 3D texture advection (Section 5.1.1); the consideration of perceptual issues, i.e., how illumination and NPR (non-photorealistc rendering) methods can be used to improve perception (Section 5.2); and linked multiple views combining 2D and 3D texture advection (Section 5.3).

5.1 Semi-Lagrange Advection in 3D

While 2D texture advection (Section 4.2) builds a powerful mechanism for visualizing 2D flows, it is rather insufficient to represent the more complex behavior of 3D vector fields. Therefore, it is desirable to develop a 3D extension of the established texture advection method. However, implementing a 3D dense representation is quite challenging because of the significant increasing complexity, i.e., the computation has to be performed for all cells inside a 3D grid, and it is also difficult to find a good visual representation of a dense collection of particle traces (discussed later in Section 5.2).

In the following, these issues are solved by adapting the semi-Lagrangian 2D texture advection approach described in Section 4.2 to 3D. This section contains parts of the original text from [195; 196].

5.1.1 3D Texture Advection on the GPU

The 3D GPU-based particle advection solves the reformulated Eq. (4.4) using Euler integration

$$\rho(\mathbf{x}, t) = \rho(\mathbf{x} - \Delta t \mathbf{v}(\mathbf{x}, t), t - \Delta t) \quad . \tag{5.1}$$

Keep in mind that Eq. (4.4) can also be solved by any other numerical integration scheme, e.g., higher order schemes like Runge-Kutta. However, this enormously increases the complexity because the additional operations that need to be performed on the GPU, like texture fetches and arithmetic operations. In this section, Eq. (5.1) is used for integration.

Preliminary Considerations

In the work of Weiskopf and Ertl [191] it has been shown that the implementation of Eq. (5.1) on the GPU leads to a 3D texture-based representation of the property field ρ and the vector field $\boldsymbol{v}$. According their implementation the physical position $\boldsymbol{x}$ and the corresponding texture coordinates are related by an affine transformation that takes into account that the physical and computational spaces may have different units and origins. Accordingly, the step size in computational (texture) space directly corresponds to the physical time step Δt. While the property texture is only updated at grid points, the lookup in the property field at the previous time step is performed at locations that may differ from exact grid positions. Therefore, trilinear interpolation is employed to reconstruct the value of the property field at the previous time step. Since any rendering operation is restricted to a 2D domain, the property field for a subsequent time step is constructed in a slice-by-slice manner. Each slice of the property field is updated by rendering a quadrilateral that represents this 2D subset of the full 3D domain. The dependent lookup in the "old" property texture can be realized by a fragment program that computes the modified texture coordinates according to the Euler integration along the flow field.

Introducing Logical and Physical Memory

The main problem with this implementation is the slice-by-slice update of the 3D texture for the property field. In many cases, an update of a 3D texture is only possible via transfer of data to and from main memory. Although OpenGL allows us to update a slice of a 3D texture (DirectX 9.0 does not support this), the speed of such an update can vary tremendously between GPU architectures because of different internal memory layouts of 3D textures (see a related discussion on read access for 3D textures [198]). Even though the OpenGL superbuffer extension [122] or the framebuffer object extension [155] allow us to render directly into a 3D texture, the fundamental issues of the speed of updating 3D textures and the extent of the vendor's support for those extensions remain. Therefore, in the following an alternative approach is presented that is based on 2D texture memory instead of 3D texture memory. 2D textures are available on any GPU, they provide good caching mechanisms and efficient bilinear resampling, and they support an extremely fast write access by the render-to-texture functionality.

For the new 2D texture-based implementation, we have to distinguish between *logical* memory and *physical* memory. Logical memory is identical as for 3D texture advection—it is organized in the form of a uniform 3D grid. Physical memory is a 2D uniform grid represented by a 2D texture. A related approach of virtual GPU memory management was taken by Lefohn et al. [95] for computing level sets on GPUs, which was later extended to generic GPU data structures [96].

In the following, the coordinates for addressing the logical memory are denoted by $\mathbf{x} = (x, y, z)$ and the coordinates for physical memory by $\mathbf{u} = (u, v)$. A slice of constant value z in logical memory corresponds to a tile in physical memory, as illustrated in Figure 5.1. Different tiles are positioned in physical memory with a row-first order. Since the maximum size of a 2D texture may be limited, several "large" 2D textures may be used to provide the necessary memory. These 2D textures are labeled

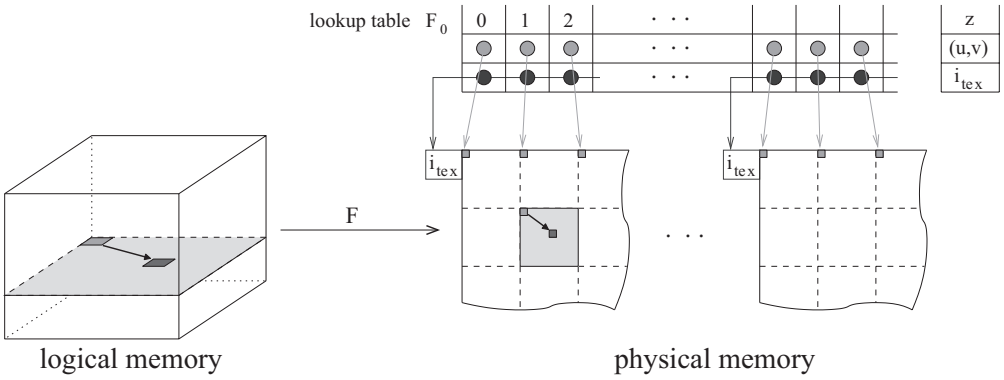


Figure 5.1: Mapping between logical 3D memory and physical 2D memory by means of a lookup table.

by the integer-valued index i_{tex}. Since all numerical operations of 3D advection are conceptually computed in logical 3D space, we need an efficient mapping from logical to physical memory, which is described by the function

$$\Phi: \quad (x, y, z) \longmapsto ((u, v);\ i_{tex}) \quad .$$

The 2D coordinates can be separated into the coordinates for the origin of a tile, $\mathbf{u}_0$, and the local coordinates within the tile, $\mathbf{u}_{local}$:

$$\Phi: \quad (x, y, z) \longmapsto (\mathbf{u}_0 + \mathbf{u}_{local};\ i_{tex}) \quad , \tag{5.2}$$

with

$$\mathbf{u}_0 = \Phi_{0,\mathbf{u}}(z), \quad i_{tex} = \Phi_{0,i_{tex}}(z), \quad \mathbf{u}_{local} = (s_x\, x, s_y\, y) \quad . \tag{5.3}$$

The function Φ_0 maps the logical z value to the origin of a tile in physical memory and is independent of the x and y coordinates. The map Φ_0 can be represented by a lookup-table (see Figure 5.1), which can be efficiently implemented by a dependent texture lookup. Conversely, the local tile coordinates are essentially identical to (x, y)—up to scalings (s_x, s_y) that take into account the different relative sizes of texels in logical and physical memory. If more than one "large" 2D texture is used, multiple texture lookups in these physical textures may be necessary. However, multiple 2D textures are only required in tiles that are close to the boundary between two physical textures because the maximum length of the difference vector for the backward lookup in Eq. (5.1) is bounded by $\Delta t v_{max}$, where v_{max} is the maximum velocity in the data set. In the case of boundary tiles another fragment program is used to reduce the number of texture samplers for the advection in interior regions.

Trilinear interpolation in logical space is implemented by two bilinear interpolations and a subsequent linear interpolation in physical space. Bilinear interpolation within a tile is directly supported by built-in 2D texture interpolation. A one-texel-wide border is added around each tile to avoid any erroneous influence by a neighboring tile during bilinear interpolation. The subsequent linear interpolation takes the bilinearly interpolated values from the two closest tiles along the z axis as input. This linear interpolation is implemented within a fragment program.

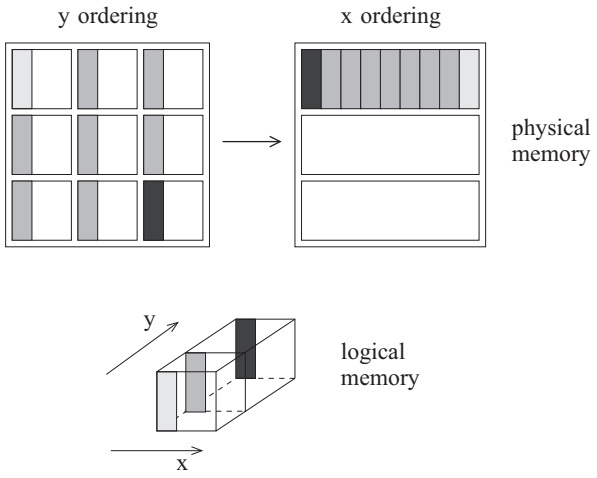


Figure 5.2: Reordering of stack direction.

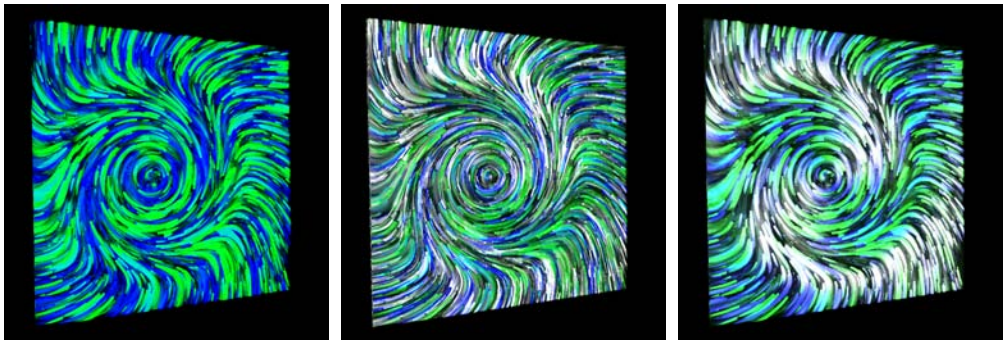
While trilinear interpolation by the above mapping scheme is necessary for the read access in Eqs. (5.1), write access is more regularly structured. First, write access does not need any interpolation because it is restricted to grid points (i.e. single texels). Second, the backward lookup for Eqs. (5.1) allows us to fill logical memory in a slice-by-slice manner and, thus, physical memory in a tile-by-tile fashion. A single tile can be filled by rendering a quadrilateral into the physical 2D texture if the viewport is restricted to the corresponding subregion of physical memory.

5.1.2 Rendering

In the following, the rendering of 3D texture advection, taking into account logical and physical memory, is described. Advanced illumination techniques are disregarded in this section; they are introduced later in Section 5.2.

The property field ρ resulting of Eq. (5.1) is visualized by volume rendering with texture slicing—similarly to 2D texture-based rendering with axis-aligned slices (see Section 1.4). Traditional 2D texture slicing holds three copies of a volume data set, one for each of the main axes. In this approach, however, only a single copy of the property field is stored on GPU. To avoid holes in the final display, the stacking direction is changed on-the-fly during advection if the viewing angle becomes larger than 45 degrees with respect to the stacking axis. Figure 5.2 illustrates a reordering of the stacking direction from $\mathtt{y}$ to $\mathtt{x}$ axis. A tile in the new stacking order is rendered in a stripe-by-stripe fashion, according to portions of tiles from the old stacking order. The reordering process takes into account that the number of tiles, their sizes, and their positions may change.

Actual volume rendering accesses tile after tile in front-to-back order by rendering object-aligned quadrilaterals. Texture coordinates are issued to address a tile in physical memory of the "large" 2D texture. Furthermore, a dependent-texture lookup in a fragment program is employed to implement post-classification. For each material in the property field, density is mapped to optical properties (color and opacity) by its corresponding transfer function. The results of different transfer functions for differ-



(a) (b) (c)

Figure 5.3: 3D advection for a tornado data set with: (a) volume rendering based on the emission-absorption model, (b) gradient-based Phong illumination, (c) line-based Phong illumination.

ent materials are added to obtain the final color and opacity. This approach retains all benefits and extensions that are available for 2D texture slicing, such as fast bilinear texture sampling (as opposed to slower trilinear reconstruction in 3D textures), additional slices with trilinear interpolation [133], or pre-integrated volume rendering [40].

For dense 3D flow representations, transfer functions are typically specified to render interesting flow regions with high opacities. Therefore, early ray termination is an effective way of accelerating volume rendering provided that a front-to-back compositing scheme is employed. Similarly to [141], the early z test can be used to skip the execution of a fragment program when a user-specified maximum opacity has been accumulated. Depending on the GPU architecture, the early z test is an efficient way to discard fragments. Terminated rays are masked in a separate rendering pass by setting the z buffer to zero (near clipping plane); the z value is set to the depth of the far clipping plane for all other pixels. Then, the depth test skips terminated rays while a slice of the volume is rendered. As the initialization of the z buffer in the separate pass consumes additional rendering time, this initialization is only performed for every n-th volume slice to achieve a compromise between perfect early ray termination and the additional costs for the separate pass. A typical value is $n = 10$. The speed-up by early ray termination is discussed in more detail in [195].

5.2 Illumination and Visual Perception

The previous rendering section has described volume rendering according to the emission-absorption model, which leads to a display similar to a self-emitting gas cloud. Although this model allows us to view different semi-transparent depth layers of a 3D flow field, it fails to explicitly visualize the orientation and relative depth of streamline or streakline structures. Figure 5.3 (a) shows a visualization according to the emission-absorption model with high opacities. The chosen camera position leads to a visualization that is essentially restricted to showing the flow on one of the faces of the cube-shaped domain boundary. The underlying data set illustrates the air flow in a tornado (view from top). Figures 5.3 (b) and (c) show that the final display has been improved by additionally applying volume illumination to the property field.

In the following three sections, two alternative methods for the volumetric illumination of the results of texture advection are investigated. The first method relies on the gradients of the property field, which serve as normal vectors for local illumination. Gradient-based lighting is a widely used approach for volume illumination in general and is directly adopted for lighting the advected properties field. The second method is based on line-oriented lighting that applies the illumination computation to streamline structures contained in the property field. This approach just needs the tangent vector of the streamlines and, thus, avoids the computation of gradients.

5.2.1 Gradient-Based Illumination

In general, volumetric illumination needs gradients as a basis for local illumination. This approach relies on the assumption that isosurface-like spatial structures of the volume are most important and, thus, should be illuminated and rendered correspondingly. Since the gradient is perpendicular to the isosurface at the respective point in the volume, the orientation of the tangent plane of the isosurface is given by the gradient. Therefore, when the gradient is available, any of the traditional local lighting models, such as the Phong or Blinn-Phong models, can be employed. The contribution from local illumination is added to the emission part in the volume rendering integral. In the discretization via texture-based volume rendering, the local illumination term is combined with the color contribution from the scalar field. The review article by Max [109], for example, provides more details on optical models for volume rendering.

Because of the generality of gradient-based illumination, it can directly be applied to the special case of the property field that results from advection. The goal is to incorporate volume illumination into the real-time advection system of Section 5.1.1 and, therefore, gradients have to be computed in real time as well. A numerical computation by central differences is employed, which delivers gradients of acceptable quality at a high speed. Gradients are stored in a grid that has the same logical and physical memory layout as the property field. Central differences exhibit a uniform access to 3D logical memory because data is fetched from neighboring grid points along the three main axes. Accordingly, a well-structured accessing scheme is also applied in 2D physical texture memory: neighboring texels in the current tile yield the x and y partial derivatives, whereas the partial derivative along the z axis is based on the two closest tiles in z direction.

The mapping from logical to physical memory according to Eq. (5.2) needs only to be computed at the four vertices that describe a single tile in physical memory. Six pairs of texture coordinates are attached to the vertices and interpolated by scanline conversion. Linear interpolation and the mapping from Eq. (5.2) are commutative: the respective tile origins $\mathbf{u}_0$ and texture indices $\mathbf{i}_{\text{tex}}$ are constant and the local coordinates $\mathbf{u}_{\text{local}}$ vary linearly because the relative distance between a central grid point and its neighbors is constant for a complete tile.

Gradients are computed after each advection and blending iteration and before volume rendering. Therefore, any gradient-based volume shading method may be applied. Figure 5.3 (b) shows an example of volume illumination by the Phong model with diffuse and specular components, which are added on top of the emissive part that is determined by the transfer function from the previous section. The same transfer function value is used for the material color for diffuse illumination; the specular

color is set to white. Other definitions of material colors could be easily incorporated, if required. Phong illumination greatly improves the perception of orientation by shading—highlights in combination with camera motion are particularly effective in revealing the orientation of particle traces.

5.2.2 Line-Based Illumination

The main problems of gradient-based illumination are increased computational costs for calculating gradients and additional memory requirements if gradients are stored in texture memory. Line-based illumination can be used to overcome both problems. The idea is to apply illumination directly to thin streamlines, whose relevant geometric information about orientation is encoded in their tangent vectors. By construction, the tangent of a streamline is identical to the vector data at the same point. Therefore, the vector field is sufficient to compute the local illumination of streamlines.

First an existing method for the lighting of curves in 3D is described which is adopted in the following for the illumination of the texture-based streamlines. According to Banks [6], local illumination can be extended to geometric objects of arbitrary dimension and codimension. We assume that an object of dimension $k > 0$ can be described as a k-manifold M embedded in Euclidean space of dimension $n > k$. The difference $n - k$ is the codimension of the object. For example, curves embedded in 3D space are 1-manifolds with codimension 2 (i.e., $n = 3$ and $k = 1$). Banks relies on Fermat's principle with light paths of minimal length to derive a model for diffuse and specular illumination.

In the following, a convention is used in which the light direction vector $\mathbf{L}$, the normal vector $\mathbf{N}$, the reflection vector $\mathbf{R}$, and the viewing vector $\mathbf{V}$ point away from the position of incidence on the illuminated object. All these vectors are assumed to be normalized to unit length.

The n-dimensional Euclidean space is decomposed into the k-dimensional tangent space $TM_{\mathbf{p}}$ and the $(n-k)$-dimensional normal space $NM_{\mathbf{p}}$—both at the point $\mathbf{p}$ of the manifold M that describes the boundary of the illuminated object. The main issue of illumination in higher codimension is that there is no unique normal vector because the normal space $NM_{\mathbf{p}}$ has a dimension $(n-k) > 1$. The basic idea is to select an appropriate vector from the normal space $NM_{\mathbf{p}}$ in order to maximize the lighting contribution according to Fermat's principle.

In this section, the case of illuminating curves embedded in 3D space is considered (i.e., $n = 3$ and $k = 1$). To compute the Phong illumination model, the reflection and normal vectors are not immediately required, but their respective dot products with the view and light vectors, which are the basis for the diffuse and specular lighting terms. According to Zöckler et al. [208], these dot products can be expressed in terms of the light vector $\mathbf{L}$, the view vector $\mathbf{V}$, and the tangent vector $\mathbf{T}$. To simplify notation, it is useful to split a general vector $\mathbf{A}$ into a tangential part $\mathbf{A}_T \in TM_{\mathbf{p}}$ and a normal part $\mathbf{A}_N \in NM_{\mathbf{p}}$, i.e., a subscript T denotes the tangential part of a vector and a subscript N denotes the normal part. Then, the diffuse illumination term is based on

$$c_d = \mathbf{L} \cdot \mathbf{N} = \|\mathbf{L}_N\| = \sqrt{1 - \|\mathbf{L}_T\|^2} = \sqrt{1 - (\mathbf{L} \cdot \mathbf{T})^2}\,, \tag{5.4}$$

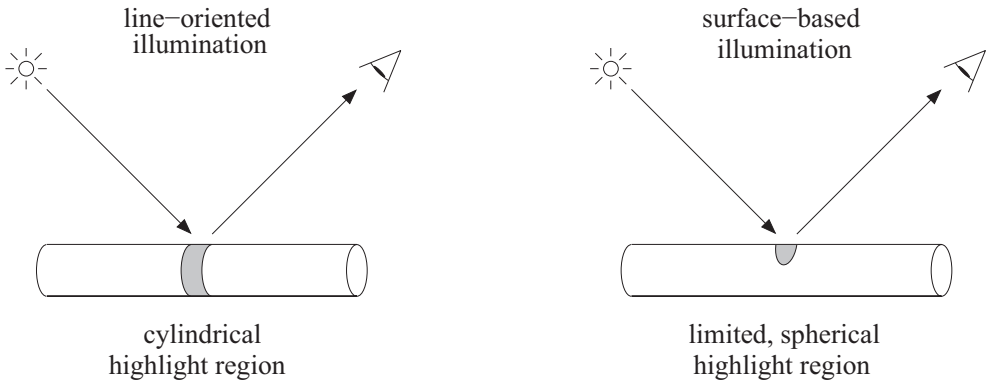


Figure 5.4: Comparison of (a) line-oriented illumination and (b) surface-based illumination, both applied to a thick cylinder that approximates a line. In the line-oriented approach, the specular highlight is erroneously present around a full cylindrical stripe. However, the highlight should only cover a limited, sphere-shaped area, as correctly computed by the surface-based approach.

whereas the dot product for the specular term can be written as

$$c_s = \mathbf{R} \cdot \mathbf{V} = -(\mathbf{L} \cdot \mathbf{T})(\mathbf{V} \cdot \mathbf{T}) + \sqrt{1 - (\mathbf{L} \cdot \mathbf{T})^2}\sqrt{1 - (\mathbf{V} \cdot \mathbf{T})^2}\,. \qquad (5.5)$$

The combined contribution of ambient, diffuse, and specular lighting can be expressed in terms of only two parameters, $(\mathbf{L} \cdot \mathbf{T})$ and $(\mathbf{V} \cdot \mathbf{T})$. In the implementation by Zöckler et al. [208], illumination is pre-computed and stored in a 2D illumination texture S. Texture coordinates $(\mathbf{L} \cdot \mathbf{T})$ and $(\mathbf{V} \cdot \mathbf{T})$ are attached to the line geometry to access the illumination texture during runtime.

This illumination method has to be slightly extended for an on-the-fly rendering of texture-based streamlines. First, the illumination model is applied to each fragment on the texture slices used for volume rendering—not to a geometrically defined curve. Second, the texture coordinates $(\mathbf{L} \cdot \mathbf{T})$ and $(\mathbf{V} \cdot \mathbf{T})$ cannot be attached to the proxy geometry of volume rendering (i.e. the slices). Instead, the vector field texture is accessed via a fragment program used during volume rendering. The fetched vector is normalized to unit length in order to obtain the tangent direction $\mathbf{T}$. Then, the two dot products $(\mathbf{L} \cdot \mathbf{T})$ and $(\mathbf{V} \cdot \mathbf{T})$ are computed and used as texture coordinates for a dependent lookup in the illumination texture S. All vector quantities are described in the object coordinate system of the vector data set. In this way, the tangent vector does not have to be transformed into any other coordinate system. For directional light, the light direction is transformed into object coordinates once per frame. The transformed view vector can be attached to the vertices of the slice polygons as a set of texture coordinates, avoiding any per-fragment transformation.

A potential problem of line-based illumination is that it can lead to inconsistent results if applied to surface-like objects. The very assumption of line-based lighting is an infinitesimally thin curve. Therefore, line-based illumination should only be applied to curve-like structures, and not to large-scale objects that typically arise from dye advection. Figure 5.4 illustrates the error introduced by applying line-based illumination to

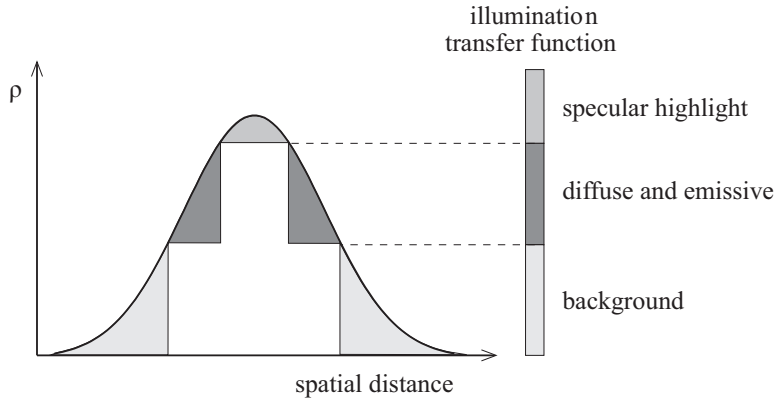


Figure 5.5: The material density profile along a straight line that crosses through a streak in the property field. The illumination transfer function, which is applied to property values, implicitly separates spatial locations within the streak into regions with specular highlights, with diffuse and emissive contributions, and with background material.

curves of finite thickness. The test object is a cylinder of finite thickness. Line-based lighting leads to a specular highlight that fills a complete ring around the cylinder because the tangent vector is the same for all those positions (even if the backfacing part might not be visible to the camera due to occlusion). In contrast, the correct illumination, as computed by the surface-based approach, leads to a smaller, sphere-shaped area that contains the specular highlight. In fact, the correct specular highlight is always part of the typically larger highlight region computed by the line-based method. The extent of possible error depends on the width of the curve: the thinner the curve, the smaller the possible room for error. In the extreme case of a one-pixel-wide curve, we achieve per-pixel accuracy with line-based lighting.

The discussion of potential error has so far been based on completely opaque, solid curves of finite thickness. However, the property field generated by advection and blending represents a material density that is mapped to colors and opacities by means of a transfer function. For this purpose, the transfer function is extended to control specular reflection and reduce the size of erroneous highlight regions. Figure 5.5 illustrates the material density profile of a single streak in the property field. Such a 1D profile is obtained from property values along a line crossing the streak. Since the streak is generated from low-pass filtered (i.e. smoothed) input noise, its profile is smooth and has its maximum at the center of the streak.

In order to modulate the specular component an *illumination transfer function* is used: only for high property values, the specular contribution is taken into account; otherwise it is multiplied by zero. In this way, the specular highlight is restricted to a small center part of the streak. Diffuse illumination and emissive colors are present in high and medium-range property values, whereas low property values correspond to background colors. Through accumulation of colors and opacities during volume rendering, the spatial concentration of specular highlights on the center of streaks remains even in the final image because a ray that hits a streak off-center only collects non-specular contributions. Similarly to limb darkening [58], which emulates a halo

effect by choosing an appropriate transfer function, a removal of specular highlights at the edges of streaks is achieved. Accordingly, this effect is called *specular edge darkening*. In contrast to the simplified illustration in Figure 5.5, this new method uses an illumination transfer function that provides a smooth transition from specular, over diffuse and emissive, to background areas in order to avoid high spatial frequencies and aliasing in the final image.

One problem that cannot be overcome by specular edge darkening is that the highlight might be located at slightly shifted positions on the streak. However, this inaccuracy does not affect the important benefits of specular edge darkening: the reduction of the size of highlights and the provision for a fine structure in streak shading.

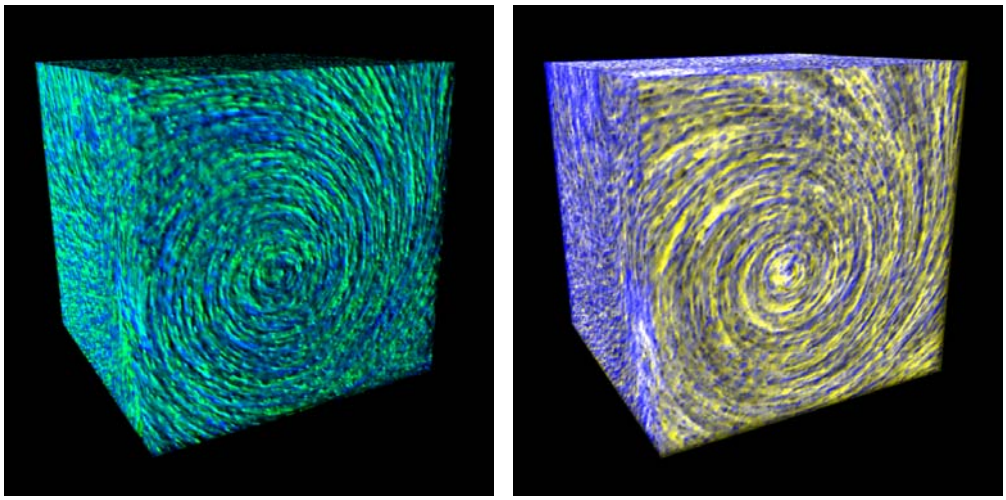
Figure 5.3 (b) and Figure 5.3 (c) compare gradient-based and line-based illumination for the example of a tornado data set. Figure 5.3 (c) demonstrates that line-based illumination is capable of displaying clear Phong highlights that are centered on the streaks due to specular edge darkening. In addition, it conveys the relative depth of streaks by means of partial occlusion and limb darkening. In contrast, gradient-based illumination in Figure 5.3 (b) exhibits high-frequency artifacts in the structure of specular highlights caused by the inaccuracies in computing gradients. The limited spatial resolution of the property field, the quantization of property values to 8-bit resolution, and the inaccuracies due to second-order central differences degrade the gradient quality. On the other hand, gradient-based illumination surpasses line-based illumination with respect to the quality of diffuse illumination because the Lambert term is less sensitive to gradient inaccuracies.

5.2.3 Perception-Oriented Illumination

So far, illumination was restricted to the Phong model, which targets a realistic look for object shading. More sophisticated local lighting models, such as the physics-based Cook-Torrance model [28], could be easily included as long as they can be expressed in terms of normal, view, and light vectors.

The goal of texture-based flow visualization is to visually represent data, not to strive for photorealism. Therefore, perception-oriented illumination plays a more important role than physically based lighting. A prominent example of a non-photorealistic illumination model is cool/warm shading [49; 38]. Here, orientation with respect to light direction is encoded by warm (yellowish) or cool (bluish) colors, respectively. Figures 5.6 (a) and (b) compare Phong illumination and cool/warm shading. The advantage of cool/warm shading is that orientation is represented by (almost) isoluminant colors. Therefore, brightness can still be used to visualize another attribute or property. The computation of cool/warm shading is strongly related to the computation of diffuse illumination—it essentially requires a dot product between light and normal vectors. Therefore, cool/warm shading can be applied for the gradient-based as well as line-based approaches.

Another means of improving the perception of texture-based flow visualization is to include additional attributes to modify the illumination. A prominent example is the depth with respect to the camera. For this purpose, color-based depth cueing is applied to imitate aerial perspective. Variations are also feasible, e.g., a subtle blue shift [38] can be included by adding a blue component to the fog color. Depth-cueing and other modifications based on additional attributes can be used with the gradient-based and



(a) (b)

Figure 5.6: Comparison of gradient-based illumination: (a) Phong illumination and (b) cool/warm shading.

line-based approaches because they can make use of the same attribute information.

As a further method, halos can be employed to visualize relative depth between line-like structures [65]: objects behind a closer streakline are partly hidden by dark halos. One way of implementing halos is to render them as thick silhouette lines. Silhouette lines are detected by examining the dot product of gradient and viewing directions because an ideal silhouette has a gradient perpendicular to the viewing vector. An additional transfer function is included to specify halos. This transfer function maps the above dot product to color and opacity. Thick silhouette lines are implemented by mapping a finite range of input values (around zero) to high opacities. This approach to halo rendering requires gradients and, thus, can only be used in combination with gradient-based illumination. As an alternative, limb darkening through high opacities in the transfer function [58] contributes to a halo effect for purely geometric reasons. Limb darkening only needs volume rendering without any illumination and accordingly can be applied to the gradient-based and line-based approaches. However, gradient-based illumination is preferable because the combination of the gradient criterion and limb darkening enhances the halo effect.

Here, we would like to point out that always several materials are advected because different line colors are an effective means of visualizing continuity along those lines [65]. For further example images or videos that demonstrate perceptual rendering approaches, including halo rendering and depth cueing, see the papers [195] and [196] as well as the accompanying web page `http://www.vis.uni-stuttgart.de/texflowvis`. The videos also show that the perception of flow direction and speed is further improved through animated visualization.

5.2.4 Performance Measurements

Table 5.1 shows performance measurements for the new 3D texture advection method on ATI Radeon X1900 and NVIDIA GeForce 8800 GTX GPUs according to the con-

	Radeon X1900		GeForce 8800	
Domain size	128^3	256^3	128^3	256^3
Advection only	187.5	26.1	200.8	129.1
Reorder stacking axis	5.1	0.9	5.6	1.4
Gradient computation	261.5	37.0	292.8	219.7
Advection and volume rendering:				
No illumination	76.0	17.9	128.0	50.8
Gradient-based illumination	59.7	11.5	102.1	30.76
Line-based illumination	61.8	16.0	120.5	45.1

Table 5.1: Performance for steady flow visualization on a 600^2 viewport (in fps).

figuration described in Section 1.5. Viewport size for volume rendering is 600^2, the size of the property and gradient fields are given in the table. Here, we use a steady vector field of size 128^3 (tornado from Figures 5.3 and 5.6). The measurements indicate that advection speed is roughly proportional to the number of texels. Typically, the computing time for advection is in the same range as the computing time for volume rendering; the exact distribution of computing time depends on several factors, including the relative size of the viewport and the number of slices for volume rendering. Volume illumination is typically slower than pure emission-absorption volume rendering. However, there is a significant performance benefit for line-based illumination over gradient-based illumination. In fact, for a property field of size 256^3, the speed of line-based illumination is only some ten percent lower than for the emission-absorption model. In contrast, the frame rates of gradient-based illumination are reduced by some forty percent. For the smaller property field of size 128^3, differences are less pronounced because computational overhead from other parts of the visualization system plays a larger role at high frame rates (at more than 50 fps). In general, all methods show that an interactive visualization is more than feasible with property fields of size 256^3. The only issue is the slow reordering of the stacking direction. Switching between stacks, however, does not occur very often in interactive applications and, therefore, this rendering bottleneck typically does not disturb the user.

Table 5.2 compares the visualization performance for unsteady and steady flow under the same conditions as for Table 5.1. Gradient-based illumination is employed for volume rendering. For unsteady flow, a new 3D texture for the vector field is transferred for each frame. Although the rendering speed is reduced for unsteady flow, the overall performance still facilitates interactive visualization for property fields up to 256^3. The measurements in Table 5.2 assume that the unsteady flow data can be completely held in main memory. An out-of-core visualization of very large time-dependent data might further reduce the rendering speed, caused by reading data from disk. However, intelligent data management and pre-fetching could be investigated in future work to hide the effects of disk access.

Table 5.3 compares the advection method of this thesis [195; 196] (implemented in DirectX) and a previous 3D texture-based method [191] (implemented in OpenGL). Here, volume rendering is used without illumination on the test platform with ATI Radeon X1900 GPU. The comparison demonstrates that the new 2D texture-based

	Radeon X1900		GeForce 8800	
Domain size	128^3	256^3	128^3	256^3
Advection only:				
Steady	187.5	26.1	200.8	129.12
Unsteady	98.2	18.9	104.5	98.76
Advection & gradient-based volume illumination:				
Steady	59.7	11.5	128.0	50.8
Unsteady	37.5	8.7	85.42	47.4

Table 5.2: Performance for steady vs. unsteady flow visualization on a 600^2 viewport (in fps).

	2D Texture		3D Texture	
Domain size	128^3	256^3	128^3	256^3
Advection only	187.5	26.1	71.0	18.3
Advection & volume rendering	76.0	17.9	25.9	6.3

Table 5.3: Comparison between 2D texture-based advection of this paper and 3D texture-based advection from [191] on Radeon X1900 with 600^2 viewport (in fps).

method outperforms the 3D texture-based method, i.e., the benefit of fast read and write access to 2D texture outweighs the additional operations for the mapping of texture coordinates from logical 3D space to physical 2D space.

All above measurements are based on volume rendering without early ray termination. As described in Section 5.1.2, the early z test can be used to implement early ray termination on the fragment-processing stage. For the example of volume rendering without illumination on an ATI Radeon X1900 GPU (as laid out in Table 5.3, left columns), early ray termination increases the rendering speed from 76.0 fps to 103.3 fps (128^3 property field) and from 17.9 fps to 21.9 fps (256^3 property field). Typically a speed-up is observed by some twenty percent.

5.3 Texture Advection on 2D Slices

As already discussed in Section 5.1, 3D texture advection provides a fast and powerful method for the interactive visualization of 3D unsteady flow. Although the problem of finding appropriate initial seed points is solved by this approach, the problem of clutter and occlusion remains [65].

Not only does a dense 3D representation pose a problem for the display of the data set by potentially occluding important information, but it also makes it difficult for the user to interact with the data set: Picking and selection in 3D is cumbersome since usual input and output devices (such as mouse and computer screen) are intrinsically

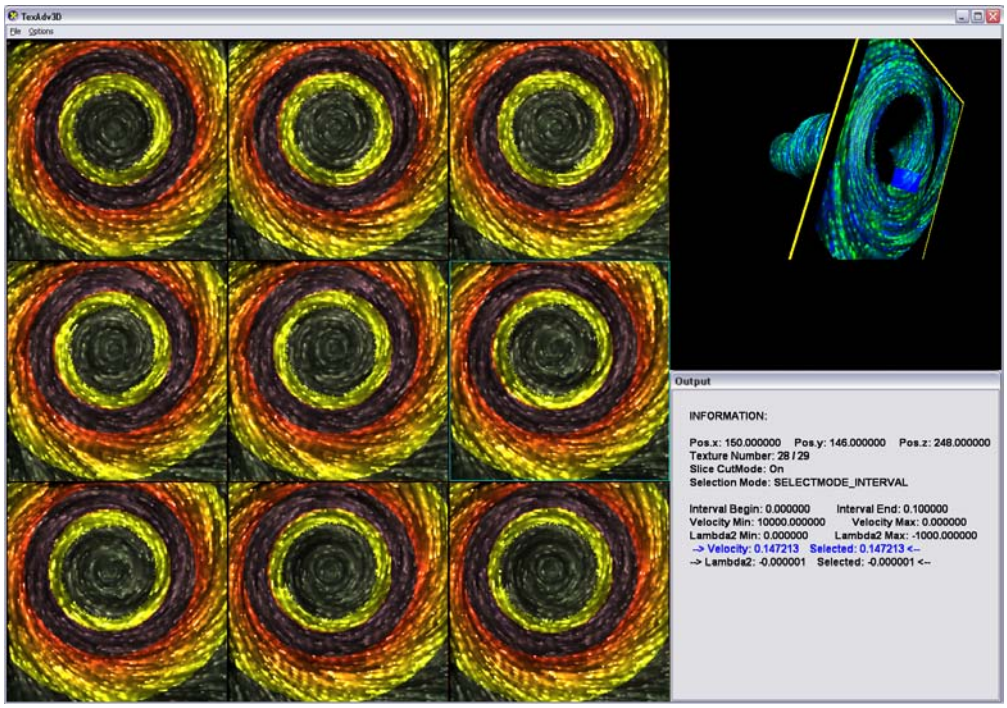


Figure 5.7: Example of linked 2D/3D texture advection: 2D representations (on the left) side-by-side with a 3D representation (upper-right part).

two-dimensional. In this section, the issues of occlusion, navigation, and interaction using 3D texture advection are addressed without losing the advantages of a 3D view according to the work of Schafhitzel et al. [152]. This section contains parts of the original text from [152].

5.3.1 Multiple Views and 3D Texture Advection

The goal mentioned above is achieved by simultaneously using dense 2D and 3D representations of the same data set, as illustrated in Figure 5.7. 3D texture advection in combination with volume rendering can be employed for the dense 3D visualization [191; 195; 196]. Adopting the traditional display of medial CT (computer tomography) or MRI (magnetic resonance imaging), the 2D representation is computed on several parallel slices through the data set, which are simultaneously shown in a tabular layout. It is crucial for this approach to tightly link these two representations in order to gain an unrestricted view on all regions of the data set and, at the same time, explore the 3D nature of the flow. Coupling between views is achieved through brushing and linking, as well as navigational slaving [182]. Brushing and linking [9; 17] automatically propagates the selection of data values by the user to other views by highlighting corresponding data values. Navigational slaving is implemented by marking the current spatial position (via 2D and 3D mouse pointers) simultaneously in the 2D and 3D views.

Another benefit of the 2D view is its immediate extensibility to multi-field visualization: An additional attribute of the data set can be shown by color coding, overlayed

on top of the vector field visualization. Typical examples of such an attribute are vortex strength by λ_2 [67], velocity magnitude, pressure, or temperature of the flow. For a more detailed discussion on the computation of flow features see Section 7.

A third goal is to achieve interactive visualization because tight linking is only useful in combination with immediate feedback to the user. Therefore, efficient visualization methods have to be employed. In the proposed system, the high processing speed of GPUs (graphics processing units) is used to achieve advection and rendering of the 3D view at interactive frame rates. The 3D computation is directly used to generate the additional 2D views. Therefore, combined 2D and 3D visualization comes with only marginal additional performance costs, as compared to a stand-alone 3D visualization.

Up to that point there exists various related work in the field of multiple views. Since there is a broad application of multiple views, they are employed for various types of visualization. Guidelines for an effective use of multiple views in information visualization are summarized by Wang Baldonado et al. [182]. The spatial organization of visual information is often oriented along tabular layouts—typical examples are scatter plot matrices [27], image spreadsheets [102], or visualization spreadsheets [25]. In general, linking and interaction between views is a key element in making multiple visual representations comprehensible and effective. Coordinated and multiple views are especially useful for interactive exploration [15]. The role of linking is investigated and different linking methods are evaluated by Plumlee and Ware [126]. Brushing and linking is a widely applied and classical method for linking highlighted regions in multiple views [9; 17]. Recently, it was shown how 3D scatter plots can be improved by linking several views [88; 125].

Advanced techniques for coordinated and linked views are often applied in information visualization, but they used to be less common in scientific visualization. A recent trend, however, is to adopt classical methods from information visualization for problems in scientific visualization. For example, focus-and-context approaches and coordinated multiple views are effective tools for flow visualization [35]. The design of transfer functions for volume visualization is another field in which coupled views are successfully applied [84]: Full 3D view, selection of features on 2D slices within the 3D domain, and exploration of structures in the multi-dimensional histogram and transfer function domain are used side-by-side.

5.3.2 Slice-Based 2D Multi-Field Visualization

This multiple views method for 2D flow visualization is directly based on the 3D texture advection implementation from Section 5.1. Planar slices are extracted from the 3D property field ρ (see Eq. (4.4) and Eq. (4.5)), reusing the results of the previous 3D computation. In this way, only minimal computational overhead is introduced for the actual display of the 2D slices.

A tabular layout is employed in order to visualize several slices simultaneously. Adopting the CT metaphor (known from the medical imaging of computer tomography), the layout is organized in a column-first order, displaying slices of increasing z value in a left-to-right and top-to-bottom fashion. (This ordering reflects the cultural influence from Latin / Western text writing and may be modified to take into another cultural background.) As an example, Figure 5.8 shows a slice-by-slice visualization

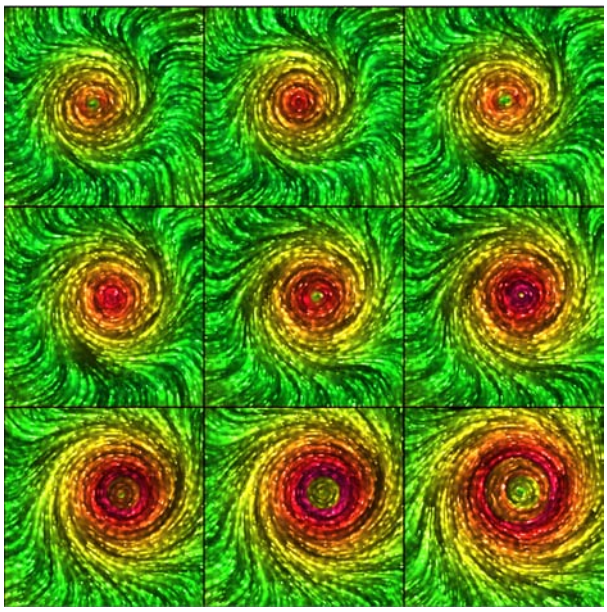


Figure 5.8: Slice-based 2D flow visualization of a tornado data set. The tabular layout displays slices of increasing z value in a left-to-right and top-to-bottom order.

of the tornado data set from Figure 5.6. Engineers are used to flow visualization on 2D slices from many existing visualization systems. Therefore, slicing is an acceptable approach to build a 3D impression of the data set through 2D visualization.

Typically, the available amount of image space is limited and, therefore, not all slices of a full 3D data set can be displayed simultaneously. In this case, only a subset of all slices is viewed at the same time, and the user can access other parts of the data set by vertical scrolling. The number of tiles is fixed along the horizontal axis, i.e., no horizontal scrolling is used. In this way, a natural and intuitive identification of the vertical axis on the screen and the z axis of the data set is established. The fixed relative position of different slices leads to an intrinsic coupling of the different views. The coupling is supported by the fact that during temporal evolution all tiles show the same time step of the visualization.

Multiple 2D slices can be implemented by accessing the previously generated 3D property field ρ. A slice is drawn onto the screen by rendering a rectangle that is textured according to the respective entries from the 3D property field. Corresponding 3D texture coordinates are attached to the vertices of the rectangle when ρ is represented by a 3D texture. In case of 2D physical memory (Section 5.1.1), 2D slices are filled with 2D texture information. If the stacking direction in 3D space (i.e., the orientation of the z axis of logical memory) is not compatible with the slicing direction for the 2D view, the stacking direction can be modified efficiently by GPU operations [195; 196]. Alternatively, a stripe-based approach can be employed to directly render from a 2D data format that is organized in any stacking direction [133].

2D flow visualization has the advantage that it can be easily extended to multi-field visualization: A second data field can be visually overlaid on top of the vector field representation. We allow for any data field whose data value is, or can be, reduced to a single component. Typical examples for such a single-component data field are

pressure, temperature, or dissipative energy in a flow. Other examples are derived from the velocity field, e.g. velocity magnitude or λ_2 [67] (see Section 7.2.2), which describes vortex strength [67]. The actual choice of the second data field depends on the application. In any case, however, the second field is a characteristic input to explore features of the flow and, therefore, is called *feature field*. An overview on flow features as well as their computation is discussed later in this thesis, in Chapter 7. The feature field serves as basis to define the degree-of-interest function. The interaction model for feature exploration is described in Section 5.3.3.

The combination of vector field visualization and the visualization of the second data field is based on color coding. The human visual system facilitates a 3D color space: Through combination of input signals from the cones in the retina, ganglion cells encode visual information in form of a one-component luminance channel and two chromatic channels [77]. Here, color tables are used to map the density values from the property field ρ to luminance contrast and encode the feature field by means of chromatic contrast. In this way, fine-grain flow structures are represented by luminance contrast, which is most suitable to display high levels of spatial detail [135]. In contrast to the 3D case, 2D vector field visualization is not affected by issues of depth perception and, thus, does not require continuity of lines. Therefore, different materials are not mapped to different colors (as for the 3D view) but combined into a common luminance representation.

The feature field is mapped to the remaining two chromatic channels. We, the authors of [152], recommend to map the single-component data value to either saturation or hue, which are intuitive concepts of color perception. Hue, in particular, is effective for our application because it allows for a visual ordering and labeling of the second data field. Although our color encoding is technically realized by a bivariate color map, only chromatic contrast is used to represent quantitative data in the sense of a traditional color map. Therefore, advanced guidelines for designing univariate color maps [183] can be adopted.

The HSV (hue, saturation, value) color space is employed to generate the color tables. The property field ρ is mapped to value, the second data field to hue. Saturation is chosen constant. Although the HSV model does not completely separate luminance and chromatic channels, it is an acceptable approximation for our range of application. Figure 5.9 illustrates color encoding for multi-field visualization on a slice through the tornado data set. The vector field is visualized by rendering the corresponding property field with luminance contrast. Velocity magnitude is used as feature field and is encoded by different hue values. Figure 5.9 shows these hue values (for fixed saturation and value in the HSV model) in the legend.

Similarly to the GPU implementation of transfer functions, the color table for the 2D view is represented by a texture. Here, a two-parameter color table is used, which is implemented by a dependent 2D texture.

5.3.3 Interactively Linking 2D and 3D Texture Advection

Since both 2D and 3D flow visualization have specific advantages and disadvantages, it is desirable to combine the strengths of both views. A 2D view is not subject to occlusion problems, facilitates the visualization of a second feature field, and supports an easy exploration of data values. A 3D view, on the other hand, is very effective in

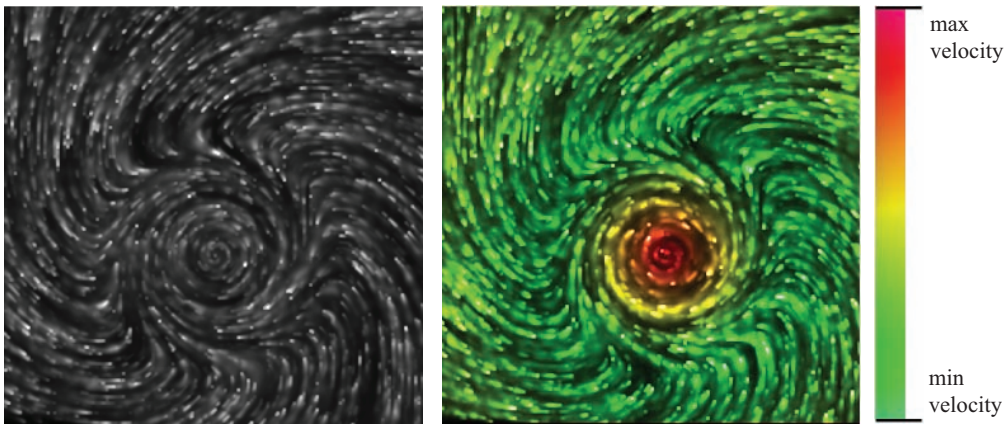


Figure 5.9: Combined visualization of vector and feature fields (image to the right). The vector field is indirectly represented by a luminance-based visualization of its corresponding property field. The feature field is described by velocity magnitude and mapped to hue values. For comparison, the left image shows only the vector field via luminance contrast.

visualizing 3D structure.

Following the description by Wang Baldonado et al. [182], three different dimensions can be distinguished for a generic multiple-view system: selection of views, presentation of views, and interaction between views. In what follows, this approach for linked flow visualization is related to these generic concepts.

