Interactive Visual Analysis of Vector Fields

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

Vorgelegt von

Sven Bachthaler
aus Kirchheim unter Teck

Hauptberichter:  Prof. Dr. D. Weiskopf
Mitberichter:  Prof. Dr. H. Theisel
Tag der mündlichen Prüfung:  05. 06. 2013

Visualisierungsinstitut der Universität Stuttgart

2013
3.1 Algorithm Overview ........................................... 52
3.2 Orthogonal Vector Field Representation ..................... 53
  3.2.1 Spatial Patterns ......................................... 53
  3.2.2 Animation and Motion Perception ....................... 54
  3.2.3 Temporal Coherence ..................................... 56
3.3 2D Algorithm ................................................... 61
3.4 2.5D Algorithm ............................................... 63
3.5 Image Compositing of Combined Approaches ................. 65
3.6 Implementation ............................................... 68
3.7 Results ....................................................... 69
  3.7.1 Performance Results .................................... 69
  3.7.2 Qualitative Results ..................................... 70
  3.7.3 Temporal Filtering ..................................... 71

II  Topology-Based Visualization 77

4 Dynamics of Lagrangian Coherent Structures 79
  4.1 Space-Time LCS Visualization ............................... 82
    4.1.1 Ridge Surface Extraction .............................. 82
    4.1.2 Visualizing Hyperbolicity .............................. 83
    4.1.3 Visualizing LCS Dynamics .............................. 84
    4.1.4 LCS Intersection Bands ............................... 85
  4.2 Implementation ............................................. 86
    4.2.1 Preprocessing ......................................... 86
    4.2.2 Interactive Visualization .............................. 87
  4.3 Results ..................................................... 87
    4.3.1 Oscillating Gyre Saddle ............................... 88
    4.3.2 Quad Gyre ............................................ 89
    4.3.3 Buoyant Plumes ....................................... 90
    4.3.4 OSCAR ................................................. 91

5 Magnetic Flux Topology of 2D Point Dipoles 95
  5.1 Dual Topology of Magnetic Fields ......................... 97
    5.1.1 Physics of Magnetostatics ............................. 97
    5.1.2 Dual Vector Field Topology ........................... 98
  5.2 Flux Topology ................................................ 100
    5.2.1 Connection Regions ................................... 100
    5.2.2 Connectrices .......................................... 102
    5.2.3 Complete Topology ................................... 103
  5.3 Connectrix Algorithm ...................................... 104
    5.3.1 Finding Connection Regions .......................... 104
    5.3.2 Constructing Connectrices ............................ 106
    5.3.3 Visualizing Magnetic Flux ............................ 107
### Contents

5.3.4 Implementation ............................................. 107
5.4 Application .................................................. 108
5.4.1 Monolayer .................................................. 108
5.4.2 Ferrogel .................................................... 108

### III Data-Domain Visualization

6 Continuous Scatterplots ........................................ 113

6.1 Mathematical Model ........................................... 114
6.1.1 Generic Model ............................................ 114
6.1.2 Case $m = n$ ............................................. 116
6.1.3 Case $m < n$ .............................................. 120
6.1.4 Case $m = 2$ and $n = 3$ ................................ 123
6.1.5 Case $m > n$ ............................................. 124
6.2 Scatterplot Algorithm for Tetrahedral Meshes ............. 124
6.3 Parallelized Computation of Continuous Scatterplots . . 127
6.4 Application .................................................. 128
6.4.1 Qualitative Results ...................................... 129
6.4.2 Performance Results ..................................... 132

7 Adaptive Rendering of Continuous Scatterplots ............. 135

7.1 Mathematical Approximation Model ......................... 136
7.2 Algorithm .................................................... 137
7.2.1 Subdivision .............................................. 138
7.2.2 Octree Hierarchy ....................................... 140
7.3 Generalization ............................................... 141
7.4 Implementation .............................................. 141
7.5 Results ...................................................... 142

8 Progressive Splatting of Continuous Scatterplots .......... 147

8.1 Footprint Computation ....................................... 148
8.1.1 Spatial Domain and Data-Set Function ................... 149
8.1.2 Scatterplot Domain ..................................... 150
8.2 Sampling and Progressive Refinement ....................... 150
8.3 Results ...................................................... 151

9 Conclusion ..................................................... 155

Bibliography ...................................................... 161

Curriculum Vitae .................................................. 173
**List of Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFD</td>
<td>computational fluid dynamics</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>compute unified device architecture</td>
</tr>
<tr>
<td>GLSL</td>
<td>OpenGL shading language</td>
</tr>
<tr>
<td>HLSL</td>
<td>high level shading language</td>
</tr>
<tr>
<td>fps</td>
<td>frames per second</td>
</tr>
<tr>
<td>FTLE</td>
<td>finite-time Lyapunov exponent</td>
</tr>
<tr>
<td>GPU</td>
<td>graphics processing unit</td>
</tr>
<tr>
<td>LCS</td>
<td>Lagrangian coherent structures</td>
</tr>
<tr>
<td>LIC</td>
<td>line integral convolution</td>
</tr>
</tbody>
</table>
Abstract

Visualization is a very active research area due to several reasons. For years, data sets have been getting larger and more complex, increasing the difficulty of handling this data. Furthermore, in technical application areas, visualization is an essential part of the engineering process. These developments drive the need for improvements of all aspects of scientific visualization, as well as the integration of information visualization techniques.

This thesis focuses on the development of visualization and analysis techniques for different types of vector fields—vector fields representing the flow of air or water, but also magnetic fields and vector fields derived computationally from scalar fields. The different techniques that were developed to handle such fields are organized in three parts: the first part presents methods that visualize vector fields in dense manner. The second part discusses methods that rely on topological approaches—the complexity of the visualization is reduced by concentrating on features of the data. In the third and final part, continuous scatterplots are introduced, which are designed to analyze correlations in multivariate data sets.

In the first part, the goal is to show as much information as possible and using every available pixel of the viewport to do so. However, one of the challenges of dense visualization methods is to maintain interactivity for high resolution visualizations. A cluster environment is used here to offer increased rendering performance and memory size for large and complex data sets. Additionally, an animation-based approach is presented that allows one to decouple the line-like patterns of LIC from the direction of animation. This decoupling is desirable since perception research suggests that LIC-based techniques combined with animation are non-optimal for local motion detection of the human visual system.

The second part focuses on topological methods to filter the data and hence, reduce the complexity of the resulting visualization. For time-dependent vector fields, Lagrangian coherent structures are used to visualize space-time manifolds that represent the topology of these fields. Furthermore, the dynamic of such fields is visualized directly on these space-time manifolds, allowing us to quantify the hyperbolicity close to the topological skeleton. In addition, another technique is presented in the second part that allows one to visualize the topology of magnetic fields based on dipoles. Here, traditional topological methods are non-optimal, hence, an alternative topology is developed that
visualizes the existence and magnitude of magnetic flux between dipoles. In the final part, the mathematical basis and several computational approaches are presented to compute continuous scatterplots. These plots are designed to work with data sets defined on a continuous domain, which is typical for scientific visualization data. In contrast to traditional scatterplots, they visualize the density in the data domain, instead of merely plotting data attached at discrete sampling positions. The additional computational approaches are an improvement of the original approach in terms of flexibility—they allow a trade-off between output quality and rendering performance, as well as the use of generic interpolation methods.
Zusammenfassung

Es gibt mehrere Gründe dafür, warum die Visualisierung ein sehr aktives Forschungsgebiet ist. Seit Jahren werden Datensätze größer und komplexer, was die Handhabung dieser Daten immer schwieriger macht. Des Weiteren ist die Visualisierung in technischen Anwendungsgebieten ein essentieller Bestandteil des Entwicklungsprozesses. Diese Entwicklungen führen zu der Notwendigkeit, alle Aspekte der wissenschaftlichen Visualisierung zu verbessern sowie zusätzlich Techniken aus der Informationsvisualisierung einzubinden.

Diese Dissertation konzentriert sich auf die Entwicklung von Visualisierungs- und Analysetechniken für verschiedene Arten von Vektorfeldern—Vektorfelder, die Luft- oder Wasserströmung repräsentieren, aber auch magnetische Felder und Vektorfelder, die rechnerisch aus Skalarfeldern abgeleitet wurden. Die verschiedenen Techniken, die für diese Felder entwickelt wurden, sind in drei Teilen organisiert: Der erste Teil präsentiert Methoden, die Vektorfelder auf eine dichte Art visualisieren. Der zweite Teil diskutiert Methoden, die sich auf topologische Verfahren stützen—die Komplexität der Visualisierung wird reduziert, indem nur die wesentlichen Merkmale dargestellt werden. Im dritten und letzten Teil werden kontinuierliche Streudiagramme eingeführt, die entwickelt wurden, um Korrelationen in multivariaten Datensätzen zu analysieren.


Der zweite Teil konzentriert sich auf topologische Methoden, um Daten zu filtern und damit die Komplexität der daraus resultierenden Visualisierung zu reduzieren. Für zeitabhängige Vektorfelder werden Lagrange-kohärente Strukturen verwendet, um Raum-Zeit-Mannigfaltigkeiten zu visualisieren, die die Topologie dieser Felder repräsentieren. Des Weiteren wird die Dynamik
dieser Felder direkt auf den Raum-Zeit-Mannigfaltigkeiten veranschaulicht, wodurch eine Quantisierung der Hyperbolizität in der Nähe des topologischen Skeletts möglich wird. Im zweiten Teil wird auch eine Technik vorgestellt, die es erlaubt, die Topologie von magnetischen Feldern zu visualisieren, die auf Dipolen basieren. Herkömmliche topologische Verfahren sind hier nicht optimal, daher wird eine alternative Topologie entwickelt, die die Existenz und Stärke des magnetischen Flusses zwischen Dipolen visualisiert.

Im letzten Teil werden sowohl die mathematische Basis als auch mehrere Berechnungsansätze für kontinuierliche Streudiagramme vorgestellt. Diese Diagramme wurden für Datensätze entwickelt, die einen kontinuierlichen Definitionsbe reich haben, wie es für wissenschaftliche Visualisierungsdaten oftmals der Fall ist. Im Gegensatz zu herkömmlichen Streudiagrammen visualisieren diese die Dichte in der Datendomäne, anstatt lediglich die Daten einzzeichnen, die an diskreten Abtastpunkten vorliegen. Die zusätzlich vorgestellten Berechnungsverfahren verbessern den ursprünglichen Ansatz in Hinblick auf Flexibilität—sie erlauben den Ausgleich zwischen Ausgabequalität und Renderungsgeschwindigkeit sowie den Einsatz von generischen Interpolationsmethoden.
Visualization is the science of creating visual representations from raw data with the intent of gaining insight into this data.

Using visualization to explain models or theories is not a recent invention—the basic idea is hundreds or even thousands of years old. Leonardo da Vinci was probably the first one to study turbulence in water. He conducted experiments approximately in 1510 A.C. and created hand-made drawings to visualize his results. However, at that time, visualization was limited by the ability and skill of human artists.

Significant changes came with the immense increase of computational power towards the end of the 20th century. During that time, the research area of visualization experienced the most notable forward leap in its development so far. Leveraging the computer’s abilities, visualization scientists were now able to create images or animations based on simulated or measured data. Furthermore, it was now possible to quickly change parameters of a visualization—unwanted data could be filtered, viewing angles could be adjusted, and important parts could be highlighted to increase the effectiveness of the resulting visualization. Being able to interactively manipulate the visualization process is an important feature that was not available prior to that time.

In recent years, visualization has become not only widely present in science, but ubiquitous in every day life as well. Probably the most prominent examples are weather forecasts that are commonly presented with visualizations of clouds or air pressure. In technical application areas, visualization is now an essential part of the engineering process. Inspired by real-world physical experiments, computational visualization is used to examine and refine the design of components or entire vehicles, with examples in both automotive and aerospace industries.

According to Möller and Tory [TM04], visualization is a research area that, historically, was often divided into two major areas called scientific visualization and information visualization. There is no exact and completely accepted definition of how to separate these two areas entirely, since examples exist that belong to both or neither area. Usually, the differentiation is performed based on the application area, whether data is physically based or abstract, and whether
spatialization is inherently given or chosen. A typical example of scientific visualization is vector data of a wind tunnel (air flow), whereas financial data is often seen as an example of a data source for information visualization. Some techniques presented in this thesis are clearly falling into the scientific visualization category. However, especially the techniques presented in the third part cannot be easily categorized into the first or the second category.

In contrast to real-world physical experiments, visualization on a computer is not restricted by the laws of nature—therefore, creating a visualization technique can use physical experiments as an inspiration, but far more advanced techniques can be realized where real-world counterparts are non-existent. A well known visualization metaphor that is based on physical experiments is a streak line: these lines are inspired from patterns of smoke in a wind tunnel. On the other hand, applying more abstract techniques like, e.g., topological methods, is only feasible in a computer-based visualization system.

Advanced visualization techniques are often based on results from other research areas such as color theory or cognitive psychology which are employed by visualization scientists to find visual metaphors. With the use of these metaphors it is possible to explain numerically, linguistically, or logically complicated relations which would otherwise be hard or even impossible to understand.

Nowadays, visualization—and especially scientific visualization—is a very important and active research area due to several reasons. Data acquired by simulations are getting larger and more complex, which leads to the problem that trivial visualization methods are not effective, or their effect is even reversed: instead of helping the user, trivial methods stand in their own way, are overstressing the user, and possibly lead to misinterpretations. Therefore, a lot of effort is put into the development of effective visualization approaches to avoid such problems.

1.1 Research Questions

In 2004, Johnson [Joh04] published a report that identified the top problems and issues in visualization research. Some of the problems mentioned by Johnson are addressed in this thesis—namely the efficient utilization of novel hardware architectures, perceptual issues, multifield visualization, and the integration of both scientific and information visualization strategies. The following paragraphs motivate in more detail how these problems are related to this thesis.

Interactivity is one important aspect of an application for scientific visualizations. Without an interactive application, the exploration of data sets is very tedious or even impossible—however, users often do not know beforehand where to look for interesting spots in the data set. Hence, gaining insight and
analyzing the data is tightly coupled with the ability of an application to produce interactive results.

Depending on data set size or complexity, the creation of such an application can be a challenging task—the data set might be too large for the computer’s memory or the computation of new images consumes too much time. Sometimes, this kind of problem vanishes over time because of Moore’s law—memory size and computing power are increasing exponentially. However, if awaiting this development is not an option, or the underlying algorithms do not scale well, more elegant solutions are required. Firstly, improving the underlying theoretical background and hence algorithms that implement this theory, and secondly, employing specialized hardware to accelerate computations. For the latter, this includes the use of a single graphics processing unit (GPU) and also the use of a cluster system in which several individual computers are bundled to increase both computational performance and available memory.

Putting aside the problem of data set size and complexity, further questions remain. One typical application area for visualization is flow simulation—such data represents the direction and magnitude of flow at the corresponding location. When considering time-independent flow, represented by a vector field that does not change over time, animation might be a helpful method to visualize the flow. However, simple animation methods do not take into account what psychology research suggests with regard to human perception. The question is: how can we combine the non-disputed power of animation with the knowledge about human perception?

Furthermore, vector fields are not restricted to flow—a magnetic field is a vector field as well, however, there is no motion of any matter involved. In contrast to flow visualization where it is often favorable to create a dense visualization, magnetic fields are better visualized using more abstract methods since individual field lines are of less interest here. However, the obvious approach of using vector field topology to support the analysis of magnetic fields proves to be less successful than expected—more suitable topological constructs are necessary to effectively help the user to find interesting structures in a magnetic field.

Finally, vector fields are commonly only one part of simulation results. Using flow simulations as an example, quantities like pressure or temperature are often of interest as well. In addition, derived quantities like the gradient or the magnitude of vectors frequently contain valuable information as well—features in these data domains are often related to interesting spots in the data set. These additional data dimensions are traditionally analyzed using parallel coordinates, scatterplots matrices, or single scatterplots. However, both parallel coordinates and scatterplots have the intrinsic property of displaying only discrete data samples—this can be a major drawback in scientific visualiza-
tion where data samples are spatially related and interpolation or reconstruction methods are applicable to obtain a continuous representation of the data. Therefore, improving the scatterplot idea further to arrive at a tool that effectively visualizes data correlations for scientific data sets remains an interesting problem.

1.2 Interactive Visualization

Visualization is a process that is formally described as a concept of data flowing through a pipeline. This visualization pipeline, as introduced by Haber and McNabb [HM90], consists of the stages data acquisition, filtering, mapping, and rendering. The result of the rendering stage can comprise both images as well as image sequences—either video animations or in the form of an interactive application.

Once the visualization pipeline is established for a data set, it creates output images by applying the aforementioned steps to this data. However, it is almost never the case that parameters of this pipeline are configured in such a perfect way that every aspect of the input data is captured with the first result. If the input data is unfamiliar or newly acquired, the need for iteratively adapting the parameters of the visualization pipeline is even more evident. The concept of exploration is oftentimes the key to identify regions of interest and finally gain insight to a data set. For this reason, the process of visualization requires feedback from the user based on the current output by allowing manipulation of each stage in the pipeline.

In practice, a major requirement for the exploration of a data set is interactivity. A system is called interactive if it is able to provide feedback to user requests in a timely manner. There is no absolute definition of “timely” in the form of a fixed time span; it must merely be short enough to allow an explorative work flow with the system. Therefore, the term interactive visualization refers to systems with highly varying feedback times—ranging from several seconds to produce a single image up to systems with real-time behavior that render several frames per second (fps).

1.3 Vector Fields

In this thesis, the focus lies on vector fields as input data to the visualization pipeline. A field is a defined region in space that stores data at distinct locations within this region. However, for most scientific data sets, it is important to mention that the domain of the field is assumed to be continuous, i.e., data is available at every location within the domain, not only at discrete locations. Depending on the type of data that is stored at these locations, the field is called a scalar, vector, or tensor field.
Mathematically, a vector field is defined as a vector-valued function that assigns each point \( \vec{x} = (x_1, x_2, ..., x_n) \) in the domain \( D \subseteq \mathbb{R}^n \) a vector \( \vec{v} = (v_1, v_2, ..., v_n) \):
\[
\vec{v} : D \rightarrow \mathbb{R}^n.
\] (1.1)

When visualizing vector data, it is important to consider the dimensionality of this data. Spatially, vector data is usually either 2D or 3D. Corresponding visualizations of such data are also called 2D or 3D, respectively. However, there is the special case of visualizations that show 3D vectors on 2D curved surfaces ("hypersurfaces") embedded in 3D space. In other words, the originally three-dimensional vector field is intersected by a hypersurface, and tangential vectors are visualized on the hypersurface—therefore, the resulting visualization is called 2.5D. Examples of 2.5D visualizations are presented in Chapters 2, 3, and 4. Such a tangential vector field \( \vec{v}_T \) can be defined on a smooth and orientable manifold \( M \) (with or without boundary) as follows:
\[
\vec{v}_T : M \rightarrow TM \quad \text{with} \quad \vec{v}_T(x) \in T_xM.
\]

This vector field maps points \( \vec{x} \in M \) to a vector in the corresponding tangent space at that point, \( T_xM \). If the vector field is not tangential, it can be modified to a related tangential vector field that stays on the manifold \( M \) by removing the normal parts from the vectors.

A vector field can also be considered as a representation of the state of a dynamical system, described by differential equations. The evolution of points in this system is given by the solutions of
\[
\vec{x}(t) = \vec{v}(\vec{x}(t), t).
\] (1.2)

Due to the similar nature of this model to fluid dynamics, vector fields are often referred to as flow. However, vector fields in fluid dynamics (in contrast to some vector fields in this thesis) have to fulfill several additional requirements, e.g., the Navier-Stokes equation, to represent physical flow.

The temporal dimension of flow data is also used to classify the corresponding vector fields as either steady if they are independent from time \( t \), and unsteady if they are time-dependent, i.e., the flow data itself changes over time. Synonyms to these terms are time-dependent, or -independent, respectively, which are used throughout this thesis as well.

In scientific visualization, vector fields are usually represented as sampled data sets—vectors are stored at discrete locations only. As a result, data values between these locations are unknown. However, for most data sets, it is a valid assumption that the actual data is smooth, well-behaved, and the underlying domain is continuous—allowing to interpolate or reconstruct data values between sample locations. The result of an interpolation or reconstruction method is an analytic description based on the available discrete data samples.
A trivial interpolation approach is called nearest neighbor, where no additional data values are computed, but the (spatially) nearest data sample is identified. The result is a piecewise constant function, since the data value of the identified sample is returned as the interpolation result.

Nearest neighbor interpolation results are often not precise enough, therefore, linear interpolation is an alternative that linearly weighs the data values of neighboring samples, resulting in a piecewise linear function. Barycentric interpolation is a representative of linear interpolation methods and is used to compute data values inside triangles or tetrahedra.

For results of higher quality compared to linear interpolation, more advanced schemes exist. For example, higher-order polynomials are introduced to allow for curvature between neighboring data points, which is determined based on the information of surrounding data samples.

1.4 Vector Field Visualization

Visualizing vector fields has been a central topic in scientific visualization for years with a variety of application examples from engineering disciplines and sciences [Wei06]. The vast amount of work in this area can be categorized according to Laramee et al. [LHD+04] in four categories: direct, geometric, dense (texture-based), and feature-based flow visualization.

Direct vector field visualization tries to represent the data without translation to more abstract approaches. One example of this category is color coding of flow velocity, which works fairly well in the case of two-dimensional domains. Additional examples of this category are given in Section 1.4.1.

Although direct flow visualization delivers an overall picture of the flow, it is not well suited to communicate its long-term behavior. Geometric flow visualization tries to overcome this issue by integrating along the flow, using geometric objects for visualizing its properties. Examples of this category are stream lines, streak lines, and path lines, which are detailed in Section 1.4.2.

Both aforementioned categories more or less depend on the appropriate placement of seed points that are the basis for this kind of visualization. However, this is oftentimes a challenging task leading to the development of dense visualization techniques that completely avoid the seed point generation problem. A dense visualization commonly uses textures that are filtered with the local values of the flow to generate a visual pattern that indicates the direction of the vector field. Several sections in this thesis are based on dense flow visualization techniques, therefore, a detailed introduction to such methods is given in Section 1.4.3.

Feature-based flow visualization can be seen as the antipole of dense approaches. Instead of capitalizing on each pixel of the image and making use of
as much room as possible to display information, feature-based visualization extracts relevant information. The result is then based on the reduced data to avoid visual complexity and clutter while delivering an efficient flow visualization at the same time. Topology-based methods are an exemplary representative of this category, as they show only the skeleton of vector fields. The key ingredients of topological methods are critical points (i.e., points where the magnitude of the vectors is zero) and separatrices, i.e., lines that separate regions of different flow behavior. Further explanations of topology-based visualization are given in Section 1.5. In some cases, these topological constructs are related to physical properties. An example of such a physically meaningful structure is the connectrix, which is presented in Chapter 5.

1.4.1 Visualization with Glyphs

One of the oldest ways of visualizing a vector field is to draw geometric primitives, or glyphs, at selected sample positions in the vector field domain. These glyphs are designed to show direction, magnitude, or both of the vector field at the current location. The by far most common glyph is the arrow. Visualizations with arrow plots are used in this thesis with the first example shown in Figure 1.2(a). On the one hand, this technique is very easy to implement and requires practically no previous knowledge of the user, on the other hand, as already mentioned, its usefulness is limited to purveying only a rough idea of the underlying vector field. Post et al. [PPWS95] give an overview of visualizations based on glyphs.

Please be aware that scatterplots can be seen as a representative of the glyph-based visualization category as well. Being part of the family of data plots, scatterplots have been proven as successful and useful diagramming techniques in descriptive statistics and information visualization. They take discrete data points with two data dimensions as input, and produce a 2D plot of those data points by drawing respective dots on a diagram with two orthogonal axes representing the two data dimensions.

Although not being able to visualize the spatial properties of a vector field, 2D scatterplots are a useful tool for identifying correlations (or the lack of correlation) between data dimensions. These plots can be either 1D (commonly referred to as “histogram”), 2D or 3D with examples shown in Figure 1.1. Higher dimensional data can be visualized with a scatterplot matrix, i.e., several 2D scatterplots are aligned in a matrix so that their data dimensions can be varied to cover all possible combinations of data dimensions. Alternatively, correlations of several dimensions can also be identified using parallel coordinates [ID90].

For three data dimensions, three-dimensional scatterplots can be applied, however, they are often affected by problems involving occlusion, as well as visual perception problems. One way of overcoming some of these problems is
Figure 1.1: Example of data plots with different dimensionality: (a) 1D, (b) 2D, and (c) 3D.

shown by Sanftmann and Weiskopf with illuminated scatterplots [SW09] and with an interpolation and projection technique for 3D scatterplots that allows the user to exchange data dimensions smoothly [SW12]. Note that scatterplots showing only two data dimensions are not affected by such problems. In this case, discrete 2D data points are taken as input, and a 2D plot of those samples is produced by drawing respective glyphs (usually dots) on a diagram with two orthogonal axes representing the two data dimensions.

A novel visual representation of scatterplots is presented in the work of Chan et al. [CCM10] where local variation in data dimensions are highlighted. This results in so-called “flow-based scatterplots” that help the user to identify interesting changes of one variable with respect to the second mapped variable. Based on ideas of flow field analysis, operations are introduced to scatterplots that efficiently allow selecting points of interest.

The SimVis system [DGH03] presents an approach that shows how the combination of scatterplots and vector field visualization can be used effectively to analyze various vector fields. With this system, it is possible to find correlations between multiple data dimensions and with brushing and linking the spatial location of interesting features found in those data dimensions can be identified and further analyzed.

Traditional scatterplots are designed to work with discrete data only, however, scientific data sets are commonly interpreted as fields where interpolation and reconstruction methods can be applied, as described in Section 1.3. Chapter 6 shows how continuous scatterplots overcome this drawback and how they are applied to scientific data sets.
1.4.2 Visualization with Lines

Falling into the category of geometric flow visualization, so-called characteristic lines are able to convey the properties of a vector field that would otherwise be hard to depict with glyphs. Representatives of this category are path, stream, streak, and time lines. Please note that these lines with the exception of time lines are identical in steady vector fields, but are different in unsteady fields. The following paragraphs elaborate on the details of these field lines.

Equation 1.2, together with the condition $\vec{x}(t_0) = \vec{x}_0$, is the starting point for an initial value problem of an ordinary differential equation. Path lines are a solution to this problem, since they form an integral curve defined by

$$\vec{x}_{\text{path}}(t_1; \vec{x}_0, t_0) = \vec{x}_0 + \int_{t_0}^{t_1} \vec{v}(\vec{x}_{\text{path}}(t; \vec{x}_0, t_0), t) \, dt.$$  

These lines are called path lines, since such curves describe trajectories of massless particles which are released into a vector field. In a real-world experiment, path lines can be obtained by photographic long-time exposure of markers that are advected by flow.

In contrast to path lines, the concept of stream lines differs with respect to the time $t$. Here, a “snapshot” of a specific point in time $t_i$ is considered, thus leading to a computation of the integral curve as follows:

$$\vec{x}_{\text{stream}}(t_1; \vec{x}_0, t_0) = \vec{x}_0 + \int_{t_0}^{t_1} \vec{v}(\vec{x}_{\text{stream}}(t; \vec{x}_0, t_0), t_i) \, dt.$$  

As opposed to path lines, stream lines are unique in the sense that they do not cross each other. This becomes apparent when considering that a particle cannot have two different velocities at the same point in a vector field.

The concept of streak lines is closely related to real-world physical experiments, where particles are continuously released at a fixed position $\vec{x}_0$ at times $t_i \in [t_{\text{min}}, t_1]$, e.g., dye released into water at a certain position, or a chimney releasing smoke into the air. A snapshot of these particles is then taken at time $t_1$, resulting in a pattern which is called a streak line:

$$\vec{x}_{\text{streak}}(t_1; \vec{x}_0, t_1) = \vec{x}_{\text{path}}(t_1; \vec{x}_0, t_i).$$  

These streak lines were generalized by Wiebel et al. [WTS+07] in the sense that the source location $\vec{x}_0$ is not necessarily fixed anymore.

The fourth representative of characteristic lines are constructed by releasing a set of particles placed on a seed curve $\vec{x}_0(s)$ simultaneously at a distinct point in time $t_0$. Again, a snapshot is taken after some time at time $t_1$. The set of particles creates a curve which is displaced by the vector field over time and is called a time line. Time lines are given by

$$\vec{x}_{\text{time}}(s, t_1; \vec{x}_0, t_0) = \vec{x}_0(s) + \int_{t_0}^{t_1} \vec{v}(\vec{x}_{\text{time}}(s, t; \vec{x}_0, t_0), t) \, dt.$$
Generalizing these characteristic lines to three dimensions results in surfaces, which are named according to their 2D counterpart.

1.4.3 Line Integral Convolution

For this thesis, especially dense, texture-based vector field visualization is of interest. Techniques of this sort use characteristic lines to create patterns that visualize the vector field. In order to avoid the problem of finding appropriate seed points for these characteristic lines, they cover the whole domain in a dense fashion.

The most prominent representative of dense, texture-based techniques is line integral convolution (LIC), introduced by Cabral and Leedom [CL93]. Next to the technique called Spot Noise, published by van Wijk [vW91] two years earlier, LIC was one of the first methods that was able to visualize the vector field in a dense fashion and to accurately reflect vector fields with high local curvature. The basic idea of LIC is to visualize the vector field with long line patterns. The basic primitive on which this technique relies on is a noise texture. This texture spans the entire vector field domain and is convolved, or “smeared”, along the path of stream lines. More precisely, the intensity $I$ of a pixel at location $\vec{x}_0$ in the resulting image is computed according to

$$I(\vec{x}_0) = \int_{-L}^{L} k(t) N(\vec{\phi}_0(t)) \, dt,$$

where $[-L, L]$ defines the support of the convolution kernel $k$, which depends on the parameter $t$ denoting the time. Furthermore, the input noise texture is denoted as $N$ and $\vec{\phi}_0(t)$ is the stream line that passes $\vec{x}_0$ at $t = 0$.

The convolution is commonly performed using a symmetric kernel, e.g., a box, tent, or preferably a Gaussian kernel. An asymmetric kernel creates directional cues which are helpful if it is important to see the orientation of LIC lines. The result of this computation is a visual pattern with a high intensity correlation along stream lines, but only little correlation perpendicular to these lines. An example of a vector field visualization using LIC with a Gaussian convolution kernel is shown in Figure 1.2(b).

The original approach of LIC was extended into several directions. One major drawback of the original approach was the slow execution speed on hardware available at that time. Stalling and Hege [SH95] improved the original approach and presented Fast LIC, which was approximately one order of magnitude faster than the original. They achieve this mainly by reusing similar convolution integrals along a stream line. More precisely, their convolution method follows a different approach than the original method of Cabral and Leedom: instead of traveling along a stream line and gathering intensity values from the noise texture, Stalling and Hege employ a depositing scheme.
that stores intensity values at pixel locations. For the convolution process, these values are collected during integration. Forsell [For94] extended the original LIC approach to work not only on uniform grids, but also on parametric surfaces like curvilinear grids. In addition, vector magnitude is visualized using an animation technique. Battke et al. [BSH97] presented a method that is based on Fast LIC and which is not restricted to curvilinear grids but works on arbitrary grids as well. However, the results presented in their paper [BSH97], are limited to relatively small grids composed of equilateral triangles. Weiskopf and Ertl [WE04] proposed an approach that overcomes this issue and works on arbitrary meshes as detailed in Section 1.4.5. Recently, Hlawatsch et al. [HSW11] published hierarchical line integral convolution, which allows one to compute long trajectories based on previously computed, shorter ones. Especially for higher-order integration in higher-order data, performance increases considerably.

Interrante and Grosch [IG97, IG98] extended Fast LIC to visualize 3D flow. To do this, they have to handle perceptual challenges like occlusion, depth perception, and visual clutter. Their approach, called Volume LIC, uses halos to improve depth perception. Occlusion and visual complexity is reduced by using a sparse noise texture. All previously mentioned approaches only operate on steady vector fields. Shen and Kao [SK97] overcome this limitation with Unsteady Flow LIC—an extension that introduces an adapted convolution method which works with path lines instead of stream lines to handle time-dependent vector fields.

1.4.4 Image-Based Flow Visualization Methods

In the dense, texture-based flow visualization category are further approaches, that, in contrast to LIC-based methods, rely on moving texels or
1.4. VECTOR FIELD VISUALIZATION

Texture-mapped polygons.

Image Based Flow Visualization (IBFV) is a technique developed by van Wijk [vW02] which is able to visualize unsteady flow. The method is based on the advection and decay of textures. This approach is image-based as it warps white noise textures according to the direction of the flow. These textures are then blended with the current image in a frame-to-frame fashion. Instead of advecting each pixel of the texture, small quadrilaterals are deformed, leading to higher performance when compared to many other texture-based visualization methods.