The proposed visualization system uses an essentially fixed selection of views. A single 3D view is employed, along with a number of 2D views for different planar slices through the data set. Only for the 2D views, a restricted selection is possible: Vertical scrolling is supported, as described in Section 5.3.2. In this way, a subset of all parallel slices can be selected by the user. Note that this subset contains only directly neighboring slices.

The selection of views is restricted, and so is their presentation: a fixed layout is chosen in which the 3D view is located in the upper-right part of the window and the 2D views are positioned in the left part (see Figure 5.10). The 2D representation covers a large portion of the available space—approximately two thirds of the window—because it needs much screen space to show several 2D views. The user may scroll the 2D views by using the keyboard. In addition to the 2D and 3D views, numerical values that correspond to user-selected elements of the data set are shown in the lower-right part of the window. Moreover, views are always shown simultaneously, without toggling between them. The fixed layout for the simultaneous presentation of multiple views has several advantages. Firstly, the time and effort required to learn the system is relatively small. Secondly, a consistent presentation is achieved. Thirdly, only minimal temporal costs are involved for switching between different 2D views and the 3D view because eye movement is very fast. In contrast, toggling between views via mouse or keyboard would lead to much lower user performance. The main disadvantage of a fixed layout with simultaneous multiple views is a demand for large image space.

The last, and most important, dimension concerns the interaction between views.

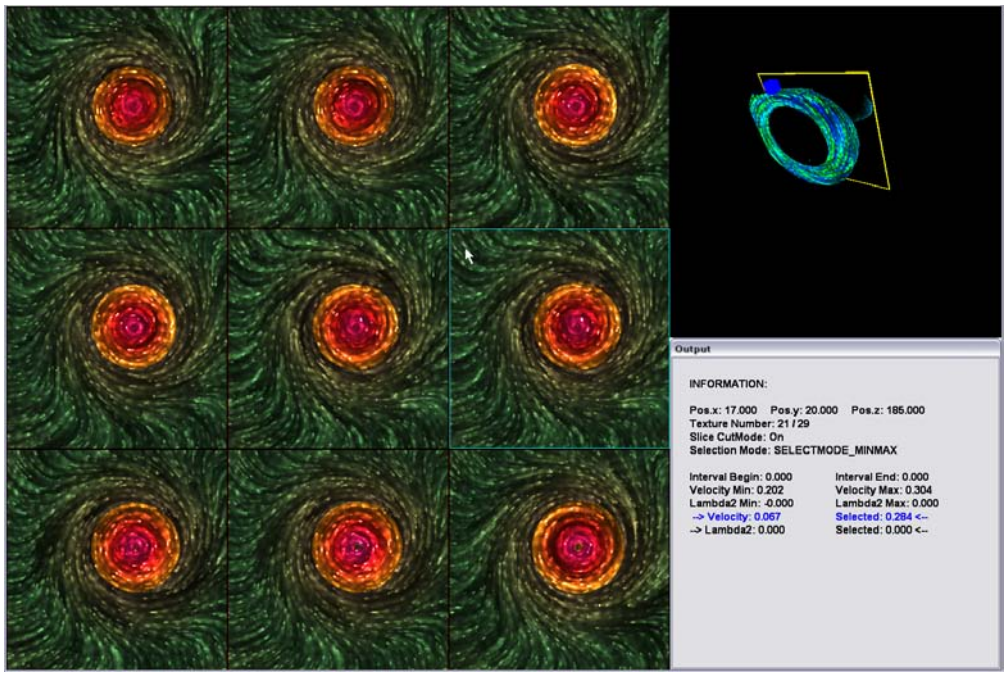
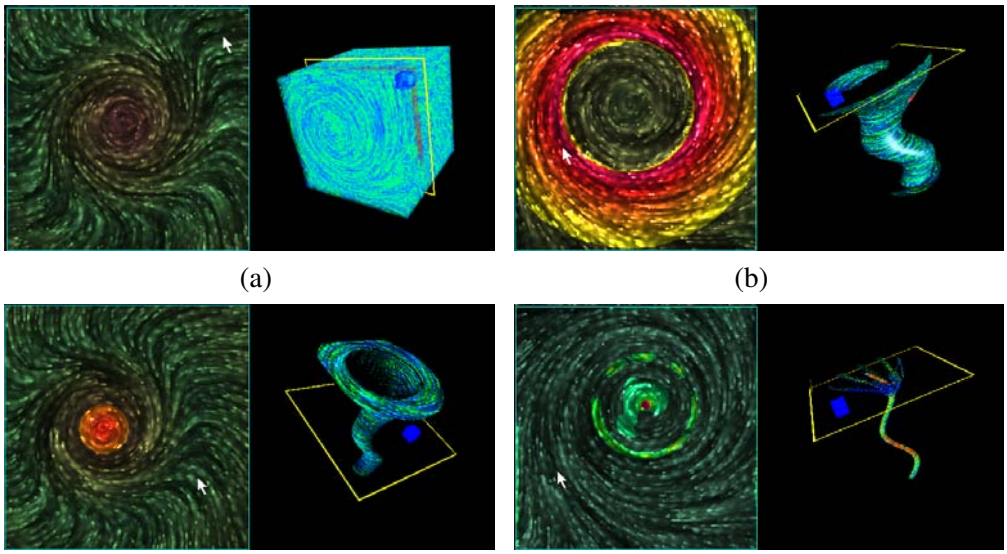


Figure 5.10: Overview of the layout for linked 2D and 3D texture advection. The 3D view is located in the upper-right part of the window and the 2D views are positioned in the left part. In addition, numerical values that correspond to user-selected elements of the data set are shown in the lower-right part of the window.

In this work, different ways of coupling the 2D and 3D views are employed. The first coupling method is based on linking that connects data in one view with data in another view. In particular, a specific type of linking is employed: brushing [9], which allows the user to highlight certain data elements in one view and automatically highlights the corresponding elements in other views. The user may highlight data in any of the 2D views by picking. Picking in 2D is much easier than in 3D because it is immediately supported by 2D input via mouse. Moreover, a 2D view is not subject to occlusion issues and avoids the problem of unwillingly selecting more than a single data point. Feature exploration is based on the feature field and, therefore, values are brushed in the domain of this second data field. In general, brushing is based on a selection of data values in an interval. (Smooth brushing [36] is not considered.) In detail, two slightly different interaction methods to define a brushing interval: The first method—called *interval picking*—allows the user to pick one point to select a single data value. The selected data range can then be extended by shifting the lower and upper bounds of the data interval by using the keyboard. The second method—named *min/max picking*—allows the user to pick several points. The brushing interval is determined by the minimum and maximum values selected so far and has to be updated when the user picks an additional value. In this way, the user can select a value range by virtually painting into the data domain.

The selected data interval is directly linked to all visible views. For the 2D views, the color table from Section 5.3.2 is extended to incorporate an additional binary parameter that describes whether a value is selected or not. The HSV-based color table



(a) (b)

(c) (d)

Figure 5.11: Illustrating feature exploration by brushing and linking. No feature is selected in (a); the left part shows the 2D view of the vector field and the velocity magnitude and the right part shows the 3D view. In (b), min/max picking is used to select a wide interval of velocity values. In (c), interval picking is used to choose a smaller interval of velocity values. Image (d) illustrates interval picking for λ_2 features.

represents this additional dimension by color saturation: Highly saturated colors are used for selected regions, less saturated colors represent deselected regions. The perceptual differences between both regions can be further emphasized by reducing the luminance contrast in deselected parts. Figures 5.11 (b)–(d) show examples: In each figure, the left image shows a 2D view with highlighted regions in saturated colors and deselected regions with low saturation and contrast. This extended color-table model is once again implemented in the form of GPU-based dependent textures. Two textures hold two different color tables for selected and deselected state, respectively. During fragment processing, one of the two dependent textures is chosen, depending on the data value.

Similarly to the 2D views, the 3D vector field representation is immediately affected by brushing. Brushing is used to define a binary degree-of-interest function for the 3D data set: Selected data values correspond to a high interest value, deselected values correspond to a low interest value. The interest value is mapped to opacity in the transfer function that governs the volume rendering to the property field ρ. Therefore, brushing only affects the definition of the transfer function and does not change any other elements of the implementation of the 3D view.

The second coupling method is based on a special form of navigational slaving, in which navigational movements in one view are propagated to other views. The current position in a 2D view is represented by a mouse pointer. Navigational coupling is employed by marking the corresponding position in 3D space via a 3D mouse pointer. Figures 5.10 and 5.11 illustrate this navigational coupling: The 2D position is shown by a white arrow as mouse pointer, while the corresponding 3D position is indicated

by a blue box that surrounds that position. Navigational slaving is intensified by simultaneously marking the current slice both in 2D and 3D views: In Figure 5.10, the 2D view is framed by a thin bluish rectangle and the corresponding slice is marked by a yellow outline in the 3D view.

Another feedback mechanism provides the user with detailed information about the selected feature region. As shown in the lower-right part of Figure 5.10, an output area displays accurate numerical data about the features of the vector field, the selected data values, and the data values at the current position of the cursor. Furthermore, the position of cursor in 3D space, the viewing and selection modes, and the number of 2D slices are shown.

Finally, the user can interact with the 3D view by changing the viewing parameters in real time. In this way, 3D navigation through the data set is possible, which facilitates improved depth perception via motion parallax and allows the user to explore any parts of the 3D data set.

Figure 5.11 illustrates how a user can perform feature exploration by brushing and linking. In Figure 5.11 (a), no feature is selected. Therefore, the left part of Figure 5.11 (a) shows a 2D view of the vector field and the velocity magnitude in slightly saturated colors. The 3D view (right part of Figure 5.11 (a)) does not show any interior details of the 3D data set because they are occluded by a dense vector field representation on the cubic boundary of the data set. The user can select interesting data intervals of the feature field to emphasize features of the data set. In Figure 5.11 (b), for example, min/max picking is used to select a wide interval of velocity values. The selected values correspond to a high velocity magnitude, which can be found in the outer parts of the tornado tube. In Figure 5.11 (c), interval picking is applied to chose a smaller interval of velocity values. In this way, the tube can be extracted more clearly. Finally, Figure 5.11 (d) illustrates interval picking for another feature field: λ_2, which indicates vortex structures. With this feature field, the vortex core can be extracted by brushing.

It is important to note that interactive feature selection and immediate feedback in the 3D view are crucial for an effective exploration. In this way, the user can mentally link his or her actions on the feature field with the effects on the overall 3D view.

5.4 Concluding Discussion

Although texture-based techniques are very popular in recent research, they are also subject to a couple of limitations that need critical discussion. Thereby the balance between performance and accuracy is an important point. Conservative geometry-based methods like stream lines, path lines or particle tracing are indeed more accurate than 3D texture advection in terms of numerical errors. The reason is that these methods are Lagrangian, i.e., the domain is not discretized. However, even when fast GPU implementations like the GPU-based particle tracing method by Kipfer et al. [83; 89] are used, Lagrangian methods lack performance in comparison to semi-Lagrangian methods like the 3D texture advection approach described in Section 5.1. This is because of the complex data structures, which are necessary to store the particles; 3D texture advection makes use of a 3D cartesian grid–the property field–which can accessed very efficiently. Furthermore, due to the transport mechanism of texture advection, it is possible to visualize flow divergences without any additional overhead—the repre-

sentation of flow divergences with Lagrangian methods is rather expensive, since new particles must be generated on-the-fly. Section 5.1 shows how 3D texture advection can be performed very efficiently by exploiting the fast access of 2D textures. Texture-based approaches are not only fast, but can also be accurate, as is shown by the 3D LIC approach by Falk et al. [42]. Although it is computationally more expensive than 3D texture advection, 3D LIC provides a dense vector field representation with the same accuracy as Lagrangian approaches at interactive framerates.

Even the problem of illuminating the streak lines resulting of 3D texture advection is solved nowadays: as described in Section 5.2.1, an on-the-fly gradient computation can be performed efficiently. However, if the texture memory is limited or the GPU is too slow, also a line-based illumination model can be applied (Section 5.2.2).

Another problem of 3D dense representations is occlusion. Section 5.3 shows how this issue can be solved by combining 2D slices with a linked 3D representation. Note that this approach is not restricted to 3D texture advection only—it is also possible to apply it to 3D LIC. Similar to the property field, the result of 3D LIC can also be represented by 2D slices, what makes it appropriate for a linked 2D/3D representation. Also masking can by applied to both approaches.

In summary, it can be observed that texture-based techniques are becoming an attractive alternative to classical geometry-based methods. Both classes of methods have their advantages and disadvantages like the memory efficiency of geometry-based approaches and the prevention of finding appropriate seed points when using texture-based approaches, but nowadays a combination of both classes might be the most efficient way to explore a 3D vector field: the fast texture-based techniques can be used for getting a global overview in order to locate the certain areas of interest; the accurate geometry-based approaches can then be used for investigating the problem more locally.

CHAPTER

6 FLOW VISUALIZATION ON CURVED SURFACES

So far, it has been shown how flow in a 3D domain can be represented by a dense representation. Another topic in flow visualization consists of the flow representation on surfaces, whether it is on geometry, like the vehicle shown in Figure 4.5, or it is on extracted surfaces, as there are path surfaces in Section 6.1 or vortex structures in Section 6.3 or surfaces extracted out of anatomical data, like in Section 6.2. This chapter discusses the application particle tracing for generating and emphasizing flow on curved surfaces.

In Section 6.1, both the path surfaces and the surface LIC representation are the result of particle tracing; in Section 6.2, particle tracing is applied to the principal direction vectors of an isosurface that is extracted out of anatomical data; and in Section 6.3, the flow around a vortex is illustrated by particle tracing. The similarity of these methods is that surface LIC should improve the perception and that the same surface LIC implementation is used. In all three methods, the surface LIC representation is used to improve the perception by providing additional information that replaces other rather unsuitable techniques that are used for this purposes so far. For example, in Section 6.1 and 6.3, surface LIC is used to emphasize the paths of flow particles on the respective extracted surfaces. This enables the user to gain a better insight and, therefore, a better understanding of the extracted structures. Questions like, "Why is the path surface that wide in this area?" and "In which direction is the vortex actually rotating?" are answered without losing any other information. Here, in both cases this texture-based representation on the surfaces replaces conventional stream lines or path lines, which though they provide the same information as the surface LIC, lead to occlusion and, therefore, to a more cluttered representation. Section 6.2 solves the same perceptual issues, even though in another manner. Here, perception is improved by almost replacing the representation of highly opaque isosurfaces by short sparse line like structures that illustrate the individual isosurface's shape. As already mentioned above, the implementation is almost identical in all three cases, only the underlying vector field is of another nature.

6.1 Stream Surfaces and Path Surfaces

In this section, the class of 1D stream lines is extended to their respective 2D representation, the stream surfaces. Both representations are directly linked to the topic of this thesis, namely particle tracing, since both representations are results of tracing particles along a given vector field. The following work was in collaboration with Eduardo Tejada from the Universität Stuttgart, who must be credited for the point-

based rendering of the stream surfaces. This section contains parts of the paper by Schafhitzel et al. [150].

Stream surfaces are defined as surfaces that are everywhere tangent to the vector field. They are effective in simultaneously displaying various kinds of information of a flow. In addition to flow direction, which is also visualized by streamlines, stream surfaces can show torsion of a vector field and convey vortex structure [47]. Despite these advantages, stream surfaces are not extremely common in flow visualization. According to [150] there exist various reasons for the lack of popularity of stream surfaces: one reason might be the higher complexity of computing stream surfaces comparing to the computation of stream lines. Here, more advanced algorithms are required. Beside the computation also the rendering suffers of the higher complexity of stream surfaces. Interactive changes of the seed curve (the natural extension of a seed point) are very difficult to visualize, since entire surfaces need to be recomputed with all their topological information. Exactly the surfaces' topology can be considered as the bottleneck of an interactive visualization because interactive changes require a fast computation of connectivity. Especially when wide stream surfaces are regarded the internal visual structure becomes more and more difficult to understand. This leads to possible perception problems. And last but not least, stream surfaces are, as the name already implies, restricted to steady flow.

In the following, all these issues are addressed by introducing a new interactive algorithm for stream surface construction and rendering. The idea is to avoid the expensive triangulation by exploiting point-based methods for rendering, i.e., stream surfaces are represented as a set of points that are started at a seed curve and integrated along a vector field. In detail, the following points are discussed in the next sections: (1) a point-based computation of stream surfaces that maintains an even density of particles on the point set surface; (2) point-based rendering by means of splatting; (3) the extension to path surfaces of unsteady flow; (4) the combination with texture-based flow visualization on stream surfaces and path surfaces to show inner flow structure on those surfaces; and (5) a mapping of all algorithms to efficient GPU implementations.

The visualization approach discussed in this section allows the interactive generation and rendering of stream surfaces and path surfaces, even while seed curves are modified by the user or time-dependent vector fields are streamed to the GPU. Figure 6.1 illustrates an example of stream surfaces generated by the new algorithm. On the left side only the stream surface is illustrated, without any information about the flow on the surface. A more detailed feedback about the flow behavior and the resulting shape of the stream surface is given on the right.

6.1.1 Previous Work

The previous work that is relevant for the stream surface method of [150] can be separated in three research areas: the rendering of point-based surface representations, the computation and extraction of stream surfaces, and texture-based flow visualization on those surfaces. While the texture-based visualization in terms of LIC and texture advection is already discussed in Chapter 4, only a brief overview of recent works in the field of point-based rendering and the computation of stream surfaces is given.

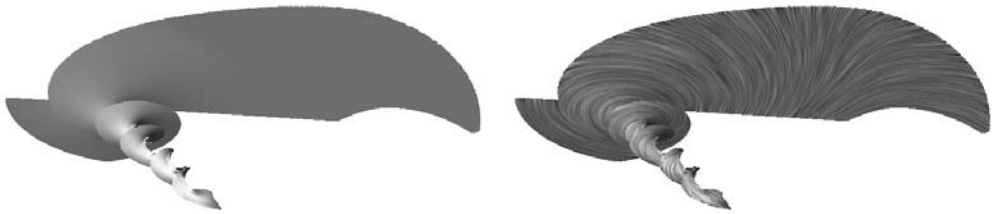


Figure 6.1: Visualization of the flow field of a tornado with: (left) a point-based stream surface; (right) the combination of a stream surface and texture-based flow visualization to show the vector field within the surface. Each stream surface is seeded along a straight line in the center of the respective image.

Point-based Rendering

Point-based methods have drawn quite some attention in the computer graphics and visualization communities over the last few years. The survey by Kobbelt and Botsch [85] provides an overview of the field. Our rendering method is based on point set surfaces (PSS), which were originally introduced for an efficient representation and rendering of point-based surfaces [2; 209]. Recently, Wald et al. [181] have developed a highly optimized CPU ray tracer for rendering PSS, and Tejada et al. [168] have described a related GPU-based ray tracing method for intersecting a primary ray with a PSS. In this paper, a PSS representation is used—in particular in combination with stored connectivity information for a fast access to neighboring points—but splatting is applied for rendering. Rusinkiewicz et al. [143] used splatting for rendering large point clouds and Zwicker et al. [209] used splatting for rendering PSS.

Stream Surfaces

The other important research topic is flow visualization based on stream surfaces. While the concept of a stream surface is straightforward, its implementation is more challenging than for streamlines because a consistent surface structure needs to be maintained. Hultquist [62] describes an algorithm that geometrically constructs a stream surface based on streamline particle tracing. In particular, his algorithm takes into account the stretching and compression of nearby streamlines in regions of high absolute flow divergence. Garth et al. [47] show how Hultquist's algorithm can be improved in order to obtain higher accuracy in areas of intricate flow. An alternative computation is based on implicit stream surfaces [176], which however cover only a subclass of stream surfaces. A related line of research addresses the issue of how stream surfaces are displayed effectively; for example, they can be chosen according to principal stream surfaces [20], rendered at several depths by using ray casting [46], or visualized through surface particles to reduce occlusion [175]. Previous methods are restricted to stream surfaces—to steady flow or instantaneous vector fields of unsteady flow—whereas this new approach is designed for steady and unsteady flow alike.

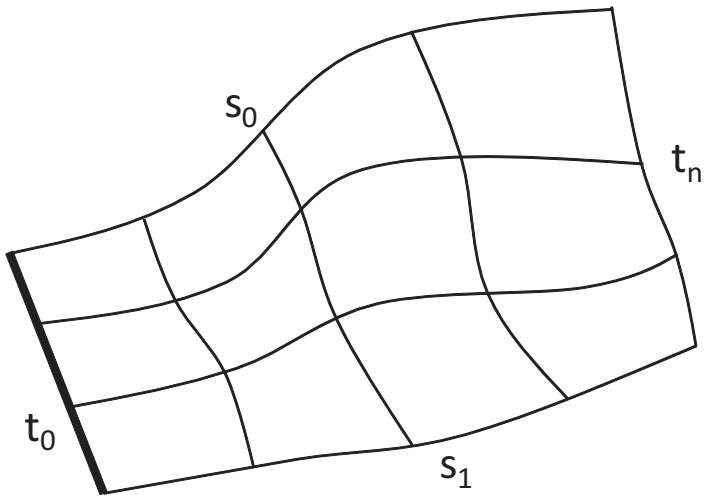


Figure 6.2: Parametrization of a stream surface accoding to Hultquist [62]: t_0 to t_n represent time lines; s_0 to s_1 is the parametrization of stream lines.

6.1.2 Definition of Stream Surfaces and Path Surfaces

Stream surfaces are surfaces that are everywhere tangent to a time-independent vector field. They extend the concept of a streamline by replacing a single seed point by a curve of seed points for which particles are traced along the vector field. According to Hultquist [62], a stream surface can be represented as a 2D parametric surface embedded in a 3D flow. Figure 6.2 shows an example. A natural choice of parameterization uses one parameter, $s \in [0, 1]$, in order to label streamlines according to their respective seed points. Assuming a parameterized representation of the seed curve, we base s on that curve parameterization. The actual streamlines are computed by solving the ordinary differential equation for particle tracing (4.3), where $\mathbf{x}$ is the particle position and $\boldsymbol{\nu}$ is the vector field at time t. The seed points represent the initial values for the ordinary differential equation. Then, the second parameter of the stream surface is the time, $t \in [0, t_{max}]$, along the streamline integration. This choice of surface parameterization results in two meaningful classes of isoparameter curves: for constant s and varying t, we obtain streamlines; for constant t and varying s, we obtain time lines, which are advected images of the initial seed line.

For stream surfaces, we assume a time-independent vector field $\boldsymbol{\nu}$. However, the above construction is already designed for time-dependent vector fields. In this case, particle tracing leads to pathlines instead of streamlines, which in turn results in the construction of *path surfaces* instead of stream surfaces.

6.1.3 Surface Generation

In this section, the algorithm for the fast generation of stream surfaces and path surfaces according to [150] is described. In order to provide an interactive tool for the generation and visualization of those surfaces, the algorithm is designed for a GPU implementation. Due to its highly parallel architecture, the GPU is well suited for computing a large number of streamlines by particle tracing. Therefore, the following discussion focuses on elements that can be mapped to the GPU programming model.

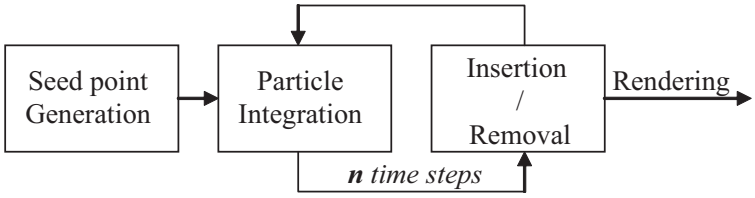


Figure 6.3: Generation of a stream surface and a path surface respectively. First, the seed points are defined followed by the integration of the particle positions. After each integration step, a particle insertion/removal method is applied, considering the distance between the neighboring particles.

In particular, the discussion considers only data structures that can be represented by textures, and to algorithms that only need information in a local neighborhood and can be executed in parallel. To the beginning, a description of of stream surface creation is given and then the minor changes required for path surfaces are discussed.

The basic algorithm consists of three parts: (1) the generation of the seed points; (2) the integration of the particles along the given vector field; and (3) insertion/removal events to maintain an evenly dense sampling of the surface by particles. Figure 6.3 shows the generation procedure. The first part is executed once only at the beginning, whereas parts (2) and (3) are repeatedly executed in an interleaved manner to incrementally construct the stream surface. The goal is to maintain a roughly even density of particles in order to obtain a good reconstruction of the surface during the point-based rendering process (see Section 6.1.4). As already pointed out by Hultquist [62], divergence of the flow can change the density of particles. Insertion and removal events that are based on a particle-density criterion play a crucial role in achieving a good distribution of streamlines.

The important data structures of the algorithm can be represented as 2D textures. One data structure is a texture, which stores the positions of the particles in the object space of the surface. The organization of this texture is rather simple: the number of rows stands for the number of particles, whereas the columns describe the number of integration steps. Actually, the number of rows has to be η times greater than the number of initial particles to allow for additional room for particles inserted during surface construction. From experience, a value of $\eta = 2$ is appropriate for typical visualizations. A second data structure—a texture holding the states—is introduced to store additional data values. This texture has the same size as the texture containing the particles and provides four additional data channels for each particle. Since we will need fast access to neighboring particles, this texture is used to store the connectivity between streamlines: it contains indices to the left and right neighbors of the respective streamline. The vector field is held in a 3D texture that is not modified by the GPU but initialized by transferring the data set from main memory to texture memory.

Initialization and Particle Integration

In the first step of the algorithm, the seed points are generated. The user defines a seed curve by placing a straight line at a specific region of interest. Seed points are distributed uniformly along this line; the distance between seeds is determined by

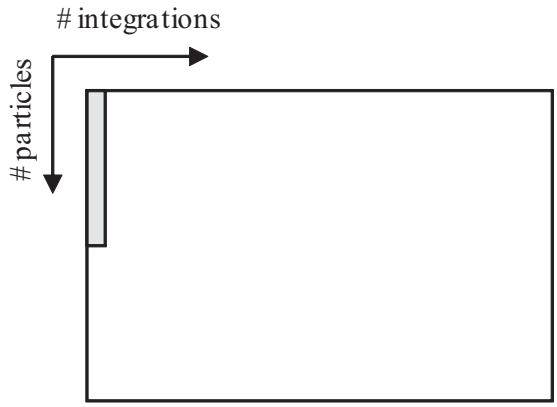


Figure 6.4: Initialization of the particles texture: only one column is rendered; the height of the strip represents the number of initial particles.

the user-specified number of initial particles. Seeding is implemented by rendering only one column into the texture containing the particle positions (Figure 6.4). The height of the quadrilateral used for rendering represents the number of initial particles. Similarly, the states texture is initialized with indices to streamline neighbors.

After initialization, the integration of particle traces is performed in step (2) of the algorithm. Eq. (4.3) is solved by applying a first-order Euler integration, but higher-order methods could be used as well. Particle tracing updates both textures in a column-wise manner, where each column corresponds to a specific time. The previous position of a particle is obtained by a texture lookup using the texture coordinates that refer to the previous column. Then, the updated position is written to the current column. Since simultaneous read and write access to textures is not supported (in DirectX) or not specified (in OpenGL), a ping-pong rendering scheme is necessary for updating the particle positions. The `states` texture is treated in the same manner to maintain consistent connectivity information.

Particle Insertion and Removal

Step (3) of the algorithm implements the insertion or removal of particles. This step relies on criteria that decide whether a particle remains, needs to be added, or has to be removed. In addition, a stream surface may tear, for example in regions of very high divergence or when the flow hits an interior boundary. Since there exists no clear definition when new particles have to be inserted to maintain an appropriate sampling of the surface or when the surface has to tear, we adopt the criteria proposed by Hultquist [62]. Let $\mathbf{P}_{i,t}$ be the position of the *i-th* particle at time t and $d(\mathbf{x}, \mathbf{y})$ be the distance between points $\mathbf{x}$ and $\mathbf{y}$. Then, a particle is inserted if

$$d(\mathbf{P}_{i,t}, \mathbf{P}_{i+1,t}) > \alpha\, d(\mathbf{P}_{i,0}, \mathbf{P}_{i+1,0}) \tag{6.1}$$

and

$$d(\mathbf{P}_{i,t}, \mathbf{P}_{i+1,t}) - d(\mathbf{P}_{i,t-1}, \mathbf{P}_{i+1,t-1}) < \beta\, d(\mathbf{P}_{i,t-1}, \mathbf{P}_{i,t})\,, \tag{6.2}$$

where α and β are usually set to 2. The first inequality tests if the current distance is larger than α times the initial distance between two adjacent particles. The second

inequality guarantees that the distance between two neighbors does not grow more than β times faster than the distance between its previous and its current position. The surface tears if Eq. (6.1) is true and Eq. (6.2) is not met. A particle dies if the distance between two neighboring particles is too small, for example, when particles enter a convergent area of the flow. A particle is removed if the following conditions are fulfilled:

$$\left(\frac{\mathbf{P}_{i,t} - \mathbf{P}_{i-1,t}}{|\mathbf{P}_{i,t} - \mathbf{P}_{i-1,t}|}\right) \cdot \left(\frac{\mathbf{P}_{i+1,t} - \mathbf{P}_{i,t}}{|\mathbf{P}_{i+1,t} - \mathbf{P}_{i,t}|}\right) \approx 1 \tag{6.3}$$

and

$$\begin{aligned} & d(\mathbf{P}_{i,t}, \mathbf{P}_{i+1,t}) < \gamma\, d(\mathbf{P}_{i,0}, \mathbf{P}_{i+1,0}) \\ \wedge \quad & d(\mathbf{P}_{i,t}, \mathbf{P}_{i-1,t}) < \gamma\, d(\mathbf{P}_{i,0}, \mathbf{P}_{i-1,0})\,, \end{aligned} \tag{6.4}$$

where γ should be less than 1. The dot product in Eq. (6.3) tests for collinearity of the particle and its neighbors. If this is true, both distances from the particle to its neighbors are checked. Eq. (6.4) defines that a particle needs to be removed if the distances to its neighbors are smaller than the distances at $t = 0$, scaled by γ.

The computation of the different criteria requires data from the local neighborhood of a particle. The temporal neighborhood (i.e., access to previous time step) is intrinsically encoded in the particles texture because a row of that texture corresponds to different time steps of the same particle. The spatial neighborhood is explicitly stored in the states texture, which holds indices to the left and right neighbors.

Particle removal is implemented by marking "dead" particles in the particles texture so that they are not processed any further during particle tracing and surface rendering. By using render targets with floating point precision, no additional color channel is necessary. There exist at least two channels, containing the neighbors which cannot be negative. If the particle dies, one of these channels is used to store this additional information, by negating its actual value. The implementation of particle insertion uses two additional textures that store intermediate results. The first one contains the positions of the new particles, and the other one contains the corresponding states. Both textures have the same height as the original textures holding the particles and their respective states, respectively. Each existing particle is tested with its right neighbor using Eqs. (6.1) and (6.2). If both inequalities are true, a new particle $\mathbf{P}'_{i,t}$ is created by linear interpolation between $\mathbf{P}_{i,t}$ and $\mathbf{P}_{i+1,t}$. Then, the particle position and connectivity are written to the additional textures. The neighbors are assigned to the new particle by using the coordinates of $\mathbf{P}_{i,t}$ as left and $\mathbf{P}_{i+1,t}$ as right neighbors.

The problem is that the intermediate textures may contain only a few particles that were actually inserted. In fact, most of the cells of those textures will contain inactive elements. Therefore, the intermediate textures need to be condensed by removing all inactive particles and putting the active particles in a consecutive order. Such a reordering is rather complicated for a GPU implementation. To solve this issue, the histogram pyramids proposed by Ziegler et al. [207] is adopted and slightly modified. The main idea is to merge the positions of the new particles, which are distributed over the whole texture. Due to the fact that the particles' positions are updated column by column, the merging algorithm is restricted to a 1D domain. In fact, all new particles are stored in one column, in which either a texel is filled with a new particle or is empty.

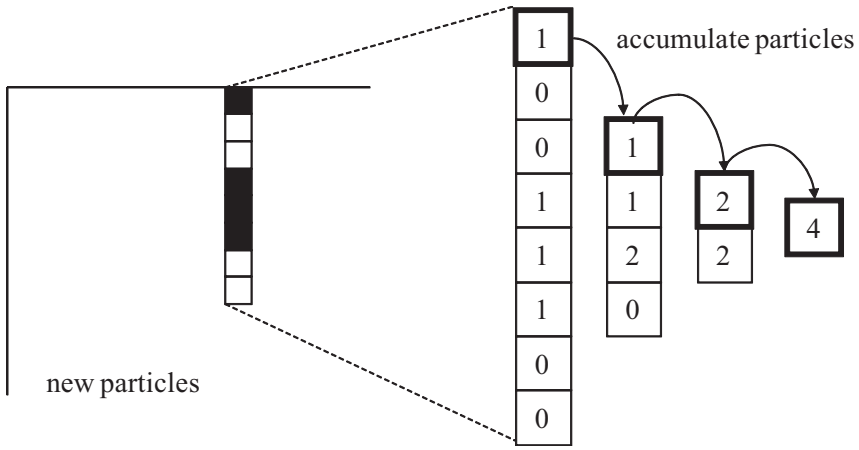


Figure 6.5: Creation of the binary tree: the new particles build the highest level. The contents are summed up until the root contains the overall number of particles to be inserted.

In [150] a binary tree is built over this column by using a pyramid stack of 1D textures, where each level of the pyramid has at least half of the height of the previous level, representing one level of the binary tree. The finest level n represents the new particle itself. In the implementation, a flag is used to notify a texel if it contains a new particle ($\phi = 1$) or not ($\phi = 0$), which serves as basis for the binary tree generation. If now level $n - 1$ of the binary tree is rendered, for example, always two texels of level n are accumulated and stored into one texel of level $n - 1$. This is continued until the root level 0 is reached, which is represented by one texel containing the overall number of new particles (Figure 6.5). The creation of the binary tree requires n render passes to build the tree levels in a bottom-up manner.

In the next step, the new particles are added to the actual particles texture. Here, the number of new particles is read back from graphics memory. Due to the small texture size—one texel—the texture read back does not affect the performance significantly. Then a quadrilateral is created with a height equal to the number of new particles. According to the histogram pyramid method, each texel rendered by the quadrilateral is numbered from 0 to $k-1$, where k stands for the number of new particles. The adapted top-down traversal algorithm by Ziegler et al. [207] works as follows. Starting from the first level of the binary tree, the key value is compared with the entry of the current cell. If the key value is smaller than the cell value, the tree is traversed downward. If the key value is greater or equal, the value of the current cell is stored and the binary tree is traversed downward following the pointer of the successor. This is repeated until the algorithm reaches level n. Finally, the value of the current cell plus the number of predecessors gathered during the traversal is compared to the key value. If the key value is smaller, the new particle corresponding to the key value is found, otherwise the successor cell is used. By rendering the position and the corresponding states at the current fragment, the new particle is inserted containing all the necessary information for the growing of the stream surface. In fact, the left neighbor is the particle $\mathbf{P}_{i,t}$, which has created the new particle, and the right one is the old neighbor of $\mathbf{P}_{i,t}$, which was $\mathbf{P}_{i+1,t}$.

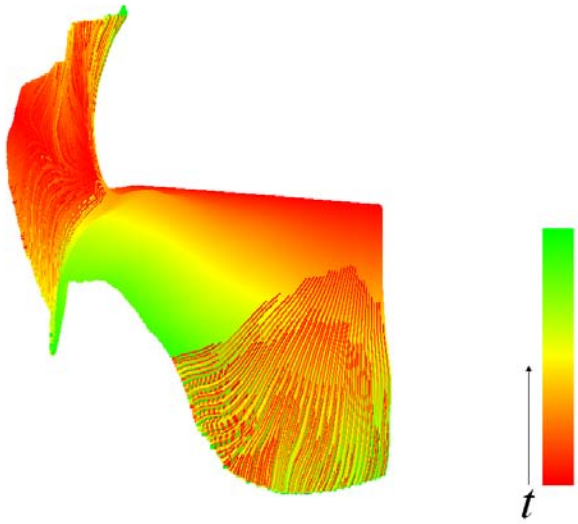


Figure 6.6: Lifetime of the individual particles. The color gradient is defined from red (at t=0) to green and illustrates the increasing lifetime. The areas with red lines at the left and bottom-right parts of the image show regions with many new streamlines.

To restore the consistency of the particle system, the old particles need to be updated as well. When a new particle is created, its index in the particles texture is yet unknown because of the particle merging mechanism. To build the connectivity information, the binary tree used to obtain the relation of the new particles to their old predecessors. Now, the tree is traversed bottom-up, from the leaves to the root. The particle $\mathbf{P}'_{i,t}$ stored in the temporary texture serves as basis of the traversal (note that it represents the leaf level n of the binary tree). While the binary tree is traversed upwards, each cell is tested if it builds the first or the second entry of a tuple. If it has a predecessor, the predecessor's value is accumulated before the algorithm ascends to the next tree level. When the root node is reached, the gathering algorithm stops and the accumulated value represents the number of predecessors. This information and the texture coordinates of the new particles are sufficient for reassigning the new neighbors. Please note that this algorithm has to be executed for all particles which have created a new one, as well as for their former right neighbors, because they also need to be informed about their new left neighbors. After this is done, the particle insertion and removal stage is finished.

The complete stream surface is constructed by applying particle insertion / removal and particle integration several times. Figure 6.6 illustrates the lifetime of the individual particles. Color encodes an increasing lifetime of particles by red to green. New particles are identified as red areas surrounded by green streamlines. The maximum integration length is user-specified.

This algorithm allows us to create stream surfaces and path surfaces alike. The only modification for path surfaces of unsteady flow is that the vector field needs to be updated each time step, which slightly affects performance but leaves the rest of the algorithm unchanged.

For the subsequent LIC calculation, which is discussed in Section 6.1.5, the vector field is needed on the surface. The remaining three channels provided by the two textures are used to store the attached vector of the flow. Regardless of whether steady or unsteady flow is visualized, only the vector used for the integration of the current time step is stored.

6.1.4 Point-based Rendering of the Surface

The fact that particles are added when divergence in the flow is present ensures a sufficiently dense sampling to cover the image space consistently. This way, only enough particles have to be generated and rendered as small point sprites in order to obtain a closed surface. However, in order to obtain lit surfaces, the normal vectors at each position on the surface must be determined. Therefore, first the normal vectors at the particles positions aer estimated, which can be performed by means of covariance analysis [61].

Given a point $\mathbf{q}$ in $\mathbb{R}^3$ and a set of points $S = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$ on the surface (the particle positions), the 3×3 weighted covariance matrix C is given by

$$C(\mathbf{q}) = \sum_{\mathbf{p}_i \in N(\mathbf{q})} (\mathbf{p}_i - \mathbf{q})(\mathbf{p}_i - \mathbf{q})^T \theta(\|\mathbf{p}_i - \mathbf{q}\|) \,, \tag{6.5}$$

where $N(\mathbf{q}) \subsetneq S$ is the set of neighbors of $\mathbf{q}$ and θ is a non-negative monotonically decreasing function. A typical choice is the Gaussian function of the form $\theta(r) = \exp(r^2/h^2)$, where h is a smoothing factor. Once the matrix C is calculated, the normal vector at $\mathbf{q}$ is estimated as the eigenvector of C corresponding to the smallest eigenvalue. Here, the *inverse power* method [130] is used to find the eigenvector.

Given the layout of the texture holding the particles, this processing can be performed in one render pass. For a given particle position $\mathbf{q}$, the set $N(\mathbf{q})$ is defined as the positions $\mathbf{p}_i$ corresponding to the particles of the previous and next time steps in the neighboring streamlines. These particles positions can be accessed using the connectivity information stored in the textures containing the particles and states. Then, the computation of $C(\mathbf{q})$ is straightforward and can be implemented in a single fragment shader, together with the inverse power method to obtain the normal vector at each particle position. This process is performed by rendering a single quadrilateral of the same size as the particles texture. The input to the fragment program are the particles and states textures. The former is used to fetch the particles' positions and the latter to fetch the neighboring particles' texture coordinates. The results of this render pass are stored in the an additional texture holding the normals.

Once the estimated normals are computed, three further render passes are performed. The final result is stored in three textures: (1) a texture with the intersection points on the surface; (2) a texture which holds the projected and lit surface; and (3) a texture with the interpolated vector of the flow at each position on the projected surface.

The process is started, for the first rendering pass, by rendering a quadrilateral for each particle centered at the particle position and perpendicular to the normal vector corresponding to the particle. The quadrilaterals are trimmed in the fragment program by means of clipping operations to obtain discs of radius b, where $0.5h < b < 1$. For this, in the vertex program, the vertex position and the position of the particle in

object-space coordinates (since the normal vectors were also computed in object-space coordinates) are attached to the vertices of the disc. The fragment program writes the position of the fragment (if not clipped) to the texture holding the intersection points.

This texture as well as the texture containing the normals are the input to the second render pass. Discs centered at each particle position are rendered as in the previous pass. The vertex program in this case fetches the normal vector corresponding to the particle and attaches it to the vertex in addition to the positions of the vertex and the particle in object space. Each fragment thus generated writes the normal vector to the RGB channels and $\omega(\|\mathbf{x}_j - \mathbf{q}_i\|, \|\mathbf{x}_j - \mathbf{z}_k\|)$ to the alpha channel of a first target texture used for storing the weighted normals. The $\mathbf{x}_j$ and $\mathbf{q}_i$ are the positions of the fragment and of the particle in object space respectively, and $\mathbf{z}_k$ is the intersection point fetched from the first texture, stored in the texel corresponding to the fragment position in clipping space. The function ω is defined

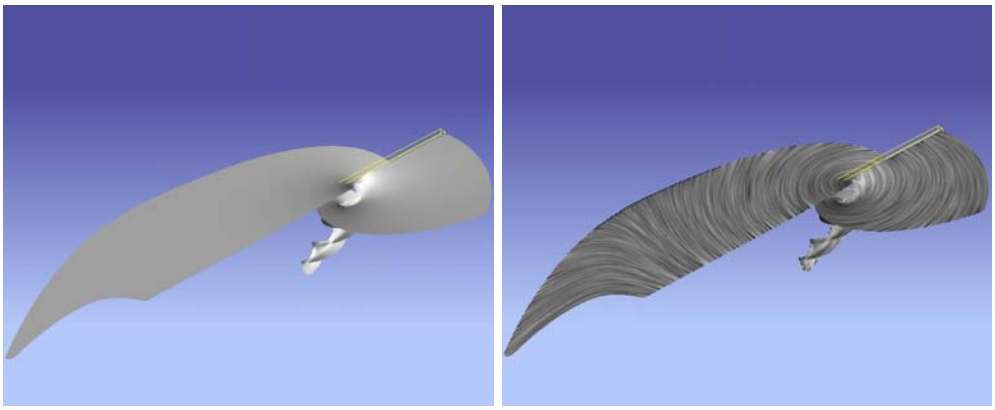
$$\omega(r, s) = \exp\left(-\frac{r^2}{h^2} - \frac{s^2}{\mu^2 h^2}\right) ,$$

where the parameter μ controls the influence of the fragments that are behind the intersection point. This parameter is chosen to avoid the influence in the result of fragments that are not in the 2D neighborhood (surface) of the intersection point. Also, to obtain sharp intersections as in Figure 6.8, it is important to test if the texture coordinates (along the x-axis in the texture) corresponding to $\mathbf{q}_i$ are in the neighborhood of the texture coordinates of the particle corresponding to $\mathbf{z}_k$. This is done to ensure that only particles in the neighboring time steps are considered to calculate the normal and velocity vectors. A second texture, which should contain the weighted vectors, is attached to a second render target, where the fragment program writes, in the RGB channels, the velocity vector at the particle position and, in the alpha channel $\omega(\|\mathbf{x}_j - \mathbf{q}_i\|, \|\mathbf{x}_j - \mathbf{z}_k\|)$. By using alpha blending, for each ray (pixel), the vectors $\sum_{\mathbf{x}_j} \omega(\|\mathbf{x}_j - \mathbf{q}_i\|, \|\mathbf{x}_j - \mathbf{z}_k\|)\mathbf{n}_i$, and $\sum_{\mathbf{x}_j} \omega(\|\mathbf{x}_j - \mathbf{q}_i\|, \|\mathbf{x}_j - \mathbf{z}_k\|)\mathbf{v}_i$ are obtained, where $\mathbf{n}_i$ and $\mathbf{v}_i$ are the normal and velocity vectors at $\mathbf{q}_i$, and the set $\{\mathbf{x}_j\}$ is the set of fragments projected onto the pixel. By normalizing these two vectors, a smoothly interpolated normal and velocity vectors for each projected position on the surface are obtained.

This fact is used in the third render pass, where a single quadrilateral covering the viewport is rendered, and each fragment generated fetches the respective weighted sum of normal and velocity vectors. The normalized interpolated normal vector is used to compute the lit surface, which is written to the second texture. The normalized interpolated velocity vector is written to the third texture. The lit surface can be then displayed, or these two textures, together with the intersections can be input to the process described in the next section, where LIC is added to the lit surface.

6.1.5 LIC on Stream Surfaces

The point-based rendering process of the previous section provides a projection of the stream or path surface onto the image plane, along with information about the 3D position on the surface and the attached normal and velocity vectors. For the LIC computation, the hybrid LIC algorithm by Weiskopf and Ertl [192] is adapted in a way



(a) (b)

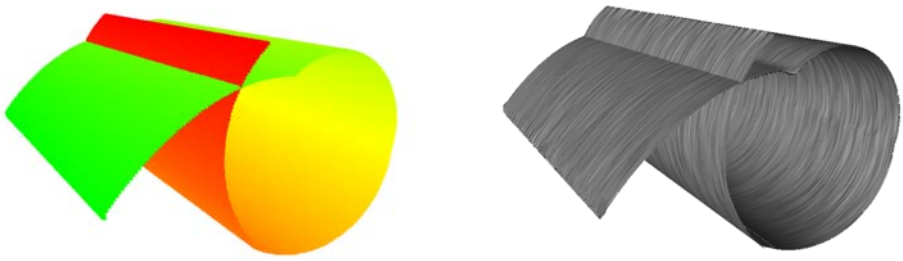
Figure 6.7: Stream surface of a 3D tornado data set: (a) without LIC texture, (b) with LIC texture.

it can be applied to the projected surfaces resulting from point-based rendering. A detailed description of the hybrid LIC method is given in Section 4.3.

The original implementation [192] was designed for older Shader Model 2.0 GPUs and uses multiple render passes to step along particle paths and to discretize the LIC integral. Current GPUs with Shader Model 3.0 support allow for a single-pass LIC implementation using loops. Algorithm 1 shows the pseudo code for this single-pass LIC computation. The two G-buffer textures, the intersections texture (object-space positions) and the vectors texture (object-space vector field), are initialized by the point-based rendering process described in Section 6.1.4. The actual particle tracing is done in 3D object space coordinates in order to achieve higher accuracy than pure image-space advection methods. Before the vector field can be accessed, the current 3D object space position is transformed to image space by applying the model, view, and projection matrices.

Algorithm 1 Shader-inspired pseudo code for surface LIC.

```
// Object-space position at current pixel:
float3 pos_obj = tex2D(intersections, pos_fragment);
float accum = 0;
// Loop along forward streamline direction:
for i=0 to imax
   // Accumulate contribution to LIC integral:
   accum = accum + k_weight(i) * tex3D(noise, pos_obj);
   // Current image-space position:
   float2 pos_img = trafo_to_image_space(pos_obj);
   // Corresponding object-space vector:
   float3 velocity = tex2D(vectors, pos_img);
   // Euler integration:
   pos_obj = pos_obj + delta_t * velocity;
endfor
// Loop along backward streamline direction:
...
```



(a) (b)

Figure 6.8: Path surface of an unsteady flow: (a) shows the time of the unsteady flow field by colors (red for for early times, green for later times); (b) illustrates the combination of the path surface and time-dependent LIC.

LIC improves the visualization on stream or path surfaces because a LIC texture provides additional information that cannot be encoded by a surface alone. Figure 6.7 shows an example: the stream surface is quite wide and without LIC lines (Figure 6.7 (a)) the flow structure within the stream surface is not displayed; in contrast, Figure 6.7 (b) shows the stream surface with LIC, conveying the internal flow structure and the flow direction. Here, the LIC texture is combined with the regularly rendered and illuminated surface (from the texture containing the lit surface) in order to show the flow and the surface shape at the same time.

The above LIC algorithm works for steady and unsteady flow alike. Since the steady case is rather simple, the focus of the following discussion lies on the unsteady scenario, which generally is challenging for texture-based flow visualization. Typically, the texture-based visualization of unsteady flow leads to smeared-out texture patterns. For example, texture advection [71; 177] constructs an overlay of streaklines (or inverse streaklines). Since streaklines may intersect each other, the weighted average of noise input from those streaklines could result in a convolution that is not restricted to a curved line. Therefore, texture patterns could be smeared out in a 2D area. Similarly, the feed forward and value depositing mechanisms of UFLIC [157; 105; 104] can lead to changing widths of line patterns.

The fundamental problem is that there is not a single, unique vector for a single spatial position in a time-dependent flow. In fact, the vector depends on how far in time the integration along a particle trace has progressed. The above texture-based methods mix, at the same position, vector fields of different time. In contrast, this surface LIC method obtains the vector field from path surface construction (Section 6.1.3), which usually yields a single vector for a certain spatial position because that spatial position is linked to a specific time. Figure 6.8 (a) shows a color coding of this time. Still, a path surface could intersect itself, which corresponds to two different times and two different vector values at an intersection point. Figure 6.8 illustrates such a self intersection. Fortunately, those intersection points typically form only a null set (i.e., a 1D line on a 2D surface) and lead to different flow regions in image space that are clearly separated by the intersection lines. As illustrated in Figure 6.8 (b), surface LIC is capable of generating crisp, line-like LIC textures for those different flow regions.