A hardware-accelerated technique called Lagrangian-Eulerian Advection (LEA) was published by Jobard et al. [JEH01] which reaches interactive frame rates as well. This approach produces images with high spatio-temporal correlation, as particle paths are integrated in a Lagrangian step, while the color distribution is updated on the grid (i.e., the Eulerian step). This hybrid approach has the advantage that the dense aspect is guaranteed by the Eulerian step, which reduces the amount of particles that are necessary for the Lagrangian step to the amount of pixels in the final image. Similar to IBFV, subsequent time steps are blended to achieve spatial coherence.

Image-Space Advection (ISA) was presented by Laramee et al. [LJH03] and incorporates features both from LEA as well as from IBFV. This approach is able to produce animated textures on arbitrary 3D surfaces based on the IBFV approach, i.e., textures are advected and blended in image-space. ISA and IBFV are technically closely related—their similarities, and also their differences, are detailed in the work of Laramee et al. [LvWJH04].

An inherent problem of the restriction to image-space is that edges of the underlying geometry in object-space are ignored, which can lead to undesired visual continuity across silhouettes or edges. As an example, ISA uses edge detection mechanisms to stop the advection process if an edge is detected at the current processing location.

1.4.5 Hybrid Physical-/Device-Space LIC

Following the basic idea of LIC, the hybrid physical-/device-space approach presented by Weiskopf and Ertl [WE04] creates a vector field visualization on 2D surfaces that are embedded in 3D space. In this thesis, several chapters attend to the problem of visualizing vector fields on curved surfaces as well. Although alternatives to the hybrid approach exist and are shown in Section 1.4.4, this approach retains the benefits of image-space techniques and avoids inflow issues at silhouette lines at the same time. Therefore, the hybrid approach of Weiskopf and Ertl was chosen as a base to build on, justifying an in-depth introduction to this technique in the following. Please note that parts of this section are adopted from our publication [BSWE06].
The hybrid physical-/device-space approach adopts a Lagrangian view to particle tracing to compute LIC-like images. Particle paths are computed as introduced in Section 1.4.2. The positions \( \vec{x} \in \mathbb{R}^3 \) are restricted to locations on the surface embedded in \( \mathbb{R}^3 \). For a tangential vector field, the resulting curves stay on the surface.

So far, the vector and point quantities are given with respect to physical space (P-space). The basic idea of image-space methods is to perform the relevant computation in image-space. In fact, the image-space operations are performed in normalized device-space (D-space), which has the extent \([0, 1]^3\). D-space is ideal to compute LIC on a per-pixel basis with respect to the image plane, which results in a largely output-sensitive algorithm and a uniform density on the image plane. On the other hand, there are some aspects that are better represented in P-space, in particular, the 3D noise input for LIC in order to guarantee frame-to-frame coherence under camera motion. The advantages of P-space and D-space representations are combined by computing particle paths in both spaces simultaneously, as illustrated in Figure 1.3.

Explicit numerical integration, such as a first-order explicit Euler scheme, works with P-space coordinates \( \vec{x}_P \equiv \vec{x} \) and the original tangential vectors \( \vec{v}_P \equiv \vec{v} \) to solve the particle-tracing equation. After each integration step, the corresponding position in D-space is computed. The vector field is no longer given on a P-space but a D-space domain, i.e., there are different representations for the vector components and the associated point on the surface. The modified particle-tracing equation then is
The crucial step in making the integration process efficient is to reduce the 3D representation of the quantities to a 2D D-space representation when possible. Since flow fields are assumed to live on opaque surfaces, only the closest surface layer needs to be considered. Here, the depth component can be indirectly computed because the depth values of the surface on which the visualization is computed are known.

The algorithm consists of two major parts. In the first part, the 2D textures for starting positions $\vec{x}_P$ and the vector field $\vec{v}_P$ are initialized by rendering the mesh representation of the hypersurface. The closest depth layer is extracted by the z-test. The P-space positions are set according to the surface’s object coordinates. The vector field texture is filled by $\vec{v}_P$, which originates from slicing through a 3D texture that holds the vector field data set. Because the vector field is usually not tangential from construction, it has to be made tangential by removing the normal component, which is computed according to the normal vectors of the surface mesh. In the second part, Equation 1.3 is solved by iterating over integration steps. This part works on the 2D $x – y$ subdomain of D-space, and it successively updates the coordinates $\vec{x}_P$ and $\vec{x}_D$ along the particle traces, while simultaneously accumulating contributions to the convolution integral.

The implementation of the complete visualization process can be split into three stages: the projection stage, which projects the surface geometry and the vector field onto the image plane, the LIC stage, which computes the line integral on the image plane, and the blending stage, which combines a LIC image with the rendered image of the surface geometry. All three stages are implemented by vertex and fragment programs to make use of the high processing speed of GPUs.

The projection stage produces three 2D textures as intermediate results: the projected vector field $\vec{v}_P$, the start point for particle tracing $\vec{x}_P$, and the rendered image of the illuminated surface. These three textures are filled in a single rendering pass by using multiple render-targets. The performance of the projection stage is comparable to the performance of rendering the surface with illumination being enabled—only a few instructions are added that project the vector field and store the initial coordinates for particle tracing.

In contrast, the LIC stage is computationally more expensive. This stage uses intermediate results from the projection part to compute the line integral. It solves the particle tracing Equation 1.3 to advance positions along stream lines. Simultaneously, contributions to the line integral are accumulated along the
stream line. The input noise is stored in a 3D texture. Potential aliasing due to perspective foreshortening is avoided by an adapted version of MIP mapping that compensates the effects of perspective projection: first, for each seed point, an independent scaling factor is computed to achieve a constant image-space noise frequency. Second, the noise scaling factor is discretized in order to achieve frame-to-frame coherence under camera motion. The result of this stage is written to a 2D texture that holds the gray-scale LIC image on the image plane. Particle tracing and integral accumulation are implemented in a shader loop that advances along the stream line. Typically, the number of iterations is between 40 and 300. Therefore, the overwhelming performance costs are associated with the LIC stage. Note that the LIC computation is only performed for pixels that are covered by the projected surface geometry, i.e., fragment processing is skipped for background pixels by means of the early $z$-test. The $z$-values for this masking are obtained from the projection stage. Masking leads to a rendering run time that is proportional to the number of visible pixels, i.e., output sensitivity is achieved to a large extent.

Finally, the blending stage combines the result of the LIC stage and the lit surface generated in the projection stage by blending and modulation. In this way, both the LIC texture and the surface geometry are visualized at the same time. This flow visualization technique completely recomputes the LIC image for each frame. Therefore, the rendering performance is not affected by deforming or changing the surface geometry, or by moving the camera. In this way, this approach is suited for interactive applications in which visualization parameters can be rapidly changed by the user.

### 1.5 Topology of Vector Fields

Classical vector field topology partitions a given vector field into areas of qualitatively different flow behavior. In order to segment a vector field in such a way, critical points must be identified. Locations $\vec{x}_c$ in the field must meet two conditions to be classified as a critical point of first order. First, for the vector field must apply $\vec{v}(\vec{x}_c) = 0$, thus, stream lines started at $\vec{x}_c$ degenerate to a single point. This single point is called stationary or, alternatively, constant orbit. The second condition requires that the determinant of the velocity gradient is non-zero at a stationary point. Then, this point represents an isolated zero and is called a critical point.

A critical point can be classified based on the behavior of the vector field around its location. This can be done by performing an eigenvector analysis of the velocity gradient at the corresponding location. Possible cases in two dimensions are sources, sinks, and saddles as well as focuses and centers. An overview is given in Table 1.1, where the classification of critical points in relation to their eigenvalues is shown.
1.5. TOPOLOGY OF VECTOR FIELDS

<table>
<thead>
<tr>
<th>two real eigenvalues</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>both positive sign</td>
<td>source</td>
</tr>
<tr>
<td>both negative sign</td>
<td>sink</td>
</tr>
<tr>
<td>opposite signs</td>
<td>saddle</td>
</tr>
<tr>
<td>two conjugate complex eigenvalues</td>
<td>type</td>
</tr>
<tr>
<td>positive real parts</td>
<td>focus source</td>
</tr>
<tr>
<td>negative real parts</td>
<td>focus sink</td>
</tr>
<tr>
<td>purely imaginary</td>
<td>center</td>
</tr>
</tbody>
</table>

Table 1.1: Classification of first-order critical points in 2D vector fields.

Saddle-type critical points play an important role in vector field topology since they give rise to distinguished stream lines—a stream line converging to a saddle-type critical point in forward-time (or converging to the same point in negative time, which means that the stream line diverges from the saddle). Please note that there is no stream line at the location of the critical point. To compute stream lines, a small step is made away from the critical point in the direction of the eigenvectors; the direction of a stream line is determined by the direction of the eigenvectors as well. These distinguished stream lines are called *separatrices*, since regions of qualitatively different flow behavior are separated by them.

Of further interest to vector field topology are periodic orbits. Periodic orbits are stream lines in a vector field that pass a certain point $\vec{x}_0$ more than once. More precisely, a periodic orbit exists in a vector field if there is a time span $T > 0$ such that $\vec{x}(t) = \vec{x}(t + T)$.

The visualization of all critical points, periodic orbits, and the corresponding separatrices is called the *topological skeleton*. This concept of vector field topology was introduced by Helman and Hesselink [HH89] to the visualization community.

1.5.1 Lagrangian Coherent Structures

Applying vector field topology to create a visualization that shows the topological skeleton of a vector field has proven useful for many applications. An inherent problem of vector field topology is, however, that it is built upon stream lines, and not path lines. Therefore, it is only able to create an instantaneous view of the vector field, which works well for stationary fields. In the case of non-stationary fields, the results of vector field topology are often inappropriate. This was shown, e.g., by Shadden et al. [SLM05] with their “double
gyre" example—separatrices are dislocated and do not show the flow separation properly.

Lagrangian coherent structures (LCS) are an alternative to vector field topology that work for time-dependent vector fields as well. LCS are not finally defined, which naturally leads to deviating definitions in the literature. Relevant for this thesis is the definition of Haller [Hal01], which states that LCS are ridges in the finite-time Lyapunov exponent (FTLE) field—a field that is computed based on path lines. Consequently, LCS are an appropriate alternative to vector field topology in unsteady vector fields, since they correctly reflect advection processes.

Since LCS are based on the FTLE, this concept is defined first. The Lyapunov exponent is employed to measure the growth of perturbations in a time-dependent field or dynamical system. Speaking informally, this value is obtained by considering two particles located closely together in the field—the advection of these two particles is then observed for a defined time span T after which the distance between the particles is measured. Non-positive separation indicates regions where the flow behaves in a predictable way, whereas positive separation is usually correlated to regions with chaotic behavior, as the particles are separated away from each other. The term “finite-time” in FTLE indicates that the practical computation of this value only approximates the limit solution, as it only considers a finite time interval for the advection analysis.

Once the FTLE is obtained for every location in the vector field, LCS can be computed based on this preliminary step. If the FTLE field is interpreted as a height field, then LCS are located at the center of elevational crests or ridges. In practice, there are no absolute values that can be used to identify ridges indiscriminantly, which means that the outcome of an LCS computation heavily depends on the time span T and the advection step size used for the computation of the FTLE field. In addition, the outcome depends on which ridges are regarded as “sharp” enough to assume a LCS at their center [SLM05]. As will be shown in Section 4, obtaining proper LCS is based on empirically finding appropriate parameter values.

1.6 Multi-Attribute Fields

Data generated in today’s simulations is not only getting larger and larger, but also more complex with respect to data dimensionality. As an example, physics-based simulations offer several data dimensions that are correlated to each other. A vector field representing air flow can be accompanied by additional data channels containing e.g., air pressure, temperature, humidity, and so on. For such data sets, scatterplots are a highly useful tool to comprehend relations between these data dimensions. To name an example, the Bernoulli
effect can be easily identified by mapping velocity and pressure to the axis of a scatterplot. According to this effect, regions of low pressure are found if high velocities are present.

To start an analysis, the user has to choose two data dimensions which are then mapped to the axis of a 2D scatterplot. Many data sets contain more information than a vector for each data sample—these additional data dimensions can contain information like temperature, pressure, or humidity to name just a few. However, even such additional dimensions are not available, Kniss et al. [KKH02] discuss an approach that can be applied in such a case. They propose mapping a scalar value to the horizontal axis and deriving a second data dimension based on this first one. Data for the second data dimension is created by computing the magnitude of the gradient of the first data dimension. If there are boundaries present in the first data dimension, they manifest themselves in the scatterplot as arc-like structures. Such boundaries can be highly interesting, e.g., in medical data sets where several layers of organic matter are close to each other.

Based on the approach of Kniss et al., not only transfer functions can be designed to reveal boundaries. Similarly, brushing and linking [BC87, GRW00] can be employed to select the arc-like structures in the scatterplot and highlight according regions in the spatial domain. Brushing and linking is a technique that connects the abstract data visualization of scatterplots with, e.g., the spatial visualization of volume rendering. Data samples that are selected in the data domain are identified in the spatial domain and highlighted there. In this way, the spatial origin of data samples selected in the data domain is visible.

1.7 Graphics Processing Units

GPUs were originally relatively simple-designed hardware used as graphical output buffer and only able to perform basic manipulations with the goal of transforming polygonal geometry into raster graphics. The programmable graphics pipeline [LKM01] introduced a considerable change as GPUs can now be used to execute arbitrary programs—even programs that are not related to graphical problem settings at all. One major difference to CPUs is the massively parallel nature of GPUs. In this thesis, CPUs with up to four cores were used to execute programs, while GPUs in the same system perform computations with 512 cores. This is highly efficient for rendering raster images, as pixels can often be colored independently from each other in a massively parallel fashion.

Generating images starts with an application executed on the CPU and ends with writing pixel data to the frame buffer on the GPU. This process, which includes several stages, is referred to as the rendering pipeline. A simplified version of the rendering pipeline is shown in Figure 1.4. The pipeline starts
with render calls that are executed on the CPU and define a set of vertices, their topology, and additional data like color, normals, or texture coordinates.

The next step in the pipeline is **vertex processing**. As the name indicates, this stage processes vertex data, however, for each vertex on an individual basis. This stage can be manipulated using a **vertex shader**, a short program which is used to transform vertices to so-called clip coordinates. However, this stage can also be used to generate any necessary data not directly related to vertices.

Recently, tesselation and geometry programs have been introduced that are located in the rendering pipeline between vertex and fragment shaders. Tesselation shaders make it possible to control the tesselation level depending on the distance to the virtual camera, allowing fine control in scenarios where dynamic level-of-detail is of interest. Geometry shaders are executed on individual output primitives and have therefore, in contrast to vertex programs, access to all vertices of a primitive. With geometry shaders it is possible to add or remove vertices as necessary, thus allowing mesh refinement or culling of unwanted primitives. In this thesis, tesselation and geometry shaders are not of interest since complex meshes are not used. Instead, textures are generated which are manipulated with fragment shaders that are located further down the pipeline.

**Rasterization** follows vertex processing and the optional tesselation and geometry shaders and filters its input of primitives by discarding those that are not contributing to the final image: primitives that are outside of the view frustum or those that are only visible from the back side. Recent generation graphics hardware allows one to program this stage as well.

One of the later steps in the rendering pipeline is **fragment processing**, where the final color and depth value of individual pixels is computed in a **fragment shader**. Here, access to texture memory also allows one to access pixel data of previous render steps. In this way, images can be created iteratively with a mechanism called **render-to-texture**, where the output is not written to the frame buffer, but into a **render target** instead.
Programming these stages with shader programs can be a challenge, especially when the desired algorithm is not related to graphic computations. Although the available programming languages like HLSL for DirectX applications or GLSL for OpenGL provide a high level of abstraction, certain limitations may require complicated approaches to create a solution which would otherwise be easy to implement, e.g., on a CPU.

To overcome the rigid programming model of the rendering pipeline and to provide hardware acceleration with GPUs at the same time, efforts were made that are now commonly referred to as General-Purpose computation on Graphics Processing Units (GPGPU). Instead of forcing a general problem into a programming paradigm using only graphical primitives, the GPGPU perspective allows one to (almost) freely program GPUs. The Compute Unified Device Architecture (CUDA) [NVI11] is such an environment that allows massively parallel execution of code on the GPU. Since CUDA is dependent on the vendor NVIDIA, alternatives were developed. The Khronos group created the Open Computing Language (OpenCL) [Khr11], which is an independent standard for various kinds of GPUs and multicore CPUs.

**1.8 Outline**

This thesis is organized in three parts that describe techniques to visualize and analyze vector fields. These three parts are organized in a way that subsequent parts handle vector fields with an increasing level of abstraction.

The first part treats methods that directly visualize vector data in a dense manner—the field is visualized by showing as much information as possible and capitalizing on each pixel of the viewport. Chapters 2 and 3 describe these texture-based mapping and rendering techniques.

The second part of this thesis increases the level of abstraction since it presents structural methods: topological features of vector fields and their visualization are the focus of this part. These feature-based techniques are described in Chapters 4 and 5.

In the third part of this thesis, the abstraction is increased again by transitioning to the data domain. Here, at first glance, the spatial aspect of data samples in a field is neglected. Instead, two data dimensions are analyzed with respect to correlations using scatterplots. Despite the decoupling of the visualization from the spatial domain, techniques like brushing and linking can be used to restore this connection again. The third part is comprised of the Chapters 6, 7, and 8. Figure 1.5 summarizes how the methods described in each chapter are organized with respect to the previously mentioned level of abstraction.
1.9 Contributions

The focus of this thesis was the development of improved visualization and analysis techniques for different types of vector fields, with the following contributions. Please note that the papers that originated from my dissertation research were published in collaboration with my PhD advisor. Other collaborators are mentioned explicitly below. Parts of these publications and material thereof are used in this thesis.

Parallel Vector Field Visualization on Curved Surfaces (Chapter 2, [BSWE06])

GPU-based visualization techniques for vector fields on curved surfaces are an alternative to full 3D approaches that offer the completeness and flexibility of the visual representation, but avoid the perceptual problems of true 3D methods at the same time. However, the approach that was used as a basis to build on is strongly influenced by image resolution and suffers from low rendering rates for high resolution visualizations. The main contributions presented in this chapter offer increased rendering performance and increased memory for larger and more complex data sets by employing a cluster environment to create the final result. A hybrid sort-first sort-last rendering scheme is applied to distribute the workload evenly to all cluster nodes and to increase the available texture memory at the same time.

The method described in this chapter is based on my diploma thesis, but was reworked for publication with additional data sets. The resulting paper [BSWE06] was written in collaboration with my diploma thesis advisors Daniel Weiskopf, Magnus Strengert, and examiner Thomas Ertl.

Animation of Orthogonal Texture Patterns for Vector Field Visualization (Chapter 3, [BW08a])

Animation is an established technique to visualize not only direction but also the magnitude of flow. However, traditional techniques that use LIC-like approaches to visualize vector fields were not designed to observe results of per-
ception research with respect to animations. In contrast, the approach presented here decouples the line-like patterns used to visualize the vector field from the direction of animation. In addition, alternative visualization methods were developed that combine traditional LIC-like techniques with the method presented in this thesis.

**Space-Time Visualization of Dynamics in Lagrangian Coherent Structures of Time-Dependent 2D Vector Fields (Chapter 4, [BSDW12])**

Lagrangian coherent structures are a means to visualize the topology of time-dependent vector fields. In this thesis, a method is presented that allows one to visualize space-time manifolds that represent the topology of a vector field. In addition, hyperbolicity and the associated dynamics are visualized directly on the space-time manifolds. In this way, these properties are not only visualized in a qualitative, but also in a quantitative way.

The method described in this chapter was developed in collaboration with Filip Sadlo and Carsten Dachsbacher.

**Magnetic Flux Topology of 2D Point Dipoles (Chapter 5, [BSW+12])**

Traditional topological methods are not optimal to analyze vector fields that are based on magnetic dipoles. An alternative topological construct is presented in this thesis that allows one to visualize the existence and magnitude of magnetic flux between dipoles.

The technique which is presented in this chapter was developed in collaboration with Filip Sadlo. To demonstrate the usefulness of this technique, data sets were used that were created by our application domain collaborators Rudolf Weeber, Sophia Kantorovich, and Christian Holm, who also helped to analyze the resulting images of our visualization software.

**Continuous Scatterplots (Chapters 6, 7, 8, [BW08b], [BW09])**

Scatterplots are a useful tool to analyze multi-attribute fields. Traditional scatterplots, however, are not able to make use of implicit data that is available in scientific data sets through means of interpolation or reconstruction. A mathematical basis for continuous scatterplots is presented in this part of the thesis to overcome this issue. The following chapters improve the original idea of continuous scatterplots to increase their flexibility with respect to interpolation or reconstruction schemes and hardware acceleration as well as algorithmic alternatives are presented that reduce the time required to compute a continuous scatterplot.

Chapter 8, which is based on our publication [HBW11], computes continuous scatterplots in an iterative way. This paper also demonstrates that this idea can be used to create continuous parallel coordinates as well. However, this thesis focuses on the computation of continuous scatterplots. This paper was first-authored by Julian Heinrich, who was supported by me in the part about continuous scatterplots.
Copyrighted material of the following papers is used in this thesis with the kind permission of IEEE:

- Animation of Orthogonal Texture Patterns for Vector Field Visualization [BW08a], © 2011 IEEE.
- Continuous Scatterplots [BW08b], © 2011 IEEE.
Part I

Visualization of Vector Fields
Many physical properties of flow can only be investigated in a true 3D environment, making it necessary to create appropriate visualizations that can handle 3D flow data as input. As shown in Section 1.4, many techniques were developed to visualize 3D flow, however, these methods usually have to forego a dense visualization due to occlusion and visual complexity problems. In order to avoid such issues, flow visualization on curved 2D surfaces embedded in 3D space (hypersurfaces) is an interesting compromise between the completeness and flexibility of the visual representation on the one hand and the reduction of perceptual problems on the other hand. Unfortunately, the performance of surface LIC is strongly influenced by image resolution, which may lead to rendering rates well below one frame per second (fps) for high-resolution visualizations. Another issue is the restriction of the amount of available texture memory, which limits the maximum size of the 3D flow data set that can be visualized.

In this chapter, both the performance and memory issues are addressed by extending and adapting surface LIC to a GPU cluster, i.e., a cluster computer with GPU-equipped compute nodes. The hybrid device/physical approach of Weiskopf and Ertl (see Section 1.4.5) is adopted as a base to build on. Image-space decomposition is used to scale the visualization speed with the number of GPU nodes, while object-space decomposition leads to a scaling of available GPU memory. Parallelizing this kind of GPU algorithm poses particular challenges that are addressed in this chapter: communication of intermediate results between GPUs, main memory, and different compute nodes; a significantly reduced locality of memory accesses caused by particle traces that cover large spatial regions (i.e., particularly less locality than for parallel volume rendering); dynamic load balancing that takes into account a strongly view-dependent behavior of surface LIC; and the combination of image-space and object-space decomposition.

Please note that the techniques presented in this chapter were already included in my diploma thesis [Bac05], however, additional data sets were evaluated for the publication of this technique [BSWE06].
2.1 Acceleration by Parallelization

There is only little previous work on parallel methods specifically related to vector field visualization. Early examples adopt multi-processor workstations, such as SGI’s 4D/340 or Cray’s T3D, to parallelize particle tracing [BMP+90, Lan94, Lan95]. For texture-based flow visualization, there exist parallel versions of LIC [CL95, ZSH97] that run on massively parallel CPU systems. Similarly, the GeoFEM [CFN02] system, which is designed for large shared memory symmetric multiprocessor architectures like the Earth Simulator, contains methods for parallel particle tracking and 3D LIC alike. Muraki et al. [MLM+03] described an approach using GPU-based cluster systems for rendering volumetric data sets with an extension for visualizing 3D LIC volumes constructed in a preprocessing step. There is a large body of research on utilizing cluster systems to improve the performance of typical computer graphics methods like ray tracing, volume rendering, and polygon-based rendering. In general, parallel visualization systems can be classified as sort-first, sort-middle, or sort-last [MCEF94]. These three approaches are differentiated based on how data is distributed among cluster nodes.

The goal of sort-first approaches is to distribute graphical primitives onto different cluster nodes as soon as possible. To do this, cluster nodes are assigned to different regions of the viewport. Since it is not possible to tell which primitive will fall into which regions of the viewport, primitives are distributed in a random fashion to the cluster nodes. Next, a simple transformation of each primitive is performed to compute its final position within the viewport. Commonly, it is sufficient to transform only the convex hull of a primitive to save computations. This transformation allows one to determine if a primitive is situated in the proper subarea of the viewport—if it is located in a different part, all corresponding data has to be sent to the proper cluster node. Once all primitives are distributed to the right cluster nodes, the remaining processing steps are performed. The final result is obtained by simple concatenation of the subareas, since there is no overlapping involved. An example of a system using this sort-first approach is WireGL [HHN+02].

The sort-middle approach differs from sort-first only marginally—primitives are transformed into image-space on the nodes, then they are transferred to the node which is responsible for the subarea of the viewport where the primitive is located. Then, the rasterization step is performed and the final result is again obtained by simple concatenation of the subareas. Implementations of such an approach were published by Fuchs et al. [FPE+89], and Akeley [Ake93].

The third approach is called sort-last, where graphic primitives are distributed to different cluster nodes at the end of the graphics pipeline. In particular, primitives are transformed and rasterized on the same cluster node disregarding the location of a primitive in image-space. For this reason, the compositing step is much more complicated than for the two previous approaches. The
final result is obtained by computing occlusion and alpha-blending and subsequent distribution to different compositing nodes. Since this compositing step is compute-intensive, dedicated hardware was developed to speed up this process. There are several systems that employ this approach for rendering [MHS99, BBFZ00, SEP+01].

In addition, hybrid approaches combining features of different partitioning strategies gained increased attention, e.g., in the form of a hybrid sort-first and sort-last method for parallel polygon rendering [SFLS00] or a hybrid object-space and image-space distribution scheme for volume rendering [GS02].

To ease the utilization of cluster systems, frameworks like CUDASA [SMDE08] or ZIPPY [FQK08] were developed. These frameworks create abstraction layers that handle necessary communication and scheduling mechanisms to employ the massive parallelism of GPU clusters, and, at the same time, reduce programming effort and complexity.

In this thesis, the performance of surface LIC is improved with respect to rendering speed as well as memory scalability by employing a hybrid sort-first sort-last rendering scheme for the cluster environment. The software architecture consists of two major elements: a user application and distributed render clients, which are executed on a PC cluster. The user application acts as the frontend that presents the final image of the distributed rendering and handles user inputs. For rendering, basic graphics functionality is sufficient and no special hardware requirements need to be met, in particular not the ones necessary for visualizing the vector data. The frontend is connected to the PC cluster via TCP/IP allowing the user to visualize data from a remote location. Each node of the cluster system runs an instance of the render client that visualizes parts of the final image depending on the user specified parameters, the number of cluster nodes used, and the partitioning scheme.

The communication scheme for parallel visualization is illustrated in Figure 2.1. The following steps are involved: First, the viewer application sends a render request to a single cluster node. Such a render request contains all information necessary to render a single frame, e.g., camera parameters, lighting conditions, and LIC parameters. Second, the request is then broadcast to all the other remaining cluster nodes using the message passing interface (MPI). This two-level communication is adopted in contrast of a direct broadcast of the user application in order to minimize the amount of data to be sent over a possibly narrow-banded TCP/IP interconnection. Third, every cluster node processes the request and renders an output image according to the given parameters. Image-space partitioning is based on stripes dividing the frame-buffer into separate areas each of which is assigned to a different render node (see Section 2.2). Additionally, the projection of the vector field onto the hypersurface can optionally make use of object-space partitioning to exploit the scalability of texture memory in a GPU-based cluster environment.
2.2. IMAGE-SPACE DECOMPOSITION

To achieve interactive frame rates for large vector fields, a sort-first approach is used. More precisely, the image plane is split into sub-images that are rendered on separate cluster nodes. Ideally, the amount of work per node is reduced to $1/n$, where $n$ is the number of sub-images (i.e., number of cluster nodes).

2.2.1 Partitioning of Image-Space

The image-space is partitioned with a collection of horizontal stripes. This partitioning affects the projection and LIC stages alike. For the projection stage, the surface geometry is not separated into different pieces for parallel rendering because the size of the surface mesh does not pose a rendering bottleneck—typical visualization meshes are visualized interactively even on a single workstation or an individual cluster node. Therefore, each node holds a copy of the

---

Figure 2.1: Overview of the communication architecture. The viewer communicates with the cluster using TCP/IP (green solid lines). Cluster-internal data transfer is driven by MPI (purple dashed lines). In the first part of the algorithm, object-space partitioning is applied to the projection of the vector field. Image-space partitioning is used for the LIC stage.

Finally, the content of the frame-buffer within the assigned stripe is read back on each node and sent to the user application. To reduce the overall amount of data to be transferred over TCP/IP, the data is compressed using a real-time compression method before sending. The user application decompresses the image tiles and composes them into a final image that is shown to the user.
Figure 2.2: Artifacts at stripe boundaries caused by missing vector field data. Red arrows highlight stripe borders.

complete surface geometry. For the sake of simplicity, the whole vector field is also completely replicated on each node. This poses a restriction for data sets that are too large to fit into the memory of an individual cluster node, however, this issue is overcome by object-space partitioning as described in Section 2.4.

In the projection stage, the viewport needs to be adjusted to produce intermediate results only for the image-space stripe that is associated with a respective render node. This is achieved by modifying the view frustum of the camera. The parts of the geometry that are not visible in the stripe are removed by view frustum clipping.

The LIC stage works directly in image-space. A render node only processes those pixels that lie within its respective stripe. In other words, the LIC computation is parallelized with a sort-first approach because the computational domain is partitioned without any overlap. Similarly, the blending stage works directly on a per-pixel basis in the image stripe.

The construction of the final image is reduced to a simple tiling of intermediate image stripes. Here, the viewer application needs an offset in addition to the content of a respective stripe to place the received frame-buffer content of a stripe at the proper position in the final image. To calculate this offset, only the heights of the stripes that are placed below the current stripe are needed. As an example, the lowermost stripe does not need any offset (this is because the frame-buffer “begins” at the lower left corner). The stripe on top of this stripe does need an offset equal to the height of the first stripe, and so on.
2.2.2 Continuous Border Transitions

A problem arises when the above stripe approach is used in combination with particle tracing in the LIC stage: A particle trace that starts within one stripe may leave that stripe and enter a neighboring stripe. In other words, particle tracing breaks the per-pixel locality that has to be assumed for naive image decomposition. Once a particle trace leaves a stripe, it has no longer access to required vector field information. This leads to clearly visible border artifacts between two stripes, as illustrated in Figure 2.2. The flow is interrupted at the border and does not continue seamlessly into the next stripe. This problem can be overcome by increasing the spatial domain of the available vector field data: The projection stage has to produce stripes with an additional area at the upper or lower parts of the stripe. Of course, this overlapping area—or “buffer zone”—is only needed if a stripe has a neighbor at the corresponding border, i.e., the uppermost and lowermost stripes only need one buffer zone at the lower or upper borders, respectively. Figure 2.3 shows how the buffer zone is constructed in the projection stage and subsequently used by the LIC stage. The buffer zone is only needed during particle tracing and can be ignored for later process stages of the visualization process. In particular, the starting points for LIC traces (in the LIC stage), the blending stage, and the read-back of intermediate results from the frame-buffer are based on the original stripe area in order to avoid unnecessary computations. The size of the buffer zone should be chosen cautiously because it can unnecessarily slow down the rendering process if set too big. Of course, if the buffer zone is too small, the previously mentioned error remains visible. The size of the buffer zone is determined by

\[
s_{\text{buffer}} = \frac{n_{\text{convolution}}}{2} v_{\text{max}} \Delta t,
\]

where \( n_{\text{convolution}} \) is the number of convolution steps (the size of the discretized LIC filter kernel), \( v_{\text{max}} \) is the maximum velocity magnitude in the data set, and \( \Delta t \) is the step size used for discretizing the LIC computation. The factor 1/2 reflects the fact that a symmetric filter kernel is used, i.e., particle traces follow one half of the filter kernel in both directions. The value \( s_{\text{buffer}} \) is the maximum
distance along a particle trace in image-space and describes the worst case when a stream line is perpendicular to the stripe border. For consistency, the parameters $v_{\text{max}}$ and $\Delta t$ need to be specified with respect to image-space as well. Note that $v_{\text{max}}$ is readily available for many applications. For example, LIC based on stream lines assumes that the vector magnitude is normalized to unit length. As another example, the representation of vector data in 8-bit texture formats gives a direct bound for the vector magnitude. The buffer zone is designed for the worst case scenario with a conservative estimate for the particle trace length. More sophisticated estimates (e.g., by considering vector field direction) might lead to a reduced size of the buffer zone, however, at the cost of a more time-consuming computation of the estimate.