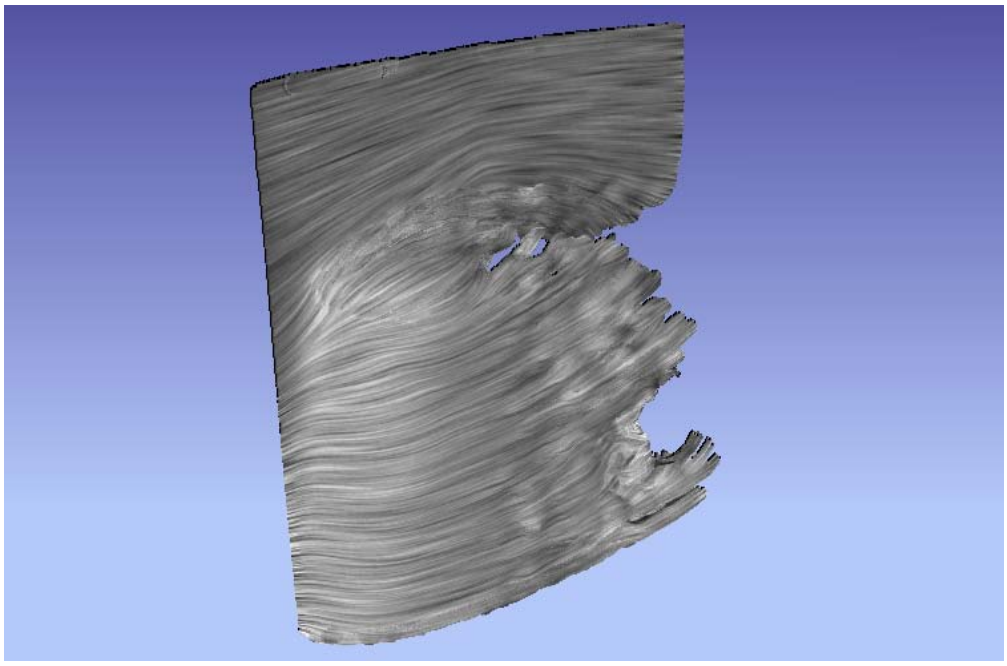


Figure 6.9: Path surface of the unsteady flow around a cylinder.

This approach achieves the essentially one-to-one correspondence between spatial positions and vectors by fixing the initial time for seeding a path surface. In other words, the advantage of clear line patterns comes at the cost of being limited to a temporally and spatially restricted region of path-surface seeding. Therefore, a real-time construction and rendering of LIC path surfaces is required in order to give the user the opportunity to interactively change the seeding parameters.

6.1.6 Performance Measurements

In order to demonstrate the interactivity of the proposed stream and path surface method, the performance of generation and rendering of the stream surfaces are measured. The method has been tested on a PC with the following configuration: a AMD Athlon 64 X2 Dual 4400+ (2.21 GHz) CPU and 2 GB of RAM. Two different GPUs have been used: an NVIDIA GeForce 8800 GTX GPU with 768 MB of graphics memory and an NVIDIA GeForce 7900 GTX GPU with 512 MB of graphics memory. The complete algorithm is implemented in DirectX 9.0 using HLSL and Shader Model 3.0. For the performance test, an unsteady data set was used, simulating the flow around a cylinder with 17 time steps. Figure 6.9 shows a visualization of the test data set. For the steady measurements, only the first time step is used. The vector field is given on a uniform grid of size $256 \times 128 \times 256$. Table 6.1 shows the results of a measurement with 256 particles that are integrated along 256 time steps. In the unsteady case, the vector field is updated each $256/17 \approx 15$ time steps. The performance for rendering the plain surface mainly depends on the size of the projected surface, which can be explained by the fragment-based surface approximation. Applying the surface LIC reduces the rendering speed by a factor of 2.5. This can be ascribed to the additional

Computation	Steady	Unsteady
Surface only [1]	63.9	*
Surface with LIC [1]	26.4	*
Integration only [2]	6.5	5.5

* The rendering speed does not differ from the steady case.
[1] Measured with an NVIDIA 8800 GTX GPU.
[2] Measured with an NVIDIA GeForce 7900 GTX GPU.

Table 6.1: Performance using the $256\times128\times256$ unsteady data set of Fig. 6.9 with 17 time steps (in fps).

render passes and the number of integrations used for solving Eq. (4.6). Since the current driver of the new NVIDIA GeForce 8800 does not provide the full power of the architecture, particulary when rendering to texture is applied, the GPU has been replaced with its predecessor, the NVIDIA GeForce 7900 GTX for measurements that need render-to-texture functionality. With that GPU, the measured frame rates are 6.5 and 5.5 fps, respectively, for integrating particles in a steady and unsteady flow, respectively. Further experiments showed that the performance of particle tracing strongly depends on the size of the vector field and the number of time steps, which corresponds to the number of texture uploads.

6.2 Visualizing Multi-Modality Data using Surface LIC

So far, particle tracing has been used to visualize the behavior of flow fields. In Section 4.2 and 5.3, texture advection has been applied on 3D unsteady fluid vector fields while in Section 6.1 it has been shown that texture based techniques can also be used to visualize the flow on a surface. The aim of the following discussion is to derestrict the surface flow visualization techniques of Section 4.3 from its explicit use on surfaces embedded in a fluid field to a more general application: for improving the perception of surface shapes.

The visual perception of a surface highly depends on the illumination and material parameters of the surface. However, the higher the transparency of an object is, the lower is the improvement illumination brings with it. This is particulary the case if more than one surface is drawn, e.g., when nested isosurfaces should be visualized. Here, it is desirable enable a good perception of the outer isosurface's shape without losing a precise visualization of the isosurface in the interior.

This scenario of rendering nested transparent objects is discussed by means of an application for medical visualization according to the paper by Schafhitzel et al. [149], where surface LIC is used for improving the visual perception of highly transparent isosurfaces. Although the main idea of drawing LIC on transparent surfaces has been already investigated by Interrante [64], the following method is fully implemented on the GPU and, therefore, supports frame-to-frame coherence as well as the interactive manipulation of the LIC parameters.

This section contains text components of the original paper [149], that was written in collaboration with my colleague Friedemann Rößler. He must be credited for the

framework providing direct volume rendering for the visualization of functional and anatomical data.

6.2.1 Combining Functional and Anatomical Data

Modern medical imaging provides a variety of techniques for the acquisition of multi-modality data. A typical example is the combination of anatomical MRI and functional brain images obtained from different modalities, to show functional processes in the spatial context of the involved brain regions. A common neuro imaging technique is functional Magnetic Resonance Imaging (fMRI), of which the major goal is to make visible the activities of the brain. This method is on the one hand used in neurology and neurosurgery for diagnosis and treatment planning, on the other hand it is a powerful tool for cognitive neuroscientists to study brain activity. fMRI takes advantage of the fact that cognitive processes lead to a local increase in oxygen delivery to activated cerebral tissue, resulting in changes of the local magnetic field. This effect, referred to as Blood-Oxygen-Level Dependent (BOLD), can be acquired with a special fMRI sequence and provides an indirect measure for the intensity of brain activation.

Usually, the data resulting from fMRI and MRI is transformed to 3D scalar-field representations to facilitate visualization. There already exist several techniques for the combined visualization of fMRI and MRI data. One example is the projection of the brain activation onto an anatomical isosurface, where the activation level is represented as color [167]. However, the projection onto the surface impairs the perception of relative depth and spatial structure. Other approaches combine several scalar visualization techniques, like direct volume rendering (DVR) and surface shaded display (SSD), i.e., an isosurface representation, in a single visualization. Grim et al. [51] developed methods to efficiently visualize multiple intersecting volumetric objects by introducing the concept of V-Objects. These abstract properties of an object are connected to a volumetric data source, respectively. Similar features are supported by other commercial [160] and non-commercial software [136; 162]. An approach for modeling and rendering complex multi-volume scenes was proposed by Leu and Chen [98; 99]. Cai and Sakas [21] proposed a method for data intermixing in direct multi-volume rendering. A survey of GPU-based volume rendering techniques was given by Preim et al. [129]. Another GPU-based multi volume approach was proposed by Rößler et al. [139] for the simultaneous rendering of functional and anatomical data. A more recent work is the dynamic shader approach [137; 138] for rendering an arbitrary number of volumes using individual automatic generated shaders.

However, partial occlusion and visual clutter that typically result from the overlay of these traditional 3D scalar-field visualization techniques make it difficult for the user to perceive and recognize visual structures. The following discussion addresses these perceptual issues by a new visualization approach for anatomical/functional multi-modalities. The idea is to reduce the occlusion effects of an isosurface by replacing its surface representation by a sparser line representation. Those lines are chosen along the principal curvature directions of the isosurface and rendered by LIC. Applying the LIC algorithm results in fine line structures that improve the perception of the isosurface's shape in a way that it is possible to render it with small opacity values. An interactive visualization is achieved by executing the algorithm completely on the GPU. Furthermore, several illumination techniques and image compositing strategies

are discussed for emphasizing the isosurface structure. The method is demonstrated for the example of fMRI/MRI measurements, visualizing the spatial relationship between brain activation and brain tissue.

6.2.2 Multi-Volume Rendering

The following visualization method is based on a multi-volume rendering approach presented by Rößler et al. [139] for the simultaneous visualization of functional and anatomical data whilst preserving their spatial relationship. This existing generic visualization framework, which facilitates the intermixing of any number of volumes and the combination of arbitrary rendering styles has been extended for this work. In its original version, the framework implements slice-based volume rendering. The data is stored in 3D textures, which are sampled by view-dependent coplanar slices (3D texture slicing) [39].

As the goal of this work consists of a focus-and-context visualization, two different visualization techniques are chosen for the volumes to be mixed. The focus is the functional data, which is represented by a brain activation data set, whereas the anatomical data builds the context. In the following, it is assumed that the context always encloses the focal region. Based on this fact, direct volume rendering is applied for rendering the brain activation and implicit isosurface rendering for the surrounding brain structure. The isosurface itself is also rendered by slice-based volume rendering, using a threshold for the isovalue. For each slice of the anatomical data set, the threshold is tested against the current density value to decide if a fragment is rendered or not. Since the brain activation is always located behind the isosurface, first the functional data and then overlay the anatomical data is rendered. This means that both objects are rendered completely independently from each other. Please note that due to this geometrical independency, different transfer functions and different shaders are attached to the objects.

Curvature Lines and Multi-Volume Rendering

Considering the functional data, the implementation of Rößler et al. [139] is used, who rendered the brain activation completely opaque. This result is stored in a render target (i.e. an image-aligned 2D texture). In the next step the isosurface is drawn. In order to apply the surface LIC algorithm of Section 4.3, the vector field representing the first principal directions is projected onto the isosurface.

The pre-computation of the curvature vector field follows for the most part the method in Section 2.2. In detail, the first principal direction is determined for each grid cell separately using the gradients of the anatomical data set. The gradients are numerically computed using central differences (see Section 2.1) or Sobel filtering. In order to achieve the desired vector field, only the eigenvector $\mathbf{x} = (u_1, v_1)$ is computed according to Eq. (2.9), where its length is defined by the first principal curvature κ_1 with $\kappa_1 > \kappa_2$. Keep in mind that this implementation assumes a linear approximation of the isosurface for each grid cell, like derived from the marching cubes algorithm.

Since the vector field does not change over time, it is pre-computed and stored in a 3D texture. According to the surface LIC algorithm [192], this texture is fetched when rendering the isosurface in order to map the vectors on the surface. The illuminated

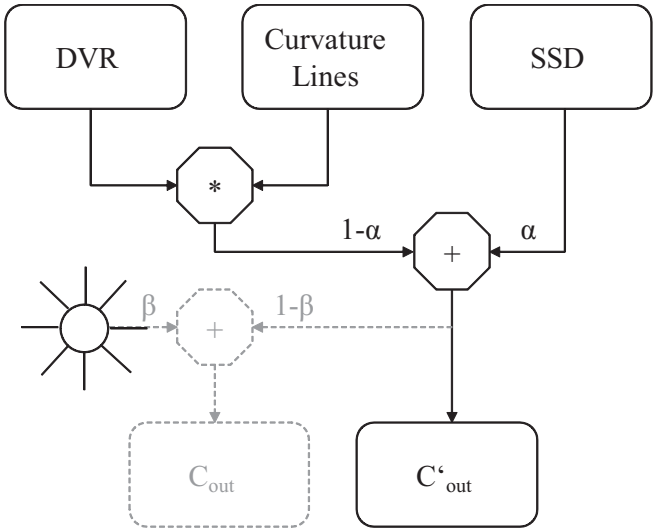


Figure 6.10: Blending of the functional data rendered by direct volume rendering (DVR) and the surrounding anatomical data using an implicit isosurface representation (SSD). The curvature lines resulting from the LIC computation affect the DVR rendering by defining the intensity of the functional data as well as the SSD of the anatomical data due to their tangential behavior. If illumination is enabled, a second blending is applied which adds the light contribution to the emissive representation.

isosurface and the positions given in object space are stored in two additional render targets. The second stage is implemented as an additional render pass. Here, a screen-filling quad is rendered to address each pixel of the viewport. In this stage, the LIC integral, Eq. (4.1), is evaluated by accessing the projected vector field and the appropriate positions in object space. This results in an intensity value, which is computed for each fragment.

The last step is the final rendering. Here, a special blending function is applied, taking into account the lines obtained from the LIC computation, the illuminated isosurface, and the already rendered brain activation. The blending function is defined as

$$\mathbf{C}'_{\text{out}} = \alpha \mathbf{C}_{\text{iso}} + (1-\alpha)(1-\mathrm{I}_{\text{LIC}})\mathbf{C}_{\text{act}} \,, \tag{6.6}$$

and describes the final output color $\mathbf{C}'_{\text{out}}$. The alpha blending between the functional data, $\mathbf{C}_{\text{act}}$, and the anatomical data, $\mathbf{C}_{\text{iso}}$, is governed by the opacity α of the isosurface, i.e. the opacity of the isosurface determines the visibility of the brain activation behind. Figure 6.10 illustrates this blending process.

The intensity of the curvature lines, I_{LIC}, further modifies the image compositing. According to the multiplication by the factor $(1-\mathrm{I}_{\text{LIC}})$, the LIC intensity provides different weights for the brain activation. As dark lines should be drawn, it is necessary to negate the intensity. Furthermore, the empty areas inside the anatomical hull have to be considered. Black empty regions would lead to a multiplication by zero, which means that in these areas no curvature lines would be visible. In order to apply the curvature lines also in these regions, the brain activations are rendered using a white

background. This does not imply any color shift of the activation volume, since activity is rendered opaque anyway.

Curve Illumination

When curve illumination is enabled, the data flow changes slightly; see Fig. 6.10. Lighting is based on normal vectors on the surface geometry, which can be related to the gradients of a texture-based representation of geometry. Since the curvature lines are computed on the image plane in a view-dependent way, a pre-computation of the gradients is not possible. Therefore, a real-time gradient computation for the curvature lines is employed. Furthermore, it must be considered that the first derivative requires neighborhood information. In [149], the gradient computation is implemented as an additional render pass directly after the LIC evaluation, which delivers the intensity values for each pixel in image space. This image can be considered as a 2D scalar field that serves as input for the gradient computation. Central differences are used to compute the 2D image-space gradient, which is combined with the surface normal to obtain the 3D normal vector in world space. In the final rendering step, this vector field is used to apply diffuse illumination of the curvature lines. This illumination component extends Eq. (6.6) to

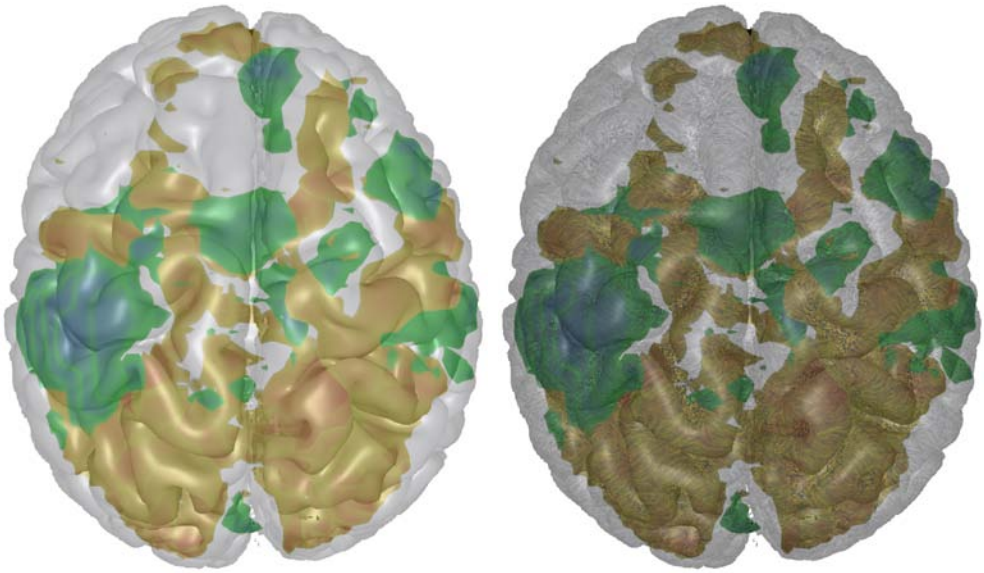
$$\mathbf{C}_{\text{out}} = \beta(\mathbf{N} \cdot \mathbf{L}) + (1 - \beta)\mathbf{C}'_{\text{out}} , \tag{6.7}$$

where $\mathbf{N}$ stands for the normal vector and $\mathbf{L}$ denotes the position of the light source. Both vectors are given in world space. Usually, only a small light contribution is sufficient for emphasizing the line structures on the isosurface. From experience, β should be chosen between 0.1 and 0.2.

6.2.3 Data Acquisition

The data used for this application was gathered from cognitive studies conducted at the Center of Cognitive Science at the University of Freiburg. In those studies, the cognitive scientists want to learn more about the links between human thoughts, feelings, and actions, and the functions of the brain. For this, a number of participants are placed one after the other in an MRI scanner; while they are performing cognitive tasks, the brain activation is measured in quickly repeated intervals. The goal of such fMRI experiments is to explore differences between these measurements. Typically, the baseline activity is measured when the subject is at rest, and other measurements are taken when the participant performs certain tasks. In the simplest experimental design, the activity in the baseline condition is subtracted from the activity measured during the performance of the cognitive tasks. Since the raw fMRI measurements contain much noise, they are not examined directly but statistically analyzed to obtain significant activations. This statistical analysis is usually done with SPM [164] (Statistical Parametric Mapping), a freely available software developed by the Wellcome Department of Imaging Neuroscience, University College London. It facilitates the analysis of whole sequences of brain imaging data that can either be a series of images from different people or a time series from the same subject.

In an example study [43], the participants performed logical reasoning problems while the brain activity was measured. During the experiment, the participants were



(a) (b)

Figure 6.11: Combined visualization of functional and anatomical data. The brain activation is rendered with DVR while the isosurface is represented by SSD: (a) Isosurface without any line structures mapped onto it. (b) A high number of curvature lines without any masking; the short lines appear as points and influence the visualization negatively. That makes the visualization appear rather dark.

asked to draw conclusions from given premises and later their responses were evaluated for logical validity. For instance, they saw two premises:

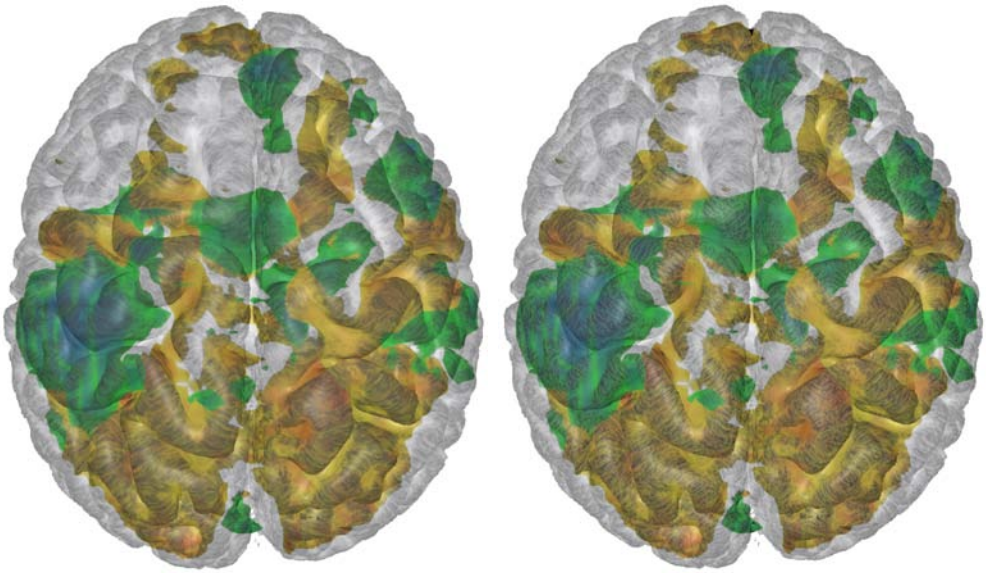
Premise 1 :	*V X*	*(V is on the left of X) .*
Premise 2 :	*X Z*	*(X is on the left of Z) .*

Afterwards, they had to decide and indicate by a key press whether the following statement logically followed from the premises:

Conclusion:	*V Z*	*(V is on the left of Z)?*

6.2.4 Results

In this section, the curvature-based multi-volume visualization is applied to the acquired fMRI data. Three different visualizations using different parameter settings are compared to the visualization achieved with the original render method by Rößler et al. [139]. In detail, the parameters are: curvature masking, noise density, and curve illumination. Figure 6.11 (a) shows an example of the combination of SSD and DVR. In this image, the volume-rendered brain activation serves as focus while the surrounding brain tissue is represented by an isosurface, which builds the corresponding context. Obviously, the main goal of this visualization is to facilitate the spatial perception of the activation areas inside the human brain. Therefore, it is necessary to have a clear visualization of both objects. In particular, the shape of both objects should be perceivable at a glance. Actually, the visualization quality depends on the materials of



(a) (b)

Figure 6.12: Improved visualization quality by manipulating the different parameters: (a) Smaller number of thicker curvature lines; in the areas of low curvature, the curvature lines are faded out completely. (b) Illuminated curvature lines. Only a small diffuse light contribution is used for emphasizing the lines' structure. The curvature lines appear more prominently without having changed their intensity.

the surrounding context and the focus object. These materials are usually defined by transfer functions, and so it depends on the user to find an appropriate setting for an optimal visualization. Therefore, the visualization quality is strongly influenced by the rendering of the context, in this case the isosurface. For example, if the isosurface is rendered opaque, the structure of the anatomy is rendered in high quality, but it completely occludes the brain activation. On the other hand, if the isosurface is chosen too transparent, the shape of the covered brain activation is clearly identifiable, whereas the quality of the isosurface suffers.

In Fig. 6.11 (b), a naive mapping of the computed curvature lines onto a selected isosurface is applied. Indeed, the structure of the isosurface is clearly perceptible, supported by the curvature lines. Nevertheless, the high number of lines drawn on the surface makes it hard to identify the shape of the brain activation behind. The visualization appears darker than the original in Fig. 6.11 (a). Furthermore, areas of low curvature, like the depressions on the cortex surface (sulcus), lead to short lines, which negatively influence the visualization. In Fig. 6.12 (a), the issue of occlusion is addressed by changing the noise intensity that influences the number and the size of the drawn LIC lines. By decreasing the number of lines, the isosurface becomes more transparent and the quality of the brain activation shape increases. The problem of noise in flat areas is solved by curvature masking, which is based on a threshold that masks lines in areas of low curvature. As a consequence of this masking, the lines in the sulcus disappear while the shape of ridges (gyrus) are emphasized by line drawing.

Figure 6.12 (b) shows the result if the density of the lines on the isosurface is fur-

Render Method	Performance in fps
Isosurface w/o LIC	42.17
Isosurface with LIC	25.67
Isosurface with illuminated LIC	25.57

Table 6.2: Performance measurements of a combined visualization of DVR and SSD. The brain activations are visualized by direct volume rendering, whereas the surrounding brain is rendered as an isosurface. Here, three different configurations for isosurface rendering are compared. All measurements are given in frames per seconds (fps), the viewport size is 800×600.

ther decreased. It is necessary to consider the disadvantages which might appear if the density is chosen too small: due to the behavior of the adapted surface LIC algorithm (see Section 6.2.2), the resulting lines are a weighted intensity of the brain activation. Actually, the weights of the streamlines are chosen with a positive offset to avoid black lines, which might affect the visualization. If the number of lines is decreased too much, either the intensity offset must be reduced or the perception of the line structures has to be improved. Indeed, the first option would lead to a higher contrast, but it would also imply a higher degree of occlusion caused by opaque lines. An alternative option is to improve the perception of line structures by illumination. A real-time gradient computation is applied as basis for lighting. The gradients determine the normal vectors for diffuse illumination (Lambert reflection). Please note that already a small contribution from illumination is sufficient for an improved perception of the lines. If the diffuse part is emphasized too much, the lines appear as bumps on the isosurface, which makes it difficult to distinguish between the original curvature given by the isosurface and the visual ridges created by illumination.

Table 6.2 shows the visualization speed of several configurations measured on a PC with the standard configuration described in Section 1.5. Considering the first two rows of the table, the lower frame rates for LIC rendering can be ascribed to the evaluation of the LIC integral. In this case, 20 LIC steps are computed in each direction, which results in 40 texture lookups for each pixel. In contrast to the LIC evaluation, the gradient computation barely influences the rendering speed. Gradients are computed by central differences. Therefore, only 4 additional texture lookups per texel are necessary, which makes it much faster than the LIC evaluation.

6.3 Line Integral Convolution on 3D Vortex Structures

In the previous two sections, two examples for the application of surface LIC were given. Although both scenarios consider completely different problems, since in Section 6.1 stream lines and path lines of a fluid flow are drawn while Section 6.2 discusses the emphasizing of surface shapes, it becomes clear that the surface LIC method [192] can be applied to any arbitrary surface that is related to an appropriate vector field.

This section returns to the usage of the surface LIC algorithm in order to visualize the motion of particles inside a flow. However, the goal of the visualization differs from

Section 6.1. Here, the behavior of fluid particles nearby vortex structures should be visualized. The explanation of the problem is rather simple: considering a steady flow field, e.g., one time step of a simulation series. Then one opportunity to investigate the behavior of the flow field is to compute its flow features. An important kind of flow features consists of vortices, since they can be directly computed from the vector field. A more detailed description on the computation of flow features is given in Chapter 7. In the following discussion, we assume the vortex regions as already computed and stored as isosurfaces, i.e., as a number of triangles. In detail, by drawing the extracted vortex geometry, the boundaries of vortex regions can be displayed. However, this way of rendering vortex structures withholds information we already have–the direction of rotation of the vortices. This information is namely given by the vector field, which is already used for extracting the vortex structures.

The idea is to extend the visualization of vortex structures by animated surface LIC. Projecting the vector field onto vortex surfaces exploits the fact that the particles' motion is almost tangential to the vortex surface. Animating those particles enables a clear perception of the particles' motion on the vortex surfaces and, therefore, represents their direction of rotation. Perception is further improved by applying two types of illumination on the surface LIC, Phong illumination and cool warm shading. The following discussion bases on the work of Schafhitzel et al. [153] and contains parts of the original text.

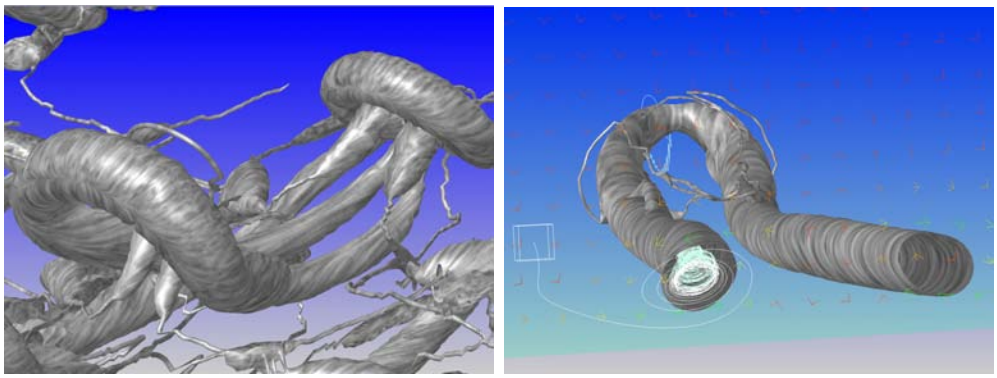
6.3.1 Vortex Visualization Process

The visualization process bases on the vortex core line detection and segmentation method of Stegmaier et al. [165]. This method delivers the vortex geometry that is used for rendering the vortex boundaries. A theoretical view on the predictor-corrector algorithm used for the vortex segmentation is given in Section 7.3.2. According to the surface LIC method by Weiskopf and Ertl [192], the positions $\mathbf{r}_D$ as well as the vector field $\boldsymbol{\nu}_P$ are created by rendering the vortex structures (see Eq. (4.8)). Since the motion of a particle nearby a vortex structure is supposed to be almost tangential to the λ_2 isosurface (see Section 7.2.2), the 3D vector field is projected onto the geometry to obtain long line-like patterns. For this purpose, the LIC integral 4.1 is evaluated in the last stage of the visualization process.

Figure 6.13 shows the results achieved with the surface LIC on vortex surfaces. In Fig. 6.13 (a) it can be observed that the stream lines rotate less than in Fig. 6.13 (b). This is due to the fast motion of the vortices in Fig. 6.13 (a), i.e., the velocity that transports the vortices is much higher than the angular velocity. Figure 6.13 (b) shows that the stream lines achieved by the surface LIC are almost identical to a 3D stream line.

6.3.2 Animated LIC

Although the standstill images of Fig. 6.13 give feedback about the trajectories of the particles surrounding a vortex structure, no information about the direction of the particles' motion is obtained so far. Considering the LIC integral of Eq. (4.1), the filter kernel $k(t)$ can be manipulated in a manner the LIC appears animated. Usually, $k(t)$ is defined to have its maximum when $t = \tau_0$, i.e., when the position of the current



(a) (b)

Figure 6.13: Surface LIC on λ_2 isosurfaces. First, the 3D vector field is projected on the isosurface, then the LIC integral is evaluated. Applying the surface LIC results in line-like pattern emphasizing the trajectories of the particles in a vortex region.

pixel is considered. Here, it doesn't matter if either a box, a tent or a Gaussian filter function is used.

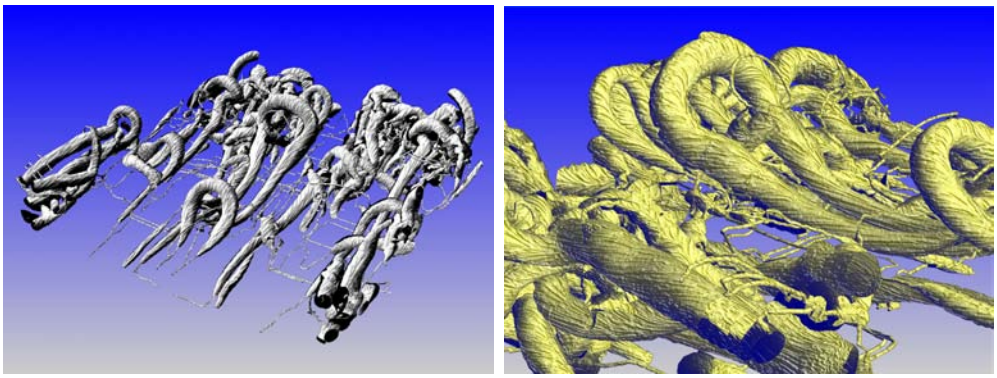
According to Cabral and Leedom [19], animation can be achieved by moving the maximum in positive direction, i.e., in direction of the parameter τ. This leads the weights to be newly distributed along the LIC integral, and so a position $\phi_0(\tau - \tau_0)$ is weighted stronger than the initial position $\phi_0(0)$. This behavior can be considered like evaluating the LIC integral for the next position on the stream line, considering $\phi_0(\tau - \tau_0)$ as starting point. If the maximum reaches the boundary of the filter kernel L, it is moved to the beginning at $-$L and the kernel shifting is repeated. This methods let the surface LIC behave like drawing animated particles moving along their trajectories.

In the case of surface LIC on vortex surfaces the direction of rotation of a vortex can be observed at a glance. Furthermore, if the vector field is not normalized, the different velocities on the vortex surface are perceptible. According to the notation of Wiebel et al. [200], a flow field can be separated into a Laplacian field and a potential field. By animating the surface LIC, although the original field is used, the user is able to distinguish between the pure rotation (potential flow) and the motion of the vortex structure (Laplacian field).

6.3.3 Particle Lighting

Another way to improve the perception of the LIC-rendered stream lines is to apply illumination. The importance of illumination has been already discussed in Section 5.2, where an on-the-fly gradient computation has been developed in order to apply several illumination models on 3D texture advection. The same idea has been adopted for lighting the surface LIC result. But in contrast to 3D texture advection [195; 196], the gradient computation only takes place in a 2D domain, what makes it much faster than its 3D counterpart.

The basic idea of this method is to calculate the gradients of the particles each timestep. This causes additional computational overhead. Anyhow, this can be reduced considering the fact, that the particles are traced on a 2D domain. The method mainly



(a) (b)

Figure 6.14: On-the-fly illuminated particles on vortex surfaces. The gradients of the stream lines resulting from the surface LIC algorithm are computed in image-space. Therefore, the resolution of the normals is directly linked to the resolution of the LIC image: (a) Blinn-Phong illumination with a low specular contribution; (b) Cool-warm shading.

consists of two steps. First, the normal of the geometry is determined, which serves as fix component perpendicular to the surface. Regarding this component, the following gradient calculation can be fully considered in 2D. Furthermore, this step creates only minimal overhead, because the normals of the geometry can also be computed during the geometry generation.

In the second step, the visualization process is extended. Therefore, an additional processing step is inserted directly after the LIC calculation. This is be done by calculating the central differences of the current point. The resulting gradient is given in D-space (see Section 4.3), and points to the direction of the strongest change. This calculation is applied for each single vortex, by rendering one screen filling quad. To avoid artefacts at the border regions of the vortices, the background is selected black. Thus, the gradient depends only on the visible neighbors, while the surrounding area is unaccounted. To obtain the normal $\mathbf{n}$, the surface gradient $\mathbf{n}_\mathrm{s}$ and the particle gradient $\mathbf{n}_\mathrm{p}$ only need to be added. Since the two gradients are given in different domains, $\mathbf{n}_\mathrm{p}$ needs to be transformed back to P-space by multiplying it with the transposed MVP (model · view · projection) matrix M_{mvp} without considering translation:

$$\mathbf{n} = \mathbf{n}_\mathrm{s} + M_{\mathrm{mvp}}^{\mathsf{T}} \cdot \mathbf{n}_\mathrm{p} \tag{6.8}$$

Finally, the illumination is computed in a separate shader program. Two different illumination techniques have been implemented: the Blinn-Phong model [124; 12] shown in Figure 6.14 (a) and Cool-Warm Shading according to [49] shown in Figure 6.14 (b).

Part III

Particle Tracing for Feature Detection in Fluid Flows

CHAPTER

7 METHODS FOR COMPUTING FLOW FEATURES

This chapter introduces the basic methods for computing flow features. Starting with a general view on the decomposition of the velocity gradient tensor, some methods for the detection of vortex regions are discussed. This is followed by the definition and extraction of vortex core lines, which are considered as 1D extremum lines of the vortex criteria. Beside vortices, also shear layers criteria are discussed as well as critical points, which play a fundamental role in some of the algorithms proposed in Chapter 8.

7.1 Decomposing the Velocity Gradient Tensor

The decomposition of the velocity gradient tensor $\nabla \boldsymbol{v}$ builds the base of the following feature detection methods. The idea is to decompose the partial derivatives of a vector field into its rotational (antisymmetric) and its shear strain (symmetric) parts. While the rotational part forms the rotation tensor Ω, the strain rate tensor is denoted by S. Both are derived from the Jacobian $J = \nabla \boldsymbol{v}$ by

$$\Omega = \frac{1}{2}(J - J^T) = \frac{1}{2}(v_{i,j} - v_{j,i}) \,, \tag{7.1}$$

and

$$S = \frac{1}{2}(J + J^T) = \frac{1}{2}(v_{i,j} + v_{j,i}) \,, \tag{7.2}$$

where the subscript comma denotes partial derivative. Since Ω only contains the components of the vorticity vector with

$$\Omega = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \quad \text{with} \quad \boldsymbol{\omega} = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \nabla \times \boldsymbol{v},$$

it is commonly known as the vorticity tensor.

Keep in mind that although S denotes the strain rate, the normal stress caused by pressure and compression as well as the viscosity are not considered so far. As already shown in Section 3.2, the strain rate tensor of a Newtonian fluid is defined by

$$T = -\left(p + \frac{2}{3}\nu \nabla \cdot \boldsymbol{v}\right) I + 2\nu S \,. \tag{7.3}$$

However, in the following only S is used, because the pressure p and the viscosity ν can be neglected if only a quantitative consideration of shear stress and rotation is desired.

7.2 Vortex Core Detection

A vortex is defined as a multitude of material particles rotating around a common center. That's the definition according to Lugt in 1979. Another, more formal definition says that a vortex is a region of complex eigenvalues of the velocity gradient tensor $\nabla \boldsymbol{v}$ (Perry and Cantwell, 1990). Hunt, Wray, and Moin also considered $\nabla \boldsymbol{v}$ by requiring it to have both a positive second invariant Q and low pressure inside vortex regions.

Actually, there exists no general definition of a vortex core, i.e., the spatial region of a vortex. Although most definitions base on the assumption of low pressure inside vortices, in the work of Jeong and Hussain [67] it has been shown that pressure also tends to have a minimum outside vortex regions, as a consequence of unsteady strain rate. On the other side, it is possible to find vortex regions without any pressure minima because it has been eliminated by viscous effects.

In the following, only two vortex core detection methods are considered in more detail, since this thesis is restricted to the use of the λ_2 and Q criteria. However, a short overview over other more or less common vortex identification criteria is given in Section 7.2.3.

7.2.1 Q Criterion

In 1988, Hunt et al. [63] defined a vortex core in dependance of the second invariant Q of the $\nabla \boldsymbol{v}$. According to Section 7.1, $\nabla \boldsymbol{v}$ can be decomposed in its antisymmetric part Ω and its symmetric part S. While Ω represents the rotational part of the motion, S contains the strain rate, i.e., the stretching and shearing of a particle.

The idea of Hunt et al. was to represent the local balance between shear strain rate and vorticity magnitude. Considering the second invariant of $\nabla \boldsymbol{v}$

$$\mathrm{Q} = \frac{1}{2}(v_{\mathrm{i,i}}^2 - v_{\mathrm{i,j}} v_{\mathrm{j,i}}) = -\frac{1}{2} v_{\mathrm{i,j}} v_{\mathrm{j,i}} \,, \tag{7.4}$$

it can be seen that if the Frobenius norm $||\mathrm{M}|| = \sqrt{\mathrm{tr}(\mathrm{M}\mathrm{M}^{\mathrm{T}})}$ is applied to Ω and S, Eq. (7.4) becomes

$$\mathrm{Q} = \frac{1}{2}(||\Omega||^2 - ||\mathrm{S}||^2) \,. \tag{7.5}$$

In other words, $\mathrm{Q} > 0$ if $||\mathrm{Q}|| > ||\mathrm{S}||$, i.e., vorticity dominates strain rate, else $\mathrm{Q} < 0$. However, although the Poisson equation for pressure

$$\nabla^2 \mathrm{p} = -\rho v_{\mathrm{i,j}} v_{\mathrm{j,i}} = 2\rho \mathrm{Q}$$

states the relation between pressure and Q, Jeong and Hussain [67] showed that $\mathrm{Q} > 0$ does not necessarily imply a pressure maximum, i.e., a pressure minimum can occur at the boundary of the $\mathrm{Q} > 0$ region.

7.2.2 λ_2 Criterion

The λ_2-criterion was proposed by Jeong and Hussain [67] in 1995. Again, the idea was to find regions of low pressure, but instead of searching for local pressure minima, a new definition was introduced. This new definition had the goal to eliminate unmeant effects like unsteady strain rate and viscous effects. Starting with the gradient of the

Navier-Stokes equations (see Section 3.2) and its decomposition into a symmetric and an antisymmetric part:

$$a_{i,j} = -\frac{1}{\rho}p_{,ij} + \nu v_{i,jkk} , \tag{7.6}$$

$$a_{i,j} = \left[\frac{DS_{ij}}{Dt} + \Omega_{ik}\Omega_{kj} + S_{ik}S_{kj}\right] + \left[\frac{D\Omega_{ij}}{Dt} + \Omega_{ik}S_{kj} + S_{ik}\Omega_{kj}\right] , \tag{7.7}$$

where the second, antisymmetric part is the vorticity transport equation, and where the Einstein sum convention is applied. Here, $a_{i,j}$ denotes the acceleration gradient, ρ the fluid density, ν the viscosity, and $p_{,ij}$ the Hessian of pressure. Inserting Eq. (7.7) in Eq. (7.6) and just considering the symmetric part, i.e., $v_{i,jkk}$ is replaced by $S_{ij,kk}$, the Hessian of pressure is defined by

$$\frac{DS_{ij}}{Dt} - \nu S_{ij,kk} + \Omega_{ik}\Omega_{kj} + S_{ik}S_{kj} = -\frac{1}{\rho}p_{,ij} . \tag{7.8}$$

As already mentioned above, the idea is to remove the parts of Eq. (7.8) that lead to unmeant pressure changes. This is on the one hand the unsteady irrotational straining, denoted by DS_{ij}/Dt, and on the other hand the viscous effects, denoted by $\nu S_{ij,kk}$. This leads Eq. (7.8) to define a new Hessian of pressure by $\Omega^2 + S^2$.

The definition of a vortex core according to [67] is that we are inside a vortex if a connection of local pressure minima is found, i.e., when the new Hessian of pressure is taken into account, the pressure needs to be a minimum at least in two of its principal directions. In detail, pressure is not required to be a 3D minimum. Formally, this definition can be expressed by the eigenvalues of $\Omega^2 + S^2$. If $\lambda_1 \geqslant \lambda_2 \geqslant \lambda_3$ are the eigenvalues, then a point $\mathbf{p}$ is part of a vortex core if $\lambda_2(\mathbf{p}) < 0$. That also leads $\lambda_3(\mathbf{p})$ to become negative and, therefore, the modified pressure is a minimum on the plane spanned by eigenvectors corresponding to λ_2 and λ_3. Note that $\Omega^2 + S^2$ is real and symmetric; therefore, it has only real eigenvalues.

λ_2 can also serve as basis for global, line-oriented vortex core detection (see Section 7.3 and Section 8.2); it has been chosen in most of the following methods because it is considered by many fluid-mechanics engineers to be most effective and reliable for detecting vortices in incompressible flows (except for very special cases with strong axial stretching [202]). A common way to visualize λ_2 vortex regions is to draw its isosurface, setting the isovalue to 0. This isosurface then encloses the entire vortex region, which is is supposed to have a negative λ_2 value.

7.2.3 Other Criteria

In this thesis, primarily the λ_2 criterion is used since it seems the most stable and reliable method at the time this thesis has been written. However, to give a short overview over other common methods for the detection of vortex cores, in this section some alternatives are discussed. According to Lugt, a vortex is found if the stream lines or path lines are closed or spiraling. However, stream and path lines are not Galilean invariant, i.e., different reference systems imply different solutions. Furthermore, stream lines and path lines are not closed in most cases, since a trajectory depends on the temporal development of the vortex and its topological changes. Another criterion consists

of the vorticity magnitude $|\omega|$. Keep in mind, that vorticity also occurs in regions of shear flow, i.e., not only vortex structures are detected by this criterion. An alternative method is the Δ method by Cheong et al. [26]. It considers the eigenvalues of $\nabla \boldsymbol{v}$ in order to find out if a stream line spirals around a point; complex eigenvalues imply this behavior. Therefore, the discriminant of the characteristic equation of $\nabla \boldsymbol{v}$ with its eigenvalues σ is taken into account. If P, Q, and R are the three invariants of the characteristic equation, then the discriminant Δ is defined by

$$\Delta = \left(\frac{1}{3}Q\right)^3 + \left(\frac{1}{2}R\right)^2 .$$

When the eigenvalues are complex, Δ is positive, i.e., $\Delta > 0$ and, therefore, the region is inside a vortex.

7.3 Vortex Core Lines

Similar to vortex cores, there exists no general definition of vortex core lines. From the point of fluid dynamics, the vortex core line describes the axis of rotation inside a vortex and can be considered as the 1D skeleton of a 3D vortex region. Because of its characteristics, a vortex core line can be used to segment [165] or track vortices [148].

The most common definition considers a vortex core line as a connection of local pressure minima on a plane perpendicular to the vortex axis. As already shown in Section 7.2, pressure is not a reliable quantity for detecting vortices and is often replaced by λ_2 or any other appropriate vortex identification criterion.

7.3.1 Formal Definition

A formal description of vortex core lines can be given by considering them as 1D height ridges [56; 37] of a specific criterion. Let the application of the criterion result in a scalar field, containing a smooth function $f(\boldsymbol{x})$. Then the condition of a local minimum can be derived from critical point theory (see Section 7.5), i.e., $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$. Furthermore, f needs to be non-degenerate, i.e., the local minima should be isolated. Per definition, a critical point is a local minimum when its Hessian $\mathcal{H}f$ containing the partial derivatives of f has three positive eigenvalues.

The generalization of minimum points to minimum lines, also called valley lines, relaxes these conditions. Since a variation of the gradient along the valley line should be allowed, a point on it is only required to be a minimum on a plane orthogonal to its tangent vector. In [113; 121], it has been shown that valley lines are required to have the minimal slope $|\nabla f(\boldsymbol{x})|$ of all points of the same elevation $f(\boldsymbol{x})$. This means that $|\nabla f(\boldsymbol{x})|$ needs to be a minimum in direction of smallest change of f, which can be determined by a plane orthogonal to ∇f. In detail, that requires the gradient to be an eigenvector of the Hessian $\mathcal{H}f$. If the eigenvalues of $\mathcal{H}f$ are defined in a consecutive order $\eta_0 \geqslant \eta_1 \geqslant \eta_2$ and $\boldsymbol{\epsilon_i}$ denote the respective eigenvectors, then a point lies on a valley line if $\eta_1 > 0$ and $\boldsymbol{\epsilon_2} = \mathrm{const} \cdot \nabla f$ are fulfilled. That implies that $f(\boldsymbol{x})$ is a minimum on the plane orthogonal to the valley line, but not necessarily on the valley line itself.

Algorithm 2 Predictor-corrector vortex detection algorithm.

```
void ExtendedBanksSinger()
{
for each remaining seed point p_0
  if p_0 is not in any previous vortex;
    determine vorticity ω_i at position p_i;
    integrate vorticity ω_i to predict new position p̄_{i+1};
    determine vorticity ω̄_{i+1} at prediction p̄_{i+1};
    p_{i+1} is the point of minimum λ_2 in the plane P⊥ ω̄_{i+1} (correction step);
    i ← i + 1;
  endif
endfor
}
```

The problem of finding valley lines was reformulated by Peikert et al. [121] by using the parallel-vectors operator. According to their definition, an extremum line of f is defined by $\nabla f \parallel (\mathcal{H}f)\nabla f$, which claims, similar to the description above, ∇f to be an eigenvector of the Hessian $\mathcal{H}f$. Also the definition of a valley line is similar: let ϵ_2 be aligned with ∇f, then $\eta_0 > \eta_2$ and $\eta_1 > \eta_2$ with $\eta_1 > 0$ must be fulfilled.

7.3.2 Predictor-Corrector Method

According to the taxonomy of Jiang et al. [68], a vortex core line can be either computed by using a *local* or a *global* method. A local method considers, as the name already implies, only the local neighborhood of the point to be classified as part of a vortex core line or not. In contrast to this, a global method considers more points, ideally all points inside a data set, in order to decide whether a position is located on a vortex core line. Parallel-vectors (PV) [121] is such a local method, because it only considers the closest neighbors incorporated in the partial derivatives of f, which should be serve as vortex core criterion, e.g., $f = \lambda_2$.

In this section, the global predictor-corrector algorithm by Banks and Singer [7; 8] is discussed. The idea of this algorithm is to start from a position that is supposed to be on the vortex core line and growing the vortex core line by integrating along a given vector field. According to Section 7.3.1, a local minimum of f is defined to be part of the vortex core line. Therefore, all 3D minima of f can be used as seed points. For the vector field, which is used for growing the vortex core line, there exist two alternatives: (1) the eigenvector ϵ_2 of the Hessian of f, that is approximately aligned to the tangent vector of the vortex core line; (2) the vorticity vector ω, that points in direction of the axis of rotation and, therefore, should also deliver an approximation of the vortex core line's tangent vector. Keep in mind, that (1) is only valid in regions close to the valley line. This can be shown by a Taylor expansion around a point $\mathbf{p}$ on the vortex core line.

In Algorithm 2 the vorticity is used, since it depends only on first order partial derivatives of the vector field. It shows the original predictor-corrector algorithm by Banks and Singer [7; 8] that is searching for local pressure minima. This algorithm was extended by Stegmaier et al. [165] by replacing pressure with the more reliable λ_2 [67]. In the following, the algorithm by Stegmaier [165] is discussed, since in this

thesis only the λ_2-based algorithm is used. The algorithm works like follows: first all the seed points are determined by searching for 3D local λ_2 minima. Starting with the global λ_2 minima, i.e., the smallest local λ_2 minima, the seed points are processed in a consecutive order. Then, for each seed point $\mathbf{p}_0$, the vorticity vector $\boldsymbol{\omega}_0$ is computed in order to integrate it. The new position $\bar{\mathbf{p}}_1$ represents the predicted next point on the vortex core line. This position is considered as predicted due to the numerical errors occurring during integration. The following correction steps tries to decrease these errors by applying a global search on the plane perpendicular to the vorticity $\bar{\boldsymbol{\omega}}_{i+1}$, which is restricted by the vortex boundaries, e.g., the $\lambda_2 = 0$ isosurface. The minimum found represents the corrected position $\mathbf{p}_{i+1}$ on the vortex core line. Then the algorithm is repeated until an exit condition is fulfilled, e.g., the vorticity becomes unsteady or the λ_2 threshold is exceeded.