2.3 Load-Balancing

To achieve optimal overall performance, every node of the cluster should be assigned an equal share of the workload. So far, image-space partitioning relies on static stripes that divide the viewport in equally sized areas. However, the determining factor for performance is not the size of the area in image-space, but the actual number of fragments of the surface geometry that need to be processed. It is obvious that a node with a stripe fully covered by the surface model is far slower than a node assigned to a completely empty region. This problem is overcome by applying a dynamic adjustment of the height of the stripes depending on the associated workload.

Two alternative methods to determine the workload were developed: a timing-based method that actually measures the workload, and a pixel-counting approach that provides an estimate for the computation time based on the number of pixels. For the timing-based method, the time needed to finish rendering is continuously measured on every cluster node. These timings are then gathered for the current frame and used to adjust the stripe heights for rendering the subsequent frame: The stripe heights are modified relative to the speed differences between nodes. The underlying assumption is that the timings are a good estimate for the rendering times of the following frame, which is reasonable when there is temporal coherence for rendering. The timing-based approach can be implemented with almost no overhead or additional processing. It just involves taking the start and end times for rendering, and transferring those times when the intermediate images are sent between nodes for final rendering. The main drawback is that the optimal size for the stripes is typically not achieved completely. This is to some extent due to inaccuracies of time measurements in connection with high frame rates. But more importantly, this issue is related to the partially violated assumption of perfect temporal coherence. If the content of the frame-buffer changes rapidly, temporal coherence between consecutive frames diminishes and the load balancing is less effective.
The second approach uses a pixel-counting algorithm to avoid the aforementioned drawbacks. The idea is to obtain an estimate of the frame-buffer content before rendering. With a good estimate, the workload can be adjusted in a way that every node gets the same number of fragments of the geometry model assigned for LIC processing. Since the workload is mainly dependent on the amount of fragments the vector field gets projected on, this approach allows for an effective load balancing. Computing the estimate has to be fast to minimize overhead; therefore, a down-sampled image of the geometry model is rendered first. To assure the accuracy of the estimation, a tradeoff between the inherent overhead and the used quality needs to be made. For all tests and performance measurements a factor of one-fifth for each dimension was found empirically. The geometry is rendered completely without shading to further speed up processing. The result of this render process is read from the frame-buffer to generate an accumulated histogram. For each row, the number of pixels covered by the geometry model is determined and summed up consecutively. Figure 2.4 shows the relationship between frame-buffer content and the accumulated histogram. The total number of pixels is divided by the number of cluster nodes to obtain the ideal number of pixels per node. Using the inverse of the histogram, equally sized intervals (where the interval size corresponds to the number of pixels per node) are mapped to actual stripe heights in the frame-buffer. The pixel-counting approach is computed with the current camera parameters and, thus, is not affected by changes of frame-buffer contents. The main disadvantage of this approach is the slightly higher rendering overhead, especially for large surface meshes.

Figure 2.4: Computing the estimate for the pixel-counting approach. Pixels of the downsampled rendering (left) are stored in an accumulated histogram (center). The stripe sizes (along the horizontal axis of the histogram) are determined by inverting the accumulated histogram for equally sized pixel intervals (along the vertical axis). The stripes are shown to the right.
2.4 Object-Space Partitioning

Not only can a cluster be used to scale rendering speed, but also to scale available memory because each node provides some fixed amount of memory. The goal to achieve the best performance by using multiple processors is often on a par with the goal to visualize very large vector fields. To display huge vector fields, the considerably larger combined texture memory of the PC cluster is used. The vector field has to be partitioned because the texture memory of a PC cluster is not available in one piece—it is distributed memory. Here, the idea is to adopt a bricking approach known from parallel volume rendering: The vector field is divided into bricks, and each of these bricks corresponds to a sub-volume of the vector field. A single cluster node works on its assigned brick. The brick size (or the number of nodes) should be chosen so that the vector field data of a brick fits in the texture memory of a single node’s GPU. Only the projection stage is affected by bricking because the 3D vector field is only used in that part of the surface LIC algorithm.

The following modifications need to be incorporated. First, it has to be ensured that a cluster node generates a projected vector field only within its own brick. Six brick-aligned clip planes are used to cut the surface geometry away for regions outside the brick. Second, a compositing step needs to be included to reconstruct the full image-space based vector field. Compositing is distributed among nodes, with the same stripe-based organization that is used for the LIC stage. Therefore, every node has to send its content of the corresponding stripe to the cluster node that is responsible for this stripe. Here, it is important that the nodes send stripes including the buffer zones. In this sort-last approach, the intermediate vector field images are composited in a back-to-front order. Since the surface geometry is opaque, alpha blending is not needed. In fact, similarly to the painter’s algorithm, incoming non-background pixels just overwrite existing pixels. The result of this compositing is a complete projection of the vector field for that stripe. The LIC stage is not affected by these modifications because it uses only the result of the projection part. Therefore, the object-space partitioning of the data set can be directly combined with the image-space decomposition for the LIC computations.

2.5 Results

All measurements were conducted on a GPU-based cluster with eight render nodes. Each node runs two AMD Opteron processors at 2.18 GHz and is provided with 4 GB of system memory. For rendering, all nodes have an NVIDIA GeForce 6800 Ultra with 256 MB of texture memory installed. The cluster’s internal communication is driven by MPI over an Infiniband interconnection that provides low latency times and data transfer rates of up to 800 MB per second. In this test environment, the PC running the viewer application is connected
Table 2.1: Performance with varying amount of nodes, rendered on a 800×800 viewport. Numbers in brackets denote speedup.

<table>
<thead>
<tr>
<th>number of cluster nodes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS static stripes</td>
<td>2.23</td>
<td>3.86 (1.7)</td>
<td>2.82 (1.3)</td>
<td>4.01 (1.8)</td>
</tr>
<tr>
<td>FPS dynamic time-based</td>
<td>-</td>
<td>4.00 (1.8)</td>
<td>5.62 (2.5)</td>
<td>6.67 (3.0)</td>
</tr>
<tr>
<td>FPS dynamic pixel-based</td>
<td>-</td>
<td>3.83 (1.7)</td>
<td>5.21 (2.3)</td>
<td>6.29 (2.8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number of cluster nodes</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS static stripes</td>
<td>4.12 (1.8)</td>
<td>5.07 (2.3)</td>
<td>5.06 (2.3)</td>
<td>6.29 (2.8)</td>
</tr>
<tr>
<td>FPS dynamic time-based</td>
<td>7.91 (3.5)</td>
<td>9.10 (4.1)</td>
<td>10.12 (4.5)</td>
<td>10.89 (4.9)</td>
</tr>
<tr>
<td>FPS dynamic pixel-based</td>
<td>7.25 (3.3)</td>
<td>8.14 (3.7)</td>
<td>8.95 (4.0)</td>
<td>9.67 (4.3)</td>
</tr>
</tbody>
</table>

To the cluster using a Gigabit Ethernet network. To demonstrate the scaling behavior of this system, results are shown that were obtained with an increasing number of nodes used for rendering on an 800×800 sized viewport using a 128^3 sized vector field. During this test series, no object-space partitioning was carried out in order to provide comparability and to avoid effects caused by a compositing stage or a corresponding communication scheme. The surface is generated by rendering the GLUT teapot that rotates around the main axis while the measurement was taken. With this constant change of the rendered image, a reproducible user interaction is simulated which allows for measuring the effects of the dynamic stripe adaption under realistic conditions. Table 2.1 documents the results for three test series, either using a static distribution with equally sized areas in image-space or using one of the two dynamic load balancing techniques from Section 2.3. The speedup obtained in the static case using all eight cluster nodes is only a factor of 2.82, compared to the single node setup. This small speedup is mainly caused by a highly imbalanced distribution of the workload throughout the cluster. With increasing number of nodes, the top and bottom stripes receive a rapidly decreasing number of fragments of the surface model, while the number of fragments for the center stripes decreases only slowly.

Adding dynamic load balancing significantly improves the performance of the complete system. For both dynamic load balancing techniques, an 8-node configuration achieves a speedup of over 4.3, with an average frame rate of approximately 10 fps. In all measurements, the timing-based method performed better than the pixel-counting approach, which is due to the higher overhead for computing the pixel-counting estimate. However, for other tests with less temporal coherence between frames, the performance gap between the timing-based and pixel-counting approaches closes. If more parameters change in-between frames, such as the geometry of the surface, a performance advantage
Figure 2.5: Resulting image of distributed rendering with eight GPU nodes. The geometric object was modeled by Sam Drake and tesselated by Amy Gooch and Peter-Pike Sloan [DGS06].

is expected with the pixel-counting approach.

Adding the object-space partitioning scheme to the projection stage allows one to increase the size of the visualized vector field at the cost of additional communication and compositing. For performance measurements, a $512^3$ vector field was used with an overall data amount of 1.28 GB. Distributing the data using all eight cluster nodes leads to nearly 200 MB of texture data assigned to each cluster node, which is close to the texture limit of the GPUs used. Using the same parameters and geometry as stated above, a rendering rate of 3.08 fps is achieved for static image-space load balancing and 3.79 fps for both dynamic load balancing methods (with eight nodes)—as a comparison, the $128^3$ sized data set renders at 3.11 fps and 3.90 fps, respectively. As described, using object-space partitioning requires an additional compositing step for the projected vector field, which is also the main reason for the measured drop in performance. Taking this overhead into account an almost optimal scaling behavior is achieved for the GPU memory. Figure 2.5 shows the result of flow visualization on a more complex surface mesh with 25000 polygons and a $512^3$ sized vector field. When rendering on a $800 \times 800$ viewport with eight nodes, 2.22 fps were obtained.
The method from the previous chapter describes how vector fields can be visualized that are technically challenging in terms of size or complexity. However, another aspect requires attention as well: how well can a human observer perceive animations in visualization? In general, animation has been used successfully for vector field visualization because it can show the direction, orientation, and especially the magnitude of a vector field. In addition, animation can mitigate curve intersection issues that occur for long path lines or streak lines of a time-dependent flow. However, there is a lack of related work that would have considered the perception of such animations for flow visualization. A recent overview of the role of perception in visualization and computer graphics is presented by Healey and Enns [HE12], however, the perception of animated patterns is not discussed in their work.

Based on results from vision research, it can be stated that previous approaches like animated stream lines are non-optimal for local motion perception. Texture-based methods significantly reduce spatial frequency along integral curves to display those curves. However, there is substantial evidence for a spatial frequency tuning of the motion detectors in our human visual system (HVS), and optimal spatial frequencies are typically much higher than the spatial frequencies produced by texture-based methods (see Section 3.2.2).

Therefore, the approach presented in our publication [BW08a] is to decouple the direction of the line-like patterns from the direction of animation. More specifically, an orthogonal vector field should be used (i.e., the original vector field rotated by \( \pi / 2 \)) to construct line-like patterns and the original vector field is used to drive the animation. In this way, the spatial frequencies along the direction of motion are determined by the spatial frequencies of the input noise (for LIC or texture advection), which are independent of the length scales along the line patterns (controlled by the filter length of LIC or texture advection). This visualization approach resembles a moving wave front of the vector field and therefore provides an intuitive analogy to the real world.

This method of the orthogonal vector field and its corresponding visualization is denoted as ortho-vis. In contrast, the traditional texture-based visualization
of stream lines or path lines is called *tangent-vis* to indicate that it shows the tangential direction of the vector field. Ortho-vis and tangent-vis can be combined to a single visualization image that retains the perceptual benefits of ortho-vis, while providing the traditional, familiar visualization via tangent-vis.

### 3.1 Algorithm Overview

This method targets vector field visualization by combining texture-based representations of the tangential and orthogonal vector field directions. The algorithm computes intermediate results of ortho-vis and tangent-vis elements in an essentially independent way, combining those results in a final image-compositing step. Figure 3.1 provides a high-level view on the data flow and computational components of the complete algorithm. Temporally coherent animation starts with generating coherently moving noise images in the first part of the algorithm. Then, the process is split in the independent computation of ortho-vis and tangent-vis. Finally, image compositing results in a combined visualization of ortho-vis and tangent-vis. On this abstract level, the above algorithm works for 2D data sets (i.e., on planar 2D manifolds) and 2.5D data sets (i.e., on curved surfaces embedded in 3D space) alike. Since an image-space approach is used for 2.5D visualization, the operations of this approach are sufficient to handle 2.5D vector fields—these operations are designed to work in image-space as well. In Figure 3.1, these operations are located in the components labeled “orthogonal vis”, “tangential vis”, and “compositing”. Although the first component, “noise transport” is designed to work in object-space, it can be implemented for both 2D and 2.5D data sets.

![Figure 3.1: High-level illustration of the structure of the complete visualization process.](image-url)
3.2 Orthogonal Vector Field Representation

The orthogonal vector field representation (ortho-vis) is introduced first in a formal, mathematical way and then the new visualization method is motivated by showing similarities to existing visualization approaches and by providing a perceptual rationale. The subsequent parts of this section discuss the animation and temporal coherence of moving visualization patterns.

3.2.1 Spatial Patterns

As introduced in Section 1.3, a tangential vector field $\vec{v}$ can be defined on a smooth and orientable 2D manifold $M$. Since $M$ is orientable, an operator $\Omega$ can be defined that rotates a vector within the tangent plane by $\pi/2$. The inverse operator $\Omega^{-1}$ yields a rotation by $-\pi/2$. The orthogonal vector field $\vec{u}$ is defined as

$$\vec{u} : M \rightarrow TM, \quad \vec{x} \mapsto \Omega \vec{v}(\vec{x}).$$

The idea is to display the orthogonal vector field $\vec{u}$ instead of the original vector field $\vec{v}$. The actual visualization relies on integral curves (stream lines in the context of flow visualization, field lines in the context of electric, magnetic, or related fields) to show the direction of $\vec{u}$. In this thesis, the focus lies on the texture-based visualization of stream lines by means of LIC (see Sections 3.3 and 3.4), but other methods such as geometrically constructed stream lines might also be employed. Figure 3.2 illustrates an example of $\vec{u}$ and $\vec{v}$ by means of a few geometric lines that represent integral curves. So far, a stationary vector field has been assumed. For a time-dependent vector field, the above approach is applied to an instantaneous vector field for a given time in order to produce a single visualization for that time.

Before animation can be considered—the main aspect of this visualization approach—the use of the rotated vector field for a single frame of the visu-

![Figure 3.2: Illustration of two perpendicular families of lines: for a circular vector field $\vec{v}$ (dashed lines) and a radial vector field $\vec{u}$ (solid lines).](image-url)
alization will be motivated. First, the mapping by the rotation operator $\Omega$ is one-to-one, i.e., the original vector field $\vec{v}$ can be recovered by applying the uniquely defined inverse operator $\Omega^{-1}$. While this argument shows that the same information content is displayed by $\vec{u}$ and $\vec{v}$, the question remains how effective the rotated vector field is for visualization purposes. Here, the special choice of the $\pi/2$ rotation angle becomes important because analogous uses of perpendicular line structures are well known and accepted in visualization. One analogy comes from the representation of 2D scalar fields by either contour lines (isolines) or gradient directions: gradients and contours are perpendicular by construction. For example, Figure 3.2, which was used to illustrate an orthogonal vector field, can also be interpreted as a visualization of a scalar field with maximum value in its center, concentric circles as contour lines (dashed), and radial gradient lines (solid). A related analogy is based on the Helmholtz decomposition of vector fields [Her58]. Adopting the notation of Polthier and Preuß [PP00], a vector field $\vec{v}$ on a 2D manifold can be written as

$$\vec{v} = \nabla \phi + \Omega \nabla \omega + \vec{\eta},$$

with the curl-free part $\nabla \phi$, the divergence-free part $\Omega \nabla \omega$, and the remaining harmonic part $\vec{\eta}$. The scalar functions $\phi$ and $\omega$ serve as potentials for the gradient field $\nabla \phi$ and the co-gradient field $\Omega \nabla \omega$ (defined as the gradient rotated by $\pi/2$).

Publications on the Hodge-Helmholtz decomposition have focused on variational, discretized computations for triangulated grids [PP00, PP03, TLHD03]. Assuming a divergence-free vector field represented by $\omega$, the orthogonal vector field approach shows the field lines of the gradient $\nabla \omega$. Similarly, a curl-free vector field based on $\phi$ would show the gradient $\nabla \phi$ by traditional visualization methods. Therefore, the orthogonal vector field visualization could be regarded as the “dual” of the traditional flow visualization. Another analogy can be found in the propagation of wave fronts. The Eikonal equation [Gol80], which, for example, applies to the propagation of action in the Hamilton-Jacobi model of classical mechanics or to light propagation, describes this propagation in terms of fronts that are orthogonal to the propagation direction, i.e., those fronts are analogues of ortho-vis, whereas the propagation vectors are analogues of tangent-vis. A related physical analogy is the propagation of water waves, where the wave fronts are perpendicular to wave propagation.

### 3.2.2 Animation and Motion Perception

So far, only static visualization by a single image was discussed. Although the spatial patterns in one frame are important, animation plays an even more crucial role in the proposed approach. The basic idea is to drive the animation
by the original vector field $\vec{v}$, i.e., the direction of motion (given by $\vec{v}$) and the integral curves in an image (determined by $\vec{u}$) are perpendicular. Figure 3.2 illustrates this approach: the solid, radial lines show the curves of $\vec{u}$, which are transported along the circular flow $\vec{v}$, leading to a counter-clockwise rotation.

Why is the decoupling of temporal evolution and spatial patterns useful? The main motivation comes from research on human visual perception. There is indication for a spatial frequency tuning of the HVS: how well we perceive motion depends on the spatial frequency of moving patterns. Low-level motion perception is based on small receptive fields that serve as local motion detectors (see, e.g., [AB87]). Vision research and physiological investigations have addressed various aspects of motion perception, including the detection and discrimination of moving patterns, the influence of contrast and color, and the breakdown of the perception of coherent motion under certain conditions. Although this topic is still an area of active research, a general observation of a frequency tuning of motion detection can be found in the literature. One interesting aspect of recent studies is that the characteristics of receptive fields may be adaptive—dependent on the stimulus. For example, Cavanaugh et al. [CBM02] describe that at low contrast, a wider spatial region (with less surround suppression) is used as input to increase sensitivity, whereas a high-contrast stimulus leads to higher spatial resolution using increased surround suppression. This principal observation can also be found in the context of motion perception [TL05], where high contrast favors the detection of high-frequency stimuli and low contrast favors large stimuli. Tadin and Lappin [TL05] report an optimal size of 0.5 deg (degrees with respect to the subtended angle as seen by the viewer) for a high contrast of 92%. Typically, texture-based vector field visualization uses patterns of high luminance contrast and, therefore, high spatial frequency patterns are appropriate. Another observation is that local and global motion detectors can be distinguished (see, e.g., [BD02]).

In this section, the focus lies on local motion detection, which is optimal for certain spatial frequencies. Bex and Dakin [BD02] report a maximum sensitivity for local motion detection for spatial frequencies around 2 cycles/deg. A similar number of 3 cycles/deg is given by Watson and Turano [WT95] as optimal motion stimulus. The actual value for the optimal spatial frequency of patterns depends on several outside parameters, but many studies agree upon frequencies somewhere around or above 2 cycles/deg. In contrast to these recommendations, typical visualization applications show extended stream lines of 100–200 pixels length. In other words, depending on viewing distance and screen resolution, commonly used spatial patterns deviate from the optimum by a factor of 5–10.
3.2. ORTHOGONAL VECTOR FIELD REPRESENTATION

3.2.3 Temporal Coherence

The intent of the ortho-vis method is to transport integral curves of the rotated vector field $\vec{u}$ along the original vector field $\vec{v}$ in order to control the spatial frequency of the transported patterns along the transport direction. One has to keep in mind that the vector fields may be time-dependent. This transport could be realized by first constructing integral curves of $\vec{u}$ for an initial time $t_0$ and then advecting those curves along $\vec{v}$ to a later time $t_1$. An alternative way is to first advect the seed points (i.e., initial noise for LIC) along $\vec{v}$ from time $t_0$ to $t_1$ and then construct the integral curves of $\vec{u}$ for time $t_1$. Figure 3.3 illustrates both approaches for the example of a shear flow. Unfortunately, these two transport approaches do not necessarily lead to the same result, as demonstrated in Figure 3.3. The first approach guarantees temporal coherence of the transported integral curves because the curves themselves are advected. The second approach makes sure that the integral curves are always perpendicular to $\vec{v}$. Since the two approaches may lead to different results, it is impossible to have a mechanism that maintains orthogonal vector field lines and achieves temporal coherence at the same time.

In the following, mathematical expressions are derived that quantitatively describe the inconsistency between the advection of orthogonal LIC lines and the from-scratch construction of those lines. Those expressions are used to provide the motivation of temporal filtering that is applied in the ortho-vis approach.
CHAPTER 3. ANIMATION OF ORTHOGONAL TEXTURE PATTERNS 57

Figure 3.4: Points and transport vectors used for the derivation of inconsistent transport of orthogonal lines.

To overcome the issues of temporal incoherence.

To begin, the distance is derived between a particle transported by the original steady vector field $\vec{v}$ and a particle transported by the orthogonal vector field $\vec{u} = \Omega \vec{v}$. Later, this result is applied to the specific case of advected orthogonal LIC lines. The derivation targets a description by means of differentials; therefore, the derivation utilizes first-order Taylor expansions to derive mathematical expressions, which is completely consistent with final results based on differential quantities.

The following notation is used for the mathematical expressions: uppercase letters label points in 2D space, and those labels are used as superscripts attached to respective variables. One variable is the position of a point, denoted by $\vec{p}$; i.e., $\vec{p}^A$ is the position of point $A$. 2D vector field values are denoted $\vec{v}$; i.e., $\vec{v}^A$ is the vector field at point $A$. Components of points and vectors are indicated by subscripts $x$ and $y$, respectively. To denote the approximation that is introduced by using Taylor expansion, the approximation sign “$\approx$” is used.

As illustrated in Figure 3.4, the following points related by first-order Euler integration for particle transport are considered:

$$
\vec{p}^A
$$

$$
\vec{p}^{B_1} \approx \vec{p}^A + \vec{v}^A \Delta t
\tag{3.1}
$$

$$
\vec{p}^{B_2} \approx \vec{p}^A + \Omega \vec{v}^A \Delta s
\tag{3.2}
$$

$$
\vec{p}^{C_1} \approx \vec{p}^{B_1} + \Omega \vec{v}^{B_1} \Delta s
\tag{3.3}
$$

$$
\vec{p}^{C_2} \approx \vec{p}^{B_2} + \vec{v}^{B_2} \Delta t.
\tag{3.4}
$$

Point $A$ is the starting point for the following discussion. The step sizes $\Delta t$ and
\( \Delta s \) correspond to transport along the vector field \( \vec{v} \) and the rotated vector field \( \Omega \vec{v} \), respectively. The rotation operator is defined in component-wise notation as

\[
\Omega \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} -v_y \\ v_x \end{pmatrix}.
\]

First-order Taylor expansion of the vector field around the point \( A \) leads to

\[
\vec{v}^{B_1} \approx \vec{v}^A + (J\vec{v}^A) (\vec{p}^{B_1} - \vec{p}^A) \approx \vec{v}^A + (J\vec{v}^A) (\vec{v}^A \Delta t)
\]

and

\[
\vec{v}^{B_2} \approx \vec{v}^A + (J\vec{v}^A) (\vec{p}^{B_2} - \vec{p}^A) \approx \vec{v}^A + (J\vec{v}^A) (\Omega \vec{v}^A \Delta s),
\]

with the Jacobi matrix

\[
J\vec{v}^A = \begin{pmatrix} \frac{\partial v_x^A}{\partial x} & \frac{\partial v_x^A}{\partial y} \\ \frac{\partial v_y^A}{\partial x} & \frac{\partial v_y^A}{\partial y} \end{pmatrix}.
\]

Without loss of generality, a Frenet frame [Gal01] is used along the stream line of \( \vec{v} \) as coordinate system. One axis of the Frenet frame points along the tangent direction \( \vec{v} \), the other axis points along the perpendicular direction \( \Omega \vec{v} \). Any orthogonal coordinate system can be transformed to the Frenet frame by rotation. After this rotation, the \( x \) axis is aligned with \( \vec{v} \) and the \( y \) axis is aligned with \( \Omega \vec{v} \). In this coordinate system,

\[
\vec{v}^A = \begin{pmatrix} v_x^A \\ 0 \end{pmatrix}.
\]

Using this coordinate system, Equations 3.5 and 3.6 lead to the component-wise expressions

\[
\Omega \vec{v}^{B_1} \approx \begin{pmatrix} 0 \\ v_x^A \end{pmatrix} + \begin{pmatrix} -\frac{\partial v_y^A}{\partial x} \\ \frac{\partial v_x^A}{\partial y} \end{pmatrix} v_x^A \Delta t
\]

and

\[
\vec{v}^{B_2} \approx \begin{pmatrix} v_x^A \\ 0 \end{pmatrix} + \begin{pmatrix} \frac{\partial v_x^A}{\partial y} \\ \frac{\partial v_y^A}{\partial y} \end{pmatrix} v_x^A \Delta s.
\]
Using the Taylor expansions for the vector field from Equations 3.7 and 3.8, the position expressions from Equations 3.1–3.4 can be combined to yield the difference vector

\[
\vec{p}^{C2} - \vec{p}^{C1} \approx \vec{p}^A + \begin{pmatrix} 0 \\ v_x^A \end{pmatrix} \Delta s + \begin{pmatrix} \frac{\partial v_x^A}{\partial x} \\ \frac{\partial v_x^A}{\partial y} \end{pmatrix} v_x^A \Delta s \Delta t \\
- \left\{ \vec{p}^A + \begin{pmatrix} v_x^A \\ 0 \end{pmatrix} \Delta t + \begin{pmatrix} 0 \\ v_x^A \end{pmatrix} + \begin{pmatrix} -\frac{\partial v_y^A}{\partial x} \\ \frac{\partial v_y^A}{\partial x} \end{pmatrix} v_x^A \Delta t \right\} \Delta s
\]

\[
= \begin{pmatrix} \frac{\partial v_x^A}{\partial y} + \frac{\partial v_y^A}{\partial x} \\ \frac{\partial v_y^A}{\partial y} - \frac{\partial v_x^A}{\partial x} \end{pmatrix} v_x^A \Delta s \Delta t .
\]  

(3.9)

If the goal would be to construct orthogonal vector field lines that exhibit velocity-dependent markers along those lines, then Equation 3.9 would appropriately describe the inconsistency between advection of orthogonal lines from the previous time step and from-scratch construction of orthogonal lines for the current time step, i.e., Equation 3.9 describes the difference in transport of point particles along \( \vec{v}_x \) and \( \nabla \vec{v}_x \), respectively. The scaling factors \( \Delta s \) and \( \Delta t \) represent the step sizes for the discretization of transport, and \( v_x^A \) can be considered as an overall scaling factor for flow velocity. Therefore,

\[
\vec{\delta}_{\text{full}} = \begin{pmatrix} \frac{\partial v_x^A}{\partial y} + \frac{\partial v_y^A}{\partial x} \\ \frac{\partial v_y^A}{\partial y} - \frac{\partial v_x^A}{\partial x} \end{pmatrix}
\]

is the interesting vector-valued measure for the inconsistencies of particle transport. In fact, the orthogonal vector field is visualized by LIC lines based on the normalized vector field. Put differently, it is not of interest how far \( \vec{\delta}_{\text{full}} \) differs in the \( y \) direction because a difference in distance along the \( y \) direction is compensated by the normalization during the LIC computation. Therefore, only the \( x \) component of \( \vec{\delta}_{\text{full}} \) is relevant for ortho-vis, leading to a single-component measure for inconsistency in orthogonal LIC transport:

\[
\vec{\delta}_{\text{LIC}} = \frac{\partial v_x^A}{\partial y} + \frac{\partial v_y^A}{\partial x}.
\]

(3.11)

The term \( \frac{\partial v_x^A}{\partial y} \) describes the curvature of the vector field, whereas the term \( \frac{\partial v_x^A}{\partial y} \) describes the shear. De Leeuw and van Wijk [dLvW93] present a detailed description and interpretation of those differential quantities of a vector field. Their discussion targets 3D flow, which can be immediately applied to the
scenario of 2D flow discussed here by restriction to two coordinates \( x \) and \( y \). Please note that a frame of reference is used in which the \( x \) axis points along the vector field and the \( y \) axis along the perpendicular vector field direction. Therefore, the measure of inconsistency is directly related to the curvature and shear of the vector field. In the example of pure shear, as illustrated in Figure 3.3, \( \delta_{\text{LIC}} \) is given by the non-vanishing shear value only. In contrast, the example of the circular flow (Figure 3.2) does not exhibit any temporal incoherence because effects of curvature and shear cancel exactly. For unsteady flow, additional incoherence effects might be introduced.

As mentioned in the beginning of this section, a two-part temporal filtering process is introduced to overcome the problem of temporal incoherence. The first part is a combination of advection and integral-curve construction: initial seed points are advected along \( \vec{v} \) from the initial time \( t_0 \) to an intermediate time \( t_i \) \((t_0 \leq t_i \leq t_1)\); then integral curves of \( \vec{u} \) are constructed at time \( t_i \); finally, those integral curves are advected along \( \vec{v} \) from \( t_i \) to \( t_1 \). This overall operation is denoted as \( T_{t_i} \). For a texture-based representation, \( T_{t_i} \) takes an initial noise image \( N \) as input and yields an image of transported integral curves. The second part applies a temporal filtering process in order to balance the conflicting goals of temporal coherence and orthogonal vector field lines. The actual visualization image at time \( t_1 \) is

\[
I = \int_{t_0}^{t_1} k(t_i) T_{t_i}(N) \, dt_i, \tag{3.12}
\]

with a filter kernel \( k(t) \) normalized according to \( \int k(t) \, dt = 1 \). This filtering process trades in clearly defined line-like patterns for a consistent and temporally coherent animation: the width of the filter interval \([t_0, t_1]\) determines the amount of “smearing out” and can be gradually adjusted. In fact, there are many important flow fields that exhibit no or only little inconsistencies and thus would not need any filtering. The circular flow of Figure 3.2 is an example of a completely consistent advection and integral-curve construction.

Animated visualization produces images for increasing end times \( t_1 \). For a constant filter width \((t_1 - t_0)\), the start time progresses accordingly. To achieve temporal coherence of the final images, the noise images \( N \) need to be temporally coherent for different times \( t_0 \), which can be ensured by advecting initial noise images along the vector field \( \vec{v} \).

An alternative approach avoids the two-part process: by reducing the length of the orthogonal LIC lines in regions of strong inconsistencies, these inconsistencies are reduced. However, this approach has the disadvantage that static images do not show the “wave fronts” of the vector field to the same degree as in the filtered version.
3.3 2D Algorithm

Here, an algorithm is described that realizes the approach from the previous section for vector fields given on planar 2D domains. All relevant information is 2D (vector field, input noise images, intermediate and final visualization images) and can be represented as 2D images, 2D textures, or 2D uniform grids. Vector data on unstructured, triangulated grids would also work because a triangle mesh can be easily rendered (i.e., rasterized) into a 2D image. The algorithm that produces the final output image can be seen as a pipeline consisting of three major stages (Figure 3.5). Each stage creates an intermediate result that is used as input for the next stage. The first stage (noise transport) implements two aspects of the abstract approach from Section 3.2.3: (1) temporally coherent input noise for different starting times \( t_0 \) and (2) the advection of noise from \( t_0 \) to the intermediate time \( t_i \). The second stage (orthogonal LIC) constructs a LIC image of the rotated vector field \( \vec{u} \) at time \( t_i \). The third stage (advection and blending) implements the transport of LIC patterns from time \( t_i \) to \( t_1 \) and computes the filter operation from Equation 3.12. In the following, the three stages are explained in more detail.

The first stage is responsible for creating a temporally coherent noise that should move according to the possibly time-dependent vector field \( \vec{v} \). Similarly to the approach of Weiskopf et al. [WEE03, Section 4], path lines are traversed from the current time step backward in time in order to accumulate noise injection input from previous times in a Lagrangian manner. This accumulation yields a convolution in time along path lines. The time span of backward particle tracing determines the scale of temporal correlation: typically, some 15–50 integration steps are appropriate for an adequate compromise between computation time and quality of temporal coherence.

The different noise injection images that serve as input for the temporal convolution need to be uncorrelated. To save memory for a large number of noise injection images, they are constructed on-the-fly by reusing a single template image. To achieve this, the template noise image must be periodic (i.e., a seamless texture), which, for example, is automatically achieved by generating a
low-pass filtered noise via filtering in Fourier space, using a fast Fourier transform. A new, uncorrelated noise image is produced from the template image by Cranley-Patterson rotation [CP76], which adds the same random shift to each point of the template image. The random shifts are applied on-the-fly while the noise injection texture is accessed. The density of the visual representation is controlled by the characteristics of noise injection. A dense representation is achieved by low-pass filtered white noise; a sparser representation is achieved by Gaussian-filtered sparse input constructed from randomly positioned dots.