7.4 Shear Layers

As shear layer regions are defined as regions of high strain, the identification of shear layers is based on the strain-rate tensor. Actually, the strain rate tensor T is defined by Eq. (7.3). However, for a qualitative consideration of shear stress S is sufficient. In order to identify a shear layer, it is preferable to compute a scalar value that represents the strength of the strain. One of the most important criteria is the Q criterion [63], which is already described in Section 7.2.1. It is used to decide whether strain rate or rotation dominates at a specific position.

Another method was proposed by Haimes et al. [54], where an eigenvalue analysis of the deviatoric stress tensor S^D is applied. The tensor S^D disregards stress components that change the volume of a fluid particle. These stress components are called *hydrostatic stresses* and can be expressed as the mean of the principal stresses:

$$p = \frac{S_{ii}}{3} \, . \tag{7.9}$$

Then the deviatoric stress tensor can be computed by removing the hydrostatic component from the strain rate tensor S, which is defined by

$$S^D_{ij} = S_{ij} - \frac{S_{kk}}{3}\delta_{ij} \, , \tag{7.10}$$

where δ_{ij} is the Kronecker symbol.

Since S^D is symmetric and positive, it has always three real eigenvalues λ^D_{Si}. The vector formed by the eigenvalues of S^D defines the principal axis of deformation and the norm of the second principal invariant is used as a measure of shear. If a transformation of the coordinate system is applied by diagonalization of S^D and the norm of the second principal invariant is computed, then the rate of shear stress according to Haimes et al. [54] is obtained by

$$S_H = \sqrt{\frac{(\lambda^D_{S1} - \lambda^D_{S2})^2 + (\lambda^D_{S1} - \lambda^D_{S3})^2 + (\lambda^D_{S2} - \lambda^D_{S3})^2}{6}} \, . \tag{7.11}$$

Another approach for shear layer identification is given by Meyer [112]. Here, the second invariant of S is used for the computation of the rate of shear stress:

$$S_M = \frac{1}{2}(S_{ii}S_{jj} - S_{ij}S_{ij}) \, , \tag{7.12}$$

where a shear layer region is detected if $S_M < 0$. Both methods shown in equation 7.11 and 7.12 are coordinate system invariant. In this thesis, both criteria by Haimes et al. and Meyer are used.

Considering the similarity of S_H and S_M, it should be mentioned that if the strain rate tensor S is used instead of the shear stress tensor T, both methods deliver the same result. The reason is that S contains no hydrostatic stresses in the case of incompressible flows, i.e., the first invariant of S is defined by $I_1 = S_{ii} = 0$.

Each of the three criteria, Q, S_H, and S_M is independent of the variation of a reference frame with constant speed; therefore, all three criteria are at least Galilean invariant. Furthermore, the criterion of Haimes et al. and the criterion of Meyer are also rotational invariant in contrast to the Q criterion, where the result depends on the orientation of the reference frame (for more detailed information see [55]). Another disadvantage of the Q criterion consists of its exclusive behavior, i.e., either a region is detected as a vortex or as shear layer.

Similar to λ_2 described in Section 7.2.2, also shear layers can be visualized by $S_H = 0$ and $S_M = 0$ isosurfaces in order to enclose complete regions of high shear stress, respectively. However, this results rather in volumetric shear regions than in shear layers. Section 8.3.1 deals with this problem. For the discussion in Chapter 8 and Chapter 9, I_2 is introduced for denoting a general shear layer criterion, which can be replaced by both S_M and S_H, respectively.

7.5 Critical Points

The mathematical definition of a critical point in 3D is given by the mapping of two tangent spaces. Let $\mathbb{M}$ be a 3-manifold and $f : \mathbb{M} \to \mathbb{R}$ a real-valued smooth function. Then a point $\mathbf{p}$ is defined as *critical* if the induced mapping of the tangential spaces $T_{\mathbf{p}} f : T_{f(\mathbf{p})}\mathbb{R} = \mathbb{R}$ is a zero map. The corresponding function value $f(\mathbf{p})$ is then called *critical value*.

We define $\boldsymbol{\nu} = \nabla f$ for the following discussion. As already mentioned above, a point $\mathbf{p}$ is critical if $\boldsymbol{\nu}(\mathbf{p}) = \mathbf{0}$, otherwise it is *regular*. Furthermore, according to Morse theory, a critical point of f needs to be non-degenerate, i.e., each critical point should be isolated. This means that $\boldsymbol{\nu}(\mathbf{p} + \epsilon) \neq \mathbf{0}$ for an infinitesimally small distance ϵ, which can be determined by requiring that the Hessian $\mathcal{H}f$ is not singular, i.e., $\det(\mathcal{H}f) \neq 0$. Additionally, each critical point must have a different function value. The set of functions fulfilling these two conditions are called *Morse functions*.

The Morse lemma also shows that each Morse function can be represented in a diagonalized quadratic form, which can be used to determine the *index* of a critical point. The index stands for the number of negative eigenvalues of $\mathcal{H}f$ and is used to classify a critical point. In 3D there exist four types of critical points: minima, 1-saddles, 2-saddles, and maxima that correspond to 0,1,2, and 3 negative eigenvalues.

Basing on the definition of Morse functions, Morse Smale complexes can be generated. A Morse Smale complex is defined as a set of *ascending* and *descending* manifolds. Both types of manifolds are clusters of integral lines, i.e., maximal paths of f in $\mathbb{M}$, where the tangent vectors are collinear to the gradient of f. Integral lines either have a common origin (ascending manifold) or a common destination (descending manifolds) that are represented by critical points of f.

CHAPTER

8 FEATURE DETECTION AND VISUALIZATION

This chapter of the thesis considers the identification of flow features by means of particle tracing. Flow features can be regarded as regions inside a fluid data set that meet specific criteria that reach from less complex quantities, like velocity magnitude, to more complex functions describing high rotation or high strain rate. Usually, flow features are directly computed from the given vector field (sometimes more information in form of multi-field data is required) and are used to investigate the flow behavior. An example is the investigation of turbulent flows that tend to amplify created vortex structures over time: here, turbulent regions can be extracted using feature detection methods. Then, the features, here the vortices, can be tracked along time using modern tracking mechanisms. This enables the user to identify critical vortices that can be backtracked in time to identify their origin.

In the following, it is shown how important particle tracing is for detecting and representing such flow features. All the methods proposed in this chapter are used for a better understanding of the flow, either by exploration of the flow in feature regions or the temporal change of a feature region itself. The discussion bases on the methods for flow feature extraction, that are introduced in Chapter 7. In detail, in Section 8.1, it is shown how feature extraction can be combined with 3D texture advection. It proves the theoretical background of the λ_2 vortex criterion by considering the particle paths within vortex regions; Section 8.2 uses the already extracted features, namely λ_2 vortex structures, to compute their geometrical simplification in terms of vortex core lines. Vortex core lines are strongly related to the particle paths considered in the previous Section 8.1 because they can be regarded as the center of the particles' rotation; finally, Section 8.3 introduces the computation of shear sheets, i.e., the 2D representation of maximal shear. This section extends the visualization of vortices to a more general feature visualization. Both the vortex core lines and the shear sheets are used for tracking in Chapter 9.

8.1 Feature-based 3D Texture Advection

Considering the 3D texture advection approach of Section 4.2 again, although an interactive representation of a flow field is achieved, the issue of clutter and occlusion remains. The reason is obvious: due to the dense representation, all parts of the domain, whether they are of interest or not, are visualized with the same particle density. This means that the (approximately) same number particles are seeded in each region of the data set and tend to occlude interesting parts that are located more behind.

The goal of this section is to show how to deal with these issues by fading out

uninteresting parts of the data set, while interesting parts are emphasized. A common way is to use flow features as importance function in order to decide whether a region is interesting or not. Therefore, the 3D texture advection approach of Section 4.2 is extended in order to extract and visualize flow features without loosing the advantage of an interactive visualization of the particles inside a flow. This section contains text from [195] and [196].

8.1.1 Previous Strategies

Most previous methods considering an importance function are geometric-based methods that explicitly extract a geometric representation of particle traces (e.g., streamlines, streaklines, or pathlines) by means of 3D positions along those traces. For geometric methods, the importance function is typically used to control the space-variant density of particles or their traces. This density can be controlled through particle seeding and particle removal.

One strategy to control the particle density is to explicitly check this density on a regular basis by counting particles and relating the number of particles to the covered region. For example, Guthe et al. [53] use this approach in combination with an octree data structure to efficiently count particles and to modify their number if needed. Another approach controls particle injection or removal by a stochastic model, typically modeled by a density distribution function as used, e.g., by Löffelmann and Gröller [106] or Weiskopf et al. [197]. With a similar method, streamline seeding can be guided by critical points of the vector field, as described by Verma et al. [179].

Another class of approaches takes into account the extended objects (e.g., streamlines) originating from particle tracing in order to control their density. An early example of this approach is image-guided streamline placement by Turk and Banks [172], which leads to an approximately even distribution of streamlines on a 2D domain. An improved method by Jobard and Lefer [72] allows for a fast computation of evenly-spaced streamlines of arbitrary density. The Chameleon system by Shen et al. [158] is a 3D example of evenly-spaced streamlines that are subsequently voxelized on a 3D texture.

An obvious strength of all geometry-oriented approaches is their high efficiency when rather sparse representations with only few particle traces are used. Typically, the computational cost increases linearly with increasing density of traces. Therefore, feature-based visualization is often used to directly control the particle or trace density in order to achieve a speed-up when only small regions of a data set are emphasized by the importance function.

8.1.2 Feature-based 3D Texture Advection

In the following, it is shown how an importance function can be integrated in the 3D texture advection technique by Weiskopf et. al [195; 196], which is described in Section 4.2. Figure 8.1 illustrates a modified volume rendering process that takes into account the importance function. The property field and the importance function are rendered simultaneously in a way that the importance value is used to modify the transfer function applied to the property value. A low importance value leads to a reduction of the transfer function value, decreasing the contribution of its color and

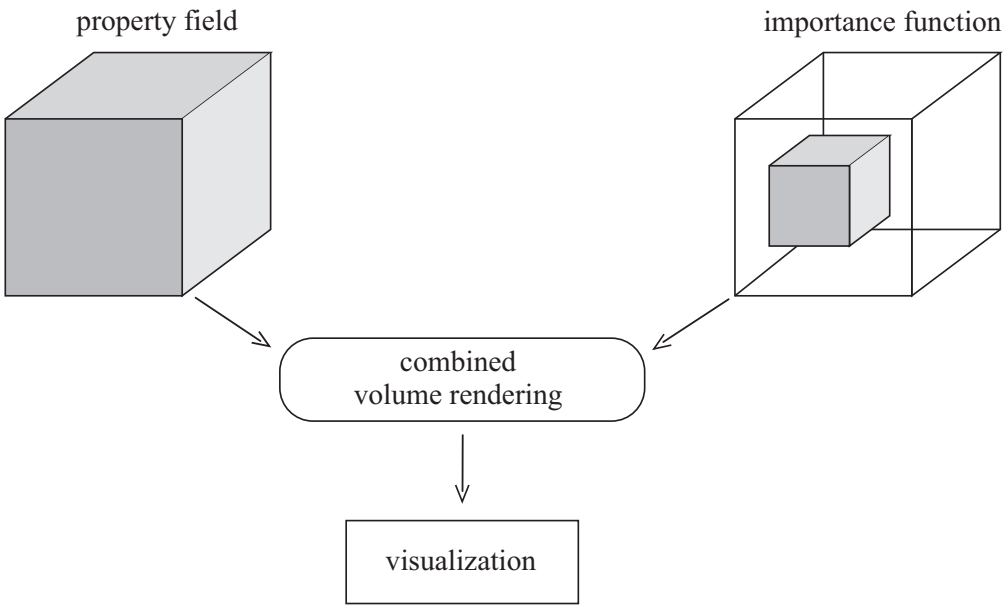


Figure 8.1: Feature-based flow visualization. A scalar-valued importance function controls the visibility of the advected property field during volume rendering.

opacity—often even to zero. Conversely, a high importance value results in a final value that is (almost) identical to the original transfer function value. This behavior is modeled in the following way: the importance value is mapped to an intermediate weighting factor; subsequently, the result of the original transfer function is multiplied by this factor. The mapping of the importance value is represented by a function ξ. A typical choice for ξ is a shifted Heaviside function

$$H_\tau(x) = \begin{cases} 0 & \text{if } x < \tau \\ 1 & \text{otherwise} \end{cases},$$

according to our convention. The parameter τ is the threshold below which the volume is completely transparent. The use of the Heaviside function leads to volume clipping because parts of the volume are virtually cut away. The threshold parameter is usually specified by the user, typically in an interactive exploration process. For more details on volume clipping see the papers on GPU-based volume clipping [190] and its application to volume illustration [16].

An alternative is a smooth map ξ that leads to a gradual transition between important regions and less relevant parts of the data set. The transition rate—the slope of ξ—and the location of the transition are normally specified by the user. A graphical way of defining a gradual transition is provided by smooth brushing [36]. In general, it is useful to specify the importance function interactively in order to incorporate the user's expertise in the data exploration process. For example, focus-and-context methods [35] could be directly used to generate the importance function.

Slice-based volume rendering can be enriched by the importance function by extending the respective fragment program. This fragment program additionally accesses

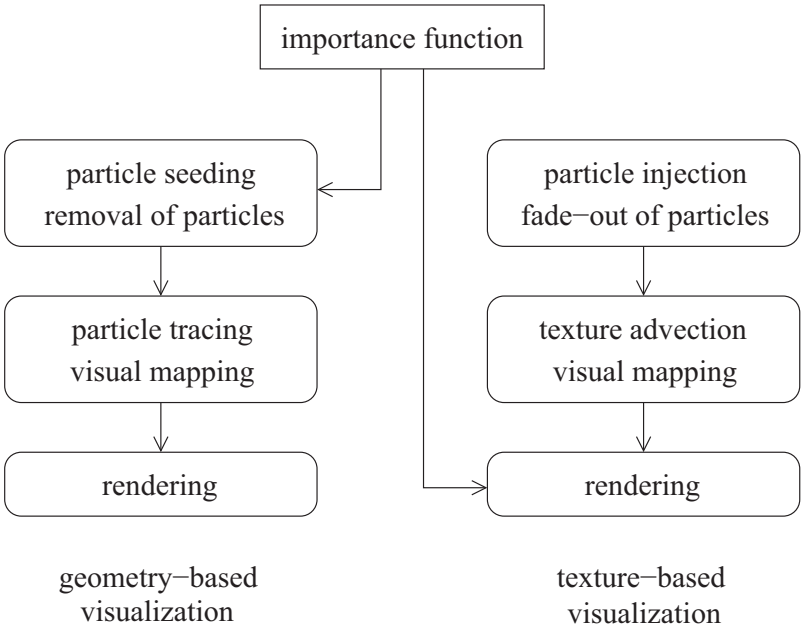


Figure 8.2: Typical pipelines for feature-based flow visualization.

the texture that holds the 3D importance function, applies the function ξ to the importance value, and multiplies the resulting value by the transfer function value. The map ξ can be implemented by a dependent lookup in a texture that represents ξ. Simple versions of ξ, such as the Heaviside function, can be mapped to arithmetic shader instructions. In some cases, the importance function need not be stored explicitly in a texture but could be constructed on-the-fly in the fragment program. An example is velocity magnitude, which is computed directly from the vector field. The usefulness of velocity magnitude as a feature measure is discussed by Jobard et al. [71] for 2D flow.

In contrast to geometric-based methods, this approach for the feature-guided visualization of texture-based flow representations uses the importance function only during the final display. Texture advection and blending, which are the analogs of particle tracing and line construction, are not affected by the feature definition.

8.1.3 Texture-based vs. Geometry-based Visualization

Figure 8.2 compares the data flow and processing steps for the geometry-oriented and texture-based approaches. The two approaches differ in the stage where the importance function influences the visualization pipeline: the geometry-based pipeline is affected already at its first stage, whereas the texture-based pipeline is only affected at the last stage of volume rendering. Of course, a geometry-oriented method could also be used to generate a dense representation that is affected by the importance function only during rendering. However, this approach is not considered in the following discussion because it would ignore a most important performance benefit of geometry-based methods and typically lead to a visualization that is less efficient than a comparable texture-based implementation.

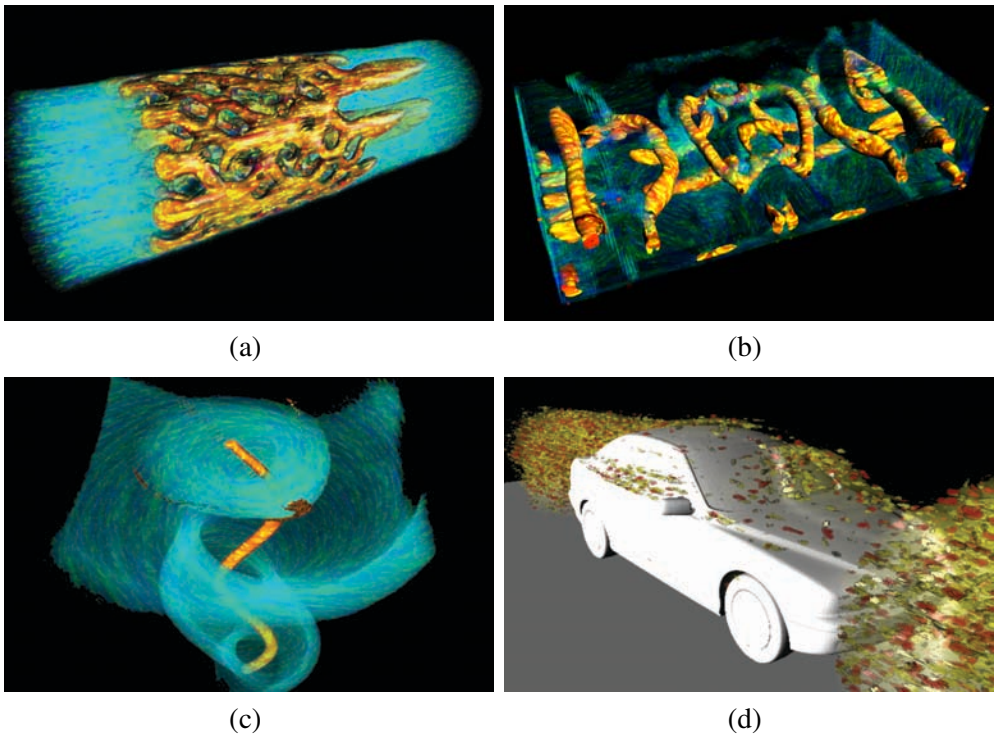
An advantage of the this implementation of texture-based visualization is the decoupling of particle tracing and feature-based display. This decoupling is useful for the following application scenarios.

One scenario supports a tight interaction between the user and the visualization. The user can quickly change the emphasized region without having to wait for an update of the seeding or particle tracing processes. This advantage is most prominent for incremental constructions of particle traces for time-dependent flow when the traces are computed step-by-step during the animated visualization. For example, texture advection just needs the current time step of the vector field to propagate the animated visualization by one time step. The iterative computation can be derived mathematically from a discretization of an exponential filter kernel for line integral convolution [41]. Texture advection can handle both steady and unsteady flow without any changes, and thus allows for a streaming-in of flow data with a concurrent streaming computation of the visualization. However, there is a tune-in phase of several time steps required at the beginning of the visualization. An analogous iterative computation can be employed for geometric traces, for example, for pathlines that are followed while unsteady flow data is streamed in. Similarly, a tune-in phase is required to obtain pathlines of a certain minimum length. The tune-in time span prevents a fast modification at the particle seeding stage of the visualization pipeline.

A second scenario is a focus-and-context visualization of 3D flow. The focus—the emphasized region (also see the first scenario)—can be rendered almost opaque, whereas the surrounding context region can be rendered more transparent. Typically, the focus covers only a small area, but the context tends to be rather spread out. Therefore, the visualization of the context requires the computation of particle traces in a large portion of the data set, which is directly supported by a full spatial coverage texture through advection. In addition, the focus and the context can be easily and quickly changed by just modifying the opacity transfer function ξ or the computation of the importance function.

A third scenario is the interactive exploration by means of dense visual representations. An extreme example is the examination of a data set by volume clipping [134], which could be described by a Heaviside transfer function ξ. A dense coverage is favorable on clipping planes to make good use of the available screen resolution. An interactive modification of clipping planes is only possible when a high-density 3D property field is available. The same argument is true even in less extreme examples that employ a spatially-restricted and highly opaque subregion of 3D space. A typical example is flow visualization in a thick boundary layer around a surface [111].

In general, the strengths and weaknesses of geometry-oriented and texture-based methods are mostly complementary to each other, which makes the two approaches valuable alternatives. The geometry-based approaches are favorable for relatively sparse representations, for non-incremental computations of particle traces, and for fixed importance functions. Conversely, the proposed texture-based method is better suited for dense representations, for an incremental and animated visualization of possibly streamed, time-dependent flow, and tight user interaction with the importance function.



(a) (b)

(c) (d)

Figure 8.3: (a) Visualization of a fluid through a tube. Velocity masking is used to emphasize regions of high velocity (yellow-red regions); (b) Benard flow, displayed by λ_2-based focus-and-context visualization. Swirling features are emphasized by high opacities; (c) Combination of λ_2 vortex extraction and velocity masking for a tornado data set. The focal point is the high-opacity yellow-red tube, as identified by the λ_2 criterion; (d) Visualization of an aerodynamic simulation from the automotive industry. Here, velocity masking is applied.

8.1.4 Examples

The effectiveness of the feature-based 3D texture advection is shown in terms of some examples (Figure 8.3). For the first example, depicted in Figure 8.3 (a), velocity magnitude is used as a feature measure. Here, velocity masking extracts the structure of the fluid streaming through a tube equipped with interior obstacles. The obstacles are not shown in order to reduce visual occlusion. Regions of high velocity are rendered in yellow-red colors and emphasized by high opacities. The context, which has lower velocity, is shown with more transparent green-blue colors. In a combined volume-rendering visualization, both the focus and the surrounding context are shown simultaneously. A more sophisticated feature is used in Figure 8.3 (b), where λ_2 vortex detection [67] is applied to a Benard convective flow. In Section 7.2.2, the λ_2 criterion is discussed in more detail. In this focus-and-context visualization, the focus region (i.e. vortex region) is displayed by materials with red or yellow colors while the surrounding flow is still visible as semi-transparent material with blue or green color. Here, the context is rendered slightly less prominent than in Figure 8.3 (a) by using a transfer function with lower opacities for the context region. Figure 8.3 (c)

demonstrates that different feature criteria can be combined in a single visualization. The focal point is the high-opacity yellow-red tube in the center of the tornado, as extracted by the λ_2 criterion. The green-blue context region is shown by velocity masking for mid-range velocities. Finally, Figure 8.3 (d) provides the visualization of an aerodynamics simulation from the automotive industry. Feature regions are again extracted by velocity masking. Here, volume rendering includes line-based illumination. These four examples demonstrate that different features or different visualization styles can be used to emphasize flow regions within the proposed system.

8.2 Topology-Preserving Vortex Core Line Detection

So far, particle tracing has been applied to several vector fields in order to display the behavior of a flow, or at least, the behavior of a generated vector field, e.g., the curvature vector field of Section 6.2. Also in the previous chapters particle tracing has been applied to the flow field, whereas the feature extraction was enabled by a masking operator.

In this section, particle tracing is applied for the detection of vortex core lines. They build an important and interesting possibility for representing vortices, since vortex core lines can be used for vortex segmentation [165] and vortex tracking [185; 170; 148]. For a more detailed discussion on vortex core lines see Section 7.3. However, although there exist various methods for the extraction of vortex core lines, one issue remains: the vortex topology. So far, vortex core lines have been considered as 1D line structures with a simple topology, neglecting any bifurcations.

The basic idea of the following method is to enrich the λ_2 criterion [67], which is a region-based, Galilean invariant, and local vortex criterion, such that it allows us to construct 1D vortex core lines. A theoretical view on the λ_2 criterion is given in Section 7.2.2. The proposed algorithm consists of the following stages: First, the scalar field of λ_2 is constructed and the corresponding isosurface is extracted. Second, the isosurfaces are reduced to 1D representations by skeletonization. These skeletons serve as initial prediction of the core lines. Third, the locations of the skeleton prediction are corrected by searching for λ_2 minima on planes perpendicular to the core line.

This vortex core line methods follows the idea of a predictor-corrector method [8; 161; 165], since it provides the advantages of being global (see Section 7.3) and avoid the usage of second order derivatives, like the parallel-vectors (PV) operator. At this point, it should be noted that the PV operator only classifies points for being part of the vortex core line or not, but it doesn't provide the correct connection of these points. One possibility of connecting the points extracted by PV is by the feature flow method [171; 170]. Here, a vector field is derived from the PV result, a vector field where critical points (see Section 7.5) occur at the location of a vortex core line, which is integrated to connect the extracted points. However, beside the requirement of higher order derivatives of λ_2, the resulting vector field cannot be used to detect bifurcations. Furthermore, due to the noise sensitive second order derivatives, a large number of unwanted core lines might be detected with this local method.

The following skeleton-based vortex core line method is supposed to achieve the same result as the PV condition $\nabla\lambda_2 \| (\mathcal{H}\lambda_2)\nabla\lambda_2$ with $\eta_1 > 0$ when $\eta_0 \geqslant \eta_1 \geqslant \eta_2$

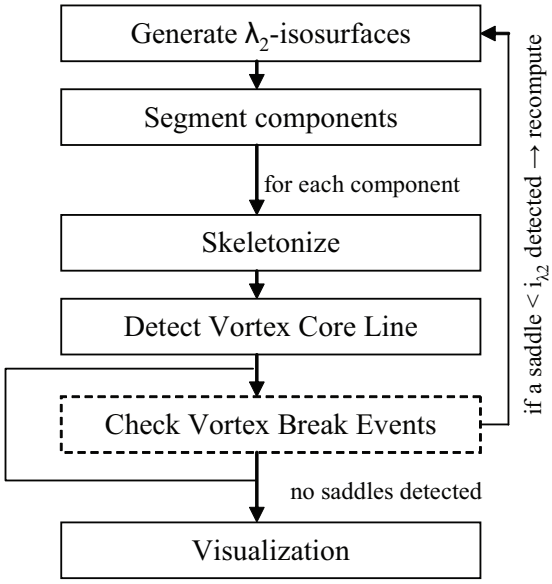


Figure 8.4: Program flow for the computation of vortex core lines.

are the eigenvalues of $\mathcal{H}\lambda_2$. This condition defines the vortex core line as a valley line of λ_2. Thereby the idea of this method is to replace pressure or vorticity, which are used for integration by the predictor-corrector methods (see Section 7.3.2) of Banks and Singer [8; 161] and Stegmaier et al. [165], respectively, by a more robust quantity: the skeleton of the $\lambda_2 = 0$ isosurface. Please note that λ_2 could be replaced by any other vortex identification criterion that defines a vortex core line as an extremum line, e.g., pressure or vorticity magnitude.

Using the vortex's skeleton, i.e., its 1D medial axis, is reasonable because it can be considered as a first approximation of the vortex core line. Furthermore, the skeleton conveys the vortex's topology. How these properties can be exploited in order to extract topology preserving vortex core lines is discussed according to the paper of Schafhitzel et al. [151] (Figure 8.4). This work was in collaboration with my colleague Joachim Vollrath.

8.2.1 Preliminary Steps

The algorithm starts by computing the λ_2 field and its corresponding isosurface. The isovalue $\lambda_2 = 0$ yields the representation of all vortical regions, enclosed by the isosurface [67]. To treat vortical regions individually, segmentation is applied to the λ_2 isosurface: individual vortical regions are identified by searching the isosurface for disjoint components. By confining the skeletonization process to each component individually a curve skeleton is obtained for each vortical region. In the remainder of this section the curve skeleton of a vortical region is referred to as *isoskeleton*.

8.2.2 Vortex Core Line Detection

According to Section 7.3, the vortex core line is defined as a connected set of plane-restricted local λ_2 minima, where the search plane is perpendicular to the tangent vec-

tor of the core line. However, the core line and its tangent vector are yet unknown during the detection phase. This makes it necessary to find an appropriate estimate of the core line. Therefore, the tangent vector of the vortex core line is approximated by the tangent vector of the isoskeleton. Since the vortex core line is generally defined as a one-parameter, 3D space curve that serves as a geometrical, and often dynamical, simplification of a vortex, the isoskeleton provides a good initial approximation of the vortex core line.

Algorithm 3 shows how the algorithm by Stegmaier [165] (Algorithm 2) is modified in a way the isoskeleton curve $\Gamma(s)$ is used to find a single-component vortex core line. Metaphorically speaking, the isoskeleton is viewed as a handrail guiding the core line search algorithm. Starting at the first point of the isoskeleton, $\Gamma(0)$, the algorithm steps along the isoskeleton until the endpoint, $\Gamma(\mathtt{numPoints}-1)$, where $\mathtt{numPoints}$ describes the number of points of the polygonal representation of Γ. At the first sample point on the isoskeleton, the plane perpendicular to the isoskeleton tangent vector $\mathrm{d}\Gamma/\mathrm{d}s$ is created to search the plane-restricted λ_2 minimum. The search region is restricted to the part of the plane contained within the λ_2 isosurface. Using the minimum found as first point of the vortex core line, $\mathtt{pos}[0]$, the algorithm loops over the remaining points $\Gamma(\mathtt{i})$ of the isoskeleton to construct the rest of the vortex core line. The tangential vector of the isoskeleton is used to predict the position of the next minimum. The predicted position $\mathtt{pred}$ is computed by moving from the current point $\mathtt{pos}[\mathtt{i}-1]$ on the vortex core line along the tangent direction $\mathrm{d}\Gamma/\mathrm{d}s$ at location $(\mathtt{i}-1)$. The idea is to move piecewise parallel to the isoskeleton, where the stepsize is defined by the distance between the sample points on the isoskeleton. Once the predicted position $\mathtt{pred}$ is found, again a plane is created, orthogonal to the tangent vector of the current sample on the isoskeleton. The λ_2 minimum on the plane defines the corrected position $\mathtt{pos}[\mathtt{i}]$ of the next sample of the vortex core line. Subsequently, the algorithm loops over all sample points on the isoskeleton. Once the endpoint of the isoskeleton is reached, the algorithm terminates. The result is a number of plane-restricted λ_2 minima connected according to the topology of the isoskeleton. Note

Algorithm 3 Detection of vortex core lines.

```
void computeVortexCoreLine(IsoSkeleton Γ)
{
// Compute the first position on the vortex core line
create plane P ⊥ dΓ/ds(0) at Γ(0);
pos[0] = min(λ2) on P;
numPoints = NumOfDiscreteSamplePoints(Γ);
// Loop along the isoskeleton
for i=1 to numPoints-1
   // Predict the new position
   pred = pos[i-1] + stepsize * dΓ/ds(i − 1);
   // Create a plane perpendicular to the isoskeleton tangent vector:
   create plane P ⊥ dΓ/ds(i) at pred;
   // Correct the predicted position
   pos[i] = min(λ2) on P;
endfor
}
```

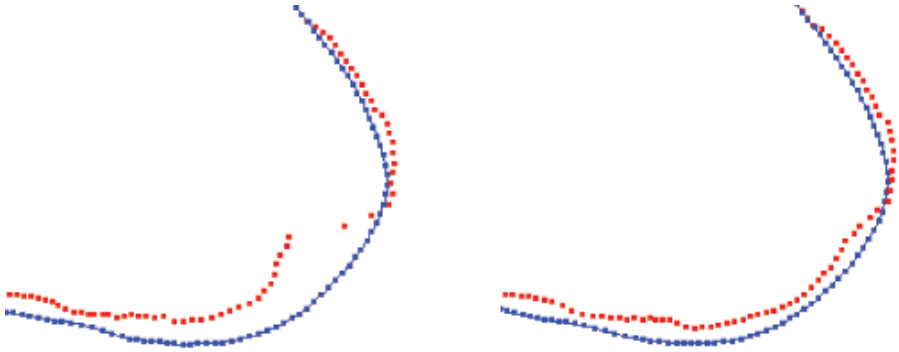


Figure 8.5: Parameterization issues: the red curve of the vortex core line grows faster than the blue curve of the isoskeleton (left). Therefore, the tangent vector for generating the search plane becomes corrupted. This problem is addressed by reparametrization of the vortex core line (right).

that the computed vortex core line is independent of the starting point of the algorithm ($\Gamma(0)$ or $\Gamma(\mathtt{numPoints}-1)$) because the orientations of the search planes are always given by $\mathrm{d}\Gamma/\mathrm{d}s$, which is independent of the order of curve traversal when computed through central differences.

One problem is the issue of different parameterizations of the isoskeleton and the vortex core line curves because one curve might propagate faster than the other, particularly in areas of high curvature. As a consequence, sample points on the isoskeleton might be wrongly related to points on the vortex core line. This problem is overcome by adaptively choosing the sampling distance along the two curves, implying an intrinsic reparameterization of the curves. In detail, for each new point on the vortex core line, the distance to its corresponding sample point on the isoskeleton is computed. If the distance is larger than a threshold, e.g., one cell, the position on the core line is recomputed using the half step size for integration. In the case the distance became larger after correction, the core line seems to grow more slowly than the isoskeleton, and therefore, the initial step size is doubled. This process is repeated until the distance between the core line and the skeleton is smaller than the given threshold. Figure 8.5 illustrates an example of the problem and its solution by reparameterization.

8.2.3 Dealing with Vortex Core Line Splits

In the following, a bifurcation point is defined as a point where a vortex core line splits into two parts (see Figure 8.6). A bifurcation is often the result of a temporal change of topology, e.g., due to *vortex reconnection*, which can be ascribed to a merge of two vortices rotating in opposite directions. Kida et al. [81] defined three types of reconnections: the vortex reconnection, the scalar reconnection, and the vorticity reconnection. While the vortex and scalar reconnection depend on isosurfaces of vorticity and any scalar quantity, respectively, the vorticity reconnection treats the topological changes of vorticity lines. Although it is known that in regions of vortex reconnection the closest vorticity lines are canceled by viscous diffusion [81], it was shown that the assumption of vanishing vorticity is only valid in 2D, not in 3D [205]. The bifurcation of a vortex core line can also be ascribed to the generation of new vor-

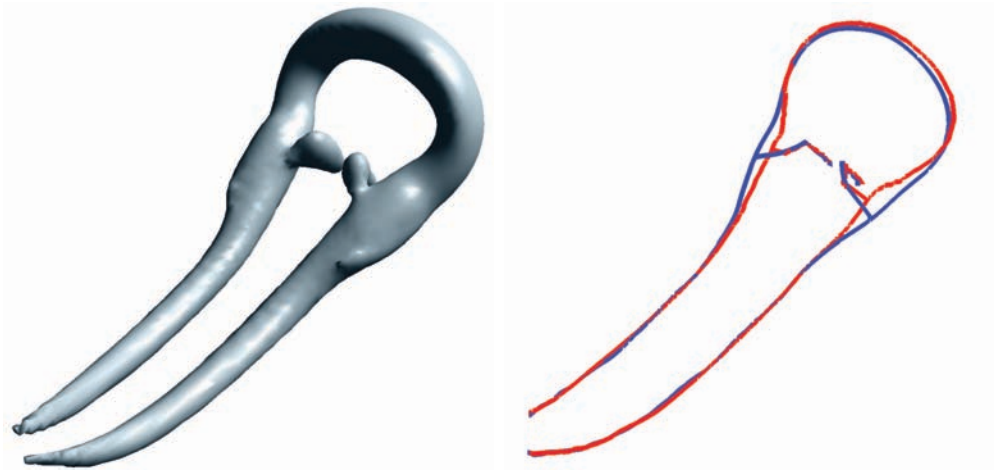


Figure 8.6: Isosurface (left) and core of a hairpin vortex in a transitional flow (right); both arms are growing temporally and are connected with the existing vortex structure. The vortex core line is colored in red and the isoskeleton in blue.

tices, e.g, in terms of an autogeneration process [1]. Here, a so-called *feeder vortex* [8] might be generated, i.e., a vortex core line that enters an existing vortex structure to spiral towards its center.

However, to the author's knowledge, there exists no formal definition of the bifurcations that might result from the previous discussion. To overcome this deficit, bifurcations are modeled by a geometric, skeleton-based approach: it is proposed that the vortex core line has the same topology as the isoskeleton of the λ_2 isosurface enclosing a vortical region. A bifurcation of the vortex core line is assumed in regions where the isoskeleton has a bifurcation, i.e., as a result of scalar reconnection with respect to λ_2.

Algorithm 4 introduces a treatment of bifurcation points on the isoskeleton and the connecting isoskeleton segments, respectively. Now, Algorithm 3 is applied to individual segments and *unsafe regions* near bifurcations (i.e. where local λ_2 minima might be shared by two or more segments, or regions where the spatial boundaries of the λ_2 minimum search are not defined clearly) are treated in the following way: the search for plane-restricted λ_2 minima is always performed in direction toward the bifurcation, starting from a remote *safe region*. Two types of segments are considered (Figure 8.7): a segment that either starts or ends at a bifurcation, and a segment that both starts and ends at a bifurcation. In the first case, the isoskeleton is traversed starting at the non-bifurcation point by adapting the order of the isoskeleton's sample points (function $\mathtt{sortSegment}(\Gamma)$). In the second case, the traversal is started in the middle of the segment in both directions (function $\mathtt{splitSegment}(\Gamma)$).

Unsafe regions are detected by testing the vortex core lines for discontinuities. Similar to previous predictor-corrector approaches [8; 165], discontinuities are identified by checking for a large Euclidean distance between the predicted and the corrected λ_2 minima. If a discontinuity is detected, adaptive search—with reduced stepsize along the isoskeleton—is employed to come as close as possible to the unsafe region by reliably computed λ_2 minimum points. Finally, the missing piece between two adjacent vortex core lines is filled by directly connecting the closest known λ_2 minimum points

Algorithm 4 Extended algorithm for vortex bifurcations.

```
void computeAllSegments(IsoSkeleton Θ)
{
numSegments = getNumberOfSegments(Θ);
// Loop over all segments
for j=0 to numSegments-1
   // Get the segment and its type
   Γ = GetCurrentSegment(Θ, j);
   type = getSegmentType(Γ);
   if(type == end-bif) // endpoint-bifurcation
       sortSegment(Γ);
       computeVortexCoreLine(Γ);
   else // bifurcation-bifurcation
       splitSegment(Γ, Γ_L, Γ_R);
       sortSegment(Γ_L);
       computeVortexCoreLine(Γ_L);
       sortSegment(Γ_R);
       computeVortexCoreLine(Γ_R);
   // connect with segments sharing the same bifurcation points
   connectCoreLineSegments();
endfor
}
```

(see function connectCoreLineSegments() in Algorithm 4). In practice it is observable that the size of the missing piece depends largely on the resolution of the vector field and does typically not exceed one cell (mostly, it is much smaller). Due to the connection step, this method only delivers a topology preserving approximation of the bifurcations.

8.2.4 Finding an appropriate λ_2 Isovalue

In Jeong et al. [67] vortical regions are defined as spatial areas where $\lambda_2 < 0$. However, in practice, the condition of $\lambda_2 < 0$ is rather insufficient because of its dependency on the accuracy of the λ_2 values. The problem of potential inaccuracies is not easy to solve, regardless of they are caused due to a resampling process, i.e. by interpolating the data, or noise which mostly occurs in experimental data. Engineers usually overcome this problem by manually determining an appropriate λ_2 isovalue. Considering the computation of vortex core lines, there are two issues to solve: on the one hand, the λ_2 isovalue has to be chosen very small (i.e., largely negative) to emphasize the vortical structures, and on the other hand, the λ_2 isovalue should not be chosen too small because it would reduce the length of the detected vortex core lines, caused by shrinking isosurfaces when $\lambda_2 \rightarrow -\infty$. This makes it desirable to compute an optimal λ_2 isovalue for each vortical region which is small enough to uncover all the individual vortex structures, but also only as small as necessary in order to prevent the vortex core lines to be shortened unnecessarily.

In this section an algorithm for determining an appropriate λ_2 isovalue is proposed. Since the desired visualization strongly depends on the user's experience, the proposed method works in a semi-automatic manner where the user's influence is given by a decay offset in terms of a parameter ϵ (i.e., a window of λ_2 values). The result of this

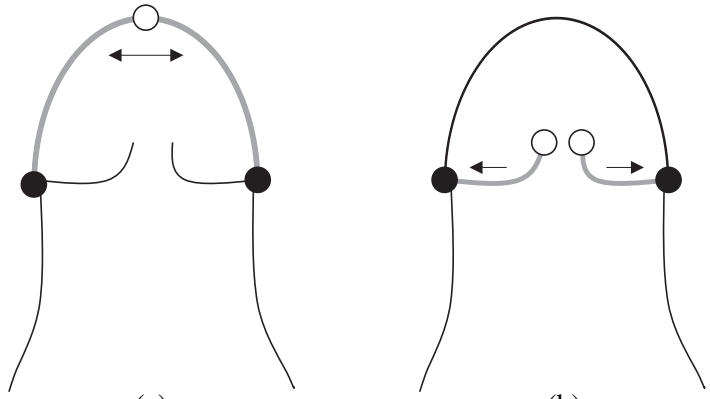
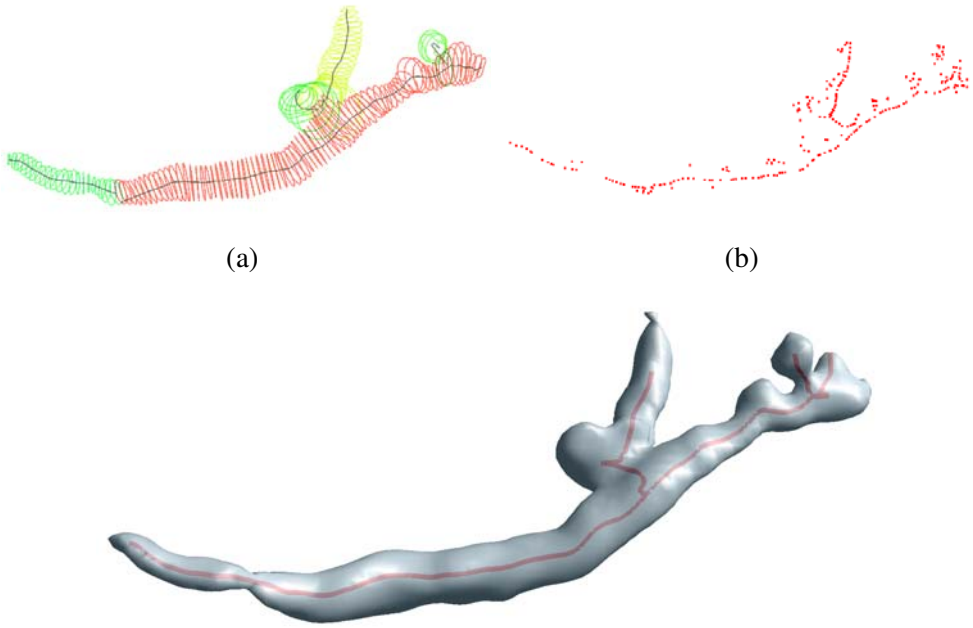


Figure 8.7: Connection types in an isoskeleton: connection of two black bifurcation points—by splitting the connecting segment and starting the search for local λ_2 minima at the white point it is assured that the search propagates toward the bifurcations (a); connection of white endpoints and black bifurcation points (b).

method should be a number of vortex structures, each holding its individual optimal λ_2 isovalue.

Typically, decreasing values of λ_2 produce a hierarchy of nested isosurfaces with changing topology (and thus with changing vortex cores), akin to the contour tree [173]. These changes of topology take place at the position of critical points (see Section 7.5) contained in the interior of the initially computed isosurface. In particular, a break of an isosurface takes place at the positions of local λ_2 maxima on the vortex core line, which is called a vortex break (to the author's knowledge there exists no general definition). Since all points are supposed to be minima on a plane perpendicular to the vortex core line, these curve related maxima can be considered as critical points, particularly as saddles in 3D. Therefore, saddle points may lead to vortex break events when the isovalue falls below the saddle's λ_2 value. In contrast, if there is no saddle on the core line, the core line is guaranteed not to break.

The possible set of vortex break events is reduced by applying the following criterion to the saddle points: $|\lambda_2(\mathbf{x}) - \mathfrak{i}_{\lambda_2}| < \epsilon$, where ϵ is the global user defined decay offset which is given as a percentaged value of the global λ_2 minimum, $\mathfrak{i}_{\lambda_2}$ the current isovalue ($\lambda_2 = 0$ at beginning) and $\mathbf{x}$ the position of a critical point, which is supposed to be a saddle in 3D and a global λ_2 maximum on the respective vortex core line. If such a saddle is found, the isosurface is recomputed with $\lambda_2(\mathbf{x})$. According to the Morse lemma, the critical points of the λ_2 field and thus also the saddles are assumed to be isolated. Therefore, the part of the isosurface within the vortical region under consideration will decompose into two disjoint components at the new isovalue. For these components, the entire process of vortex core line detection, including the aforementioned vortex break detection, is performed in a recursive manner (see Figure 8.4). The recursive way of operation guarantees the core detection to operate locally on vortical regions, traversing the hierarchy of nested components of the λ_2 isosurfaces for different isovalues from top to bottom. In addition, it allows for the detection of different vortex cores to be based on different values of λ_2. It furthermore offers control to the user via the parameter ϵ.



(a) (b)

(c)

Figure 8.8: (a) The approach by Stegmaier et al. extracts components of the vortex core line that are almost equivalent to our results but remain unconnected, (b) the λ_2-based parallel-vectors approach extracts unconnected points on the vortex core line; several outliers are present as well as sections where no core line points were detected, (c) the skeleton-based approach faithfully extracts a vortex core line that consists of several connected components.

At this point it should be mentioned that the quality of the segmentation strongly depends on the quality of λ_2. Theoretically, assuming an infinitely high resolution of the underlying grid, the λ_2 criterion should be able to identify whether two approaching vortex core lines really merge or one vortex only winds around the other. However, in practice this is not always the case. Here, the λ_2 criterion usually considers two vortex core lines that are close to each other as connected.

8.2.5 Vortex Core Line Visualization

For λ_2 isosurface extraction a marching cubes implementation with topological guarantees [103] is used. This implementation is beneficial because individual vortical regions are identified based on the topology of the isosurface geometry, as described in the following paragraph. In order to process vortices individually, a segmentation of the λ_2 isosurface into disjoint components is carried out. Given the output of the isosurfacing algorithm as a triangle soup encoded in indexed triangle and vertex lists, such components are identified by using the instance of a shared vertex between triangles as a grouping criterion.

For this method the isoskeleton, which guides the core line extraction algorithm, is required to be a smooth (C^1 continuous) curve. The skeletonization algorithm pro-

Figure 8.9: Semi-automatic isovalue determination: original vortical region (left); decomposition with color coding by isovalue (middle) and $\epsilon = 5\%$; the respective vortex core lines (right).

posed by Cornea et al. [29] has the property of producing smooth curves and was thus the method of choice. Clearly, not all skeletonization algorithms produce smooth curves. Since any isoskeleton can be smoothed after extraction, our algorithm is essentially independent of the actual skeletonization algorithm.

In the following, the skeleton-based algorithm is compared to other vortex core line extraction methods: the algorithm of Stegmaier et al. [165] is a variation of the approach by Banks and Singer [8], which constructs a vortex core line from initial seedpoints, following the vorticity vector. It is therefore intrinsically incapable of modeling vortex bifurcations, since the vorticity vector field does not reproduce the bifurcations of the core line. The same holds for the algorithm by Theisel et al. [170]. In Section 4 of their paper they model various notions of bifurcation in space-time. However, a model for sole bifurcation in space is not given and can therefore not be detected by their method. Furthermore, the algorithm by Stegmaier et al. may erroneously identify a coherent vortical region as disconnected. Finally, the parallel-vectors approach by Peikert and Roth [121] for finding minimal lines is sensitive to noise due to the evaluation of the Hessian (see also Section 7.3). Figure 8.8 compares the proposed skeleton-based approach [151], the approach of Stegmaier et al., and the parallel-vectors operator for a specific vortex in experimental wake flow behind a cylinder at Reynolds number 540 (measured by tomographic PIV).

To illustrate the effect of the semi-automatic λ_2 isovalue determination, the skeleton-based algorithm is executed with ϵ at 0% and at 5% of the absolute range of λ_2 in the dataset. Figure 8.9 shows that the ϵ parameter gives control over the degree of decomposition of vortical regions. While this user-guided solution is a step in the right direction, a fully automated procedure of isovalue determination would be preferable and could be investigated in future research.

In this approach, the accuracy of the extracted vortex core line is a function of the initial prediction by the isoskeleton. Thus, good results cannot be guaranteed for arbitrarily bad isoskeletons. Figure 8.9 demonstrates this issue. On the bottom of the right structure a part of the vortex core line cannot be detected due to a missing isoskeleton. Without going into further details of the particular skeletonization approach, please note that skeletonization is still an active field of research and that there is a number of

competing approaches (of which very recent work by Hassouna and Farag [57] seems promising). Furthermore, provided that a correct and smooth isoskeleton is extracted, the core line prediction and correction process might be iterated in order to improve the accuracy of core line computation (by using the extracted core line as the prediction for the next iteration). While we, the authors of [151], leave a theoretical proof of convergence as future work, our initial results give reason to believe that such a conjecture is based on solid grounds.