Inflow and outflow at boundaries of the domain often cause problems for texture-based methods. This issue is addressed in two ways. First, the injection noise is periodic and thus virtually infinite in size. Second, the vector field is clamped at the boundary, making it virtually infinite as well. Therefore, particles can be traced beyond domain boundaries. Another issue is divergence or convergence of the flow, which could change the spatial frequency of injected noise by stretching or compression. Due to the limited integration length in time (some 15–50 integration steps), this problem does not lead to serious artifacts except for extremely large absolute values of divergence. Finally, the temporal convolution of uncorrelated noise images leads to reduced contrast. The convolution corresponds to a summation of (approximately) independent random variables, resulting in a normal distribution of values according to the central limit theorem. Contrast is restored by histogram equalization.

The result of the first stage is a noise texture that moves along the vector field $\vec{v}$ and serves as input to the second stage. The second stage creates LIC lines that visualize the orthogonal vector field $\vec{u}$ at a fixed time that corresponds to the current visualization time, which is similar to the spatial filtering process described by Weiskopf et al. [WEE03]. The rotation of the original vector field $\vec{v} = (v_x, v_y)$ is computed by a mapping to $\vec{u} = (-v_y, v_x)$. Usual particle tracing and LIC integration are performed with the orthogonal vector field. Vectors are normalized to unit length to obtain LIC lines of equal length. Boundaries of the domain are taken into account by stopping the LIC integration once a particle trace crosses a boundary. Similarly to stage one, contrast is enhanced by histogram equalization.

The third stage of the pipeline transports LIC patterns from the second stage and evaluates the filter operation from Equation 3.12. The goal of the third stage is to produce a temporally coherent and consistent visualization with line patterns that are (approximately) perpendicular to the vector field. Several intermediate images $T_i$ must be computed and stored to implement the filtering from Equation 3.12. This flexible and accurate approach was constructed mainly for evaluation and comparison purposes. The additional work and memory consumption for generic filtering can be avoided by restriction to an exponential filter kernel, which can be discretized in the form of a recurring application of the over operator (i.e., alpha blending with weights $\alpha$ and
(1 – \(\alpha\)) [EJW05]. The alpha value determines the falloff of the exponential filter. One image used for blending is the result of the second stage; the other image is the visualization result of the previous time step, transported to the current time step by semi-Lagrangian advection. This incremental computation of exponential filtering is mainly recommended for real-time rendering for interactive visualization.

If memory consumption and additional computation time are not limiting factors, then the generic filtering process can be used in order to obtain images of higher quality. The inexpensive, incremental exponential filtering process is compared with generic filter kernels in a parameter study in Section 3.7.3.

A different approach avoids the third stage completely. This approach adapts the length of orthogonal LIC lines in those regions where inconsistencies are present. To make this possible, the inconsistency is precomputed and used as a scaling factor at every point of the vector field to shorten the LIC lines proportional to the amount of inconsistency. The results of this approach are shown in Section 3.7.3.

### 3.4 2.5D Algorithm

This section describes the visualization of tangential vector fields on curved surfaces embedded in 3D space. The same basic pipeline is adopted as for 2D vector fields (see Figure 3.5), but some modifications and extensions are necessary that are specific to 2.5D data. The following discussion is restricted to those modifications. The input vector field may either be given as a 3D texture that is intersected by the surface or it is attached to the vertices of the surface. Since this algorithm is designed for tangential vector fields, a possibly non-tangential vector field is made tangential by subtracting the normal component of a vector.

The first stage of the pipeline (noise transport) adopts the hybrid physical/device-space approach of Weiskopf and Ertl which is detailed in Section 1.4.5. This LIC-based technique is turned into temporal convolution by the following modifications. First, particle paths are traced along path lines backward in time only. Here, the vector field is not normalized to unit length. Second, a single input noise is replaced by uncorrelated noise inputs for different times, according to a Cranley-Patterson rotation. Noise is modeled as 3D solid texture in order to achieve temporal coherence even under camera rotations, i.e., noise is attached to the surface geometry in object-space. The result of the first stage is a temporally coherent noise image that moves along the surface of the scene geometry.

The following steps are performed in image-space only, reminiscent of image-space advection techniques [LJH03, vW03]. The second stage (orthogonal LIC) takes the original vector field given in 3D space, rotates it by \(\pi/2\) around the
local normal vector of the surface, and projects the orthogonal vector field onto image-space. The rotation is determined in object-space by computing the cross product of the surface normal and the tangential vector. The subsequent projection to image-space yields a 2D vector field with respect to image-space coordinates. Finally, LIC is performed in image-space, based on the noise image from stage one and the image-space vector field. Particle tracing for LIC is stopped at the boundaries of the object; the silhouette of the object is determined according to a mask that contains the classification of pixels as foreground or background pixels. Because a complete LIC is evaluated, any filter kernel can be chosen. In contrast, image-space advection techniques [LJH03, vW03] are restricted to an exponential kernel, which yields lower image quality than the Gaussian kernel typically used in this implementation (see the discussion of filter quality by Weiskopf [Wei09]).

The third stage (advection and blending) consists of the following components: projection of the original, non-rotated vector field onto image-space; semi-Lagrangian image-space advection of the visualization result from the previous time step; and blending of the advected image with the image from stage two. The projection of the vector field is similar to the projection in stage two. Blending is a simple 2D image operation. Instead of the inexpensive blending operation which results in an exponential filtering, a generic filter kernel can be used here as well. To do this, the same steps are necessary as for the temporal filtering in 2D. In particular, intermediate images of the second stage are stored and a weighting and accumulation process is performed using an arbitrary filter kernel. Similarly to the 2D algorithm, the alternative approach of reducing LIC length in regions of inconsistency can be implemented by modifying stage two; in this case, stage three is not needed.

Semi-Lagrangian image-space advection can cause problems due to inflow at silhouette lines. To avoid inflow of background color, a modified bilinear interpolation is employed within the previous visualization image. This special filter works basically the same way as the standard bilinear filter—except for background texels, which are weighted zero. To decide whether a texel belongs to the background or to the surface geometry, the same mask is used as in stage two. If all four texels lie on the background, a gray-scale value of 0.5 is assumed. Currently, internal edges are neglected, i.e., image information could be transported across such edges. The approach of Laramee et al. [LJH03] could be included to overcome this issue. For the final display, the texture from stage three is modulated by a rendered image of the surface geometry to simultaneously show the vector field texture and the surface shape. This implementation supports the Blinn-Phong model and cool-warm shading [GGSC98] for surface illumination. Bump mapping is also available as an option to emphasize the structure of the vector field texture, mimicking the embossing of flow structures [UIL+04]. Here, the resulting texture from stage three is interpreted as a height field that perturbs the normal vectors.
3.5 Image Compositing of Combined Approaches

Conventional LIC images that show lines along stream lines have the advantage that the direction of the flow is directly perceived in such an image. In contrast, the proposed orthogonal LIC approach has the advantage that the spatial frequency of the moving pattern can be tuned to allow optimal perception of the animated flow. In this section, the computation of conventional tangent-vis within the proposed framework is presented and then image-compositing methods are proposed to combine tangent-vis with ortho-vis. The motivation is to create an approach that benefits from the two different LIC methods and overcomes their respective drawbacks.

To begin, the modifications required to compute intermediate tangent-vis images are discussed. Since tangent-vis and ortho-vis mainly differ in the direction of their field lines, only the changes of the ortho-vis pipelines for 2D (Section 3.3) and 2.5D (Section 3.4) are of interest. The most important difference is that the original vector field, not the rotated vector field, is used: stage two in Figure 3.5, which computes orthogonal LIC, is replaced by a LIC computation that works with the original vector field. Similarly, the 2.5D algorithm avoids the rotation of the vectors before projection to the image plane. Otherwise, the principal processing pipelines are identical for ortho-vis and tangent-vis, including stage one (transport of noise) and stage three (temporal filtering).

The next aspect concerns the actual combination of the orthogonal and tangential representations. These two representations are combined by compositing in image-space using the LIC density (i.e., the gray-scale value that originates from the LIC computation). The LIC density computed for the orthogonal and tangential LIC images is denoted as $I_{\text{ortho}}$ and $I_{\text{tangent}}$, respectively. Often, those LIC images are immediately interpreted as gray-scale values for final display. However, this approach takes into account the possibility of some additional color map applied to the LIC images before the final result is produced. The respective color-mapped images are denoted $C_{\text{ortho}}$ and $C_{\text{tangent}}$.

Based on this terminology, two different compositing models are proposed. The first one is called the *engraving* model and is formulated by the compositing equation

$$C_{\text{comp}} = \begin{cases} C_{\text{ortho}} & \text{if } I_{\text{ortho}} > I_{\text{tangent}} \\ C_{\text{tangent}} & \text{otherwise} \end{cases}$$

which is applied to each pixel independently. The engraving approach essentially results in a maximum computation: the maximum of the two LIC densities fully determines the final color. For the special case of an identity color map, which implies a gray-scale image, Equation 3.13 is reduced to the maximum operator $I_{\text{comp}} = \max(I_{\text{ortho}}, I_{\text{tangent}})$. This expression motivates the term “engraving”: the LIC values can be interpreted as depth values of a
height-map; only the deepest engravings survive the compositing (i.e., the subsequent application) of intermediate engravings. Visually, the engraving effect is especially clear in combination with bump mapping, which is demonstrated in Section 3.7.2 and Figure 3.11.

The engraving approach is easily computed, it has a simple interpretation, and it immediately works with height-maps and bump mapping for embossing. However, one possible shortcoming is that engraving only keeps the LIC lines with large values, i.e., dark lines of one representation can never occlude bright lines of the other representation. To overcome this problem, an alternative compositing scheme is proposed, called interweaving. The basic idea is that the color and brightness of LIC structures should be chosen independent of their relative depth, leading to interweaving patterns reminiscent of the interweaving of threads in fabrics. Independence is achieved by adding height-map textures $H_{ortho}$ and $H_{tangent}$, respectively. The height-maps are produced by the same pipeline as the original LIC density textures, but are based on completely independent input noise. The compositing equation of interweaving is

$$C_{\text{comp}} = \begin{cases} 
C_{\text{ortho}} & \text{if } H_{ortho} > H_{tangent} \\
C_{\text{tangent}} & \text{otherwise}
\end{cases}$$

According to the value stored in the height-map texture, either the tangential LIC image or the orthogonal LIC image is drawn. By doing this, LIC images of both types are drawn on top of each other in an alternating fashion, thus creating the desired interweaving effect. The input noise for the height-map computation is always based on filtered white noise in order to densely cover the whole domain by height-field values, without any holes. The height-map texture provides a certain amount of control over the appearance of the final interweaving pattern: by modifying the scale of the white noise the width of the resulting (invisible) depth LIC lines can be controlled, and therefore how often one of the visible interweaving LIC lines “changes depth”.

In contrast to the height-map textures $H$, the density LIC textures $I$ may use different noise models. For example, moderately sparse noise together with asymmetric filtering is recommended in order to visualize downstream flow direction for tangent-vis according to the idea of Oriented LIC (OLIC) [WGP97]. For visual consistency, the ortho-vis image should also be based on a similar kind of sparse noise in this case. However, there is no natural downstream direction for the orthogonal vector field and, thus, a symmetric filter kernel like the Gaussian function should be applied for ortho-vis.

Different parameters and compositing models for combined LIC are compared in Figure 3.6. An example of interweaving is depicted in Figure 3.6a), where LIC lines of one type are running on top and below LIC lines of the other type in an alternating fashion. The data set is a circular flow. Figure 3.6b) applies
Figure 3.6: A circular vector field visualized with combined LIC: (a) interweaved LIC, (b) engraved LIC, (c) engraved LIC with dense noise.

The engraving model to the same visualization scenario. Zoomed-in views of these two figures demonstrate that engraving and interweaving lead to different occlusion behavior. Similarly to color weaving [UIM+03], appropriate color mapping is recommended for effective engraving and interweaving: the two combined LIC images use two different colors that are highly saturated and approximately isoluminant. Approximate isoluminance is instrumental in avoiding induced perception of shape and other features that could affect the visualization. Isoluminant colors can be conveniently generated by the method of Kindlmann et al. [KRC02]. The colors are chosen to be highly discriminable in order to achieve easy separation of the line structures and good visual continuity [IG97]. In addition, colors are chosen that induce the same depth perception to prevent conflicting depth cues (e.g., red and blue would be bad choices because they induce different perceptual depth).

Figure 3.6c) shows another example of engraving. Here, sparse noise for the density LIC images is replaced by white noise to achieve a denser visual representation. In addition, the isoluminant color table is substituted by a grayscale mapping. Although bump mapping or embossing is not applied, this image provides a subtle depth impression due to perceptually induced shape recognition connected to luminance changes. Gray-scale mapping is recommended only in combination with the engraving model because this compositing scheme guarantees consistency of grayscale values and depth. Furthermore, Figure 3.6c) demonstrates that combined LIC also works with dense noise.
3.6 Implementation

The GPU implementations of the 2D and 2.5D algorithms are based on C++, DirectX 9.0, and HLSL for shader programming. The above algorithms are mapped to vertex and pixel shaders, the data structures are realized by 2D or 3D textures. Shader model 3.0 is essential because loops in pixel shaders are necessary.

For the 2D implementation of ortho-vis with incremental temporal filtering, each operation in the pipeline of Figure 3.5 is mapped to one pixel shader program that works on 2D images represented by 2D textures. Similarly, the analogous pipeline is implemented for tangent-vis, except for the missing rotation of the vector field. For steady vector fields, multiple render passes are not required for any of the stages. The template noise is precomputed on the CPU and low-pass filtered in Fourier space. Data between different stages is transferred as 2D textures (16-bit floating point format) filled by means of the render-to-texture functionality. Semi-Lagrangian advection uses the built-in bilinear interpolation within 16-bit floating point textures.

For the 2.5D implementation with incremental temporal filtering, each stage is essentially mapped to two shaders and two render passes: one shader implements the different variants of projecting the vector field onto the image plane; the subsequent shader is responsible for the actual particle tracing and/or integration. Data between stages is transferred as 2D textures with 32-bit floating point format. Semi-Lagrangian advection is based on a modified version of bilinear interpolation (see Section 3.4) that is explicitly implemented in a pixel shader. Once again, ortho-vis and tangent-vis are based on essentially the same processing pipelines and implementations, except for the difference in the computation of the vector field.

Image compositing for combined LIC is implemented by one image-space operation that takes the intermediate results from ortho-vis and tangent-vis as input and that outputs the composited image. The engraving model works directly on the LIC density values of the intermediate images. For the interweaving model, additional height-map textures have to be generated during the computation of ortho-vis and tangent-vis. Here, it is not necessary to use multiple render passes to create the additional textures. Instead, the additional rendering is performed in parallel by writing to multiple render-targets.

Time-dependent flow requires minor changes in the implementation—the first stage is modified, since the coherent noise is accumulated over past time steps. Here, for each of these time steps, the vector field is found by interpolating between the previous and next time step of the flow. The noise values are accumulated by using multiple render passes. The second and third stage remains unchanged.

The generic temporal filtering process requires a few changes to the aforemen-
tioned processing pipeline. First, intermediate, temporally unfiltered images of different times are stored in additional textures. Those images are the result of the second stage in the pipeline of Figure 3.5. Stage three (semi-Lagrangian advection and alpha blending) is disabled. The textures with the unfiltered images are organized as a ring buffer that keeps track of the time tag of the buffer textures: the oldest texture is overwritten with the current result of the second stage. The number of elements in the ring buffer determines the filter size for temporal filtering. The actual filter process reads all unfiltered intermediate images, applies the convolution kernel, accumulates the filtered values, and writes the final result. This is implemented using multiple render passes: the images stored in the ring buffer are weighted and accumulated inside this loop with ping-pong rendering. Therefore, the complexity of the generic temporal filtering algorithm is linear with respect to the filter size, whereas the incremental filtering by recurring alpha blending is independent of the filter size (i.e., the alpha value).

The alternative approach of shortened LIC lines needs an additional texture which is precomputed to store the amount of inconsistency at all texel locations. During the second stage, this texture is accessed to retrieve and apply the scaling factor for the LIC length.

### 3.7 Results

#### 3.7.1 Performance Results

The following tests were conducted on a Windows PC with Intel Dual Core CPU (1.86 GHz), 2 GB RAM, and NVIDIA GeForce 7900GS GPU with 256 MB of texture memory. Figure 3.7 documents timings (in milliseconds) for rendering a single image of orthogonal vector field visualization. Each plot shows measurements for a 2D data set (Benard convection, depicted in Figure 3.8) and a 2.5D data set (a spherical object with a vector field given on a 3D texture). Figure 3.7 (left) shows the behavior for varying integration length in the

![Figure 3.7](image-url)  
**Figure 3.7:** Performance results for varying parameters and squared viewports: number of virtual noise textures (left), LIC integration length (middle), resolution (right). All vertical axes show rendering times in ms/frame.
3.7. RESULTS

(first stage of Figure 3.5 (noise transport), i.e., different temporal convolution of virtual input noise images. The viewport size is $512^2$ and the LIC convolution length is $2 \times 100$. The 2D case exhibits an almost linear behavior. The 2.5D algorithm shows an unexpected increase of rendering time for long convolution lengths, which might be explained by an influence of the texture cache.