8.3 Visualizing Shear Stress

Shear layers are generated by viscosity effects, e.g., in boundary layers and free shear layers, and build another feature that can be extracted directly from a flow field. According to Section 7.1, the velocity gradient tensor can be decomposed into two parts: one representing the rotation, the vorticity tensor, and one containing the shear stress, the strain rate tensor. Shear stress describes, as the name already implies, the rate of shear that impacts on a fluid particle. Considering the strain rate tensor of Eq. (7.2), two types of forces can impact on a fluid particle: the strain forces and the shear forces. In practice, the measure of shear is of prime importance because it can be used for detecting boundaries within a fluid data set, e.g., the boundary between a high and a low velocity region. Furthermore, shear layers play an important role in the generation and interaction process of vortices, which is discussed later in Section 9.3.

Considering again the strain rate tensor S, it is more desirable to work with one scalar value reflecting the shear stress than with a more complex tensor. Both approaches by Haimes et al. [54] and Meyer [112] try to consider primarily the fraction of shear stress of the strain rate tensor by applying rather similar techniques: Haimes et al. first computed the deviatoric shear stress tensor in (Eq. (7.10)) in order to neglect hydrostatic stresses, i.e., the strain forces that only change the volume of a fluid particle. Then the second invariant (7.11) is used, simply based on the fact that it provides a scalar value that is independent of the selected coordinate system. In contrast to Haimes et al., the approach by Meyer directly works on the strain rate tensor S. Based on the fact that we are mainly interested in the shear parts of the tensor, he built a function $\phi = S_{12}^2 + S_{13}^2 + S_{23}^2$ that only considers the entries of S representing the shear in a direction. However, ϕ is not invariant under rotation, i.e., changing the coordinate system would lead to different results. He found out that, fortunately, in the second invariant of S of Eq. (7.12) ϕ is the dominating term and, therefore, the invariant can be used to represent shear stress.

Although some mathematical formulations of shear layers have been proposed, an appropriate visual representation is another open question. From the visualization's point of view, a common visualization of a scalar field is by direct or indirect volume visualization (see Chapter 1.4). Considering shear layers, this leads to a conflict because indirect volume visualization would display shear layers as flat but volumetric regions. The physical understanding is, however, that shear layers are objects of codimension 1.

This section describes how shear layers can be displayed according to their physical understanding. The discussion only considers the graphical representation of shear layers in a steady vector field and serves as staring point for the discussion

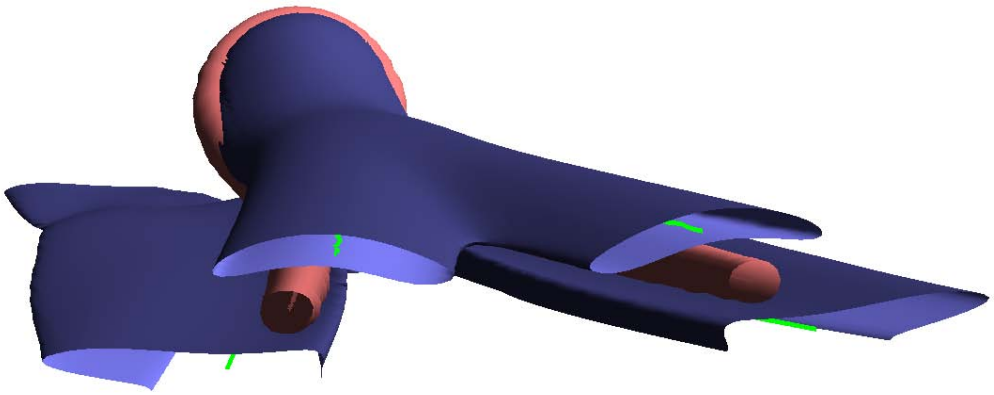


Figure 8.10: Ridge lines of I_2 inside an I_2 isosurface (green) surrounding a hairpin vortex (red). It shows that the I_2 function is not monotonic what leads to multiple, sometimes even parallel, ridge lines inside a shear layer.

in Section 9.3, which clarifies the role of shear layers in 3D time dependent flows. In contrast to the work of Sahner et al. [145], where vortex cores and shear layers result from a topological consideration as 0D, 1D and 2D structures, this work defines vortex cores as 1D and shear layers as 2D structures of local extrema [37; 56; 121]. In order to discuss the rendering of shear layers in general, for the following discussion I_2 replaces the two shear layer identification criteria S_H [54] and S_M [112] (see Eq. (7.11) and Eq. (7.12)). Therefore, I_2 can be substituted by S_M and S_H anytime.

8.3.1 Shear Layers vs. Vortices

In 3D, shear layers are, similar to vortex regions, defined as 3D regions where the boundaries are given by the respective shear stress criterion. As already discussed in Chapter 7, for both criteria the boundaries can be represented by isosurfaces of isovalues $I_2 = 0$ and $\lambda_2 = 0$, respectively. However, one of the main differences consists of the geometrical shape of the regions: while vortex regions are usually tubular, shear layers are, as the name already implies, more flat and stretched on a plane orthogonal to their medial axis. Furthermore, shear layers might be influenced by roll-up events, that lead the isosurfaces' shape to become concave and folded (discussed later in this thesis in Chapter 9). Figure 8.10 shows both structures: the more tubular vortex is drawn in red while the folded shear layer is represented in blue.

Taking into account the more complex geometrical shape of shear layers compared to vortex structures, a more detailed discussion concerning the behavior of I_2 and λ_2 is required: in most cases it is expected that a vortex structure has exactly one vortex core line, i.e., one axis of rotation. Since most recent vortex criterions define a vortex core line as connected minima of pressure (or in the case of λ_2 as a connection of λ_2 minima) on planes $\mathcal{P}$ perpendicular to the vortex core line, the pressure is assumed to increase monotonically on $\mathcal{P}$. This is except for bifurcations [151], i.e., regions where a vortex branches and, therefore, more than one vortex core line occurs inside a λ_2 isosurface. In contrast to λ_2, I_2 is not monotonic on $\mathcal{P}$, i.e., note that more than

Figure 8.11: Line vs. sheet: while a line serves as an appropriate simplification of a vortex (vortex core line), shear layers are represented as sheets of maximal shear.

one extremum line might occur inside a $I_2 = 0$ isosurface. This fact makes it rather difficult to consider shear layers in a similar simplified manner as vortices, i.e., as 1D representation. Moreover, the physical understanding of a vortex region is that particles rotate around one axis while inside shear regions the particles are deformed. Fig. 8.11 shows an example: in the case of a vortex a 1D representation seems to be sufficient. But in the case of shear layers the stretching of particles is more planar since there must exist a sheet where the particles' deformation is maximal.

In this thesis, volumetric shear layers are represented according to their 2D *shear sheet*, which is defined as 2D height ridge of the underlying shear criterion. In detail, each shear sheet represents the 2D manifold of highest strain inside a shear layer.

8.3.2 Introducing Shear Sheets

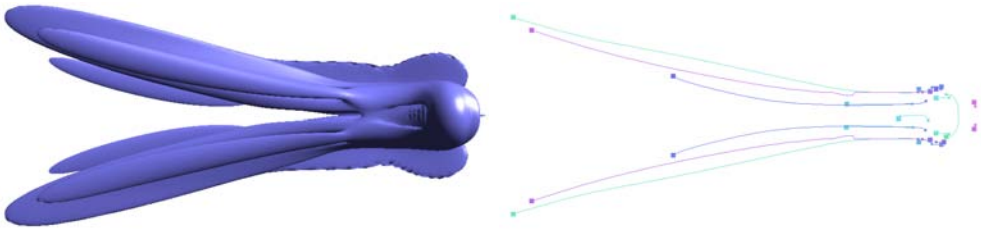
The following subsections describe the formal definition and computation of the shear sheets. The visualization of shear layers is discussed in Section 8.3.3.

Formal Definition

The formal definition of height ridges is followed according to Eberly [37] (further discussions on height ridges can be found in [56] and [121]): let f be a smooth function in a 3D domain, ∇f its gradient, and $\mathcal{H}f$ the Hessian; since the Hessian is real and symmetric, it has three real eigenvalues $\eta_1 \leqslant \eta_2 \leqslant \eta_3$. Then a position $\mathbf{x}$ is on a d-dimensional height ridge in 3D if

$$\begin{aligned} \nabla f(\mathbf{x}) \cdot \boldsymbol{\epsilon}_i &= 0 \text{ for all } i = 1, ..., 3-d \\ \text{and } \eta_{3-d} &< 0, \end{aligned} \tag{8.1}$$

where $\boldsymbol{\epsilon}_i$ denotes the $i-th$ eigenvector of $\mathcal{H}f$. Setting the dimension d of the height ridge to 1, the formal definition of a ridge line is achieved, where for each point $\mathbf{x}$ on it the gradient ∇f needs to be colinear to the eigenvector $\boldsymbol{\epsilon}_3$. This leads $\mathbf{x}$ to be an extremum on the plane spanned by $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$. The second condition of Eq. (8.1) states that both eigenvalues of the plane's tangent vectors need to be negative, i.e., that $\mathbf{x}$ is a maximum on the plane. In order to obtain 2D shear sheets, these conditions are relaxed in a way that $f(\mathbf{x})$ is only required to be a maximum in the direction of the plane's normal. Setting $d = 2$ in Eq. (8.1), the condition $\nabla f \cdot \boldsymbol{\epsilon}_1 = 0$ is achieved with $\eta_1 < 0$. In fact, this tells us that a shear sheet, i.e., $f(x) = I_2(x)$, is perpendicular to the eigenvector of the smallest eigenvalue η_1. Since η_1 is supposed to be negative, we know that $\mathbf{x}$ is a maximum in the direction of $\boldsymbol{\epsilon}_1$, which can be considered as the shear sheet's normal.



(a) (b)

Figure 8.12: Height ridges of I_2: (a) shows the I_2 isosurface of a shear layer, (b) the respective skeleton. For the skeleton, the largest principal axis ϵ_3 was used. Color-coding has been applied for a more precise distinction of the different 1D height ridges.

Computation

The shear sheets are computed in two stages: (1) the ridge lines of I_2 are computed; (2) each ridge line is used as seed line for the shear sheet computation. Computing the ridge lines first is reasonable because

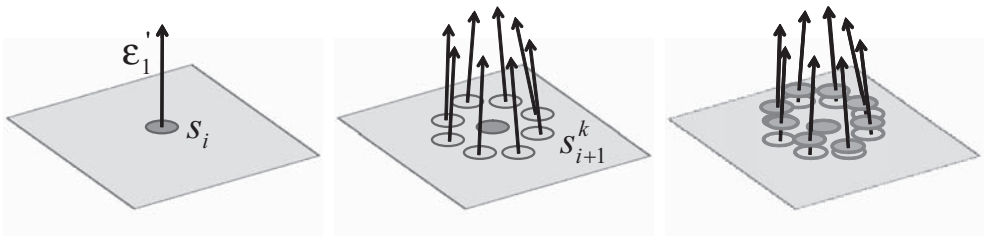
- the local maxima on the lines provide more seed points than 3D maxima of I_2,
- the condition of shear sheets is weaker than for ridge lines. Therefore, it is guaranteed that ridge lines are a subset of shear sheets,
- the lines can be used for tracking (see Chapter 9).

As already mentioned above, shear layers have a global center–the shear sheet–where shear stress is maximal. This means that though the I_2 isosurface encloses a 3D region of high shear stress, there exists a global center inside each shear layer. Adopting the classification of shear layers according to the taxonomy proposed by Jiang et al. [68], where vortices are classified as global features due to their global rotation axis, shear layers can also be considered as global due to their global sheet of maximal shear.

Therefore, the advantages of global and local methods are exploited by combining them in a predictor-corrector manner. The used global 1D line-based method is supposed to be more concise than a local 3D region, while the local I_2 method is Galilean invariant, what makes it immediately applicable for moving shear layers. The

Algorithm 5 Computation of I_2 ridge lines.

```
void computeShearRidgeLines()
{
for each I2 maximum x0
   while ridge line grows
      compute eigenvector ε3' of H(I2) at xi
      integrate ε3' to predict next position xi+1'
      get ε3 at xi+1' to compute plane P ⊥ ε3
      corrected position xi+1 is the maximum on P
   endwhile
endfor
}
```



(a) (b) (c)

Figure 8.13: Shear sheet generation process with respect to Algorithm 6: (a) first, the seed point $\mathbf{s}_i$ is drawn with its surface normal $\boldsymbol{\epsilon}_1'$. Furthermore, the plane $\mathcal{P}$ is created perpendicular to the surface normal; (b) second, n points $\mathbf{s}_{i+1}^{k\,\prime}$ are distributed on $\mathcal{P}$ around $\mathbf{s}_i$; (c) third, the new points are corrected by searching for maxima of I_2 along the normal vectors $\boldsymbol{\epsilon}_1^k$.

predictor-corrector algorithm of Section 7.3.2 is modified in a way it becomes applicable for the detection of ridge lines inside shear layers. Algorithm 5 shows the corresponding pseudo code. Note that $\boldsymbol{\epsilon}_3$ is the eigenvector to the largest eigenvalue η_3. The algorithm is performed as long as there is a local minimum of I_2 which is not located in an already detected shear layer. Starting from this I_2 minimum, the ridge line grows by integrating the eigenvector $\boldsymbol{\epsilon}_3'$. The position found is then considered as a location close to the actual ridge line, called predicted position. In a next step, the eigenvector $\boldsymbol{\epsilon}_3$ is computed again at the predicted position in order to find the actual I_2 minimum on a plane $\mathcal{P}$ perpendicular to $\boldsymbol{\epsilon}_3$. This corrected position is considered as the next point on the ridge line. Figure 8.12 shows the ridge lines of an I_2 isosurface.

Since the isosurfaces of I_2 are always closed, except for regions close to a wall, there exists at least one maximum on a plane orthogonal to the ridge line, i.e., there always exists at least one ridge line. Therefore, ridge lines provide a good basis for the computation of shear sheets. As discussed above, the sampled ridge lines are used as seed points for the following shear sheet computation. Here, the predictor-corrector algorithm of Banks and Singer [8] is adapted a second time in order to deal with normal vectors instead of tangents: according to Eq. (8.1), a shear sheet is orthogonal to the direction of the smallest principal direction of I_2. This defines the eigenvector $\boldsymbol{\epsilon}_1$ to be the normal of the shear sheet. In fact, the predictor-corrector algorithm can now be applied in an inverse order as it is used to compute the ridge lines. Algorithm 6 shows that case.

The main difference is that the prediction step predicts the positions on a plane in order to correct them in the following along a line. In practice, eight points circular around a seed point $\mathbf{s}_i$ are distributed (Figure 8.13 (a)). Then, the positions are corrected by iteratively searching the maximum of I_2 along the respective eigenvector $\boldsymbol{\epsilon}_1^k$ (Figure 8.13 (b) and (c)). In order to obtain an appropriate sampling of the shear sheet, the deflections of the eigenvectors $\boldsymbol{\epsilon}_1^k$ to the seed point's normal $\boldsymbol{\epsilon}_1'$ are taken into account. This can be considered as the curvature of the shear sheet that is used to find an optimal distance between the new particles and the seed point. Actually, for each new point the angle $\cos(\alpha) = \boldsymbol{\epsilon}_1' \cdot \boldsymbol{\epsilon}_1^k$ is compared to a threshold in order to decide if the distance to the seed point should be selected larger or smaller. This results in different curvature-dependent distances between each of the eight points and

Algorithm 6 Computation of shear sheets.

```
void computeShearSheets()
{
for each sample on the I_2 ridge line s_0
  while X is not empty
    compute eigenvector ε'_1 of H(I_2) at s_i ∈ X
    create plane P ⊥ ε'_1
    distribute n points s^k_{i+1}' on circle on P around s_i
    compute search vector ε^k_1 at each new point s^k_{i+1}'
    corrected positions s^k_{i+1} are the maxima of I_2 along ε^k_1
    if any position of s^k_{i+1} is too close to an already existing point
      remove old seed point s_i of X;
      break;
    else
      add positions s^k_{i+1} to set X and remove old seed point s_i of X
    endif
  endwhile
endfor
}
```

the seed point. Furthermore, the new particles must be tested for their distance to all other points to avoid oversampling, which is simply done by traversing a kd-tree holding all positions of the particles to compare the distance between them and a new particle against a threshold. On the other hand, the surface might be undersampled if the distances between the respective points become too large. At the time the points $\mathbf{s}^k_{i+1}{}'$ are seeded, the final distance between the points is not known—it is only possible to estimate if the points are diverging or converging. This is be done by computing α. Furthermore, because its dependance on the second order derivative of I_2, which is also the result of a derivation of the original vector field, this method might suffer from noise. Therefore, the eigenvectors $\boldsymbol{\epsilon}_1$ are filtered.

8.3.3 Visualization

Applying the algorithm described above results in a number of points representing the shear sheet. The shear sheets are visualized as illuminated surfels, using the eigenvector $\boldsymbol{\epsilon}_1$ as surface normal. In Figure 8.14 the I_2 shear sheets are drawn as green surfaces surrounded by the black wireframe representation of the I_2 shear layers. Phong illumination can easily applied by using the eigenvector $\boldsymbol{\epsilon}_1$ as surface normal. This can be done very efficiently by pre computing the eigenvectors and either storing them into a 3D texture (which are fetched while drawing the surfels) or attaching them to the vertex position. Both possibilities have their advantages and disadvantages: using a 3D texture requires only the vertex positions to be send to the GPU—the texture coordinates can be computed on the GPU by only one division. However, a 3D texture is necessary, which requires a lot of memory. Attaching the surface normal to the vertex, no memory-consuming texture is necessary, but the amount of data sent to the GPU is doubled. Rendering is performed by drawing point sprites, which are always aligned perpendicular to the view vector.

Figure 8.15 shows another example. The green shear sheet is located above and

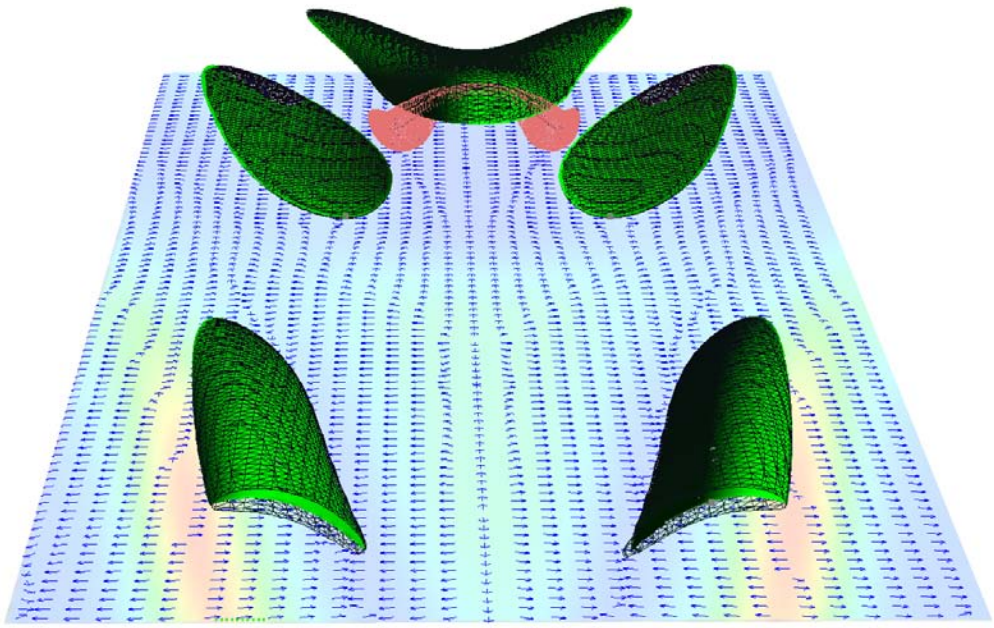


Figure 8.14: The graphical simplification according to the physical understanding. The vortex core lines is represented as a black 1D structure, surrounded by a red λ_2 isosurface. Shear sheets are represented as green 2D structures covered by a black I_2 isosurface. All isosurfaces are drawn as wire frame representation.

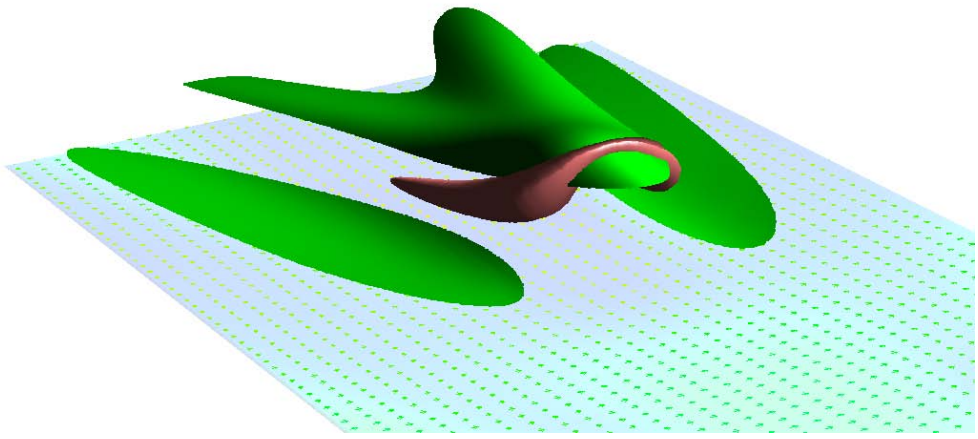


Figure 8.15: Closer view of the rear part of Figure 8.14. It is noticeable that the green shear sheet interacts with the red vortex: the shear sheet covers most of the vortex structure and possesses a bulge close to the vortex head. This plays a crucial role for the temporal evolution of the vortex and the shear layer.

below the red λ_2 vortex structure. The shape of both structures display their interaction. The importance of shear layers in vortex dynamics as well as their temporal evolution are discussed in Chapter 9 in this thesis.

CHAPTER

9 FEATURE TRACKING

In the previous Sections 8.2 and 8.3 the computation of 1D and 2D skeletons of vortices and shear layers is discussed. Considering these skeletons from a point of view that only takes into account steady vector fields, they can be used for a segmentation of the respective feature [165]. However, in practice, time-dependent vector fields are of much higher interest than single time steps because most flows underly a temporal variation.

A typical example is the exploration of wind flows around vehicles in wind tunnels. As a goal of the development process it is of high interest that a future vehicle runs safely and, as a more recent topic, that it does not consume too much gas. For this purpose, coherent structures might be extracted and investigated for their temporal development. Vortices, certainly, build the most interesting coherent structure, since they are considered as a measure of turbulence. In turn, the turbulence is tightly linked to the roadholding of a vehicle as well as to its gas consumption (vortices behind the rear-view mirrors). Of course, this is only one specific example, but it makes it interesting how these flow features change temporarily in their kinematic properties, like size, shape, energy, and strength.

In this chapter, the mapping of flow features at time t to their corresponding structures at time $t + \Delta t$ is discussed (tracking). Starting with a more general algorithm for the tracking of vortex structures and shear layers, the discussion will give an in-depth analysis of the interaction and dependency between vortices and shear layers considering vortex dynamics. Again, the importance of particle tracing methods is demonstrated, particularly when time-dependent flows are considered. The following discussion bases partially on the results presented in the paper [148] and contains parts of the original text.

The tracking algorithm for vortices and shear layers has been developed in collaboration with Kudret Baysal and Ulrich Rist from the Institute for Aerodynamics and Gasdynamics at the University of Stuttgart, who must be credited for their contribution in terms of the theoretical background in fluid dynamics used for the following methods.

9.1 Particle-based Feature Tracking

The extremum lines of feature criteria like λ_2 (vortex core line) or I_2 (lines of maximal shear) play an important role for the following tracking algorithm. A theoretical view on these 1D height ridges is given in Chapter 7; methods for their computation are presented in Sections 8.2 and 8.3.

9.1.1 Sampling the Extremum Line

The reliability of extremum lines of a feature function for tracking is best shown by means of an example: let us consider a particle at an arbitrary position inside a vortex region, i.e., inside a $\lambda_2 = 0$ isosurface. Then, its motion strongly depends on the selected reference frame: in the first case, the reference frame moves along with the vortex structure; in the second case, a static reference frame is used. While in the first case the motion of the particle is observed as a pure rotation, the same particle describes a convection in the static reference frame. In detail, the motion of the particle inside a vortex region consists of the vortex structure's motion and the rotation around the vortex axis. Therefore, inside a vortex region the naive use of the velocity vector for tracking might lead to a particle leaving the vortex region already after a short time Δt, particularly if it is located nearby the vortex boundaries. This is due to the change of the spatial extents of the vortex region along time, and the motion of the particle which might depart more and more from the vortex center. In order to enable vortex tracking two conditions must be fulfilled: (1) the deviation of the particle from the vortex center as a result of rotation should be as small as possible; (2) an appropriate sampling of the vortex structure must be maintained.

The only spatial region which guarantees a particle's motion without the rotation around the vortex axis, is the vortex core line. Furthermore, the vortex core line provides the following advantages: (1) it can be regarded as a kind of medial axis of a vortex, i.e., placing the particles on the core line provides an appropriate sampling of the structure; (2) in all likelihood the particles will not leave the vortical region after integration. This is due to the definition of the vortex core line, namely the connection of local λ_2 minima, which is expected to change least from one time step to the next because of the temporally smooth behavior of λ_2. In other words, in all likelihood, the local minima at time t are expected to remain negative at $t + \Delta t$ and, therefore, to lie inside the vortex boundaries.

Conditions (1) and (2) can also be applied on other criteria like I_2 without any restrictions. Here, an extremum line of I_2 defines the line of maximal shear; regardless if the function is monotonic or not, the extremum line lies inside the shear layer and provides the most certain area for seeding particles for tracking. This is due to the required temporally smooth behavior of the criteria, which leads the particles on an extremum line to exceed the isovalue of the function last (for λ_2 and I_2 the isovalue is 0).

9.1.2 Predictor-Corrector Tracking Algorithm

Tracking works, similar to the vortex core line detection, in a predictor-corrector manner. In order to track a flow feature, the extremum line is sampled, where for each sample point a particle $\mathbf{p}_t$ is integrated along the velocity field. For this purpose, again the ODE for Lagrangian particle tracing in Eq. (4.7) is evaluated, usually applying a higher order integration scheme, like fourth order Runge-Kutta (see Section 2.4). Once the new position $\mathbf{p}'_{t+\Delta t}$ is predicted, the nearest structure is determined. This is done by measuring the particle's euclidian distance to all extremum lines. Furthermore, the particle is tested if it is located inside the extremum line's surrounding isosurface. If this test succeeds, the new particle's position is corrected to the nearest particle on the

extremum line $\mathbf{p}_{t+\Delta t}$. Here, each feature structure has its own counter which holds the number of particle hits and the respective source feature ID from where the particles were emitted. In order to maintain an appropriate sampling rate, the extremum line found is resampled for further tracking.

Although it is expected that most particles from one structure hit the same structure at $t+\Delta t$, it might happen that more than one structure were hit by particles holding the same source feature ID. This event can be ascribed to close feature structures or topological changes of a feature along time. For example, a vortex can be split or merged with another vortex structure as described in Section 8.2.3 during a vortex reconnection process. Note that the correctness of the extremum lines depends strongly on the underlying detection method. Although the method of Stegmaier et al. [165] provides a high quality of the detected vortex core lines, it is not guaranteed that all actually connected structures are detected as connected. As already mentioned in Section 8.2.3 in this thesis, there exists no work guaranteeing the topological correctness of vortex core lines (as well as for lines of maximal shear). This issue is solved by taking into account all features hit by any particles, except for the features hit by a relatively small number of particles with respect to the number of emitted particles or the length of the feature hit by the particles. This prevents us from discarding of possibly important feature structures and is expressed by the following two conditions which must be fulfilled:

$$\frac{n_i^{[t]}}{n_o^{[t]}} \geqslant a \quad \text{and} \quad \frac{n_i^{[t]}}{n_o^{[t-1]}} \geqslant b, \tag{9.1}$$

where $n_i^{[t]}$ stands for the number of incoming particles and $n_o^{[t]}$ for the outgoing particles, respectively. The first condition claims that a structure was hit by at least a times its length, the second condition ensures that an appropriate number of emitted particles hit the current structure. In the implementation of [148], the best results are obtained by setting both ratios to $5 \sim 10\%$. Once a feature is hit, its ID is added to a list holding all the necessary information which is used for further tracking.

9.1.3 Graphical Representation of Tracked Features

For the graphical representation of vortices and shear layers the representation according to the work of Stegmaier [165] has been adapted. In the original version, each vortex core line is assigned to its respective λ_2 isosurface representing the 3D vortex region. All the data is stored in a scene graph that enables the user to pick the vortex structures individually. For tracking, this scene graph has been extended to: (1) hold also shear layers and their corresponding extremum lines; (2) enable the tracking of extremum lines.

The graphical representation builds the end of a user-driven input sequence:

1. A structure is marked for tracking: the user is able to pick an arbitrary isosurface. Here, it doesn't matter whether it is a vortex or a shear layer.

2. The user applies tracking of the selected structure: the particles of the respective extremum lines are integrated along the vector field. The corresponding structures of the next time step are assigned to the selected structure. Here, it also

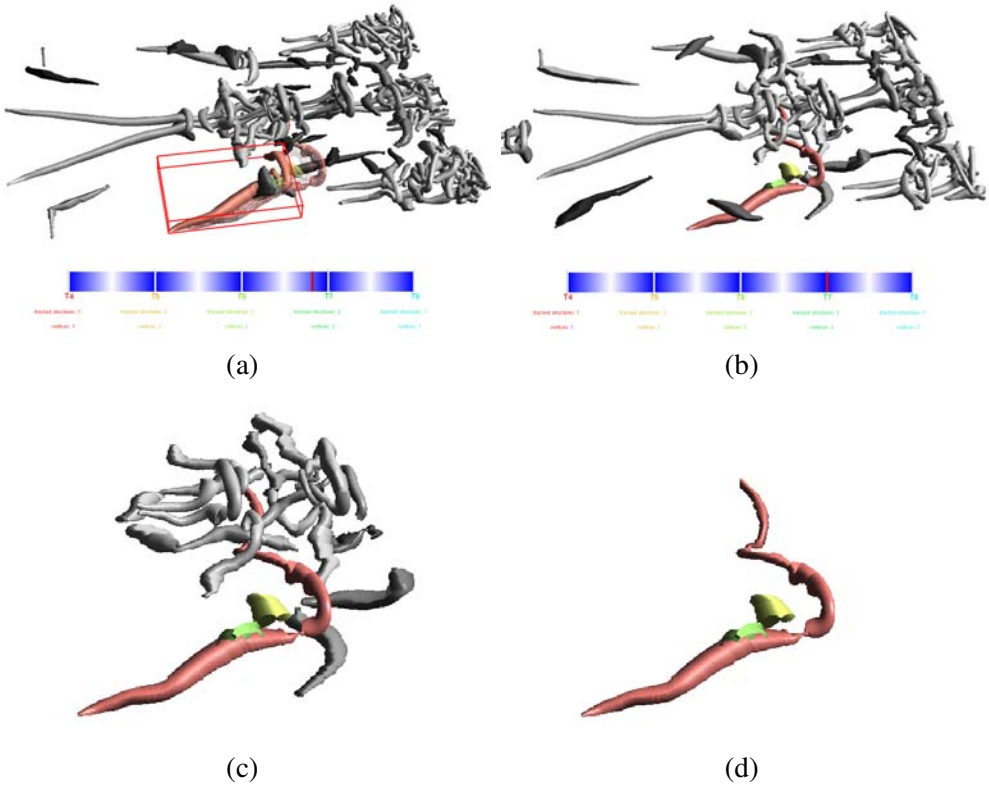
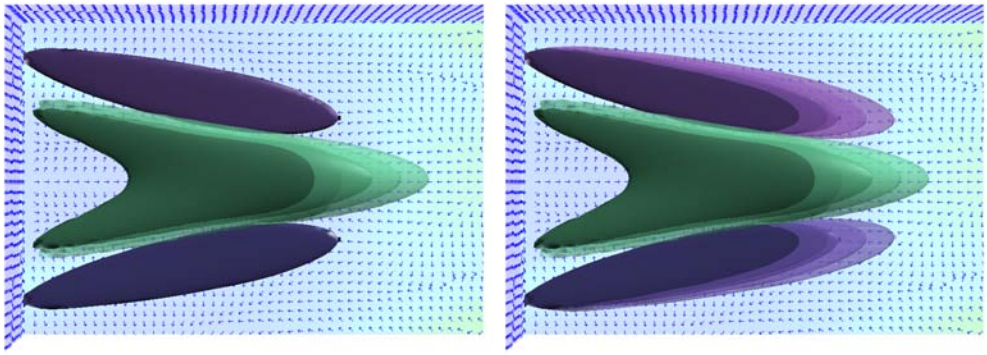


Figure 9.1: A focus and context visualization of a simulation of a laminar-turbulent transition on a flat plate [5]. The flow moves from left to the right: (a) a vortex structure is selected for tracking at an arbitrary time—here $t = 4$, showed by the blue timeline. The wire frame representation shows the forward motion of the vortex; (b) the context is switched to a particular time of interest; (c) due to the underlying scene graph the context can be scaled down; (d) only the tracked vortex structure is displayed. The other vortex structures created due to temporal split events are clearly perceptible.

doesn't matter if the structure hit by the particles is of the same feature type as the selected structure or not.

3. The result is visualized: a combination of a 3D representation of the scene and a time line enables the user to observe the temporal development of the selected structure. By moving the cursor along the time line, the position and the shape of the selected feature is represented according to its attributes to the selected time step.

The usefulness of the proposed system is demonstrated in Figure 9.1. This sequence considers a simulation of a laminar-turbulent transition on a flat plate, where only λ_2 vortex structures are drawn. This example fully exploits the underlying scene graph: first a vortex is selected and tracked. By using the time line, the user decides to explore the vortex structure at time $t = 7$ in more detail. Therefore, he changes the context to $t = 7$ in order to gain knowledge about the state of the whole data set. Then the context is decreased by fading out uninteresting vortex structures until only



(a) (b)

Figure 9.2: Tracking of a single shear layer (a) or multiple shear layers (b): the user is able to influence the selection of the structures exploiting the underlying scene graph. The motion of the structures is perceptible by alpha-blended isosurfaces. The user is further able to manipulate the temporal visualization using the time line showed in Figure 9.1 (a) and (b).

the selected vortex structure remains.

Figure 9.2 shows another example. Here, shear layers are tracked. Keep in mind that the visualization system is able to track an arbitrary number of features.

9.2 Temporal Development of Vortices

While in the previous section the tracking of flow features is discussed in general, this section has its focus on the temporal development of vortex structures. For this purpose, vortex dynamics are taken into account, i.e., the interaction between different coherent structures (here: vortices). The basic idea is that the temporal development of a single vortex structure mainly depends on the external influence of other vortex structures. Therefore, a generated vortex can be increased or decreased in its strength by neighboring vortex structures. The knowledge about these influences is crucial because often so-called critical vortices are only critical as a result of the external influence.

In the following, the tracking algorithm of Section 9.1 is adapted in a way it considers vortex dynamics. Thereby the tracking process is used to find the moment of strongest perturbation, i.e., the time and the location where a selected vortex structure is perturbed most by another vortex structure.

9.2.1 Vortex Dynamics

The consideration of vortex dynamics enables a more detailed insight into the temporal behavior of coherent structures. In detail, vortex dynamics connect the evolution and interaction of coherent structures to their spatial extents. A simple method for the consideration of vortex dynamics consists of the analysis of dynamical values, e.g.,

kinetic energy or enstrophy:

$$E_{kin} = \frac{1}{2}\int_V \boldsymbol{v}\,(\boldsymbol{x},t)^2\,d^3x, \tag{9.2}$$

$$E_{rot} = \frac{1}{2}\int_V \boldsymbol{\omega}\,(\boldsymbol{x},t)^2\,d^3x, \tag{9.3}$$

where V represents the 3D domain and $\boldsymbol{\omega}$ denotes the vorticity. Thereby, the problem arises of a proper definition of the region of integration. In test cases with two vortex structures like collision of vortex rings [81], it is sufficient to integrate over the whole domain. In real flow fields containing hundreds of structures of interest, each of them needs a more specified definition of the integrated volume. Reasonable choices are the spatial extents of the vortical regions.

Another technique supports the investigation of the interactions between vortex structures and other flow features, e.g., shear layers or other vortices. Particularly the interaction between vortex structures is rather important, e.g., in 2D vortex merging [73] and in 3D collision of vortex rings [81]. The information obtained by identification, segmentation, and tracking of vortex structures facilitates an intensified study of the interaction between several vortex structures. The fundamental equation for the analysis of vortex dynamics is the Biot-Savart law, which computes the induced velocity field from vorticity:

$$\boldsymbol{v}_{ind}(\boldsymbol{x},t) = \frac{1}{4\pi}\int_V \frac{\omega(\boldsymbol{x}',t)\times(\boldsymbol{x}-\boldsymbol{x}')}{|\boldsymbol{x}-\boldsymbol{x}'|^3}\,d^3x'. \tag{9.4}$$

In general, a velocity field can be represented as the sum of rotational and irrotational parts. The irrotational part is determined by the boundary conditions. The rotational velocity, denoted as $\boldsymbol{v}_{ind}$ in Eq. (9.4), is completely determined by the vorticity field. By computing the velocity induced by the vorticity field of a single vortex structure S_1, the influence of this structure on its environment is determined. Furthermore, by applying Eq. (9.2) or Eq. (9.3) to the induced velocity field, where the integration domain is restricted to a vortex structure S_2, the influence of S_1 to S_2 is determined.

9.2.2 Taking into Account Vortex Dynamics

The influence of one vortex structure on another is described by two characteristic integral values: the kinetic energy and the enstrophy. The physical understanding is that the kinetic energy represents the translational part of influence and the enstrophy the rotational part of influence of one vortex structure to another. Both the kinetic energy and the enstrophy depend on a given velocity field and the derived vorticity, respectively, which need to be computed first. For this purpose, the Biot-Savart equation (Eq. (9.4)) is evaluated as described in Section 9.2.1 using the segmented λ_2 isosurfaces as an appropriate representation of vortex structures (see Section 9.1.3). Each vortex holds a pointer to a discrete grid containing the positions inside the structure which can be used to access and store the vorticity and the induced velocity field, respectively. This makes the evaluation of Eq. (9.4) rather simple. For each cell inside the destination vortex, it is integrated over all cells of the source vortex to sum up their

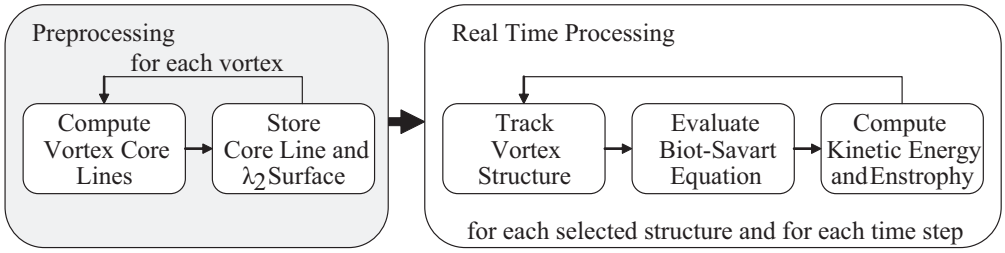


Figure 9.3: Program flow of the vortex tracking framework takin into account vortex dynamics: in the first stage the vortex core lines are computed in order to segment the vortex structures according to the λ_2 criterion. All the pre-computed structures are stored to provide a fast access during the second stage. In this stage, the structures are tracked and computed for their influence on each other. This stage can be interactively manipulated by the selection of the structures to be tracked.

distance weighted vorticity. The algorithm finishes when each of the destination cells is filled with a velocity vector.

In the next step, the resulting velocity field is used to compute the kinetic energy induced by the source vortex. The kinetic energy is computed according to Eq. (9.2). For computing the enstrophy in Eq. (9.3), first the vorticity of $\boldsymbol{v}_{\text{ind}}$ must be derived.

Figure 9.3 shows the overall program flow taking into account vortex dynamics, which is separated into two independent parts: the preprocessing, where the core lines and the vortex segmentation are computed, and the real time processing which consists of the tracking and the consideration of vortex dynamics. In contrast to the first stage, the second stage can be widely influenced by the user.

Though Algorithm 7 only considers the enstrophy, the computation of the kinetic energy follows the same scheme. Although, for the sake of simplicity, this example assumes the vortex structure to be selected at time $\mathrm{t} = 0$, the implementation allows for the selection of an arbitrary vortex structure at any time t. Note that due to topological changes along time, the variable *selectedVortex* may consist of more than one structure. In the first loop, all vortex structures are considered to compute their influence on the selected structure. Here, the selected structure itself is excluded. After the evaluation of the Biot-Savart equation, the enstrophy is computed and stored in the case it has the maximum influence. The variable *selectedVortex* represents the ID of the selected vortex at any time. Once the strongest items, i.e., the items with the highest effect, are computed for each time step, the overall maximum perturbation is obtained by comparing the results of the different time steps. Then, the tracking is started again at time $\mathrm{t_p}$, the time of the strongest perturbation, to deliver a representation in which the affected vortex as well as the strongest source of perturbation are included. This plays a crucial role for the following visualization.

9.2.3 Visualization

The example in Figure 9.4 extends the tracking method by taking into account vortex dynamics in order to find the time where the selected structure is perturbed strongest by another vortex. The computation results in a list containing the most impacting vortices

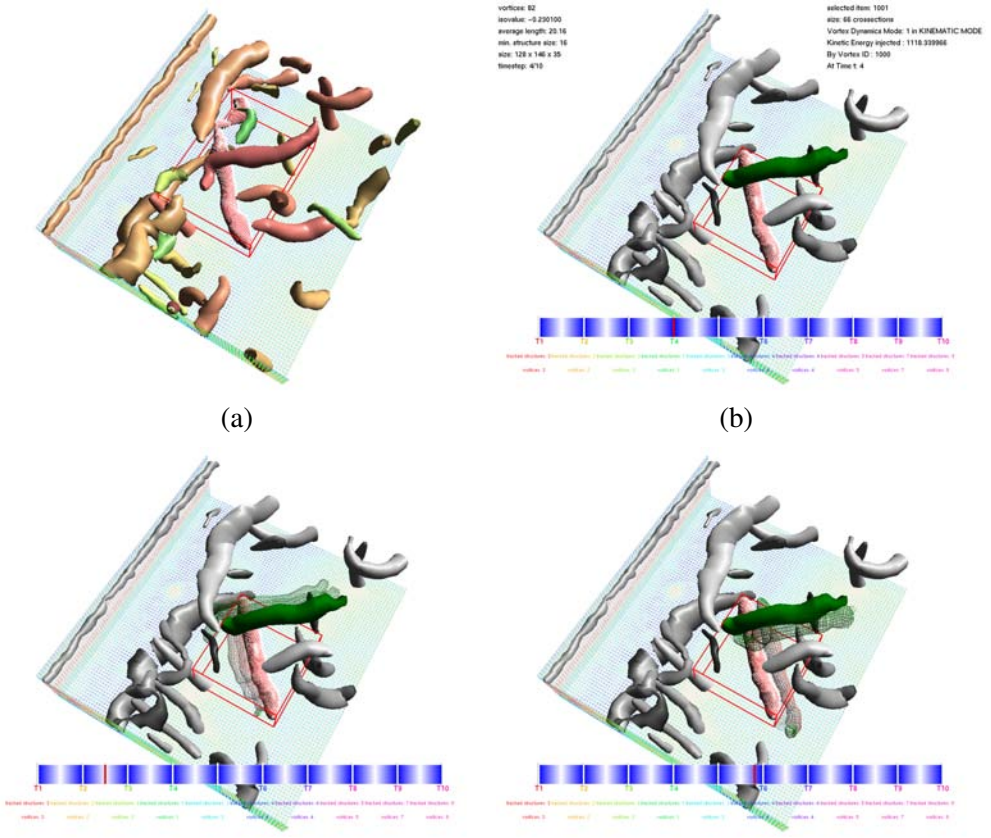
and the appropriate information in terms of kinetic energy and enstrophy, respectively. The results can be visualized with respect to both quantities, whereas the user is able to switch between both representations.

The resulting time t_p defines the time of the strongest perturbation, and therefore, the visualization is set to t_p for both the focus and the context. Figure 9.4 the detection and visualization of t_p is shown. By selecting a vortex at an arbitrary time (Figure 9.4(a)), the vortex with the strongest influence is computed, displayed as green vortex in Figure 9.4(b) at time t_p. Starting from here, both structures are tracked forward and backward to gain information about their origin and their further development (Figures 9.4(c) and (d)). Additionally, the detailed information about the detected and the selected vortex, e.g., the energy induced, IDs, length, etc. is displayed as text.

Although in this section only vortex-vortex interactions are discussed, vortex dynamics are not restricted to vortex structures only. Also shear layers have their contributions to the overall behavior of a flow, i.e., the Biot-Savart equation is also applicable to shear layers and other coherent structures. However, due the dependency of the Biot-Savart equation (Eq. (9.4)) on the vorticity, shear layers are expected to have less influence than vortices. A more detailed investigation on the influence of shear layers

Algorithm 7 Vortex tracking taking into account enstrophy.

```
void getstrongestPerturbation(id selectedVortex)
{
// loop over all time steps
for t=minTime to maxTime
// loop over all vortices except for the selected one
   vector indvel[lastVortex];
   int trackIDs[2];
   for i=firstVortex to lastVortex at time t
      indvel[i] = BiotSavart( i, selectedVortex);
      float en = computeEnstrophy(getVorticity(indvel));
      if max(en)
        setStrongestVortex(i, t);
      endif
   endfor
   selectedVortex = trackStructures( selectedVortex, t+1);
endfor
// get the time of strongest perturbations and vortex ID
strongestVortex = getStrongestPerturbation();
t_p = getTimeOfStrongestPerturbation();
selectedVortex = getSelectedVortexAtTime(t_p);
trackIDs[0] = selectedVortex;
trackIDs[1] = strongestVortex;
// then track all the structures
int resultIDs[2][maxTime - minTime];
for t=t_p to minTime
   resultIDs[0,1][t] = trackStructures(trackIDs, t - 1);
endfor
for t=t_p to maxTime
   resultIDs[0,1][t] = trackStructures(trackIDs, t + 1);
endfor
}
```



(a) (b)

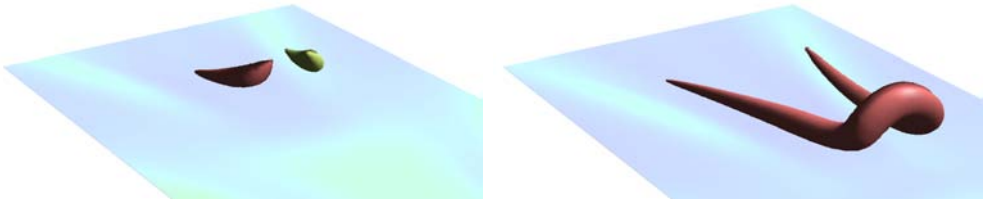
(c) (d)

Figure 9.4: The visualization of an experimental data set [147]. The data was measured using tomographic PIV and shows a circular cylinder wake at Reynolds $Re = 360$. The cylinder, which is located parallel to the left edge of the data set, is not visualized. In this example, vortex dynamics is considered: (a) initial situation at $t = 0$. All vortices are color coded according to their strength (weak = bright green, strong = red). The red bounding box gives feedback about the vortex selected for tracking; (b) the moment of strongest perturbation at $t_p = 4$. The representation has changed to time t_p, where the green vortex influences the selected structure (red); (c) backward tracking. The wire frame representation shows where the structures came from; (d) forward tracking. The further development is visualized.

to vortex structures is given in the next section.

9.3 Temporal Development of Shear Layers

So far, a technique for the representation of shear layers—the shear sheets in Section 8.3—as well as an appropriate tracking method (in Section 9.1) have been discussed. However, the role of shear layers in vortex dynamics has not been considered yet. This section should give more insight in the temporal development of shear layers and their interaction with vortex structures.



(a) (b)

Figure 9.5: Figure (a) shows two small vortex structures which had just appeared before they grew up to form a hairpin vortex in (b). However, it is not perceptible how the vortex had been generated.

Figure 9.5 builds the starting point of this discussion. It shows the development of the same vortex structure at different times. In Figure 9.5(a) the vortex suddenly appears; two unconnected λ_2 vortex structures are visible. In the previous time step (not shown), no vortices were perceptible at all. A few time steps later, the vortex has fully developed its shape to a hairpin-like vortex. This example illustrates the lack of information caused by considering only vortex structures.

In fluid mechanics, particularly in regions of overlapping of high vortical motion and shear stress, the quantity of shear stress became of interest in flow field feature analysis [24; 202; 87]. The theoretical background of shear layer-vortex generation can be found in the work of Ho and Huerre [59] and in the work of Williamson [201]. More recent publications considering the roll-up effect of shear layers to vortex structures are proposed by Lepage at al. [97] and Ponta [127] for instance. A combined visualization of shear stress and vortex regions in terms of combining the Q (see Section 7.2.1) and the M_z [55] criterion was also shown in the work by Sahner et al. [145]. The following discussion tries to clarify why the combination of those criteria is not sufficient in general.

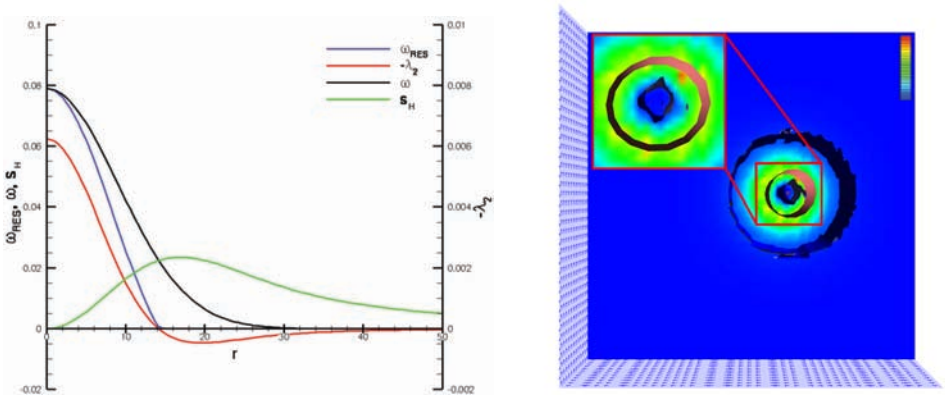
Based on the theoretical work above, an appropriate time-dependent visualization of shear layers is implemented. First the theoretical background of shear layers in vortex dynamics is discussed, followed by a discussion of an appropriate visualization of shear layers in time-dependent flows.

9.3.1 Shear Layers in Vortex Dynamics

The generation of new vortex structures is of major interest in the exploration of flows: first, the evolution and relation of shear stress and vortex strength is shown by means of the Oseen vortex; second, the influence of shear layers during vortex merge events is discussed; Third, the generation of new vortex structures as a result of the roll-up of a shear layer is considered.

Oseen Vortex

The Oseen vortex is a solution of the Navier-Stokes equation (see Section 3.2) and can be considered as a realistic test case for the investigations of vortex and shear layer



(a) (b)

Figure 9.6: (a) 1D representation of an Oseen vortex: the results of vorticity (black), the vortex identification criteria λ_2 (red) and TDM [87]–triple decomposition of the relative motion (blue)–and a shear layer identification criterion S_H (green); (b) 3D representation of the Oseen vortex. Shear stress is visualized as bluish $I_2 = 0$ isosurface and as color coding on the opaque surface. The red isosurface represents the vortex. The color of the vortex isosurface represents the vortex strength, i.e., the normalized vorticity $|\omega|$. Particularly in the zoomed view an overlap is visible.

identification criteria. It is defined by the cylindrical velocity equations:

$$\begin{aligned} u_r &= u_z = 0 \\ \text{and } u_\Theta &= \frac{\Gamma}{2\pi r}\left(1 - e^{\frac{r^2}{4\nu t}}\right), \end{aligned} \tag{9.5}$$

where Γ is the circulation, r the radius, ν the kinematic viscosity, and t is the time.

In Fig. 9.6(a) two different vortex identification criteria have been applied for the detection of an Oseen vortex: λ_2 [67] and the triple decomposition method of Kolar [87], where r is the radius from the vortex center, $\boldsymbol{\omega}$ denotes the vorticity, and $\boldsymbol{\omega}_{RES}$ the residual vorticity, i.e., the vorticity describing the ridged body rotation of a particle. In 2D it is defined as $\boldsymbol{\omega}_{RES} = \boldsymbol{\omega} - \boldsymbol{\omega}_{SH}$, where $\boldsymbol{\omega}_{SH}$ is the vorticity caused by pure shear. Here, both vortex criterions are combined with the shear layer criterion S_H by Haimes et al. [54].

The results of Figure 9.6 show clearly that there exists an overlap between vortex and shear layer regions. The left image shows a 1D representation while the right image shows the same vortex in 3D by repeating the 2D vector field along the z-axis. The opaque plane serves as cut plane where the color coding represents the I_2 values. Here, the color gradient goes from blue (weak) to red (strong). The inner red-colored tubular surface represents the vortex; here, a $\lambda_2 = 0$ isosurface is chosen. The dark-blue surfaces stand for the $I_2 = 0$ isosurface. Particularly Fig. 9.6(b) shows the overlap of vortex and shear layer regions. The overlap region is emphasized in a zoomed view on the upper left. It is noticeable that one $I_2 = 0$ isosurface surrounds the λ_2 isosurface while another $I_2 = 0$ isosurface is situated inside the vortex. The values for shear strain S_H and vortex strength λ_2 or $\boldsymbol{\omega}_{RES}$ in Fig. 9.6(a) demonstrate that also in vortex regions shear strain can have the same order as the maximum shear strain. Hence in

the overlapping regions both features are of interest for kinematical and dynamical investigations. For a more detailed discussion on this behavior we refer to the work of Wu et al. [202], Chakraborty et al. [24] and the discussions between both groups [23; 203].

Considering overlapping regions, the usage of a criterion which combines the detection of shear layers and vortex regions, like the Q [63] or the M_z [55] criterion, would deliver insufficient results. This is because the exclusive behavior of those methods; here, either a region is dominated by strain or by rotation. However, the physical correctness of overlapping vortices and shear layers in three dimensional flow fields can be derived from the incompressible velocity-vorticity form of the Navier-Stokes equation

$$\frac{\partial \omega}{\partial t} + (u \cdot \nabla)\,\omega = (\omega \cdot \nabla)\,u + \nu \nabla^2 \omega \; .$$

A more detailed consideration of the Navier-Stokes equation is given in Section 3.2. The first term on the right-hand side represents the vortex stretching, i.e., the amplification of vorticity by stretching; the second term represents the diffusion term. In detail, this term shows that vorticity in three-dimensional flow fields is not only affected by advection and diffusion like passive scalars; it is also affected by stretching of fluid elements [82]. Hence, if the information given by the shear layers were neglected, this would cause a deficit in the analysis of fluid dynamics.

Vortex Merging

The merging of two-dimensional co-rotating vortices is a well-investigated research topic in fluid dynamics, e.g. in meteorology and geology [131; 50]. In the following, it is shown that local vortex identification criteria are unreliable for the exploration of merging vortices and how the consideration of shear layers can help to solve this issue. Thereby, the overlapping of rotational motions seems to be the main reason for the inaccuracy of vortex criteria considering merging vortices. The example in Fig. 9.7 shows this behavior: here, two co-rotating vortices are considered; there exists one rotation around the respective vortex center, and one rotation is around the center of the connecting line between both vortex centers (global center). The overlapping of two rotational motions causes basically two misleading observations of the features.

- The follow-up motion behind the vortex structures induces regions of high vorticity values and low shear stress values. The commonly used vortex identification criteria define these regions as vortex regions.

- The vortex identification criteria locate two local extrema and a saddle point between the local extrema [87].

Figure 9.7 shows this behavior, where the colored lines illustrate the contour lines of S_H (the lines are colored by the value of S_H). Red-colored regions of λ_2 are no vortex regions. The vortex strength goes from orange (weak) to blue (strong). One possibility to solve this issue might consist of the usage of a rotating reference system [74] with an angular velocity similar to the angular velocity of the rotating vortices around the global center. Although the usage of the rotating reference system leads to a slight correction with respect to the identification of the follow-up motion regions as vortex

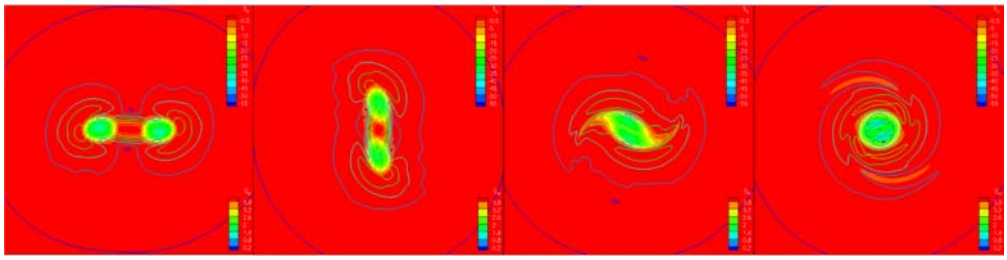


Figure 9.7: The interaction of two co-rotating Oseen vortices with the generation of a new vortex structure. λ_2 is represented by color coding, S_H by contour lines. The upper color legend represents λ_2; S_H is represented by the lower color legend.

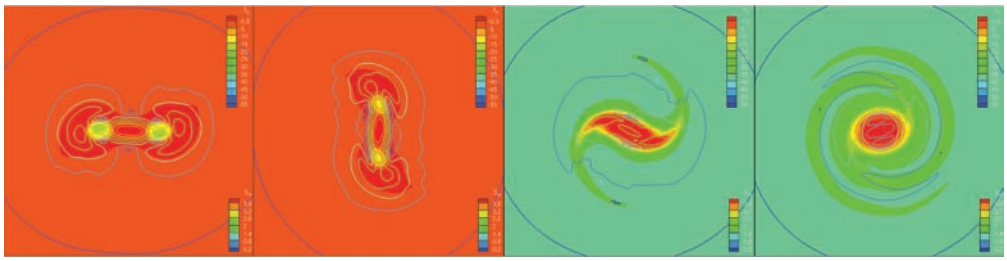


Figure 9.8: As same as figure 9.7 with the difference of a rotating refence system for the post-processing. Again, the upper color legend represents λ_2, whereas S_H is represented by the lower color legend.

regions (Fig. 9.8) and the unmeant saddle point is eliminated, it leads in the end to a new problem: at a later state of the merging process a new vortex is generated which dominates the flow field. This makes the flow field become more and more comparable to a solid body rotation. Then, the reduction of the angular velocity of the whole flow field with the angular velocity of a point in the rotational field causes a vanishing of the rotational motion in the quasi solid body rotation and to the observation of rotating outer regions which are detected as vortex regions (Fig. 9.8). A switching back to the original reference system with respect to the new vortex structure as proposed by Josserand et al. [74] has two disadvantages: (1) the analysis of the interactions of the flow field features is not straightforward; and (2) the positive effect of the new reference system is limited to a part of the whole vortex merging process—the time it becomes unsuitable is mostly unknown.

In contrast to vortex identification criteria [55], the used shear layer identification criteria are also invariant against rotation. Hence the investigation of the shear layer regions for the merging of co-rotating vortices is more reliable than the investigation of vortex regions. The investigation is straightforward, since no change of the reference system is required.

Vortex Generation

The relevance of the consideration of shear layers for the understanding of fluid dynamics can be observed by the influence of shear layers in the evolution of a new vortex structure. Here, the generation of the new vortex structure is comparable to the roll-up

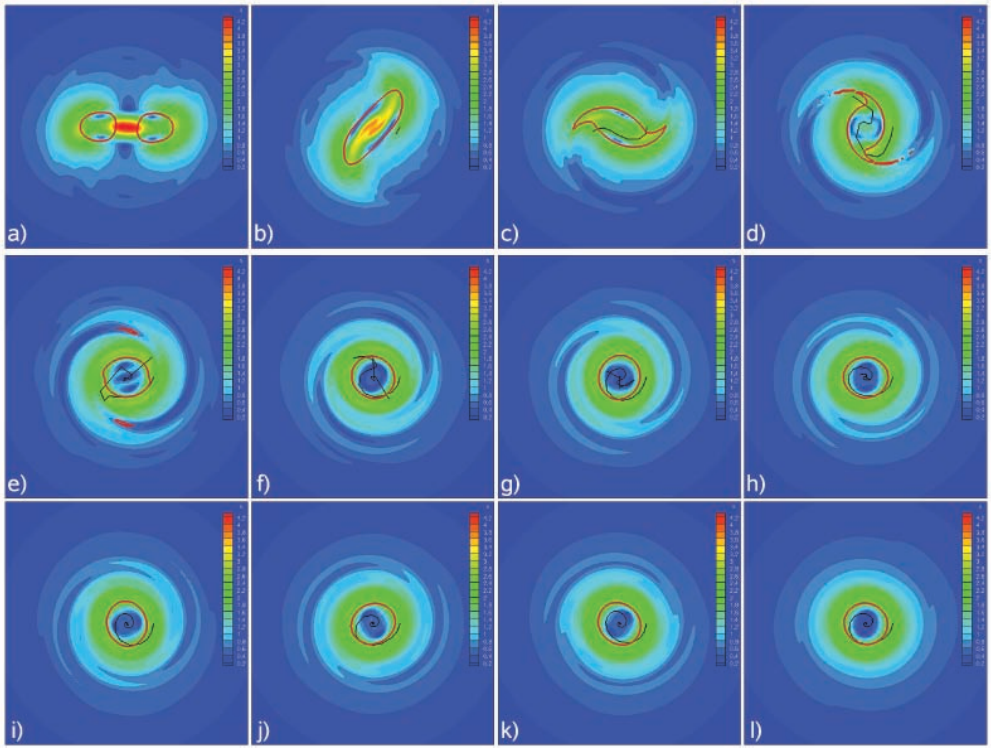


Figure 9.9: Shear layer visualizations for the merging of two co-rotating Oseen vortices. The red line represents the $\lambda_2 = 0$ isocontour.

of shear layer structures [59; 201], which is discussed later in this section. Figure 9.9 shows the different stages of 2D vortex merging. $\lambda_2 = 0$ is represented as contour lines and S_H as color coding. In Figure 9.9 (a) two initial Oseen vortices (regions inside of the red marked circles), which are surrounded by high shear stress regions, are shown. The maximum at the global center is red-colored, the color gradient for the shear stress goes from blue (weak) to red (high). Figure 9.9 (b) and (c) show three changes; the two vortex structures merged to one, the overlapping of high shear regions with the vortex, and the deformation of the surrounding shear layer. In Figure 9.9 (d) to (i) merging is completed and the new vortex structure become cylindrical while the regions of high shear layer regions inside the vortex region vanish. Finally, in Figure 9.9 (j) to (l) the shape of the new vortex structure becomes more and more comparable to a single Oseen vortex with a surrounding shear layer structure.

The black lines in Figure 9.9(b) to (l) represent a streakline at the respective time started at a point in a surrounding shear layer region. The streaklines visualize the transport of fluid particles from surrounding shear layer regions into the newly generated vortex structure. A model which considers the transport of fluid particles from shear layer regions into newly generated vortex structures for cylinder wake cases is given by Ponta [127].

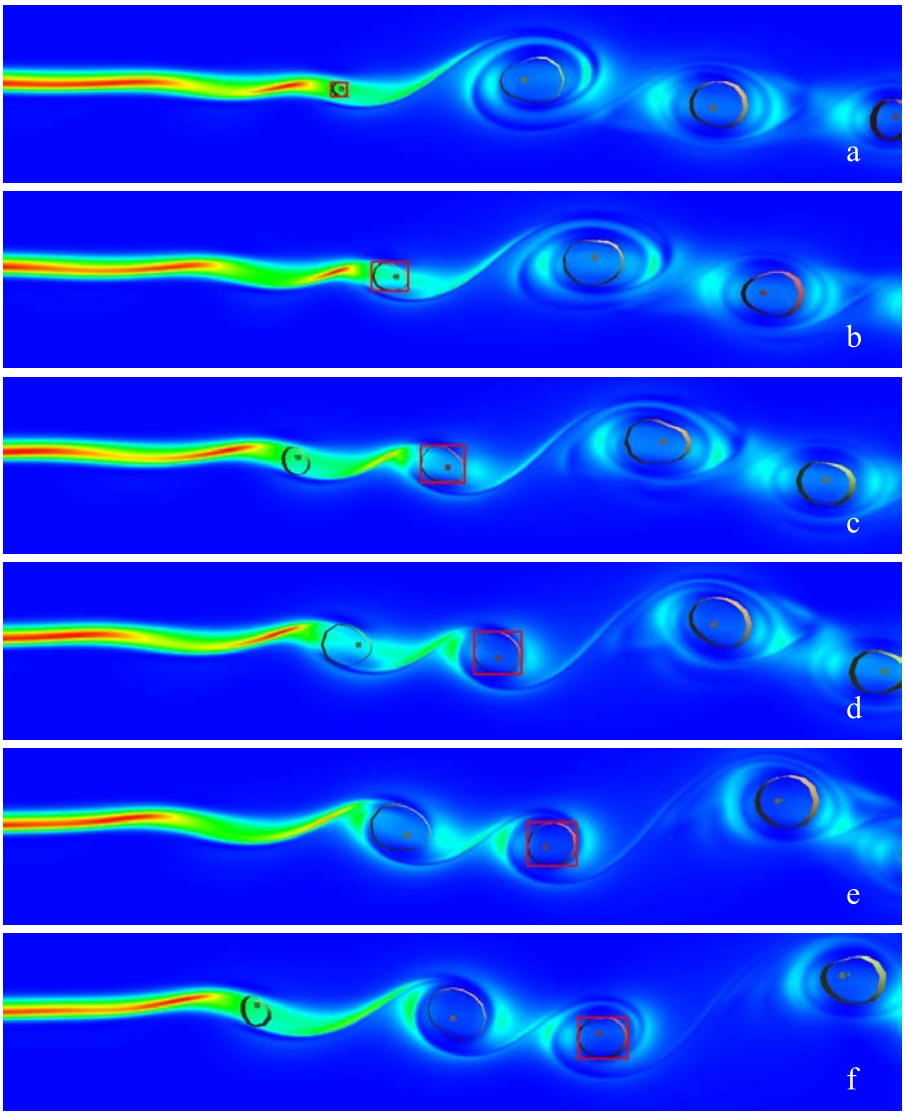


Figure 9.10: Formation of new vortex structures in the velocity gradient region between two free streams [3]. The red bounding box surrounds a specific vortex at different times.

Free Shear Layer

The generation of new vortex structures as a result of the roll-up of shear layers is widely investigated [59; 97; 127; 201]. The main focus of research of vortex dynamics in free shear layers is on the mixing of two parallel streams [59] and cylinder wakes [201]. The roll-up of shear layers in free shear layers is primarily initiated by linear instabilities of the vorticity distribution where the established two-dimensional waves grow exponentially with the downstream distance and roll-up into vortex structures.

Figure 9.10 shows the formation of new vortex structures in the velocity gradient region between two free streams. In Figure 9.10 (a) to (d) the development of the instability of the shear layer region (false color representation: blue regions are regions

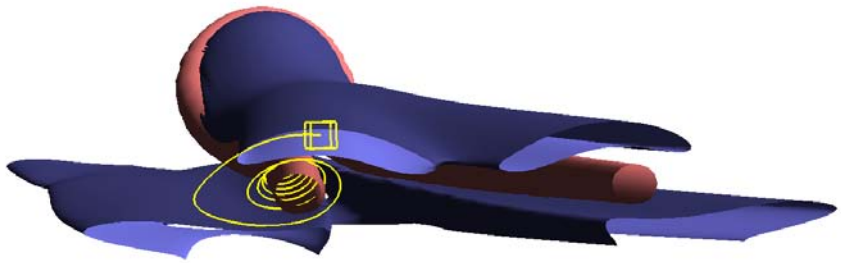


Figure 9.11: Path line of a particle started inside a shear layer (blue). It is noticeable that the particle enters a new vortex region (red $\lambda_2 = 0$ isosurface). This leads to the observation that the new vortex region is generated by shear layers.

of low shear stress, red regions are regions of high shear stress) and the generation of a two-dimensional wake can be observed. A further observation is the roll-up of this two-dimensional wake to a new vortex structure (regions inside red contour lines) as well as the overlap of shear layer and vortex regions. Figure 9.10(e) to (f) show the final stages of the development of the new vortex structure with vanishing shear stress inside the vortex structure. This is comparable to the 2D vortex merging of co-rotating vortex structures.

Another possibility to emphasize the importance of a shear layer region in the formation of new vortex structures is the observation of pathlines. In Fig. 9.11 a path line is shown which starts in the high shear stress region of the free shear layer. The tracked fluid particles follow the high shear stress region until they enter into the generated vortex structures and are transported downstream as a part of the vortex structures.

9.3.2 Visualization of Shear Layers in Time-dependent Flow

The importance of an appropriate shear layer visualization has been shown in the previous section. In the following, the methods for visualizing and tracking shear layers described in Section 8.3 and Section 9.1 are used to investigate the temporal development of shear layers and vortex structures. The scene graph used for the tracking algorithm in Section 9.1.2 is able to hold both structures, the shear layers and the vortices. If a shear layer is selected for tracking, the particles are integrated along the vector field and tested for intersection with both kinds of features. If, for example, a vortex structure is hit by a large amount of shear layer particles, the shear layer is assumed to create a new vortex as it is discussed in Section 9.3.1 for vortex generation. For the tracking of shear layers, the 1D ridge lines of I_2 are sampled which have been computed before in a pre computation step according to Algorithm 5.

In this section, the effectiveness of the tracking method is discussed by experts in the field of fluid dynamics (two researchers from the Institute for Aerodynamics and Gasdynamics from the University of Stuttgart; both are specialists in vortex and shear layer detection methods). For this purpose, a simulated time-dependent data set of a laminar-turbulent transition on a flat plate is investigated using the visualization techniques proposed. In the following, the term *plate* is be used, which actually stands

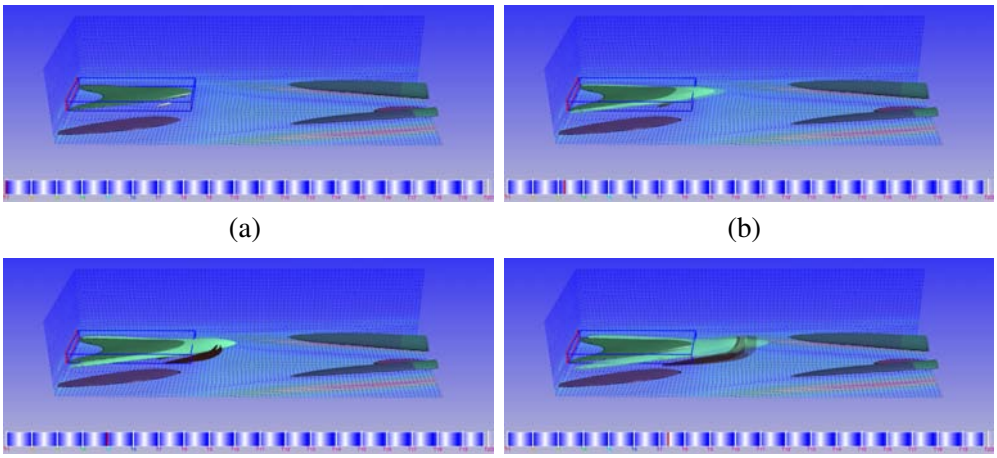
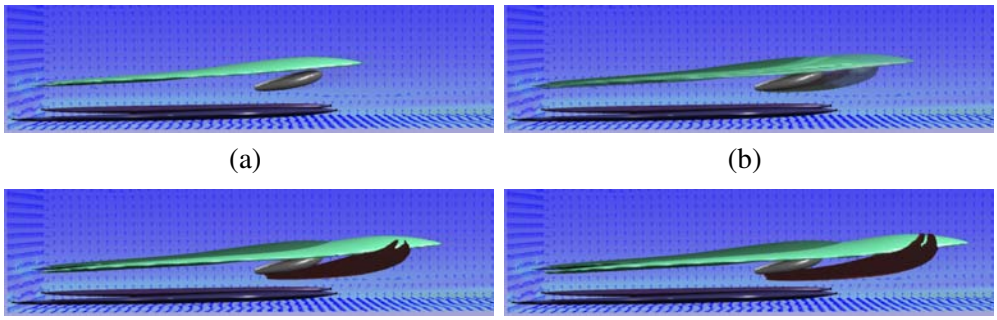


Figure 9.12: Tracking of coherent structures: first an arbitrary shear layer is selected by the user (a). The time line gives feedback about the visible time step. The selected structure is drawn as opaque green I_2 isosurface (context), surrounded by a bounding box. The color gradient for the shear layers goes from blue (weak) to green (strong), vortices are red-colored; the tracked structure is represented as transparent surface (focus). The development of the shear layer is clearly perceptible: first it only moves from left to right (b), then a vortex structure is hit by the shear layer at $t = 5$, which is in addition to the shear layer, also tracked (d).

for the (invisible) plate at the bottom of the data set.

The observation of flow field features using different perspectives (shown in Figure 9.12 - 9.15) emphasizes the importance of shear layer structures in the investigation of the spatial and temporal development of vortex structures. Here, a new vortex structure appears below the the selected shear layer structure in Figure 9.12(a). In detail, the generated vortex can be considered as a result of high shear stress, particulary of the shear layer above and below the later vortex position. The deformation of the shear sheet around the legs of the vortex structure can be interpreted as a roll-up of the shear layer structure and as an important factor in the generation of turbulence.

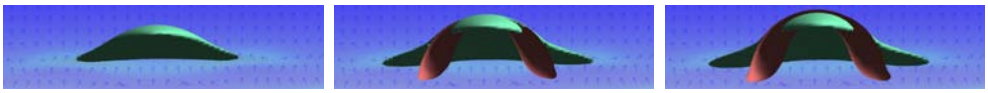
The development of both structures, the shear layer and the vortex, during the observed time period is quite similar. Both structures move with similar velocity downstream and both structures are affected by stretching. There is also a movement away from the flat plate for both structures observable (Fig. 9.13). This motion seems to be faster in the downstream parts of the structures than in the upstream parts. Furthermore, the movement away from the flat plate seems to be more significant for the head of the vortex structure than for the head of the shear layer structure. This leads to an intersection of both structures in Fig. 9.14(b) and 9.14(c). This intersection of both structures is much better visible in Fig. 9.15, where the shear sheet is visualized instead of the shear layer isosurface. This leads to the observation (also in Fig. 9.15) that both structures are aligned, which demonstrates the importance of the investigation of both flow field features and their interaction.



(a) (b)

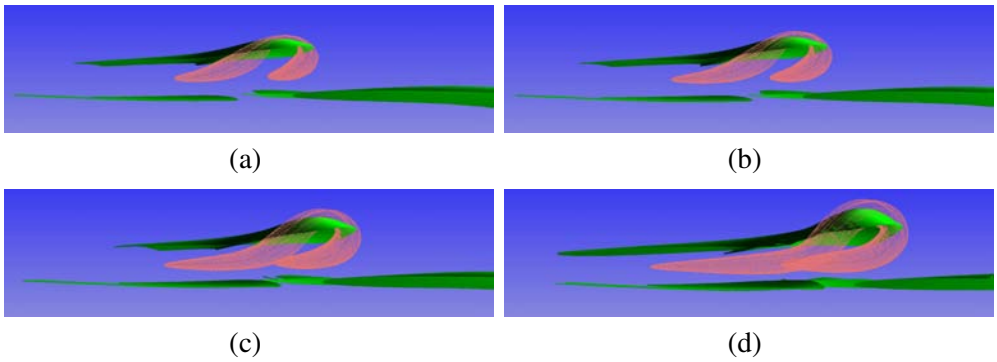
(c) (d)

Figure 9.13: This figure shows the same scenario as Fig. 9.12. Here, the tracking process, i.e., forward and backward tracking, is started at $t = 5$ for both, the shear layer and the vortex structure. Setting the context to $t = 1$ leads to the sequence shown above. The green shear layer is located on top of the developing vortex (gray). Then the shear layer is getting stretched while its head begins to surround the vortex, and finally, to intersect it. This illustration shows that the shear layer serves as impulse for the generating vortex, i.e., it amplifies the vortex. However, it is not clear what exactly happens inside the intersection region.



(a) (b) (c)

Figure 9.14: The front view shows the vortex-shear layer interaction in more detail: before the vortex is generated, the shear layer seems to be small. This changes once the vortex begins to grow. Then the shear layer grows, too. However, even in this sequence it is not clear where the shear layer intersects the vortex. In figure (b) it seems to go right through the vortex, while in figure (c) it is located underneath the vortex structure.



(a) (b)

(c) (d)

Figure 9.15: Visualizing the shear sheet instead of the I_2 isosurface results in a more precise representation. Here, it is clearly perceptible that the shear sheet slightly intersects the vortex. The investigation of this intersection regions allow an advanced investigation of the dynamics in fluid mechanics by taking into account shear layers during the vortex generation process.

Part IV

Light Tracing

CHAPTER

10 LINEAR LIGHT TRACING

So far, particle tracing has been used for tracing particles along a given vector field that either describes the motion of a particle inside a flow field (Part II of this thesis) or results from computations like the eigenvector field of $\mathcal{H}\mathrm{I}_2$, which is used for finding lines of maximal shear (Part III of this thesis). This chapter considers particle tracing from another point of view: here, light rays are traced along their way from the light source. In this case, the particles' trajectories are defined by the connection between two points in space (the view ray), which can also be formulated as Lagrangian ODE with a vector field $\boldsymbol{v}$. In the linear case, this vector field is constant per ray and can be computed directly by the distance to the front clipping plane, the field of view, and the aspect ratio. In the nonlinear case, which is discussed in the next section, the vector field is space variant and depends on spatial locations.

There always exists an underlying function that defines the change of the radiance along the ray which needs to be evaluated. This chapter considers this scenario in terms of volume rendering (see Section 1.4 for further information), where ray casting is applied for sampling the view ray.

10.1 Raytracing Multifield Data

As already shown in Section 8.1, ray casting is well-suited for the interactive visualization of flow features. However, since the interactivity strongly depends on the parallelism of graphics hardware, i.e., on an appropriate GPU-implementation, also the data structures of the vector and scalar quantities need to be optimized for the GPU. Thereby uniform regular grids (see Section 1.3) build the most efficient data structure for GPUs, since they can be simply represented by textures. On the GPU usually only one texture fetch is necessary to obtain the desired vector or scalar at a certain position. But the memory requirement of uniform grids depends on the overall resolution of the data set, i.e., if a higher resolution is desired at a specific region, then the resolution is increased at each position within the data set, even if a lower resolution would also be sufficient.

Therefore, in practice, more complex grid types are used in order to reduce the amount of data without losing the precision at specific regions within the data set (adaptivity). The most popular adaptive grid type is the unstructured triangle (2D) and tetrahedra (3D) grid, respectively. But due to the high costs, particularly for interpolating the data, also other adaptive grid types are common in practice, like the adaptive mesh refinement data (AMR). First introduced by Berger and Oliger [11], AMR has found various applications such as in astrophysics, weather and fluid dynamics sim-

ulation for engineering. Kaufmann and Mueller [78] give a comprehensive general overview of volume rendering, while Pfister [123] presents hardware accelerated techniques. Both Park et al. [119] and Kähler et al. [75] visualize AMR data interactively in a multipass approach by partitioning into bricks and the use of tree data structures for brick selection on the CPU. Park et al. render bricks with hardware-accelerated hierarchical splatting, whereas Kähler et al. rely on the classical slicing-based approach. Recently, Sadlo et al. [144] proposed a method for the computation of height ridges on AMR data.

This section describes how GPU-based ray casting can be modified for an interactive application to AMR grids in order to visualize flow features. In a first step, the entire AMR data set is mapped to graphics memory by employing GPU implementations of complex data structures, like octree textures [94] or adaptive page tables [96]. While octree textures provide a compact representation at the cost of rendering speed, adaptive page tables offer high frame rates but consume more memory. Both approaches have been implemented and compared. In the second step, a GPU-based single pass ray caster is applied on the optimized data structures. Here, the original GPU-based ray casting implementation by Stegmaier et al. [166] for uniform grids is adapted for managing also AMR grids.

This section reflects the discussion of the work by Vollrath et al. [180] which was in collaboration with my colleague Joachim Vollrath who did the major part of the implementation as part of his diploma thesis.

10.1.1 Octree Texture

An octree represents a uniform cartesian grid at any depth and is therefore well suited to store AMR data. The GPU implementation represents each inner node with a cube of $2 \times 2 \times 2$ texels. These texels represent the eight children of an octree node and are either interpreted as a reference pointer to the respective child if it is an inner node or as a scalar sample if the child is a leaf. This is achieved by using the alpha component to indicate the node type (empty, leaf or inner node) and the RGB-components as references within the octree texture or as scalar samples. The octree is traversed in a top-down process on the GPU, details are found in the original article by Lefebvre et al. [94].

Texture Construction

The adaptive grid employed in the target CFD postprocessing tool is organized as a hierarchical data structure where each individual cell of the coarsest resolution can be considered as the root of an octree. The octree texture is constructed by reconstructing a full octree covering the entire domain of the grid from these subtrees (Figure 10.1) and store the octree nodes in a 3D texture in depth-first order.

As described above, one color channel of each texel indicates the type of the child node. The scalar sample contained in a leaf can then be stored in one of the remaining three channels, leaving only two channels to store gradients for shading computations. This issue is solved by storing the gradient in spherical coordinates in the remaining two channels and convert it back into cartesian coordinates in the fragment program.

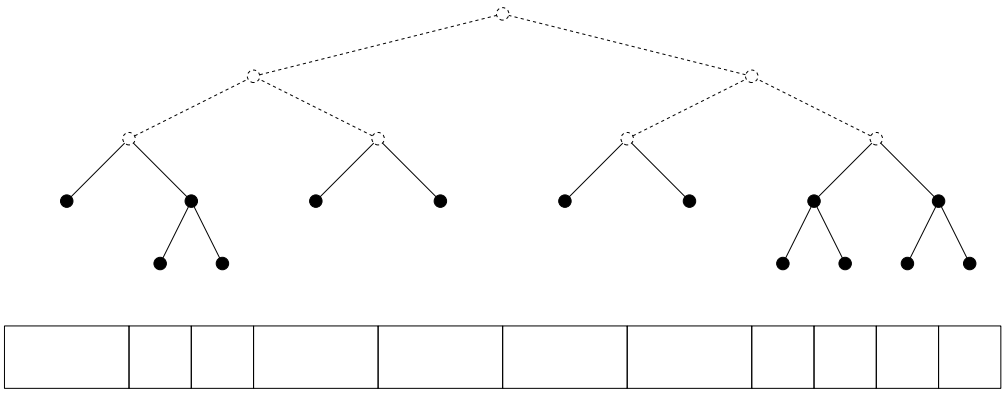


Figure 10.1: Tree interpretation of an adaptive 1D grid. The dotted part has to be reconstructed.

Interpolation

Lefebvre et al. [94] proposed a hardware accelerated interpolation scheme for the octree texture which requires all leaves of the octree to be at the same depth. An alternative interpolation scheme is proposed by Benson and Davis [10] that constructs a virtual uniform grid of the desired resolution on which trilinear or tricubic interpolation can take place. In contrast to those methods, an interpolation scheme that is more suitable for graphics hardware is discussed, which guarantees continuous interpolation within octree leaves of the same depth. Strictly speaking, the interpolation across different depths will be discontinuous, but in practice hardly any visible artifacts will occur as it is shown in Section 10.1.3.

For a given sampling position the octree traversal returns the sample at the nearest neighbor cell. The other seven of the eight samples needed for trilinear interpolation are retrieved by performing traversal with the sampling position offset by half the size of the nearest neighbor cell.

Interpolation of samples of the same depth in the tree is trivial and shall not be further detailed whereas the interpolation of samples at different depths is more interesting. This is done with the aid of a level of detail (LOD) approach. Together with the octree texture a LOD texture is constructed which contains the averaged values of the child nodes for each internal node of the octree. Due to the favorable ratio between the number of internal nodes and the number of leafs (1:7 for a full octree), the additional storage requirements are tolerable. Assume that for a certain sampling position the sample value of the leaf at depth d containing this sampling position and the values at the seven neighboring cells needed for trilinear interpolation have been retrieved. Furthermore, the difference in depth of the neighbor cells to the current cell should be restricted to one. Then, for each neighbor cell, the following cases may arise:

- The neighbor cell is at depth d: Its sample is used for interpolation.
- The neighbor cell is at depth $\mathrm{d}+1$: The averaged value at depth d from the LOD texture is used for interpolation (Figure 10.2 (a)).
- The neighbor cell is at depth $\mathrm{d}-1$: Its sample is used for interpolation, which is

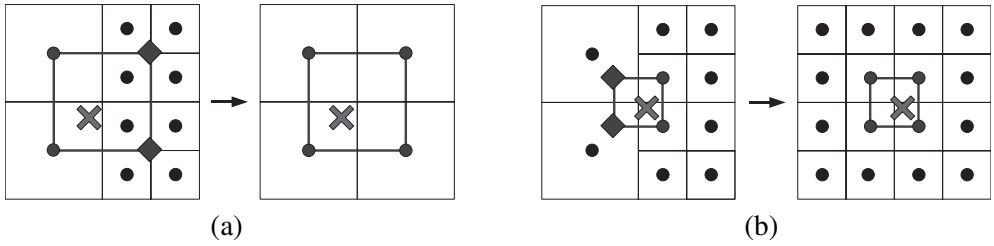


Figure 10.2: The octree interpolation scheme in 2D. The sampling position is marked with a cross, required samples of a different refinement level are marked with a diamond.

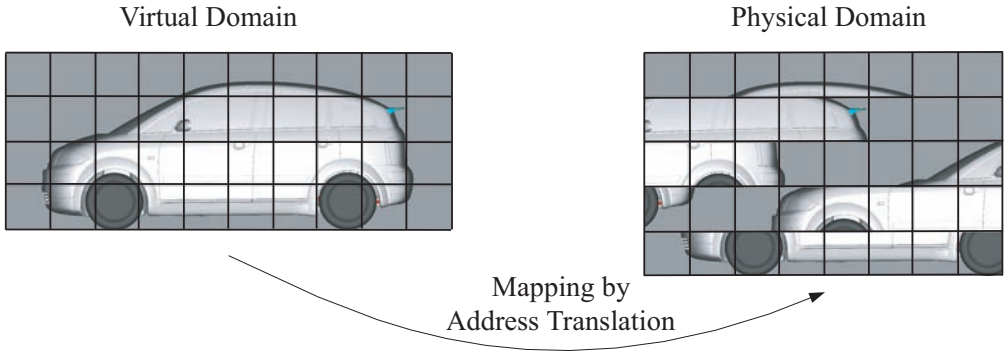


Figure 10.3: Mapping between the virtual and physical domain

equivalent to subdividing the cell into eight children with the same value (Figure 10.2 (b)).

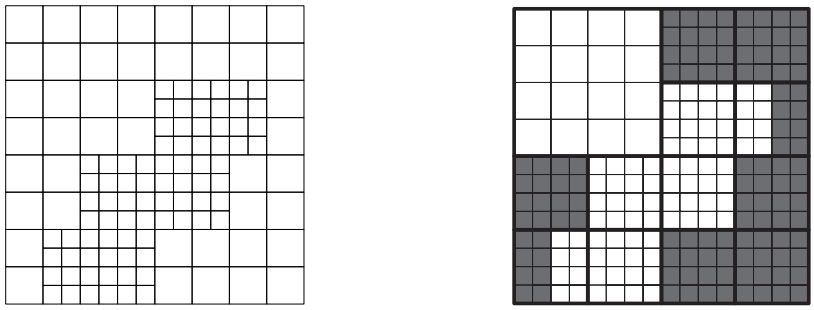
10.1.2 Adaptive Page Table

The second method for storing AMR data efficiently for GPU-based ray casting is an adaptive page table presented by Lefohn et al. [96]. Here, a dynamic adaptive multiresolution GPU data structure performs page table address translation on the GPU by mapping from a virtual domain to the physical domain of graphics memory. The idea is to partition space into pages with identical number of samples, stored in a page texture (Figure 10.3). A page table texture handles the mapping, accomplishing adaptivity by mapping multiple virtual pages to a single physical page. A cell-centered version of this data structure has been implemented, which is refer to it as *adaptive page table* to simplify matters.

The mapping of virtual to physical pages is realized by storing a reference in the RGB color channels of each texel of the page table texture. The alpha channel stores a scaling factor depending on the refinement level of the page that is addressed.

Texture Construction

For a given page size p, the adaptive grid is partitioned into pages with $p \times p \times p$ samples. However, the refinement level of the cells covered by a page may vary. To



(before) (after)

Figure 10.4: Partitioning of a grid consisting of 126 cells with $p = 4$. The grid is represented by 13 pages with a total of 208 samples. Oversampled cells are highlighted in gray.

avoid undersampling, a page is then recursively subdivided into sub-pages (each with p^3 samples) of a higher sampling rate until the level of refinement matches that of the smallest covered cell. This leads to an oversampling of some cells (Figure 10.4) and increases storage requirements as it is discussed in Section 10.1.3.

Since the pages are equally sized there is no need for any sophisticated packing in the page texture – it is simply filled with pages in the order of occurrence during partitioning.

Interpolation

The adaptive page table offers constant access complexity and can exploit hardware filtering since physical pages are stored coherently in texture memory. However, since it is not guaranteed that pages adjacent in the virtual domain are adjacent in graphics memory, continuous interpolation is ensured by additionally sharing one layer of cells between physical pages. The number of samples thus increases to $(p+2)^3$ in the cell-centered implementation. Depending on the page size p this produces an overhead:

$$o(p) = \frac{(p+2)^3 - p^3}{p^3} = \frac{6}{p} + \frac{12}{p^2} + \frac{8}{p^3}\,. \tag{10.1}$$

Table 10.1 implies that it might be desirable to choose large page sizes to reduce this overhead. However, with increasing size, the aforementioned overhead through oversampling increases, too. In practice, the actual behavior strongly depends on the structure of the adaptive grid. Therefore, it makes sense to run the partitioning algorithm for various values of p and to select the best size for the dataset.

p	8	16	32	64	128
$o(p)$	56.25%	26.56%	12.89%	6.35%	3.15%

Table 10.1: Overhead through sharing between physical pages

Model	Audi	BMW	Motorbike
Refinement levels	4	5	6
Number of Cells	2338083	3346309	5146028
Octree levels	11	13	13
Octree size (MB)	10.89	15.26	23.39
LOD size (MB)	1.02	1.43	2.19
Number of pages	1606	2621	3467
Page size (MB)	37.02	60.07	80.09
Index size (MB)	0.07	1.25	2.5

Table 10.2: Memory Requirements of 3 different AMR grids

	Audi		BMW		Motorbike	
	OCT	APT	OCT	APT	OCT	APT
DVR	0.7	23.7	0.8	27.3	0.7	9.35
ISO	0.4	14.9	0.4	17.5	0.27	7.0

Table 10.3: Performance in frames/second. Abbreviations: OCT = Octree Texture, APT = Adaptive Page Table, DVR = direct volume rendering, ISO = isosurface rendering.

10.1.3 Results and Discussion

Table 10.2 shows the memory footprints for three datasets. It shows that the octree approach is notably more memory-efficient. The page table approach consumes roughly three and a half times more memory due to the previously addressed overheads.

Table 10.3 shows performance measurements on a 3.4 GHz Pentium IV machine with a GeForce 7800 GTX card in a 500^2 viewport. Obviously, the adaptive page table dramatically outperformes the octree texture due to its better access complexity and native hardware filtering. Altogether, three to four times larger memory requirements for up to 40 times more performance with the adaptive page table must be considered a fair deal. Nevertheless, for very large AMR datasets the octree texture may be the only viable approach.

Figure 10.5 shows that the interpolation scheme for the octree texture produces images of higher quality compared to the adaptive page table, mainly because we currently employ a cell-centered implementation of the adaptive page table which produces more visible artifacts.

Figure 10.6 demonstrates the ray caster implementation for AMR data. For all images in Figure 10.6 direct volume rendering of the velocity magnitude has been applied. The color gradient resulting from the transfer function goes from red (slow) to green (fast). In the front of the car, a bright bump is visible; air impounds in front of the car. The isosurface in the next figure emphasizes this bump. In the third image,

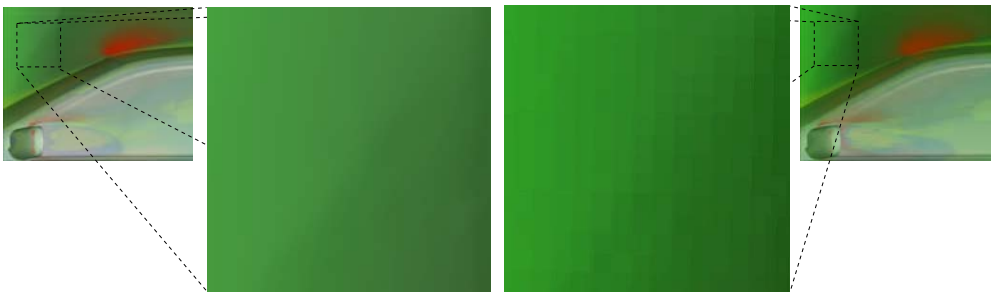
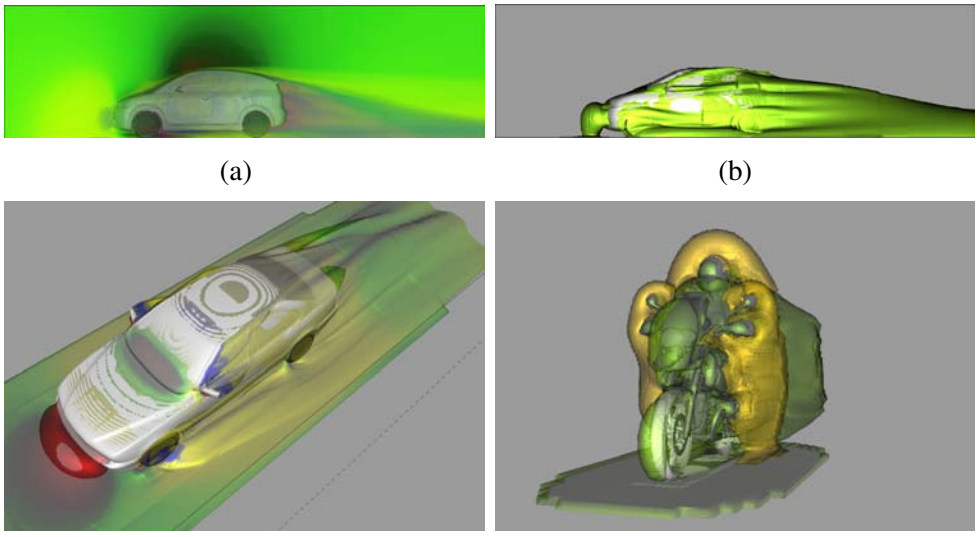


Figure 10.5: Interpolation artifacts: octree texture (left), adaptive page table (right).



(a) (b)

(c) (d)

Figure 10.6: Interactive ray casting of AMR data. Starting upper left: (a) direct volume rendering of the velocity magnitude; (b) opaque velocity magnitude isosurface; (c) semi-transparent velocity magnitude isosurface colored by static pressure; (d) combined direct volume rendering and isosurface of the velocity magnitude.

static pressure is mapped on the isosurface (gradient from blue to red). Behind the rear view mirrors low pressure is noticeable. The last image shows two different isosurfaces around a biker. The yellow isosurface illustrates slower velocities that can be ascribed to the turbulences occur close to the driver.

10.2 Atmospheric Scattering

Applying particle tracing in form of tracing photons on their way to the eye is a broad topic. Although the application area is completely different, the method in Section 10.1 and the method which is discussed in following, perform basically the same algorithm: light particles are captured along a light ray by evaluating an underlying function. This function can either be given by a discrete scalar field like in Section 10.1: here each entry in a grid cell represents a value of a discretely sampled function, e.g., the

velocity magnitude. For the sake of completeness, it should be mentioned that the light contribution is usually determined by a transfer function which maps the discrete scalar values to colors.

As already mentioned above, the following method fundamentally does the same: it evaluates a function along a light ray. But in contrast to the previous section, where direct volume rendering has been applied for flow visualization, the underlying function consists of the atmospheric scattering function, which is evaluated for each color channel separately. This eventually makes a transfer function unnecessary.

In the last years, several methods were developed to simulate atmospheric scattering: in [34] light beams are simulated; a model for drawing the sky is discussed in [128; 66] ,including the colors of sunrise and sunset; and the earth is viewed from space in the work of Nishita et al. [116]. The attenuation of the incident light was considered at each point of the atmosphere as well as the attenuation from this point to the viewer. Furthermore, additional to the sky's color, the colors of clouds and sea were discussed. Other models for rendering the sky in the daytime were proposed by Preetham [128] and Hoffman [60]; a physics based night model was described by Jensen [66]. An extension of Preetham's model was proposed by Nielsen [115] by considering a varying density inside the atmosphere. In 2002 Dobashi et al. [34] proposed a GPU based method to implement atmospheric scattering using a spherical volume rendering. They solved the discrete form of the light scattering integral by sampling it with slices. Depending on the shape of the slices, which are planar or spherical, atmospheric scattering effects as well as shafts of light, as occurring between clouds can be visualized. In 2004 O'Neil [117] proposed an interactive CPU implementation, solving the scattering integral by ray casting. This method avoids expensive volume rendering by representing the atmosphere by only two spheres, one for the outer and one for the inner boundary of the atmosphere. Ray casting is used to solve the discrete form of the scattering integral by separating the ray into segments to calculate the attenuation of the light from the sun to the viewer for each sample point. A GPU-based implementation of this method can be found in [118; 187; 188], whereby the latter two also consider terrain rendering.

The following discussion extends the idea of [117; 118], where the scattering integral is partially pre computed. The idea is to fully pre compute the scattering integral in order to release the GPU of expensive real-time computations. Furthermore, though only pre computed data should be accessed, elevation in form of terrain rendering should be supported as well.

10.2.1 Equations of Atmospheric Scattering

This section gives a short introduction to the equations of Rayleigh and Mie scattering. First the scattering of air molecules is discussed, that is described by the Rayleigh scattering equation:

$$I_p(\lambda, \theta) = I_0(\lambda) K \rho F_r(\theta) / \lambda^4 \,, \tag{10.2}$$

where λ is the wavelength of the incident light at $\mathbf{P}$ and θ the scattering angle between the viewer and the incident light at $\mathbf{P}$. I_0 stands for the incident light, ρ for the density ratio (given by $\rho = \exp(-h/H_0)$), depending on the altitude h and scale height $H_0 = 7994m$. F_r denotes the scattering phase function, which indicates the directional

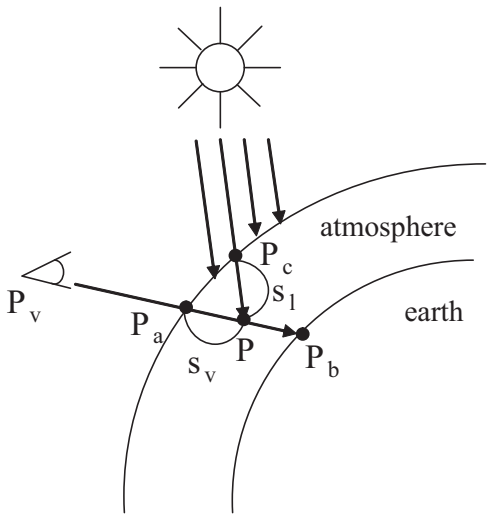


Figure 10.7: Calculation of light reaching $\mathbf{P}_v$.

characteristic of scattering ($F_r = \frac{3}{4}(1+\cos^2(\theta))$). K describes the constant molecular density at sea level:

$$K = \frac{2\pi^2(n^2-1)^2}{3N_s} ,$$

where N_s is the molecular number density of the standard atmosphere and n is the index of refraction of the air.

Equation 10.2 states the relation between scattered light and the wavelength of the incident light and describes the strong attenuation of short wavelengths. To determine the light intensity reaching $\mathbf{P}_v$ in Figure 10.7, two steps have to be performed: first, for each point $\mathbf{P}$ between $\mathbf{P}_a$ and $\mathbf{P}_b$, the light reaching this point needs to be attenuated; second, for each point $\mathbf{P}$, the resulting light intensity needs also to be attenuated on its way to the viewer at $\mathbf{P}_v$. The attenuation between two points inside the atmosphere is determined by the optical length, which is computed by integrating the attenuation coefficient β, which describes the extinction ratio per unit length along the distance s_v. It is given by:

$$\beta = \frac{8\pi^3(n^2-1)^2}{3N_s\lambda^4} = \frac{4\pi K}{\lambda^4} , \tag{10.3}$$

and is integrated along the distance S and yields

$$t(S,\lambda) = \int_0^S \beta(s)\rho(s)ds = \frac{4\pi K}{\lambda^4}\int_0^S \rho(s)ds . \tag{10.4}$$

The scattering of aerosols is described by the Henyey-Greenstein function [30]. To adjust the rate of decrease of the aerosol's density ratio, the scale height H_0 is set to 1.2km [154]. The optical length of aerosols is the same as for air molecules, except for the $1/\lambda^4$ dependance. According to Nishita[116], after inserting the equations above to Equation 10.2, the light intensity at $\mathbf{P}_v$ is computed by:

$$I_v(\lambda) = I_s(\lambda)\frac{KF_r(\theta)}{\lambda^4}\cdot\int_{\mathbf{P}_a}^{\mathbf{P}_b}\rho\exp(-t(\overline{\mathbf{PP}_c},\lambda)-t(\overline{\mathbf{PP}_a},\lambda))ds . \tag{10.5}$$

10.2.2 Real-Time Atmospheric Scattering

In current approaches for computing atmospheric scattering effects, the performance mainly depends on the complexity of the scene. Therefore, the scattering integral of Eq. (10.5) is solved for each vertex belonging to the scene geometry. Though these approaches are quite fast by applying modern graphics hardware, especially when planets are restricted to simple spheres, the interactivity suffers immensely by an increasing complexity of the scene. This is the case if one intends to render the structure of planets. In the following it is described how this lack of performance can be avoided by pre computing the scattering integral. After pre computation the result is stored in a 3D texture. Therefore, the number of instructions on the GPU is strongly decreased; eventually only one texture fetch is necessary for obtaining the light attenuation value.

Creating the Scattering Texture

The scattering equation of Eq. (10.5) defines the light contribution reaching the observer's eye when it is placed at a position $\mathbf{P}_v$ and his view ray penetrates the atmosphere between $\mathbf{P}_a$ and $\mathbf{P}_b$. Basically, the light is attenuated two times at each point on the connection $\overline{\mathbf{P}_a\mathbf{P}_b}$. The first time from the light source to a position $\mathbf{P}$ on the view ray, and the second time from $\mathbf{P}$ to the observer's position $\mathbf{P}_v$. Therefore, the resulting light contribution depends on four variables: the observer's position $\mathbf{P}_v$, the position of the light source $\mathbf{P}_c$, and the entry and exit position of the atmosphere $\mathbf{P}_a$ and $\mathbf{P}_b$. Furthermore, approximating the integral as a Riemann sum requires an additional sample variable $\mathbf{P}$. A naive pre computation of the scattering integral would require a nine-dimensional domain: the position of the sun; the position of the observer; and the view direction. Each of these variables is expressed by a three dimensional vector.

O'Neil [117] suggested to simplify the computation of the optical depth in Eq. (10.4) from the light source to $\mathbf{P}$ by parameterizing each point $\mathbf{P}$ by its height and its angle to the Sun. This simplifies Eq. (10.5) enormously because the pre computed values can be used to determine the optical depth from the light source to the sample point $t(\mathbf{PP}_c, \lambda)$ as well as for the attenuation from the sample point to the observer. Only a 2D texture is required for holding these values. Nevertheless, the integral from $\mathbf{P}_v$ to $\mathbf{P}$ remains. In order to solve this issue, O'Neil's approach has been extended by considering the view direction as well. This also extends the pre computation texture to 3D.

The algorithm works as follows: first an appropriate parametrization of the 3D texture is defined. As discussed above, a number of parameters must be considered, which have to be reformulated to parameterize a three-dimensional lookup table. For the sake of simplicity, the observer is assumed to be situated inside the atmosphere. The general case is discussed later in this section. Basically, the observer can be situated at an arbitrary position, looking in an arbitrary direction at an arbitrary daytime. Considering the observer's position $\mathbf{P}_v$ and his view direction $\mathbf{R}_v$ at a specific daytime, then the view angle θ can be determined by

$$\cos(\theta) = \frac{1}{|\mathbf{P}_v||\mathbf{R}_v|} < \mathbf{P}_v, \mathbf{R}_v > = \frac{1}{|\mathbf{P}_v{}'|} < \mathbf{P}'_v, \mathbf{R}'_v > , \tag{10.6}$$

where $\mathbf{P}'_v = (0, |\mathbf{P}_v|, 0)^T$ and $\mathbf{R}'_v = (\sin(\theta), \cos(\theta), 0)^T$. This means that each actual position and the corresponding view direction can also be described only by its height

Algorithm 8 Computation of the 3D lookup texture.

```
void createLookupTexture()
{
while angleViewer < resZ
  // get angle θ
  θ = GetViewAngle(angleViewer);
  // generate a view vector
  R'_v = vec3d(sin(θ), cos(θ), 0);
  // loop over all view angles to the light source
  while angleSun < resY
    // get angle δ
    δ = GetLightAngle(angleSun);
    // loop over all heights of the camera
    while height < resX
      // get current height inside the atmosphere
      h = fRadIn + ((fRadOut − fRadIn) · height)/(resX − 1);
      // generate the position vector
      P_v = vec3d(0, h, 0);
      // finally, compute the light scattering
      color = ComputeScattering(P_v, δ, R_v);
    endwhile
  endwhile
endwhile
}
```

$h = |\mathbf{P}_v|$ and the view angle θ. Exploiting this behavior, the camera is placed at each height inside the atmosphere to send out the view rays in each direction.

The distance $\overline{\mathbf{P}_a\mathbf{P}_b}$ is obtained by considering the atmosphere's boundaries as spheres, one representing the inner and one the outer boundary. Then only $\mathbf{R}'_v$ needs to be tested for an intersection with both spheres. Keep in mind that the pre computation regards the planet's surface as a simple sphere. How to deal with terrains, though, is described in Section 10.2.2.

Finally, the light source has to be introduced to the model. As discussed above, the attenuation from a light source to an arbitrary position can also be described by the height of the sample point and the angle δ to the light source. For solving Eq. (10.5) at a position $\mathbf{P}'_v$ and a view ray $\mathbf{R}'_v$, the light attenuation $t(\overline{\mathbf{P}\mathbf{P}_c}, \lambda)$ and $t(\overline{\mathbf{P}\mathbf{P}_a}, \lambda)$ can easily be computed by sampling along $\mathbf{R}'_v$ and evaluating Eq. (10.4) according to the height of the sampling position and the angle δ. Thus, the angle δ builds the third parameter of the 3D texture.

If the observer is located outside the atmosphere, the situation changes. Since there is no light scattering outside the planet's atmosphere, the distance of the observer to the planet is not accounted in the pre computation step. Nevertheless, viewing the planet from outside builds a special case, in which the computation can be considered the same for each position, regardless if the camera is situated on the atmosphere's outer boundary or the camera is far away. Thus, the camera has to be virtually moved towards the planet until it hits the outer boundary of the atmosphere. Only one additional height must be considered during pre computation.

The code sample in Algorithm 8 shows how simple the 3D lookup texture is created. The texture is given with its sizes in the x, y, and z direction. For each voxel, the

corresponding angles and the height is used for evaluating the light scattering integral. Here, the discrete form of Eq. (10.5) is solved.:

$$I_v(\lambda) \quad = \quad I_s(\lambda)\frac{KF_r(\theta)}{\lambda^4}\sum_{i=0}^{k}\rho\exp(-t(\overline{P_iP_c},\lambda)-t(\overline{P_iP_a},\lambda))\Delta s \quad , \quad (10.7)$$

where P_i is the ith point on the distance from P_a to P_b and Δs is the sampling distance. Due to the fact that the whole scattering integral is pre computed, the sample rate k as well as the sample rate for the optical depths can be set very high.

Applying the Scattering Texture

After the lookup texture is computed, the calculation of the light intensity is quite simple and needs only few instructions on the GPU. During the rendering phase, two independent scene objects are drawn: a sphere representing the sky, and the planet's surface. In Section 10.2.3 the rendering of the terrain is discussed in more detail. First, the rendering of the sky is discussed, followed by a detailed discussion on the combination of light scattering and elevation data.

In this model, the sky is represented by rendering only one tesselated sphere which is placed nearby the outer boundary of the atmosphere. The sphere needs to be only visible from its interior, i.e., if the observer looks towards the sky. Otherwise, if the observer looks towards the surface of the planet, the attenuated light directly influences the terrain rendering and, therefore, no outer sphere must be drawn. Therefore, culling needs to be enabled for both the sphere and the terrain.

First the view ray $\mathbf{R}_v = \mathbf{P}_g - \mathbf{P}_v$ is computed, where $\mathbf{P}_g$ stands for the position of the current vertex. The view ray describes the paths of the light particles, whereby the amount of the arriving light particles is computed by the scattering function, which is stored in the pre computed 3D texture.

If the observer is outside the atmosphere, the camera is moved to the outer boundary. Then, the height is obtained by $h = |\mathbf{P}_v|$ (this assumes the planet's origin at $(0,0,0)^T$) as well as the cosine of the view angle $\cos(\theta) = \frac{1}{|\mathbf{P}_v||\mathbf{R}_v|} < \mathbf{R}_v, \mathbf{P}_v >$ and the sun angle $\cos(\delta) =< \mathbf{P}_c, \mathbf{P}_v >$. After rescaling all three parameters to $[0,1]$, the lookup texture is fetched. Because of the nonlinear behavior of the scattering function, an interpolation error caused by fetching the textures is unavoidable—otherwise the scattering integral must be recomputed for each point between the already points, what makes the precomputation senseless. Also previous methods had to deal with this issue—O'Neil [117; 118] applied barycentric interpolation on the rendered triangles. In this approach, this interpolation error is reduced by implementing the texture fetch as a fragment program instead of a vertex program. This minimizes the interpolation error since the sample frequency of the fragment shader is much higher and the texture fetches much faster than in a comparable vertex shader implementation. However, the inaccuracy introduced by linear interpolation depends on the resolution of the precomputed scattering texture.

In contrast to the sky, the light contribution on the terrain is much more complex. If the texture lookup is simply applied to any other vertices of the scene geometry, like the vertices representing the terrain, the computation fails. Figure 10.8 shows this case. This behavior results from the pre computation, which assumes the ray to

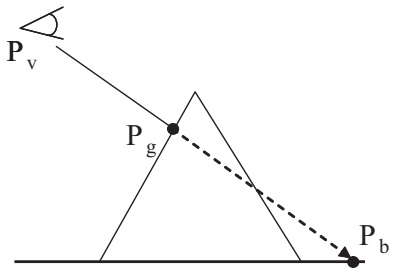


Figure 10.8: Obtaining the wrong light intensity: instead of $\overline{\mathbf{P}_v\mathbf{P}_g}$ the distance $\overline{\mathbf{P}_v\mathbf{P}_b}$ was used for the pre computation

intersect a simple sphere without considering the height field of the terrain. Thus, the light scattering is computed for the whole distance $\overline{\mathbf{P}_v\mathbf{P}_b}$. In order to get only the light scattered along $\overline{\mathbf{P}_v\mathbf{P}_g}$, the pre computed values have to be analyzed. The pre computed light scattering along $\overline{\mathbf{P}_v\mathbf{P}_b}$ can be formulated as the scattering along $\overline{\mathbf{P}_v\mathbf{P}_g}$ plus the scattering along $\overline{\mathbf{P}_g\mathbf{P}_b}$. Rewriting Eq. (10.5) to obtain the light scattering along $\overline{\mathbf{P}_v\mathbf{P}_g}$ would lead to

$$I'_v(\lambda) = I_s(\lambda)\frac{KF_r(\theta)}{\lambda^4} \cdot \left(\int_{\mathbf{P}_v}^{\mathbf{P}_b} \rho \exp(-t_l - t_v)ds - \int_{\mathbf{P}_g}^{\mathbf{P}_b} \rho \exp(-t_l - t_v)ds \right) , \quad (10.8)$$

with $t_l = t(\overline{\mathbf{P}\mathbf{P}_c})$, $t_v = t(\overline{\mathbf{P}\mathbf{P}_a})$ and $\mathbf{P}$, which is parameterized by s; both terms can be obtained fetching the pre computed texture. In fact, the first term is obtained anyway, simply using the current parameters as input. The second light contribution can be determined easily by moving the camera to the intersection point $\mathbf{P}_g$ without changing the view angle. Then the lookup texture is accessed a second time to subtract the result from the color value of the first texture lookup. By using this mechanism, it is possible to obtain the light contribution for any point inside the atmosphere.

Finally, the correct illumination of the terrain is discussed. Although the light scattering between object and viewer can be computed according to Eq. (10.8), the light contribution of the illuminated terrain has been not considered so far. The outgoing sun light reaching the terrain also underlies the light attenuation between the light source and the terrain as well as between the terrain and the observer. For this purpose, the light intensity I_g reaching the terrain geometry is considered, which is defined by

$$I_g(\lambda) = I_s(\lambda)\frac{KF_r(\theta)}{\lambda^4} \cdot \rho \exp(-t(\overline{\mathbf{P}_g\mathbf{P}_c}, \lambda)) . \quad (10.9)$$

I_g represents the incident light intensity for illuminating the terrain geometry. Here, a Lambert reflection is applied, which considers the cosine of the angle φ between the incident light and the terrain's normal as the intensity of the light absorbed by the terrain. Introducing the diffuse reflection to Eq. (10.9) yields

$$I_g(\lambda) = I_s(\lambda)\frac{KF_r(\theta)}{\lambda^4} \cdot \rho \cos(\varphi) \exp(-t(\overline{\mathbf{P}_g\mathbf{P}_c}, \lambda)) . \quad (10.10)$$

Additionally the attenuation of the reflected color needs to be attenuated on its way to the viewer. As defined by Nishita [116], the attenuation has to be multiplied with the intensity of the terrain geometry

Algorithm 9 Light contribution of an arbitrary scene geometry.

```
void renderIlluminatedGeometry()
{
Compute R_v
R_v = normalize(P_g − P_v);
If the camera is outside the atmosphere, move it to the outer boundary
dist = Intersect(P_v, R_v, SphereOut));
If dist > 0
   P_v = P_v + R_v · dist;
endif
Compute h, θ and δ
h = MapToUnit(|P_v|);
θ = MapToUnit(< P_v, R_v >);
δ = MapToUnit(< P_c, P_v >);
Get the light intensity without considering the height field
I_v = tex3D(texPre3D, vec3d(h, δ, θ));
Move the camera to the intersection point to obtain the offset
h′ = MapToUnit(|P_g|);
δ′ = MapToUnit(< P_c, P_g >);
I_off = tex3D(texPre3D, vec3d(h′, δ′, θ));
Correct the intensity
I′_v = I_v − I_off;
Now compute the light contribution of the terrain
F_r = tex1D(texPhase, θ);
t_gc = tex2D(texPre2D, h′, δ′);
t_gv = tex2D(texPre2D, h′, MapToUnit(< P_g, R_v >));
I_gv = I_s K F_r 1/λ^4 · ρ· < N_g, P_c > ·exp(−t_gc − t_gv);
Finally, get the overall light intensity at P_v
I″_v = I_gv + I′_v;

return I″_v;
}
```

$$I_{g\nu}(\lambda) = I_g \exp(-t(\overline{P_\nu P_g}, \lambda)) \; . \tag{10.11}$$

This equation describes the light contribution of the terrain without considering the light scattering along the distance $\overline{P_\nu P_g}$. Therefore, Eq. (10.8) has also to be taken into account

$$I''_\nu(\lambda) = I_{g\nu} + I'_\nu \; , \tag{10.12}$$

where I''_ν stands for the overall intensity reaching the observer's eye if he is looking at the rough surface of a planet. While Eq. (10.8) is implemented in terms of two texture fetches of the pre computed 3D texture, the easiest way to obtain I_g and $I_{g\nu}$ is to adopt the 2D lookup texture described in the work of O'Neil [117]. As this texture stores the optical depth for each point inside the atmosphere, the light attenuation between the light and the geometry can be fetched as well as the attenuation between the geometry and the observer. For the other parameters, shader constants are used, except for the phase function $F_r(\theta)$, which is pre computed as 1D texture. The pseudo fragment shader code in Algorithm 9 demonstrates the small number of instructions used for this complex computation.

Further Suggestions

This section discusses some practical issues when the 3D scattering texture is applied for rendering terrains. The main problem of this approach is its instability at the horizon of a scene. The reason lies in the pre computation, where for each position and view ray of the observer is decided whether the inner or the outer sphere should be intersected. This is because each of the observer's view rays either see the terrain, represented by the inner sphere, or the sky, but never both. At the horizon, we have a sampling problem: one time the view ray is tested for an intersection with the inner sphere, another time it is tested for an intersection with the outer sphere. This leads to flickering artifacts when the observer is moving over the terrain.

The first and simpler possibility for minimizing this problem is to generate two instead of one 3D lookup texture: one for the inner and one for the outer sphere. Then the light contribution values at the horizon can be pasted to the undefined entries, i.e., the cells in the 3D texture where the respective sphere isn't hit.

The second but more complex possibility is to use only one 3D texture. Let us consider an observer parameterized by the height and a view ray $\mathbf{R}_v$. Again, for the sake of simplicity, only plain spheres are considered. Then the light scattering between observer and ground can also be expressed by the pre computed values of the outer sphere. The procedure is simple: the scattering between the observer and the ground (inner sphere) is exactly the same as the scattering between the ground and the outer sphere minus the scattering between the observer and the outer sphere. For both cases the negative view ray $-\mathbf{R}_v$ must be used. This enables the pre computation of the scattering texture only for the outer sphere and flickering is avoided completely.

10.2.3 Planetary Terrain Rendering

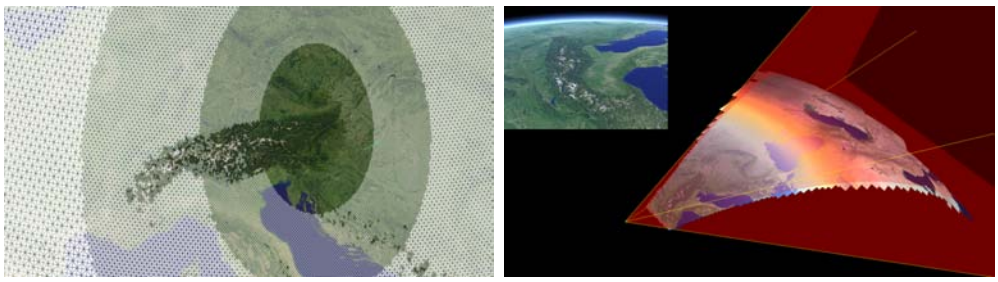
For this implementation, an existing terrain renderer implementation by Röttger et al. [142] has been extended for visualizing spherical objects.

One adaption consists of the dynamic mesh refinement (geomorphing). As defined in [142], the mesh refinement criterion f is defined by:

$$\mathrm{f} = \frac{\mathrm{l}}{\mathrm{d} \cdot \mathrm{C} \cdot \max(\mathrm{c} \cdot \mathrm{d2}, 1)} , \tag{10.13}$$

where l is the distance to the viewer and d is the length of the block (i.e., to the beginning the entire tile, then the block of the quadtree—see [142] for more details), which needs to be refined. The constant values C and c are standing for minimum global resolution and the desired global resolution. The surface roughness value is defined as $\mathrm{d2} = 1/\mathrm{d} \max|\mathrm{dh_i}|$, where $\mathrm{dh_i}$ is the elevation difference between two refinement levels. If $\mathrm{f} < \mathrm{l}$, the mesh needs to be refined. Applying this criterion to a spherical terrain representation, the viewpoint and the vertices need to be in the same coordinate system. To obtain a correct length l, the spherical transformation of the terrain needs to be taken into account (see Figure 10.9 (a)). Therefore, if we assume the position $\mathbf{P}_v$ of the viewer given in world space, an inverse spherical mapping of $\mathbf{P}_v$ is necessary.

For further optimization, spherical view frustum clipping has been implemented. The terrain consists of several tiles, which can differ in the resolution of their local



(a) (b)

Figure 10.9: (a) adaptive mesh refinement; (b) a two stage clipping algorithm is applied, the red planes represent the viewing frustum. First, whole tiles are clipped. Then, clipping is performed on a per vertex stage. Upper left: the corresponding image reaching the observer.

height field but not in their size in world space. Thus, a tile with a higher local resolution implies a higher level of refinement in world space. Due to their constant size these tiles are well suited as input for the clipping algorithm. The clipping algorithm consists of two stages: in the first stage, all non-visible tiles are clipped. This stage is accomplished before the mesh refinement by a comparison of the vector representing the view direction and the four vertices of the tile to be tested. If the cosine between those vectors is negative, the tile is visible. Thus, the first stage can be considered as crude back face culling, just working with whole terrain tiles. The second stage implements a smoother clipping taking into account the grid refinement. The quad tree (for further information see [142]) that holds the geometry is traversed down for testing each vertex that is situated in the center of the current quad against the four clipping planes. If the visibility test fails, the quad is clipped by removing the entry of the quad tree. Figure 10.9 (b) shows the two-stage clipping (center) and the resulting visualization (upper left).

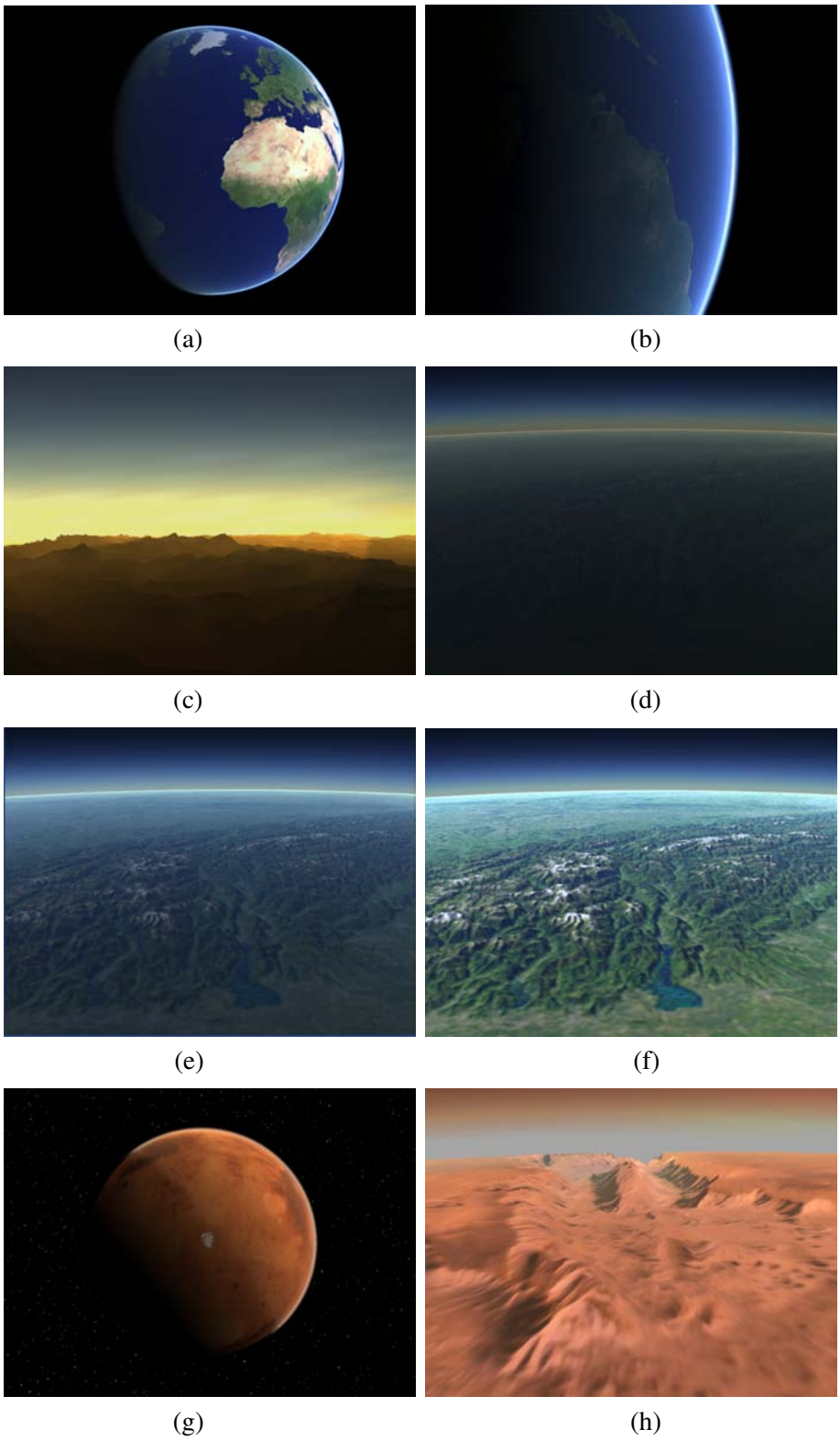
10.2.4 Performance Measurements and Examples

Due to its small number of real-time GPU instructions, this method outperforms previous ray tracing methods for atmospheric scattering. Drawing only the atmosphere, i.e., it is mapped onto two spheres with 128^2 vertices, the frame rate is measured with 752.79 fps for a NVDIA 8800 GTX graphics card. A comparable GPU-based implementation can be found in the GPUGems2 article by O'Neil [118]: here, approx. 151 fps are measured with the same configuration.

When terrain rendering is applied, the performance strongly depends on the efficiency of the underlying terrain rendering algorithm. Measuring two data sets, an Earth and a Mars data set with $257^2 \times 24 \times 12$ and $65^2 \times 24 \times 12$ data points, respectively, the frame rate for terrain rendering with atmospheric effects enabled varies between 51.38 fps and 90.32 fps for the Earth data set. Rendering the Mars data set is faster due to its lower resolution and lies between 71.79 fps and 85.44 fps on a machine with the standard configuration according to Section 1.5. The varying frame rate is tightly linked to the respective level of detail (LOD): the closer the camera is to the surface, the higher is the LOD. If only the atmosphere is rendered, i.e., only the two spheres of

128×128 vertices, 752.79 fps are achieved.

Figure 10.10 (a)–(c) show the earth data set. Daytime can be manipulated interactively, even when the camera approaches the planet. Figure 10.10 (c) shows the color shift shortly before sunrise. Figure 10.10 (d)(e)(f) show the Alps viewed from Italy at different day times, starting with the sunrise over to the early morning hours to midday. This sequence illustrates the dependency of the light scattering on the angle to the sun. Figure 10.10 (g) and (h) demonstrate the flexibility of the proposed method by replacing the Earth's atmosphere with the Martian atmosphere by simply modifying the molecular density and the wavelength of the incident light. Additional dust particles, which mainly influences the color of the Martian atmosphere, are unaccounted. Finally, (i) demonstrates how the spherical refinement mechanism works.



(a) (b)

(c) (d)

(e) (f)

(g) (h)

Figure 10.10: (a) the Earth viewed from space, (b) South America after sunrise, (c) sunrise over the Alps, (d) sunrise (e) early morning, (f) midday, (g) Mars viewed from space, (h) Valles Marineris with Martian atmosphere, (i) adaptive mesh refinement.

CHAPTER

11 NONLINEAR LIGHT TRACING

This chapter generalizes the ray tracing method used in Chapter 10 for the simulation of physical effects. For this purpose, the so far linear rays are from now on given as nonlinear functions. Gröller [52] first proposed a generic CPU implementation of a nonlinear ray tracer with various applications. The first GPU-based implementation was by Weiskopf et al. [194]. This algorithm builds the base of the following discussion and, therefore, it is introduced first. In the second part of this chapter, the generation of panorama maps as an application of non linear ray tracing is discussed.

11.1 Nonlinear Ray Tracing on the GPU

The following discussion denotes the rays started at the observer and penetrating the scene as *light rays*, because light is transported from object to the observer.

11.1.1 Algorithm

The algorithm of Weiskopf et al. [194] is built upon previous GPU-based implementations of a linear ray tracer [22; 132]. Basically, the $n_{rays} \times n_{objs}$ problem, i.e., n rays are intersecting n objects is extended to a $n_{rays} \times n_{objs} \times n_{segs}$ problem, where n_{segs} are the piecewise linear segments used for bending the rays. Here, nonlinearity is achieved by sampling an underlying function that influences the segments' direction.

Therefore, two loops are required: an inner loop, solving the $n_{rays} \times n_{objs}$ problem, and an outer loop, iterating over all n_{segs}. The algorithm works like follows: first, the rays are initialized by storing each ray in two textures, one holding the position and one holding the direction. This is done by rendering one quadrilateral that covers the image plane, using a pixel shader program for the computation of the initial directions.

Once both textures are initialized, the ray integration, i.e., the bending of the rays, is performed by evaluating a system of ODEs

$$\begin{aligned} \frac{d\mathbf{x}(\tau)}{d\tau} &= \boldsymbol{\nu}(\tau) \,, \\ \frac{d\boldsymbol{\nu}(\tau)}{d\tau} &= \mathbf{f}(\mathbf{x}(\tau), \boldsymbol{\nu}(\tau), ...) \,, \end{aligned} \tag{11.1}$$

where $\mathbf{x}$ is the position of the ray, $\boldsymbol{\nu}$ its direction, $\mathbf{f}$ a force function, and τ the parametrization of the ray. The dots represent possible additional parameters for the propagation

of light. Eq. (11.1) can be solved by explicit numerical integration schemes, like the first-order Euler integration,

$$\begin{aligned} \mathbf{x}_{i+1} &= h\mathbf{v}_i + \mathbf{x}_i \,, \\ \mathbf{v}_{i+1} &= h\mathbf{f}(\mathbf{x}_i, \mathbf{v}_i, ...) + \mathbf{v}_i \,, \end{aligned} \tag{11.2}$$

with the stepsize h and the index i for the points along the polygonal approximation of the light rays. At each integration step read access to the textures with index i and write access to the textures with index $i+1$ is required. After each integration step, the two copies are alternatingly exchanged in a ping-pong rendering scheme. The numerical operations for Eq. (11.2) are implemented in a pixel shader program that writes its results to multiple render targets, namely the $\mathbf{x}$ and $\mathbf{v}$ textures. Once again, the fragments are generated by rendering a single viewport-filling quadrilateral.

In the last step, the ray-object intersection is computed. A ray segment is defined by the line between $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$. The intersection test is performed for each ray segment individually according to the ray-triangle intersection by Carr et al. [22] and the ray-sphere intersection by Glassner [48]. During the ray-object intersection also local illumination is computed according to the Blinn-Phong [12] model.

Since the goal of GPU-based nonlinear ray tracing is a fast, preferably interactive representation, some acceleration techniques have been applied like *early ray termination* or adaptive step size. For a more detailed discussion see [194].

11.1.2 Applications

In [194] some examples of nonlinear ray tracing are given, mainly of the field of dynamical systems or astrophysics. Thereby only the input function $\mathbf{f}$ for solving Eq. (11.1) is exchanged. The most complex example consists of the visualization of black holes, applying the Schwarzschild metric to GPU-based nonlinear ray tracing. This example results in a long complex shader code for evaluating Eq. (11.2). A simpler example is the rendering of rays in a medium with varying index of refraction [206] or the motion in a radial potential fied [52].

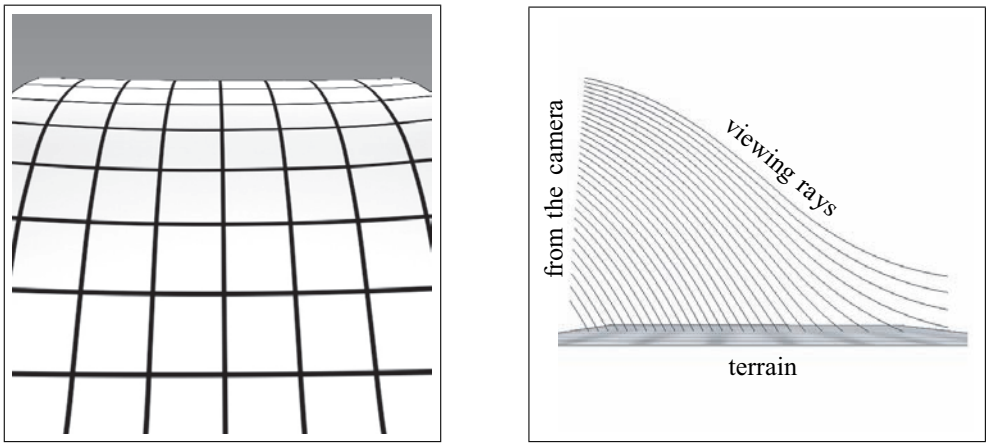
The next section discusses another application of nonlinear ray tracing: the generation of panorama maps.

11.2 Generation of Nonlinear Panorama Maps

So far, some examples of nonlinear ray tracing have been described. In contrast to those examples, the following discussion describes a more practical application of the algorithm of Section 11.1.1: the interactive generation of panorama maps according to [42]. This work was in collaboration with my colleague Martin Falk, who developed the panorama generation framework in terms of his study thesis and must be credited for the practical realization.

11.2.1 Requirements for Rendering Panorama Maps

When drawing panorama maps sooner or later the big question arises, ”how can all roads and trails be appropriately visualized in a single image without occlusion while



(a) (b)

Figure 11.1: (a) Progressive perspective is often used in panoramas to enhance the range of vision. The foreground is more map-like, the background appears to be more realistic. (b) The viewing rays are bent upwards towards the background.

the shape and the landscape are preserved?" One answer could be nonlinear projection, i.e., the manipulation of the human's eye projection to achieve the desired visualization. Also panorama maps make use of nonlinear projections. Traditionally, the generation of panorama maps is a time consuming process because they need to be hand-drawn by a specialist. And this might take years.

In the following, nonlinear ray tracing is used for the much faster, computer-aided generation of panorama maps. This doesn't substitute the specialist but it optimizes the duration of the generation process. This new technique follows the procedures of Berann's (the most popular visualizer of panorama maps) techniques [120] in wide parts. These are:

Orientation and perspective North orientation is not compulsory in a panorama map. The perspective improves realism.

Projection plane Berann tilted the landscape toward the horizon downwards, which combines the advantages of both the low and the high viewing point. Figure 11.1 shows an example.

Vertical exaggeration Vertical scaling must be applied to the elevation for an improved perception of small mountains.

Rotation and shifting mountains Panorama maps exhibit local changes in the topography of the landscape compared to topographic maps.

Generalization As oblique aerial photographs feature too much unwanted details, the landscape has to be more abstracted in a drawn panorama.

11.2.2 Modeling Ray Deflection

As already discussed in Chapter 10, ray tracing can be considered as a tracing of light particles along a given vector field. In contrast to linear ray light tracing, in the case of

nonlinear ray tracing the vector field changes; the rate of change depends on the second order derivative of the particle location: the force function $\mathbf{f}$. Starting with the initial situation that the vector field reflects the perspective of the human eye, the force field changes the direction, given by the vector field, of the moving particle. Keep in mind that the light particles are actually transported from the light source to the observer. However, the idea of ray tracing is to compute the path of each light particle starting from the observer to decrease the computational overhead caused by light particles that never arrive at the observer.

According to Eq. (11.1), ray bending can be expressed by an arbitrary function $\mathbf{f}$. For the generation of panorama maps $\mathbf{f}$ is defined by

$$\mathbf{f}(\mathbf{x}, \mathbf{v}) = g w_{\mathtt{dist}} w_{\mathtt{dir}} l(\mathbf{x}) \mathbf{F}(\mathbf{x}) \ , \tag{11.3}$$

where $w_{\mathtt{dir}}$ is a view dependent weight with $w_{\mathtt{dir}} = 1 - \cos^2\alpha$ with maximum $\pi/2$, when α is the angle between viewing direction and deflection, $w_{\mathtt{dist}}$ denotes a distance weight, g a global weight of $\mathbf{f}$, and $l(\mathbf{x})$ a local influence weight for masking purposes. In order to achieve a non-deflected area close to the viewer, the weights $w_{\mathtt{dir}}$ and $w_{\mathtt{dist}}$ are applied, where $w_{\mathtt{dist}}$ behaves proportional to the distance between the ray segment and the viewer.

In the following, two different approaches for the generation of $\mathbf{F}(\mathbf{x})$ are discussed.

Normal Map Approach

The first model is actually a 2D model for deflecting rays in a 3D domain. Here, the slope of the elevation data is considered by computing its first order derivative, resulting in the slope's normal. Since the normal is directly linked to the shape of the elevation, it can be used for a terrain-dependent repelling and attracting of the light ray.

Since only 2D height fields are processed by the framework, it makes sense to generate the normal field also in 2D, i.e., the normals are directly computed in the 2D domain. The terrain should serve as an attracting object, that leads $\mathbf{F}(\mathbf{x})$ to

$$\mathbf{F}(\mathbf{x}) = -\mathbf{n}\left(\mathbf{x}_{\text{proj}}\right) \ , \tag{11.4}$$

where $\mathbf{n}$ returns the normal at $\mathbf{x}_{\mathtt{proj}}$, which is the projection of $\mathbf{x}$ onto the xz-plane. The normals do not have to be scaled since the weighting is applied during ray integration according to Eq. (11.3).

The general approach for generating the normal map according to [42] is to consider the terrain from the bird's eye view. Orthographic projection is used to prevent perspective distortions which occur in a top view with perspective projection since the terrain is usually not flat. Figure 11.2 illustrates the projection process. If a 2D height field is available, the projection can be skipped.

Volumetric Approach

In the second model, a more global 3D deflection field is be created. Here, instead of only considering the local neighborhood of the projected position, all deflection forces are regarded. This is done by accumulating them to a single force.

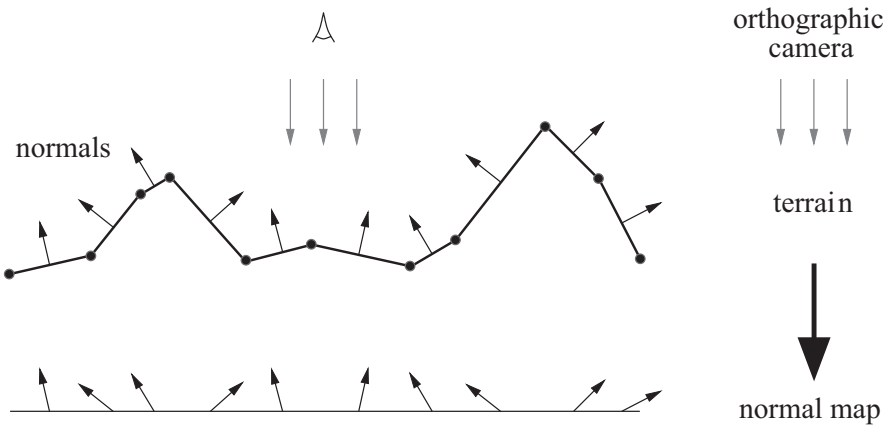


Figure 11.2: The normal map is received by a orthographic top view of the landscape considering only the triangle normals.

In the following, a 3D unform grid is used for representing the 3D deflection field. For each cell of the grid, the deflection potential of the whole terrain surface is considered. In practice, a loop is applied over all triangles of the terrain data in order to compute their influence on each grid cell. Similar to the normal map approach of Section 11.2.2, for the deflection force the negative surface normal is used. Therefore, the influence of a triangle to a grid cell is defined by

$$\mathbf{F}_i = -\mathbf{n}_i A_i \mathbf{f}_d(d_i/r) \,, \tag{11.5}$$

where $\mathbf{n}_i$ is the triangle's normal, A_i is the area of the triangle, f_d a linear, quadratic, or exponential fall-off function with distance between the triangle's center and the grid point's distance d_i and the radius of influence r as input parameters.

The calculation of the distance d_i between triangle i and the grid cell is done by a simple ray-plane intersection. Thereby the normal of the triangle serves as direction of the ray, the position of the grid cell as origin of the ray. If the intersection point lies outside the triangle, i.e., at least one barycentric coefficient is negative, the negative coefficients will be set to zero. Hence the corrected intersection lies on the border of the triangle. Triangles with a negative distance, i.e., triangles that face away from the cell center, are discarded. Finally the previously weighted forces $\mathbf{F}_i$ of all triangles are summed up and stored into the corresponding cells of the cartesian grid.

In Figure 11.3 the shortest distance d_2' between the second triangle and a cell is depicted in gray because the point of intersection does not lie inside the triangle. The distance d_2 is therefore measured between the cell center and the triangle border. The third and the fourth triangle are discarded since they are facing away.

Comparing Both Approaches

Both approaches, the normal map and the volumetric approach have their advantages and their disadvantages. While only the volumetric approach is able to consider overhanging rocks and caves, the normal map provides a much higher resolution. Both approaches differ substantially in terms of computational time and memory consumption. The normal map is created in a single render pass in a fraction of a second. In

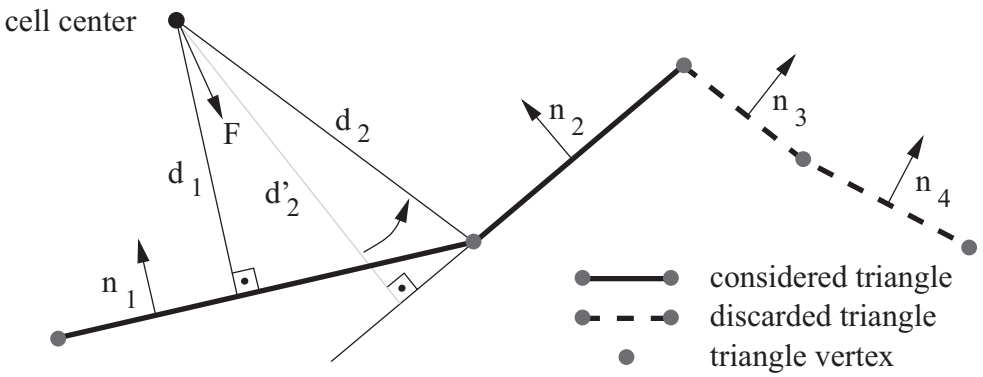


Figure 11.3: Computing the influence of four triangles illustrated by a cross-section of a mountain. Each triangle is weighted over the shortest distance towards the cell center. As the intersection of d_2' (gray) lies outside triangle 2, it is shifted onto the triangle border yielding d_2. Triangle 3 and 4 are neglected because they face away from the cell center.

contrast, the computation of a $64 \times 16 \times 64$ force field for 42,544 triangles of the Whistler data set took on our system, an AMD64 X2 Dual Core 4600+, 15 seconds for a radius of 6 voxels and 1 minute 24 seconds for a radius of 16 voxels. For the data set of Kronplatz with 6,400 triangles, it took 3.7 seconds and 19 seconds respectively.

The choice whether the normal map or the volumetric approach is more suitable depends on the terrain itself. As the normal map approach directly employs the normals of the terrain, it emphasizes subtle and local structures, which could lead to unwanted ray deflection in data sets covering vast areas. Therefore, the volumetric approach is more applicable for panorama maps of large scale landscapes such as large regions or even countries. The normal map should rather be used for small sceneries like ski maps covering only a few mountains or single mountain ranges.

11.2.3 Generating Nonlinear Panorama Maps

Since the process of generating panorama maps is quite complex, this method cannot claim to perform this in an automatic manner. Therefore, still an experienced user is required. However, this method can claim to make this process faster, by providing a tool that releases the user completely from drawing, which is the most time consuming part of the process. This makes it absolutely important to provide more than the basic algorithm of Section 11.2.2; a framework for the interactive generation of nonlinear panorama maps is desirable.

Our original work [42] discusses the details of the framework; in this thesis, only a brief overview of the interaction mechanisms is given. Considering the local influence parameter $l(\mathbf{x})$ of Eq. (11.3) again, it should be mentioned that the user is able to modify this parameter interactively. This is done by drawing in a 2D texture in the case a normal map is used, and in a 3D texture when the volumetric approach is used. The drawing is done by mouse interaction, which can be used to mark regions on a 2D plane within the graphical representation. This drawing plane is represented by a xz plane, that can be moved along the z-axis in the case of the volumetric approach. The

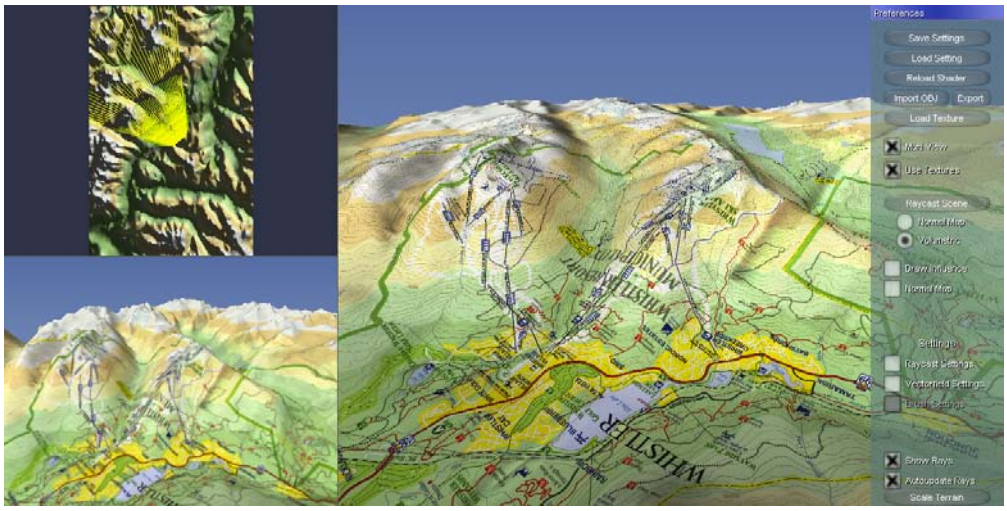


Figure 11.4: Linked multiple views showing Whistler, Canada: top view with curved rays (top left), undistorted view (bottom left), and scene with non-linear perspective (right).

parameter $l(\mathbf{x})$ consists of a horizontal and a vertical component, whereby the vertical component also enables the negation of vertical forces. This parameter, representing the local influence, can be used to assign the rate of influence of each individual structure of the terrain.

Flat mountains are often the unmeant result of nonlinear ray tracing and can be ascribed to the vertical deflection of the light rays. Since an unmeant vertical deflection can be triggered almost everywhere inside the domain, often at locations far away from the intersection where the local vertical deflection is meaningful, this issue is solved by enabling vertical scaling. Again, the user draws interactively into a 2D scaling texture which is used to scale the y component of the terrain up to 1.5 times the original height. Initially flat regions can be exaggerated to appear more realistic.

The feedback of the user interactions is delivered within a multiple view representation: the painting texture is displayed in combination with the semi-opaque terrain and the bent rays can be visualized additionally to gain knowledge about the ray trajectories.

Figure 11.4 shows the multiple view representation. On the upper left corner, a top view of the bent rays is drawn. In the lower left corner, the scene is rendered with linear projection, i.e., the original scene without any manipulations. The actual result is shown on the right hand side. Additional preferences can be set in the window at the right border.

Figure 11.5 shows the nonlinear panorama map of Whistler obtained with the framework. For a better comparison, the boxes show the original part of the image with linear perspective projection. The progressive perspective leads the mountains in the background to appear smaller without losing the impression of their shape while the valley in the foreground appears magnified. Furthermore, horizontal deflection leads to straightened valleys, while vertical deflection reveals important objects to become visible, e.g., lakes or hidden trails.

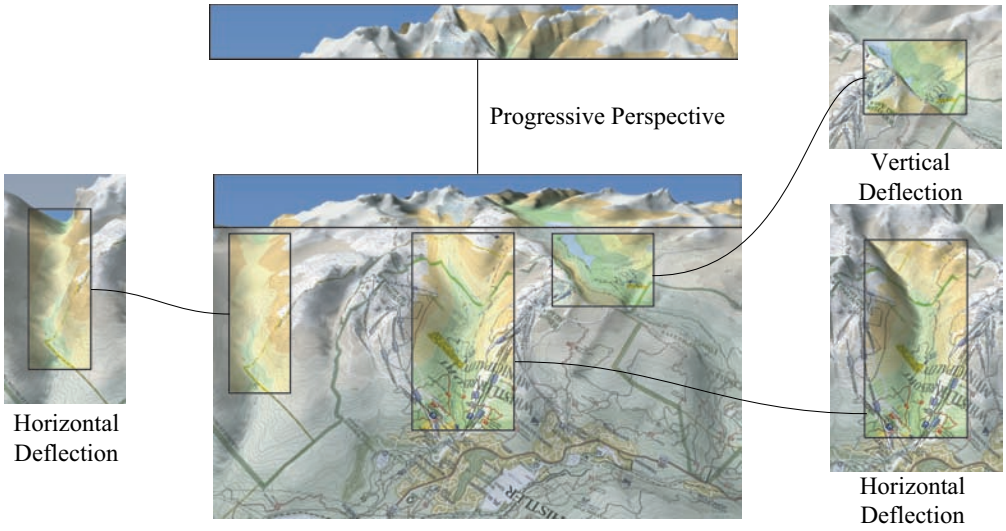


Figure 11.5: Effects of non-linear ray tracing. The boxes outside show the same detail without ray deflection. The valleys are both straightened by the use of horizontal ray deflection to lessen occlusion. The lake on the right becomes more visible through vertical deflection. To flatten the mountain ridges in the background a progressive perspective is used.

CHAPTER

12 CONCLUSION

In this thesis, it has been shown that particle tracing is the crucial part of a wide number of algorithms, whether they are used for visualization or computer graphics. First it should be mentioned that in the last four years when the algorithms of this thesis were developed, graphics hardware has changed immensely—keep in mind that the performance for GPUs has been growing much faster than for CPUs predicted by Moore's law. This behavior is also reflected by the algorithms proposed in this thesis: while the 3D texture advection method in Section 5.1 was originally developed for the ATI X800 series, which only supports vertex and fragment shader 2.0 (no loops, very limited number of instructions, etc.), the shader used for path surface generation exploit vertex and fragment 3.0 features like loops and branching. Furthermore, it should be mentioned that all GPU-based algorithms in this work, without any exception, could also be implemented on the CPU; the issue is to clarify whether a GPU-based implementation is more efficient than a CPU implementation. In the most cases, the coauthors and me had to balance between accuracy and speed. Up to now, GPUs are not able to perform *double* accuracy (64 bits) efficiently, i.e., GPUs are used for that computations where *float* accuracy (32 bits) leads to sufficient results. That makes this thesis to become a mixture of GPU-based and CPU-based algorithms.

The organization of this thesis according to the flow diagram in Figure 12.1 also reflects the contributions of this work. The method of tracing particles along a given direction builds the focus of all proposed algorithms, which are illustrated by the boxes in the diagram. The ellipsoidal elements contain the attributes that are used to distinguish the different methods. The first question is if the particles are traced along a vector field (as it may result from simulations, measurements, or any other computations) or if only a scalar field is given. Please note that in this diagram the term "scalar field" also includes any position-dependent function. However, in most cases, the input scalar field builds at least the potential for a vector field, which is eventually used for particle tracing.

We are first following the left branch, where a vector field already exists. Adopting the concept of volume visualization, where one must distinguish direct volume rendering, which is mostly used for a dense representation of the 3D domain, and indirect volume rendering, which extracts a specific isosurface in terms of an isovalue, we need to decide between a dense representation of the entire volume and the extraction of features. Line Integral Convolution, shortly LIC, is referred to as a dense representation and can be applied on 2D and 3D domains. As in the example of a 2D application, in Section 6.3, a method that applies LIC on extracted vortex isosurfaces has been discussed. Particle integration is performed in a 2D/3D hybrid domain, where virtual

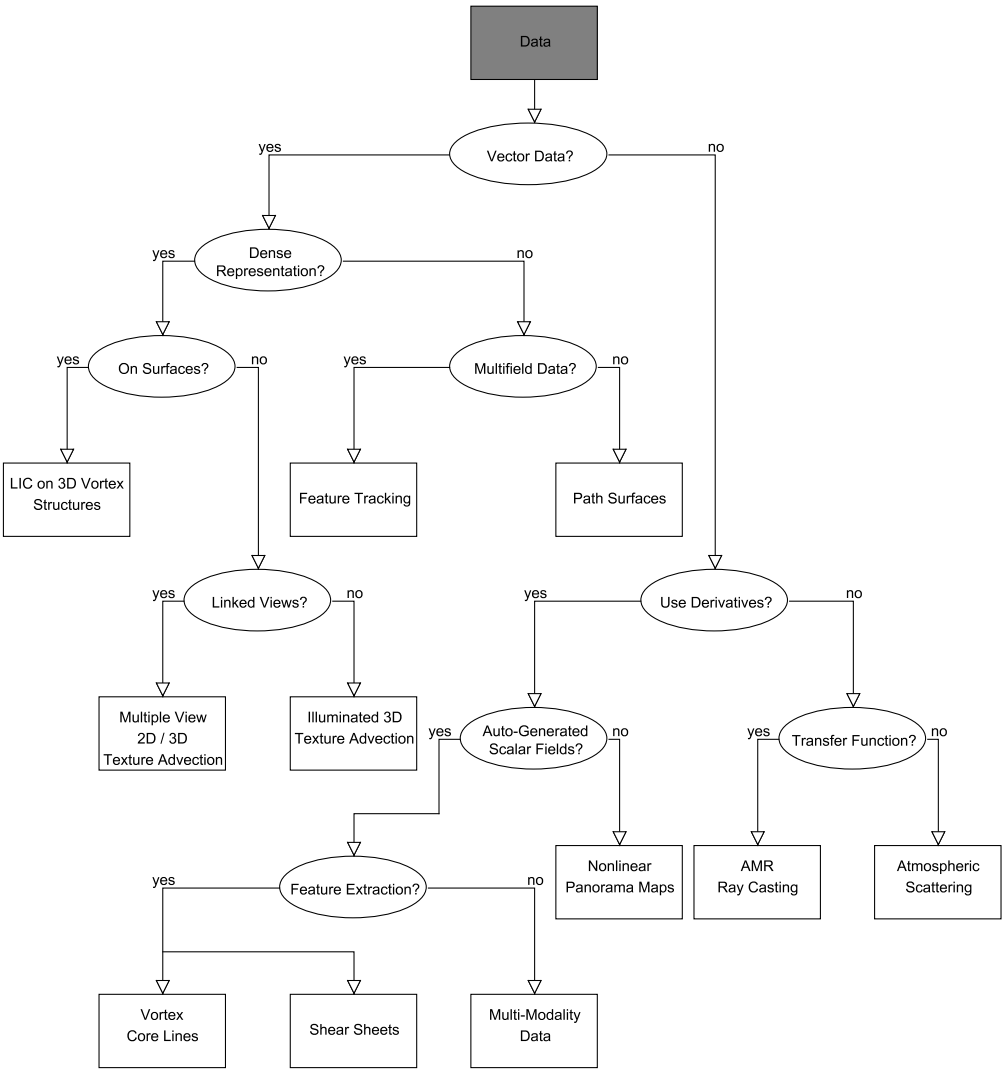


Figure 12.1: Flow diagram illustrating the flexibility of particle tracing approaches.

particles are moved along the projected vector field. Actually, this method manipulates the original vector field by projecting it onto the vortex surface. However, the aim of this method is to show the direction of rotation of a vortex structure and, therefore, the motion perpendicular to the vortex boundary represented by a λ_2 isosurface can be neglected.

Adopting this method for 3D applications and replacing the computationally expensive LIC integral by a step-by-step evaluation results in 3D texture advection, where particle tracing is of a slightly different nature. Here, for each grid cell inside a 3D domain, only one integration step is performed in order to obtain the density value that reaches a certain grid cell. The domain is discretized according to the Eulerian point of view, but the Lagrangian particle tracing remains. Applying this approach results in a dense representation of moving particles that appear in the form of streak lines. Due to its nature, this approach always computes the particle motion inside the complete 3D domain. This makes it well suited for 3D domains, though it is too expen-

sive for objects of codimension > 0. Section 5.1 has shown how 3D texture advection can be efficiently implemented on the GPU. Here, illumination plays a crucial role for a better perception and, therefore, the texture advection approach is enriched by an on-the-fly gradient computation.

Unfortunately, since the visualization of 3D texture advection is done by direct volume rendering, this approach also suffers from occlusion. Section 5.3 describes how to solve this issue by multiple linked views where a 2D and a 3D representation of a flow data set are combined. Both the 2D and the 3D representation are rendered by texture advection in combination with color coding, which represents the selected isoslabs, i.e., intervals of isovalues. The two-dimensional representation allows the user to interactively select an isovalue or an interval (isoslab) of a related scalar field representing a flow feature of the data set (velocity magnitude or λ_2 vortex strength). Keep in mind that particle integration is only performed once for both the 2D and the 3D representation together. Only single slices of the data set are rendered twice. Though this approach provides more flexibility for user interaction in contrast to the pure 3D texture advection, it suffers in its performance because of the multiple rendering passes.

If no dense representation is desired, the vector field can be used for extracting and tracking spatial structures. Note that this diagram distinguishes between flow features which are computed from derived scalar fields, and flow structures which are computed directly from the vector field. The question that arises here is if multifield data is available or not. If not, the vector field can be used to compute stream or path surfaces, depending on whether a vector field is time-dependent or not. Section 6.1 describes how the GPU can be used to enable an interactive computation of path surfaces. Here, interactivity plays a crucial role for the investigation of a flow field in order to find vortex structures. Particle tracing is performed twice: first, a set of particles is started at a seed line and integrated along the time-dependent vector field. Between each integration step, additional tests are performed, e.g., whether particles need to be inserted or removed to guarantee an appropriate sampling of the surface; second, particle tracing is performed again to compute LIC on the extracted path surface. This additional rendering pass makes sense because it enables a visual insight into the trajectories of the single particles and, therefore, helps to reveal the reason why a surface becomes wider or tighter. Another scheme that is quite similar to the generation of path surfaces is used for feature tracking. Let us assume that we have an extracted flow feature, i.e., a 2D surface embedded in a 3D domain enclosing the feature region and a 1D representation like vortex core lines or extremum lines of maximal shear. According to the algorithm in Section 9.1, those line representations are used for seeding particles that are, again, moved along the vector field. If the particles hit a structure at the following time, it is assigned to the structure inside which the particle has been started. However, this process requires multifield data because flow features need to be extracted first before they can be tracked.

Let us consider the branch on the right hand side that considers the particle tracing mechanisms that actually have only a scalar field for input. The first and most crucial question clarifies if the vector field required for particle tracing is derived from the scalar field. Note that first order derivatives serve as gradients while the Hessian can be used for computing the second order derivatives along a given direction. The second order derivatives stand for the change of the first order derivatives and result in

a force field; those fields come into play in the case of nonlinear ray tracing. Furthermore, we distinguish between potentials that are completely automatically generated and those which can be manipulated by the user. An example of a user-manipulated potential consists of the gravity field used for the generation of panorama maps. The algorithm in Section 11.1 shows how rays are bent to achieve a nonlinear projection of a panorama. Basically, the light particles (the photons) follow a trajectory which is computed by the current velocity (and direction) and the acceleration impacting the particle. The new velocity is determined by the old velocity and the acceleration, whereby the acceleration is defined by the gravity of the terrain. Therefore, for the computation only the starting velocity and the force field need to be known to perform the integration along the bent view ray.

In the case of flow feature detection the potential results from a flow feature computation on the flow field. Both methods presented, the 1D vortex core lines in Section 8.2 and the 2D shear sheets in Section 8.3, depend on the relation between the first and the second order derivatives of a flow feature field. However, both metods generate their own vector fields in order to extract the desired feature. While for vortex core lines the vector field consists of the eigenvector of the second order derivative that points in the direction of the next local λ_2 minimum, the vector field for the shear sheets is used for finding the direction where the shear value decreases most. For vortex core lines, a classical Lagrangian particle tracing is performed; the resulting stream lines represent the vortex core lines. For shear sheets, first particles are seeded around a center and are integrated in a second step along the given vector field.

If no derivatives of the scalar field are used, the particle traces need to be described by another criterion. If the view rays are considered as paths for light particles, this method is known as ray tracing; if no secondary rays are considered, as ray casting. Ray casting is an alternative method for evaluating the volume rendering integral of Eq. 1.2 and, therefore, it is a common method for the visualization of a scalar function. In direct volume visualization, the mapping of the derived gray values to colors is achieved by a transfer function, which is also used for ray casting of adaptive mesh refinement data (AMR). Here, the particle paths are also given by the perspective of the human eye and penetrate an AMR mesh. In Section 10.1, a GPU-based algorithm for the interactive ray casting of AMR grids is discussed. This method enables the user to investigate quantities that are, beside the vector field, also relevant for flow visualization like pressure, temperature, vorticity magnitude, etc. In contrast to AMR ray casting, for the rendering of light scattering no transfer function is needed. Here, the particle path is not only given by the perspective projection, it is also given by the connection between a sample point on the ray and the light source and represents the light particle that first leaves the light source to be attenuated twice by aerosols and air molecules: first from the sun to the sample point and then on its way to the observer. Actually, this algorithm performs iterative ray casting. As already mentioned above, no transfer function is required because the color shift is computed for each wavelength independently. Section 10.2 gives more insight in the fast computation of atmospheric effects caused by light scattering.

Let us return to the statement which was made at the beginning of this thesis: “A wide range of problems in visualization and computer graphics, which apparently have no commonness, can be still reduced to a common denominator: the motion of massless particles”. Considering Figure 12.1 again, we can see that indeed particle

tracing plays a crucial role in many algorithms for visualization and computer graphics and, therefore, the statement above can be approved. However, a critical comparison of pros and cons of the application of particle tracing is necessary. Some of the most important points are:

pros	cons
+ easy to implement	- inaccuracies due to pre computations
+ well-suited for GPU	- often possible, but not always most efficient
+ flexible	- knowledge of seed points required

Starting with the pros, typically the implementation of particle tracing is rather easy. From the practical point of view, one can implement a generic particle tracing module, which uses a vector field as input and can be applied to different kinds of applications. Here, the difficulty lies more in the preprocessing of the actual input data, i.e., to bring it in an appropriate form. An example can be found in Section 8.3, where the visualization of shear stress is discussed. The preprocessing step also includes the formal definition of the problem: we need a vector field for the particle integration, which is used to correct the position of a particle to be part of a shear sheet. Formally this is expressed in Eq. (8.1), which defines a shear sheet as 2D height ridge of I_2. The question is still the same. How can we generate a vector field out of the information we have? The flow expert would wonder why it is absolutely necessary to solve this problem using a vector field, but this issue will be discussed later. In Section 8.3, this problem was solved by pre computing the eigenvector ϵ_1 of the Hessian of I_2. This results in a vector field consisting of eigenvectors, which are, per definition, not aligned. This can create a number of unmeant critical points and, therefore, the vector field must be further processed. Finally, at some point, the vector field is stored into a texture and at last we are able to run our general particle tracing module.

This example should just clarify the effort of preparing the input data to be ready for particle tracing, while the implementation of the particle tracing module itself is the simplest part in this process. This leads us to two points on the cons side of the table. First, this preparation process requires a lot of pre computations, which often need to be cached and, therefore, might lead to inaccuracies due to numerical errors, discretization, and interpolation. Second, the question might arise if particle tracing is always the most efficient way to solve a problem. The answer must be: naturally not. Revisiting the flow expert again, he might suggest to generate an isosurface out of Eq. (8.1) or to apply local differential operators on the derivations of a λ_2 scalar field in order to detect vortex core lines, instead of tracing particles along a pre computed vector field. Without going into detail here, but the reader can see that the simple discussion on the implementation of a problem can lead to a more general discussion on a more abstract level–in this case, the discussion on the effectiveness and efficiency of local or global methods.

Particle tracing is well suited for a parallel implementation and, therefore, for a realization on the GPU, as demonstrated in various examples within this thesis. Path surfaces, for example, can only be computed interactively because of the exploita-

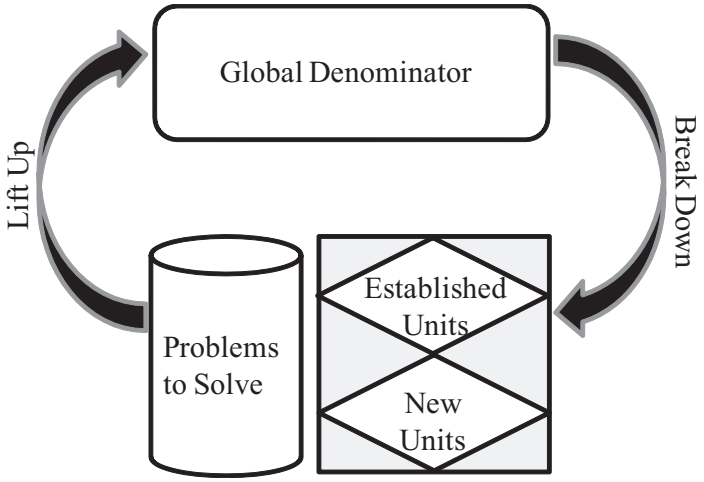


Figure 12.2: The development process exploiting an abstract intermediate representation: problems are lifted to an abstract level where a common nominator has been found. The realization is in form of breaking down the abstract problem to individual units, which are either established methods–existing knowledge can be exploited–or need to be developed newly.

tion of the parallel units on the GPU. However, this benefit becomes useless when the seed points of the particles are not known. Disregarding dense vector field representations, where the particles are distributed in the whole domain according to any specific conditions, the computation of appropriate seed points can be very expensive. Lines of maximal shear stress (in Section 8.3.2), for example, are computed by integrating along the eigenvector field of the Hessian of I_2. Points within the domain can be classified as seed points by a local consideration: if a point is a 3D maximum, then it must be part of the line and, therefore, can be serve as seed point. However, considering the generation of topology-preserving vortex core lines (in Section 8.2.2), the seed point is computed by searching on a plane perpendicular to the λ_2 isoskeleton, which is a global, more expensive approach. This makes the particle approach–in this case–completely uninteresting for an interactive computation and visualization of vortex core lines of time-dependent vector fields. However, in the case of computing vortex core lines for a single time step, the duration of finding the seed points does not matter what makes this method a very attractive alternative to local methods like the parallel vectors operator [121].

Last but not least there remains the flexibility of particle tracing, which was demonstrated very detailed in this thesis in terms of examples and, therefore, does not have to be discussed again.

At this point, the methods proposed in this thesis are summarized with respect to the foregoing discussion by describing the lessons learnt and the actual contributions to computer graphics, considering this work on a more abstract level. In my opinion, the different projects within this thesis show that the correct strategy to solve a problem is not the consideration if the CPU or the GPU is more appropriate to enable a fast representation–it is more the transformation of the respective problem to a more abstract level (like the ODE for Lagrangian particle tracing). This abstract level can

be considered as the “common denominator”, which was already mentioned in the introduction of this thesis. The advantage of this abstract level is that it can be broken down to individual units, whereby each of them solves a specific problem. This can be done on the theoretical as well as on the implementation level. Then most of those units can be replaced by established and well-investigated methods; on the implementation level, where those units appear as algorithms, the question if a module should be realized on the CPU or on the highly parallel GPU becomes unnecessary in many cases. Figure 12.2 illustrates this concept.

This work realizes this concept in the following way: the common denominator has been found in the Lagrangian ODE for particle tracing; the input must be a vector field; Additionally, it should be possible to attach scalar fields to the vector field, which are evaluated during integration (e.g., for ray casting). These are the ingredients for our concept: the input data and the knowledge that particle tracing needs to be performed. Now the breaking-down step consists of a split-up into established units and units which have to be newly developed. The established unit can be the GPU-based particle tracer or the rendering module; a new unit can be the computation of a vector field according to a specific equation. This concept shows how existing experience can be exploited and the research process can be more focused on the fundamental parts.

The area of research (in terms of visualization and computer graphics) where particle tracing is employed most was and will be flow visualization. But new insights in the field of fluid dynamics will also give the opportunity to gain new insights due to flow visualization–and again particle tracing will play a crucial role. Considering the field of flow features, the research beyond the detection of vortices is just at the beginning. New feature criteria will be found and new flow structures will be extracted by particle tracing. In this work, some methods for an appropriate representation of vortices and shear layers have been proposed and also the temporal change of these quantities has been investigated. However, the interaction between shear layers and vortices is not fully explored and, therefore, feature tracking will become an important technique for getting new insights. Also the topological changes of flow features are an interesting topic in research. Maybe the question how a vortex changes in its topology, i.e., if it merges or splits, can be answered in the near future. This makes it necessary to find new methods for tracking those structures using continuous vector fields—particle tracing will always be a part of the development process.

BIBLIOGRAPHY

[1] R. Adrian. Hairpin vortex organization in wall turbulence. *Physics of Fluids*, 19:041301, 2007.

[2] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *IEEE Visualization 2001*, pages 21–28, 2001.

[3] A. Babucke, M. Kloker, and U. Rist. Numerical investigation of flow-induced noise generation at the nozzle end of jet engines. *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, 96:413–420, 2007.

[4] S. Bachthaler, M. Strengert, D. Weiskopf, and T. Ertl. Parallel Texture-Based Vector Field Visualization on Curved Surfaces Using GPU Cluster Computers. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06)*, pages 75–82, 2006.

[5] S. Bake, D. Meyer, and U. Rist. Turbulence mechanism in Klebanoff-transition. a quantitative comparison of experiment and direct numerical simulation. *Journal of Fluid Mechanics*, 459:217–243, 2002.

[6] D. C. Banks. Illumination in diverse codimensions. In *Proc. ACM SIGGRAPH 1994*, pages 327–334, 1994.

[7] D. C. Banks and B. A. Singer. Vortex tubes in turbulent flows: identification, representation, reconstruction. In *Proc. IEEE Visualization '94*, pages 132–139, 1994.

[8] D. C. Banks and B. A. Singer. A predictor-corrector technique for visualizing unsteady flow. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):151–163, 1995.

[9] R. A. Becker and W. S. Cleveland. Brushing scatterplots. *Technometrics*, 29(2):127–142, 1987.

[10] D. Benson and J. Davis. Octree Textures. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 785–790, 2002.

[11] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53(3):484–512, Mar. 1984.

[12] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198. ACM Press, 1977.

[13] R. P. Botchen, D. Weiskopf, and T. Ertl. Texture-Based Visualization of Uncertainty in Flow Fields. In *Proceedings of IEEE Visualization '05*, pages 647–654, 2005.

[14] R. P. Botchen, D. Weiskopf, and T. Ertl. Interactive Visualization of Uncertainty in Flow Fields using Texture-Based Techniques. In *Electronic Proceedings International Symposium on Flow Visualization '06*, page Electronic, 2006.

[15] N. Boukhelifa and P. J. Rodgers. A model and software system for coordinated and multiple views in exploratory visualization. *Information Visualization*, 2(4):258–269, 2003.

[16] S. Bruckner and E. Gröller. VolumeShop: An interactive system for direct volume illustration. In *Proc. IEEE Visualization*, pages 671–678, 2005.

[17] A. Buja, J. A. McDonald, J. Michalak, and W. Stuetzle. Interactive data visualization using focusing and linking. In *Proc. IEEE Visualization*, pages 156–163, 1991.

[18] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, 1994.

[19] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *Proc. ACM SIGGRAPH*, pages 263–270, 1993.

[20] W. Cai and P.-A. Heng. Principal stream surfaces. In *IEEE Visualization 1997*, pages 75–80, 1997.

[21] W. Cai and G. Sakas. Data intermixing and multi-volume rendering. *Computer Graphics Forum*, 18(3):359–368, 1999.

[22] N. Carr, J. Hall, and J. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, 2002.

[23] P. Chakraborty and S. Balachandar. Comment on "axial stretching and vortex definition". *Physics of Fluids*, 18:029101, 2006.

[24] P. Chakraborty, S. Balachandar, and R. Adrian. On the relationships between local vortex identification schemes. *Journal of Fluid Mechanics*, 535:189–214, 2005.

[25] E. H. Chi, J. Riedl, P. Barry, and J. Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, 4:30–38, 1998.

[26] M. S. Chong, A. E. Perry, and B. J. Cantwell. A general classification of three-dimensional flow fields. *Physics of Fluids*, 2:765–777, 1990.

[27] W. S. Cleveland. *The Elements of Graphing Data*. Wadsworth, Monterey, California, 1985.

[28] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, 1982.

[29] N. D. Cornea, D. Silver, X. Yuan, and R. Balasubramanian. Computing hierarchical curve-skeletons of 3D objects. *The Visual Computer*, 21(11):945–955, 2005.

[30] W. Cornette and J. Shanks. Physical reasonable analytic expression for the single-scattering phase function. *Applied Optics*, 31(16):3152–3160, 1992.

[31] N. Corporation. Nvidia Opengl extension specifications webpage, 2004.

[32] N. Corporation. Nvidia CUDA programming guide, 2008.

[33] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical report, 1994.

[34] Y. Dobashi, T. Yamamoto, and T. Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 99–107, 2002.

[35] H. Doleisch, M. Gasser, and H. Hauser. Interactive feature specification for focus+context visualization of complex simulation data. In *Proc. EG / IEEE TCVG Symposium on Visualization*, pages 239–248, 2003.

[36] H. Doleisch and H. Hauser. Smooth brushing for focus+context visualization of simulation data in 3D. In *Proc. WSCG*, pages 147–155, 2002.

[37] D. Eberly. *Ridges in Image and Data Analysis. Computational Imaging and Vision*. Kluver Academic Publishers, 1996.

[38] D. Ebert and P. Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proc. IEEE Visualization*, pages 195–202, 2000.

[39] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. A K Peters, 2006.

[40] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. SIGGRAPH / EG Workshop on Graphics Hardware*, pages 9–16, 2001.

[41] G. Erlebacher, B. Jobard, and D. Weiskopf. Flow textures: High-resolution flow visualization. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*, pages 279–293. Elsevier, Amsterdam, 2005.

[42] M. Falk and D. Weiskopf. Output-sensitive 3D line integral convolution. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):820–834, 2008.

[43] T. Fangmeier, M. Knauff, C. C. Ruff, and V. Sloutsky. fMRI evidence for a three-stage model of deductive reasoning. *Journal of Cognitive Neuroscience (in press)*, 18(3):320–334, 2006.

[44] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[45] J. H. Ferziger and M. Perić. *Computational methods for fluid dynamics*. Springer, Berlin, 1999.

[46] T. Frühauf. Raycasting vector fields. In *IEEE Visualization 1996*, pages 115–120, 1996.

[47] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann. Surface techniques for vortex visualization. In *EG/IEEE VGTC Symposium on Visualization (VisSym 2004)*, pages 155–164, 2004.

[48] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, 4 edition, 1993.

[49] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452, 1998.

[50] R. Griffiths and E. Hopfinger. Coalescing of geostrophic vortices. *Journal of Fluid Mechanics*, 178:73–97, 1987.

[51] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. Flexible direct multi-volume rendering in interactive scenes. In *Proceedings of Vision, Modeling, and Visualization*, pages 386–379, 2004.

[52] E. Gröller. Nonlinear ray tracing. visualizing strange worlds. *The Visual Computer*, 11(5):263–276, 1995.

[53] S. Guthe, S. Gumhold, and W. Straßer. Interactive visualization of volumetric vector fields using texture based particles. In *Proc. WSCG*, pages 33–41, 2002.

[54] R. Haimes and D. Kenwright. On the velocity gradient tensor and fluid feature extraction. In *Proc. AIAA 14th Computational Fluid Dynamics Conference*, pages 3288–3297, 1999.

[55] G. Haller. An objective definition of a vortex. *Journal of Fluid Mechanics*, 525:1–26, 2005.

[56] R. M. Haralick. Ridges and valleys on digital images. *Computer Vision, Graphics, and Image Processing*, 22(1):28–38, 1983.

[57] M. S. Hassouna and A. A. Farag. On the extraction of curve skeletons using gradient vector flow. *Proc. ICCV 2007*, pages 1–8, 2007.

[58] A. Helgeland and O. Andreassen. Visualization of vector fields using seed LIC and volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):673–682, 2004.

[59] C.-M. Ho and P. Huerre. Perturbed free shear layers. *Ann. Rev. Fluid Mech.*, 16:365–424, 1984.

[60] N. Hoffman and A. Preetham. Real-time light-atmosphere interactions for outdoor scenes. *Graphics programming methods*, pages 337–352, 2003.

[61] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *ACM SIGGRAPH 1992 Conference*, pages 71–78, 1992.

[62] J. P. M. Hultquist. Constructing stream surfaces in steady 3D vector fields. In *IEEE Visualization 1992*, pages 171–178, 1992.

[63] J. C. R. Hunt, A. A. Wray, and P. Moin. Eddies, stream, and convergence zones in turbulent flows. Technical Report CTR-S88, 1988.

[64] V. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In *Proc. ACM SIGGRAPH*, pages 109–116, 1997.

[65] V. Interrante and C. Grosch. Strategies for effectively visualizing 3D flow with volume LIC. In *Proc. IEEE Visualization*, pages 421–424, 1997.

[66] H. Jensen, F. Durand, J. Dorsey, M. Stark, P. Shirley, and S. Premoze. A physically-based night sky model. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 399–408, 2001.

[67] J. Jeong and F. Hussain. On the identification of a vortex. *J. Fluid Mech.*, 285:69–94, 1995.

[68] M. Jiang, R. Machiraju, and D. Thompson. Detection and visualization of vortices. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*, pages 295–309. Elsevier, Amsterdam, 2005.

[69] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Hardware-accelerated texture advection for unsteady flow visualization. In *Proc. IEEE Visualization*, pages 155–162, 2000.

[70] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Lagrangian-Eulerian advection for unsteady flow visualization. In *Proc. IEEE Visualization*, pages 53–60, 2001.

[71] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Lagrangian-Eulerian advection of noise and dye textures for unsteady flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):211–222, 2002.

[72] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proc. EG Workshop on Visualization in Scientific Computing*, pages 43–56, 1997.

[73] C. Josserand and M. Rossi. The merging of two co-rotating vortices: a numerical study. *European Journal of Mechanics B/Fluids*, 26:779–794, 2007.

[74] C. Josserand and M. Rossi. The merging of two co-rotating vortices: a numerical study. *European Journal of Mechanics B/Fluids*, 26:779–794, 2007.

[75] R. Kähler and H.-C. Hege. Texture-based volume rendering of adaptive mesh refinement data. *The Visual Computer*, 18(8):491–492, 2002.

[76] J. T. Kajiya and B. P. V. Herzen. Ray tracing volume densities. *ACM SIGGRAPH Computer Graphics (Proceedings of SIGGRAPH Š84)*, 18(3):165–174, 1984.

[77] E. R. Kandel, J. H. Schwartz, and T. M. Jessell, editors. *Essentials of Neural Science and Behavior*. Appleton & Lange, Norwalk, 1995.

[78] A. Kaufman and K. Mueller. Overview of volume rendering. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*, pages 127–174. Elsevier, 2005.

[79] R. Kelso, T. Lim, and A. Perry. An experimental study of a round jet in a cross-flow. *Journal of Fluid Mechanics*, 306, 1996.

[80] I. Kenneth E. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, 1999.

[81] S. Kida and M. Takaoka. Vortex reconnection. *Annual Review of Fluid Mechanics*, 26:169–189, 1994.

[82] S. Kida, M. Takaoka, and F. Hussain. Collision of two vortex rings. *Journal of Fluid Mechanics*, 230:583–646, 1991.

[83] P. Kipfer, M. Segal, and R. Westermann. Uberflow: a GPU-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, 2004.

[84] 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.

[85] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[86] J. J. Koenderink. *Solid Shape*. MIT Press, 1990.

[87] V. Kolar. Vortex identification: New requirements and limitations. *International Journal of Heat and Fluid Flow*, 28:638–652, 2007.

[88] R. Kosara, G. N. Sahling, and H. Hauser. Linking scientific and information visualization with interactive 3D scatterplots. In *Proc. WSCG Short Communication Papers*, pages 133–140, 2004.

[89] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A particle system for interactive visualization of 3D flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[90] R. Laramee, G. Erlebacher, C. Garth, T. Schafhitzel, H. Theisel, X. Tricoche, T. Weinkauf, and D. Weiskopf. Applications of Texture-Based Flow Visualization. *Engineering Applications of Computational Fluid Mechanics*, 2(3):264–274, 2008.

[91] R. S. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. H. Post, and D. Weiskopf. The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum*, 23(2):143–161, 2004.

[92] R. S. Laramee, B. Jobard, and H. Hauser. Image space based visualization of unsteady flow on surfaces. In *Proc. IEEE Visualization*, pages 131–138, 2003.

[93] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, 2001.

[94] S. Lefebvre, S. Hornus, and F. Nyret. Octree Textures on the GPU. In *GPU Gems 2, Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 595–613, 2005.

[95] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, 2004.

[96] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[97] C. Lepage, T. Leweke, and A. Verga. Spiral shear layers: Roll-up and incipient instability. *Physics of Fluids*, 17:031705, 2005.

[98] A. Leu and M. Chen. Direct rendering algorithms for complex volumetric scenes. In *Proceedings of the 16th Eurographics UK Conference*, pages 1–15, 1998.

[99] A. Leu and M. Chen. Modeling and rendering graphics scenes composed of multiple volumetric datasets. *Computer Graphics Forum*, 18(2):159–171, 1999.

[100] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[101] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[102] M. Levoy. Spreadsheets for images. In *Proc. ACM SIGGRAPH*, pages 139–146, 1994.

[103] T. Lewiner, H. Lopes, A. W. Vieira, and G. Tavares. Efficient implementation of Marching Cubes' cases with topological guarantees. *Journal of Graphics Tools*, 8(2):1–15, 2003.

[104] G.-S. Li, X. Tricoche, and C. Hansen. GPUFLIC: interactive and accurate dense visualization of unsteady flows. In *Proc. Eurovis 2006 (EG / IEEE VGTC Symposium on Visualization)*, pages 29–34, 2006.

[105] Z. P. Liu and R. J. Moorhead. AUFLIC: An accelerated algorithm for unsteady flow line integral convolution. In *Proc. EG / IEEE TCVG Symposium on Visualization*, pages 43–52, 2002.

[106] H. Löffelmann and E. Gröller. Enhancing the visualization of characteristic structures in dynamical systems. In *Proc. EG Workshop on Visualization in Scientific Computing*, pages 35–46, 1998.

[107] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[108] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, 2003.

[109] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[110] N. Max and B. Becker. Flow visualization using moving textures. In *Proc. ICASW/LaRC Symp. Visualizing Time-Varying Data*, pages 77–87, 1995.

[111] N. Max, R. Crawfis, and C. Grant. Visualizing 3D velocity fields near contour surfaces. In *Proc. IEEE Visualization*, pages 248–255, 1994.

[112] D. Meyer. Direkte numerische Simulation nichtlinearer Transitionsmechanismen in der Strömungsgrenzschicht einer ebenen Platte. *PhD Thesis, University of Stuttgart*, 2003.

[113] H. Miura and S. Kida. Identification of tubular vortices in turbulence. *Journal of the Physical Society of Japan*, 66(5):1331–1334, 1997.

[114] H. P. Moreton. Simplified curve and surface interrogation via mathematical packages and graphics libraries and hardware. *Computer-Aided Design*, 27(7):523–543, 1995.

[115] R. Nielsen. Real time rendering of atmospheric scattering effects for flight simulators. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2003.

[116] T. Nishita, T. Sirai, K. Tadamura, and E. Nakamae. Display of the earth taking into account atmospheric scattering. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 175–182, 1993.

[117] S. O'Neal. Real-time atmospheric scattering. www.gamedev.net/reference/articles/article2093.asp, 2004.

[118] S. O'Neal. Accurate atmospheric scattering. *GPU Gems*, 2:253–268, 2005.

[119] S. Park, C. L. Bajaj, and V. Siddavanahalli. Case study: interactive rendering of adaptive mesh refinement data. In *VIS 2002: Proceedings of the conference on Visualization 2002*, pages 521–524, 2002.

[120] T. Patterson. A view from on high: Heinrich Berann's panoramas and landscape visualization techniques for the US National Park Service. *Cartographic Perspectives*, 36:38–65, 2000.

[121] R. Peikert and M. Roth. The parallel vectors operator – a vector field visualization primitive. In *Proc. IEEE Visualization '99*, pages 263–270, 1999.

[122] J. Percy. OpenGL extensions. ATI presentation at ACM SIGGRAPH 2003, `http://www.ati.com/developer`, 2003.

[123] H. Pfister. Hardware-accelerated volume rendering. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*, pages 229–258. Elsevier, 2005.

[124] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[125] H. Piringer, R. Kosara, and H. Hauser. Interactive focus+context visualization with linked 2D/3D scatterplots. In *Proc. Coordinated & Multiple Views in Exploratory Visualization (CMV'04)*, pages 49–60, 2004.

[126] M. Plumlee and C. Ware. An evaluation of methods for linking 3D views. In *Proc. Symposium on Interactive 3D Graphics*, pages 193–201, 2003.

[127] F. Ponta. Effect of shear-layer thickness on the strouhal-reynolds number relationship for bluff-body wakes. *Journal of Fluids and Structures*, 22:1133–1138, 2006.

[128] A. Preetham, P. Shirley, and B. Smits. A practical analytic model for daylight. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100, 1999.

[129] B. Preim, W. Spindler, and H.-O. Peitgen. Interaktive medizinische Volumenvisualisierung – ein Überblick. In *Proceedings of SimVis 2000*, pages 69–88, 2000.

[130] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1995.

[131] A. Provenzale. Transport by coherent barotropic vortices. *Ann. Rev. Fluid Mech.*, 31:55–93, 1999.

[132] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, 2002.

[133] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proc. EG / SIGGRAPH Workshop on Graphics Hardware*, pages 109–118, 2000.

[134] 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*, pages 233–240, 1999.

[135] B. Rogowitz and L. Treinish. How NOT to lie with visualization. *Computers in Physics*, 10(3):268–274, 1996.

[136] C. Rorden. MRIcro. http://www.sph.sc.edu/comd/rorden/mricro.html, 2005.

[137] F. Rößler, R. P. Botchen, and T. Ertl. Dynamic Shader Generation for Flexible Multi-Volume Visualization. In *Proceedings of IEEE Pacific Visualization Symposium 2008 (PacificVis '08)*, pages 17–24, 2008.

[138] F. Rößler, R. P. Botchen, and T. Ertl. Utilizing Dynamic Shader Generation for GPU-based Multi-Volume Raycasting. *Computer Graphics and Applications*, page to appear, 2008.

[139] F. Rößler, E. Tejada, T. Fangmeier, T. Ertl, and M. Knauff. GPU-based multi-volume rendering for the visualization of functional brain images. In *Proceedings of SimVis 2006*, pages 305–318, 2006.

[140] R. J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2006.

[141] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl, and W. Straßer. Smart hardware-accelerated volume rendering. In *Proc. EG / IEEE TCVG Symposium on Visualization*, pages 231–238, 2003.

[142] S. Röttger, W. Heidrich, P. Slusallek, and H.-P. Seidel. Real-time generation of continuous levels of detail for height fields. In *Proc. WSCG '98*, pages 315–322, 1998.

[143] S. Rusinkiewicz and M. Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *ACM SIGGRAPH 2000 Conference*, pages 343–352, 2000.

[144] F. Sadlo and R. Peikert. Efficient Visualization of Lagrangian Coherent Structures by Filtered AMR Ridge Extraction. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1456–1463, September-October 2007.

[145] 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, 2007.

[146] T. Saito and T. Takahashi. Comprehensible rendering of 3-D shapes. *Computer Graphics (ACM SIGGRAPH 90)*, 24(4):197–206, 1990.

[147] F. Scarano, C. Poelma, and J. Westerweel. Towards four-dimensional particle image velocimetry. In *7th International Symposium on Particle Image Velocimetry*, 2007.

[148] T. Schafhitzel, K. Baysal, U. Rist, D. Weiskopf, and T. Ertl. Particle-based vortex core line tracking taking into account vortex dynamics. In *Proceedings International Symposium on Flow Visualization '08*, 2008.

[149] T. Schafhitzel, F. Rößler, D. Weiskopf, and T. Ertl. Simultaneous Visualization of Anatomical and Functional 3D Data by Combining Volume Rendering and Flow Visualization. In *Proceedings of SPIE Medical Imaging 2007: Visualization and Image-Guided Procedures*, pages 650902 1–9, 2007.

[150] 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.

[151] T. Schafhitzel, J. Vollrath, J. Gois, D. Weiskopf, A. Castelo, and T. Ertl. Topology-Preserving λ2-based Vortex Core Line Detection for Flow Visualization. *Computer Graphics Forum (Eurovis 2008)*, 27(3):1023–1030, 2008.

[152] T. Schafhitzel, D. Weiskopf, and T. Ertl. Interactive Exploration of Unsteady 3D Flow with Linked 2D/3D Texture Advection. In *Proceedings of the 3rd International Conference on Coordinated and Multiple Views in Exploratory Visualization (CMV 2005)*, pages 96–105, 2005.

[153] T. Schafhitzel, D. Weiskopf, and T. Ertl. Interactive Investigation and Visualization of 3D Vortex Structures. In *Electronic Proceedings International Symposium on Flow Visualization '06*, 2006.

[154] S. Sekine. Optical characteristics of turbid atmosphere. *J Illum Eng Int Jpn*, 71(6):333, 1992.

[155] SGI. OpenGL extension registry. Web site: `http://oss.sgi.com/projects/ogl-sample/registry`, 2006.

[156] H.-W. Shen, C. R. Johnson, and K.-L. Ma. Visualizing vector fields using line integral convolution and dye advection. In *Proc. Volume Visualization Symposium*, pages 63–70, 1996.

[157] H.-W. Shen and D. L. Kao. A new line integral convolution algorithm for visualizing time-varying flow fields. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):98–108, 1998.

[158] H.-W. Shen, G.-S. Li, and U. D. Bordoloi. Interactive visualization of three-dimensional vector fields with flexible appearance control. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):434–445, 2004.

[159] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Comput. Graph.*, 24(5):63–70, 1990.

[160] Siemens. Medical Solutions. Syngo 3D Offline fMRI. http://www.medical.siemens.com, 2005.

[161] B. A. Singer and D. C. Banks. A predictor-corrector scheme for vortex identification. Technical Report ICASE 94-11, NASA CR-194882, 1994.

[162] K. Singh. MRI3DX. http://www.aston.ac.uk/lhs/staff/singhkd/mri3dX/, 2005.

[163] J. Smagorinsky. General circulatio experiments with the primitive equations. *Mon. Weather Rev.*, 91(3):165–216, 1963.

[164] SPM. Statistical Parametric Mapping. http://www.fil.ion.ucl.ac.uk/spm/, 2005.

[165] S. Stegmaier, U. Rist, and T. Ertl. Opening the can of worms: An exploration tool for vortical flows. In *Proc. IEEE Visualization '05*, pages 463–470, 2005.

[166] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005.

[167] R. Stokking, K. Zuiderveld, and M. Viewgever. Integrated volume visualization of functional image data and anatomical surfaces using normal fusion. *Human Brain Mapping*, 12(4):203–218, 2001.

[168] E. Tejada, J. Gois, L. G. Nonato, A. Castelo, and T. Ertl. Hardware-accelerated Extraction and Rendering of Point Set Surfaces. In *EG/IEEE VGTC Symposium on Visualization (Eurovis 2006)*, pages 21–28, 2006.

[169] A. Telea and J. J. van Wijk. 3D IBFV: Hardware-accelerated 3D flow visualization. In *Proc. IEEE Visualization*, pages 233–240, 2003.

[170] H. Theisel, J. Sahner, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Extraction of parallel vector surfaces in 3D time-dependent fields and application to vortex core line tracking. In *Proc. IEEE Visualization '05*, pages 631–638, 2005.

[171] H. Theisel and H.-P. Seidel. Feature flow fields. In *Proc. EG / IEEE TCVG Symposium on Visualization '03*, pages 141–148, 2003.

[172] G. Turk and D. Banks. Image-guided streamline placement. In *Proc. ACM SIGGRAPH*, pages 453–460, 1996.

[173] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *SCG '97: Proc. Computational Geometry*, pages 212–220, 1997.

[174] J. J. van Wijk. Spot noise – texture synthesis for data visualization. *Computer Graphics (Proc. ACM SIGGRAPH 91)*, 25:309–318, 1991.

[175] J. J. van Wijk. Flow visualization with surface particles. *IEEE Computer Graphics and Applications*, 13(4):18–24, 1993.

[176] J. J. van Wijk. Implicit stream surfaces. In *IEEE Visualization 1993*, pages 245–252, 1993.

[177] J. J. van Wijk. Image based flow visualization. *ACM Transactions on Graphics*, 21(3):745–754, 2002.

[178] J. J. van Wijk. Image based flow visualization for curved surfaces. In *Proc. IEEE Visualization*, pages 123–130, 2003.

[179] V. Verma, D. T. Kao, and A. Pang. A flow-guided streamline seeding strategy. In *Proc. IEEE Visualization*, pages 163–170, 2000.

[180] J. E. Vollrath, T. Schafhitzel, and T. Ertl. Employing Complex GPU Data Structures for the Interactive Visualization of Adaptive Mesh Refinement Data. In *Proceedings of the International Workshop on Volume Graphics '06*, pages 55–58, 2006.

[181] I. Wald and H.-P. Seidel. Interactive ray tracing of point based models. In *Symposium on Point Based Graphics*, pages 9–16, 2005.

[182] M. Q. Wang Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for using multiple views in information visualization. In *Proc. Advanced Visual Interfaces*, pages 110–119, 2000.

[183] C. Ware. Color sequences for univariate maps. *IEEE Computer Graphics and Applications*, 8(5):41–49, 1988.

[184] Z. U. A. Warsi. *Fluid Dynamics - theoretical and compuational approaches*. CRC Press, 1999.

[185] T. Weinkauf, J. Sahner, H. Theisel, and H.-C. Hege. Cores of swirling particle motion in unsteady flows. *IEEE Transactions on Visualization and Computer Graphics (Proc. Visualization 2007)*, 13(6):1759–1766, 2007.

[186] D. Weiskopf. Dye Advection Without the Blur: A Level-Set Approach for Texture-Based Visualization of Unsteady Flow. *Computer Graphics Forum (Eurographics 2004)*, 23(3):479–488, 2004.

[187] D. Weiskopf, M. Borchers, T. Ertl, M. Falk, O. Fechtig, R. Frank, F. Grave, A. King, U. Kraus, T. Müller, H.-P. Nollert, I. Mendez, H. Ruder, T. Schafhitzel, S. Schar, C. Zahn, and M. Zatloukal. Visualization in the einstein year 2005: a case study on explanatory and illustrative visualization of relativity and astrophysics. *Visualization, 2005. VIS 05. IEEE*, pages 583–590, 2005.

[188] D. Weiskopf, M. Borchers, T. Ertl, M. Falk, O. Fechtig, R. Frank, F. Grave, A. King, U. Kraus, T. Müller, H.-P. Nollert, I. Mendez, H. Ruder, T. Schafhitzel, S. Schar, C. Zahn, and M. Zatloukal. Explanatory and illustrative visualization of special and general relativity. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):522–534, 2006.

[189] D. Weiskopf, R. Botchen, and T. Ertl. Interactive Visualization of Divergence in Unsteady Flow by Level-Set Dye Advection. In *Proceedings of SimVis 2005*, pages 221–232, 2005.

[190] D. Weiskopf, K. Engel, and T. Ertl. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.

[191] D. Weiskopf and T. Ertl. GPU-based 3D texture advection for the visualization of unsteady flow fields. In *Proc. WSCG Short Communication Papers*, pages 259–266, 2004.

[192] D. Weiskopf and T. Ertl. A hybrid physical/device-space approach for spatio-temporally coherent interactive texture advection on curved surfaces. In *Proceedings of Graphics Interface 2004*, pages 263–270, 2004.

[193] D. Weiskopf, M. Hopf, and T. Ertl. Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *Proc. VMV*, pages 439–446, 2001.

[194] D. Weiskopf, T. Schafhitzel, and T. Ertl. GPU-Based Nonlinear Ray Tracing. *Computer Graphics Forum (Eurographics 2004)*, 23(3):625–633, 2004.

[195] D. Weiskopf, T. Schafhitzel, and T. Ertl. Real-time advection and volumetric illumination for the visualization of 3D unsteady flow. In *Proc. Eurovis 2005 (EG / IEEE VGTC Symposium on Visualization)*, pages 13–20, 2005.

[196] D. Weiskopf, T. Schafhitzel, and T. Ertl. Texture-Based Visualization of 3D Unsteady Flow by Real-Time Advection and Volumetric Illumination. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):569–582, 2007.

[197] D. Weiskopf, F. Schramm, G. Erlebacher, and T. Ertl. Particle and texture based spatiotemporal visualization of time-dependent vector fields. In *Proc. IEEE Visualization*, pages 639–646, 2005.

[198] D. Weiskopf, M. Weiler, and T. Ertl. Maintaining constant frame rates in 3D texture-based volume rendering. In *Proc. IEEE Computer Graphics International (CGI)*, pages 604–607, 2004.

[199] L. Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, 1990.

[200] A. Wiebel, C. Garth, and G. Scheuermann. Localized Flow Analysis of 2D and 3D Vector Fields. In *Data Visualization 2005: Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization 2005 (EuroVis 2005)*, pages 143–150, 2005.

[201] C. Williamson. Vortex dynamics in the cylinder wake. *Ann. Rev. Fluid Mech.*, 28:477–539, 1996.

[202] J.-Z. Wu, A.-K. Xiong, and Y.-T. Yang. Axial stretching and vortex definition. *Physics of Fluids*, 17:038108, 2005.

[203] J.-Z. Wu, A.-K. Xiong, and Y.-T. Yang. Response to "comment on "axial stretching and vortex definition"". *Physics of Fluids*, 18:029102, 2006.

[204] D. Xue, C. Zhang, and R. Crawfis. Rendering implicit flow volumes. In *Proc. IEEE Visualization*, pages 99–106, 2004.

[205] M. Yamada, Y. Ono, A. Hayakawa, M. Katsurai, and F. W. Perkins. Magnetic reconnection of plasma toroids with cohelicity and counterhelicity. *Physical Review Letters*, 65(6):721–724, 1990.

[206] G. D. Yngve, J. F. O'Brien, and J. K. Hodgins. Animating explosions. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 29–36, 2000.

[207] G. Ziegler, A. Tevs, C. Theobalt, and H.-P. Seidel. On-the-fly point clouds through histogram pyramids. In *Workshop on Vision, Modeling, and Visualization (VMV 2006)*, pages 137–144, 2006.

[208] M. Zöckler, D. Stalling, and H.-C. Hege. Interactive visualization of 3D-vector fields using illuminated stream lines. In *Proc. IEEE Visualization*, pages 107–113, 1996.

[209] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop 3D: An interactive system for point-based surface editing. *ACM Transactions on Graphics*, 21(3):322–329, 2002.