Figure 3.7 (middle) reports timings for varying LIC integration length (for stage two of Figure 3.5). The integration length is given as the length in one direction, i.e., the total number of integration steps is twice the displayed number. The viewport size is $512^2$ and the temporal convolution length is 16. Finally, Figure 3.7 (right) illustrates the influence of the viewport size, for constant temporal convolution length (16 steps) and constant LIC convolution length ($2 \times 100$ steps). Please note that the $y$ axis has a quadratic scale. As expected, all plots show almost linear behavior for the varying parameters. Therefore, quality (i.e., longer integration or more pixels) can be gradually balanced with rendering speed. In general, typical 2D and 2.5D visualizations render at some 10 frames per second, which facilitates interactive applications.

When interweaved LIC is enabled, the computation time for rendering one frame increases by approximately 60%, regardless of the other parameters. This is an expected result since additional render-targets are used, and therefore the execution time for the pixel shaders of stages one and two increases.

3.7.2 Qualitative Results

First, the orthogonal vector field visualization is compared with the existing method of tangential stream line LIC. In Figure 3.8, convection flow is used to present the two different LIC approaches for the 2D case. Figure 3.8a) shows the conventional way of visualizing the vector field using the standard LIC approach, whereas Figure 3.8b) shows the method of orthogonal vector field visualization.

Figure 3.8: 2D vector flow visualized in two ways: (a) conventional tangent LIC, (b) orthogonal vector field visualization.
Figure 3.9: Flow visualization on curved surfaces: (a) conventional, tangent LIC, (b) orthogonal vector field visualization.

visualization. Figure 3.9 illustrates tangent-vis and ortho-vis side-by-side for a 2.5D data set from an automotive CFD simulation.

Different compositing models for combined LIC were already discussed in Section 3.5 in Figure 3.6. In addition, with combined LIC several parameters can be changed in order to modify the look of the resulting images as shown in Figure 3.10. Depending on how much detail in the visualization is desired, the density of the sparse noise for the conventional stream line LIC image as well as for the orthogonal LIC image can be adjusted. Furthermore, the integration length for the two different LIC images can be modified.

The orthogonal vector field visualization can also be combined with conventional LIC images in 2.5D: in Figure 3.11a), a uniform vector field is projected onto the surface of a torus. Here, the engraving compositing scheme is applied, which immediately leads to a height-field interpretation. The height-field character is emphasized by bump mapping applied to the composited image (bump-mapping increases the contrast of the lines, making them even better perceivable). In Figure 3.11b), the same technique is used, except for additional color coding of the magnitude of the vector field. The color image is blended with the LIC image, providing an additional cue for the velocity of the vector field.

3.7.3 Temporal Filtering

Figure 3.12 shows the 2D orthogonal vector field visualization of a shear flow. As discussed in Section 3.2.3, a shear flow is a challenging example because of substantial inconsistencies in the transport of orthogonal LIC patterns. In Figure 3.12a), the inexpensive exponential filter is used with an alpha value of 0.075. When alpha blending is being used to perform temporal filtering, a
Figure 3.10: Different parameters for interweaved LIC: (a) integration length for conventional LIC lines four times smaller than for orthogonal LIC lines, (b) integration length for orthogonal LIC lines four times smaller than for conventional LIC lines, (c) density of orthogonal LIC lines is three times higher than for conventional LIC lines.

Figure 3.11: An example of combined LIC visualizing the tangential vector flow on the surface of a torus: (a) bump mapping improves the perception of the LIC patterns, (b) velocity magnitude is additionally encoded by using a color map.
smaller alpha value leads to a wider filtering and thus to a larger blurring of
the image in regions of inconsistency. This is helpful in perceiving the over-
all motion of the vector field. From experience, useful alpha values are in the
range of 0.03–0.3, depending on the animation speed and structure of the vec-
tor field.

In Figure 3.12b), again an exponential filter is used; however, this time the
version is used that results from the generic filtering approach with 25 inter-
mediate images. This version shows slightly less pronounced artifacts because
the generic filtering pipeline performs a completely Lagrangian particle trans-
port. In contrast, the alpha blending of the incremental filtering process im-
plies the use of semi-Lagrangian advection, which leads to numerical diffu-
sion (see discussion of Weiskopf et al. [WEE03]). Figure 3.12c) is created with a
Gaussian filter kernel using the generic filtering pipeline. This kind of filtering
reduces irritating patterns in the region of highest velocity because the Gaus-
sian function exhibits a faster falloff in frequency space than the exponential
function [Wei09]. These irritating patterns can be observed in Figures 3.12a)
and b) as slightly undulating patterns.

Figure 3.12d) shows the result of filtering with the sinc() kernel. In theory, the
sinc() filter provides perfect temporal low-pass filtering; however, the filter
has to be of infinite length in order to reach this quality. A relatively short filter
length of 15 is used in this implementation and the rest is cut off of the infi-
nite filter function. Nevertheless, the results show better visual quality than
the exponential filter of length 25. Finally, Figure 3.12e) illustrates box filter-
ing, which serves as a negative example: the region of high velocity is washed
out and additional aliasing patterns appear. The reason for the bad quality of
3.7. RESULTS

Figure 3.13: Comparison of Gaussian filter kernels of varying length. The graph on the left shows the velocity profile for this example. (a) Length 10, (b) length 20, (c) length 30, (d) length 40, (e) length 50.

The box filter is its high contents of spurious high-frequency contributions in frequency space. In summary, Figure 3.12 demonstrates that the generic filtering pipeline along with appropriate low-pass filters (like Gaussian or sinc(·)) provides best quality. Nevertheless, exponential filtering—even in the form of incremental alpha blending—leads to acceptable results. Therefore, the incremental filtering process is an acceptable option for fast, interactive visualization.

Figure 3.13 demonstrates the effects of increasing filter length. In this case, the Gaussian filter is chosen and its kernel length is varied from 10 up to 50. As the filter length increases, more images of previous time steps are included in the filtering process. For each rendered frame, the filtering process advects the images of the filter support along the original vector field (see Section 3.2.3 for more details). Therefore, an increase in the slope of the LIC patterns can be seen in regions of increasing velocity. This behavior is a by-product of the proposed temporal filtering technique and can be observed regardless of the choice of filter kernel. This effect might be misleading in static images, since the direction of the flow in the shear area seems to change for increasing filter lengths. However, according to observations, this misinterpretation is less likely in the animated visualization. The advantage of an increased filter length lies in its improved temporal coherence. Based on experience, insufficient temporal coherence results in clearly visible shower-door effects, whereas stronger temporal coherence with a filter length of approximately 30 or greater reduces the shower-door effects almost completely. Additionally, the hypothesis is put forth that strong temporal coherence and reduced shower-door effects facilitate the perception of consistent flow; however, this hypothesis requires further investigation by user studies.
Yet another way of overcoming the problem with shearing flow is shown in Figure 3.14. For comparison, a) and b) show the results of recurrent alpha blending and a Gaussian filter, respectively, whereas c) was generated without any temporal filtering. Here, the length of orthogonal LIC lines is modified proportionally to the strength of the shear in the visualized vector field. This way, the lines become very short in regions of strong shear to avoid that the LIC lines are “torn apart”. The disadvantage of this approach is that it works only with animated visualization—in static images, the underlying vector field is harder to recognize because the line patterns become degenerate.
Part II

Topology-Based Visualization
The techniques presented in the previous part of this thesis visualize vector fields in an unfiltered way. This can be a drawback for highly complex data sets, especially in three dimensions. In this part, topology-based methods will be applied to filter the data. Therefore, the resulting visualization is reduced by removing unessential parts of the visual representation. In other words, features of data sets are extracted and visualized with the goal of simplifying the analysis of these data sets.

Feature extraction techniques, providing a condensed representation of the essential information, are often applied to the visualization of vector fields. A prominent concept revealing the overall structure is vector field topology [HH89]. Whereas vector field topology is directly applicable only to steady or quasi-stationary vector fields, Lagrangian coherent structures (LCS) [Hal01] are popular for the analysis of time-dependent vector fields. A short introduction to LCS is given in Section 1.5.1.

LCS are a time-dependent counterpart to separatrices, which are stream lines that separate regions of different behavior. LCS have been increasingly subject to research in the last decade and can be obtained as maximizing curves (ridges) in the finite-time Lyapunov exponent (FTLE), a scalar field measuring the separation of trajectories [Hal01]. FTLE computation is, however, an expensive task because at least one trajectory needs to be computed per sample point. LCS behave as material lines under the action of time-dependent flow, i.e., they are advected and exhibit negligible cross-flow for sufficiently long advection time intervals, as reported by Haller [Hal01], Lekien et al. [LCM+05], and Sadlo et al. [SUEW12]. This property gives rise, e.g., to the acceleration technique by Sadlo et al. [SRP11] based on grid advection.

Our approach [BSDW12], that will be detailed in this chapter, adopts the concept of hyperbolic trajectories and space-time streak manifolds. Previous work by Sadlo and Weiskopf [SW10] generalized vector field topology to time-dependent vector fields by replacing the role of stream lines by generalized streak lines [WTS+07]. In this way, critical points turn into degenerate streak lines and separatrices turn into streak lines (space-time streak manifolds) converging toward these degenerate streak lines in forward or reverse time.
Hyperbolic trajectories can be seen as constituent structures in time-dependent 2D vector field topology. As mentioned, space-time streak manifolds—the time-dependent counterpart to separatrices—can be constructed alone from hyperbolic trajectories—no dense sampling is required in contrast to the FTLE approach. However, a major limitation with hyperbolic trajectories is the difficulty of their integration. Although the integration error tends to grow exponentially in linear vector fields, it is usually negligible due to comparably short advection times and low separation rates along common trajectories. Unfortunately, this is not the case in typical hyperbolic configurations due to large separation rates and the fact that both forward and reverse integration are subject to repulsion from one of the LCS (see Figure 4.1).

Hyperbolic trajectories coincide with the intersection of forward and reverse LCS; since ridges in forward FTLE represent repelling LCS whereas those in reverse FTLE are attracting, the trajectory is repelled from the former in forward and from the latter in backward direction. The method presented in this thesis has two advantages: First, avoiding the integration of hyperbolic trajectories by replacing them with intersections of LCS, and second, revealing tangential dynamics in LCS, accomplished by line integral convolution. By treating time as an additional dimension, a stationary visualization is obtained that conveys the overall structure in space-time.

Several approaches for obtaining seeds for hyperbolic trajectories exist: by intersecting ridges in hyperbolicity time [Hal00], ridges in FTLE [SW10], and constructing streak manifolds from them, or by building a time-dependent linear model from critical points [ISW02]. This kind of visualization of hyperbolic trajectories is, however, restricted to LCS geometry, i.e., the dynamics in the vicinity of the hyperbolic trajectories is not conveyed. Furthermore, the hyperbolicity of the vector field is typically analyzed by requiring negative determinant of the velocity gradient. This approach fails in providing insight into the role and importance of hyperbolic trajectories. In contrast, the presented LIC-based visualization captures the configuration of the flow in the neighborhood of hyperbolic trajectories and also in general along LCS. One example is the discrimination of almost parallel flow configurations from strongly hyperbolic ones, as demonstrated in Figure 4.2. This provides increased insight in the overall dynamics, interplay, and importance of LCS, which allows for, e.g., a qualitative analysis of mixing phenomena.
Figure 4.1: Space-time ridge surfaces in forward (red) and reverse (blue) finite-time Lyapunov exponent together with cross section at final time step (colored). The space-time intersection curve in the center (white) represents a hyperbolic trajectory. Traditional integration of the hyperbolic trajectory (yellow curve) from the initial intersection is difficult due to exponential growth of error.

Figure 4.2: Traditional visualization of a time-dependent vector field by time series of the finite-time Lyapunov exponent (left) is difficult to analyze and does not convey the dynamics inside its ridges (LCS). The space-time representation (right) reveals the overall structure and makes the dynamics inside the LCS visible by line texture patterns. Close-ups: in contrast to the traditional 2D visualization, different dynamics along intersection curves (almost parallel flow on the left vs. strongly hyperbolic flow on the right) are apparent.
4.1 Space-Time LCS Visualization

The visualization technique presented in this thesis builds on the fact that time-dependent vector fields can be turned into stationary ones by treating time as additional dimension. This approach is common in the field of differential equations, where non-autonomous systems are made autonomous. Hence, 2D time-dependent vector fields \((u(x,y,t), v(x,y,t))\) are converted into steady 3D vector fields \((u(x,y,t), v(x,y,t), 1)\), which are denoted as space-time vector fields. All following steps of the algorithm (see Figure 4.3) take place in this space-time domain. Since 2D path lines represent stream lines in space-time, 3D stream line integration is used over advection time \(T\) inside the space-time vector field to generate a flow map \(\phi(x,y,t) \mapsto (x', y', t + T)\). Then, for each time slice \(\bar{t}\) of the space-time stack, the traditional 2D FTLE is computed according to Haller [Hal01] as

\[
1 / \lvert T \rvert \ln \sqrt{\lambda_{\text{max}}[(\nabla_2 \phi(x,y,\bar{t}))^\top \nabla_2 \phi(x,y,\bar{t})]},
\]

with \(\nabla_2 = (\partial / \partial x, \partial / \partial y, 0)\) and major eigenvalue \(\lambda_{\text{max}}(\cdot)\). Then, LCS are extracted from the resulting stack of traditional 2D FTLE fields by ridge surface extraction, discussed in Section 4.1.1.

Due to the previously mentioned material advection property of LCS, these surfaces represent stream surfaces in the space-time vector field, i.e., they are tangent to the space-time flow. This allows a direct application of LIC techniques, which is described in Section 4.1.3. By doing this, LIC visualizes the dynamics of path lines along which the LCS are advected, and hence the dynamics within the LCS. As intersections of stream surfaces are stream lines, the space-time intersection of these LCS surfaces from forward and reverse FTLE represents a counterpart to hyperbolic trajectories.

In Section 4.1.2, the investigation of these intersection curves is addressed in terms of hyperbolicity, again based on LCS. Restricting the LIC visualization to bands around the intersection curves helps to avoid occlusion and visual complexity, as shown in Section 4.1.4.

4.1.1 Ridge Surface Extraction

According to Eberly [Ebe96] and following the approach of Sadlo and Peikert [SP09], ridge surfaces are extracted from the stack of 2D FTLE fields as height ridges of co-dimension one from the 3D space-time FTLE field. This approach is used here as well to avoid the problems that would arise from stitching of the individual ridge curves from the 2D FTLE fields. Furthermore, ridges are typically non-manifold, which would cause further issues. Since Eberly’s formulation [Ebe96] is local and relies on higher-order derivatives, it
is subject to erroneous solutions. It is therefore common practice to apply filtering. In this thesis, the filtering process described by Sadlo and Peikert [SP07] is applied: since only sufficiently sharp FTLE ridges represent LCS, ridge regions are suppressed where the modulus of the eigenvalue of the Hessian is too low. Further, a minimum FTLE value is required, which is coupled to the minimum separation strength of the LCS. Finally, small ridges are suppressed by filtering ridge surfaces by area. As described by Sadlo and Peikert [SP07], a least-squares approach is used to prevent noise amplification during estimation of the gradient and Hessian. Figure 4.1 shows examples of ridges extracted from a stack of forward and reverse-time FTLE: repelling LCS (ridges in forward FTLE) colored red and attracting ones (ridges in reverse FTLE) blue. The space-time structure of the field is revealed including the intersection curves. However, this does not convey hyperbolicity aspects, e.g., it does not disambiguate intersection curves representing strong hyperbolic trajectories from weak hyperbolic ones. This motivates the visualization of hyperbolicity on LCS as described in the following section.

4.1.2 Visualizing Hyperbolicity

To help the user in the investigation of hyperbolic effects, and hyperbolic trajectories in particular, hyperbolicity is mapped to saturation, as shown in Figure 4.4. The hyperbolicity definition is based on the one by Haller [Hal00]. Haller defines hyperbolicity as the sign of the determinant of the velocity gradient of the original 2D vector field at the respective space-time location. However, in order to obtain a scalar value, the determinant of the velocity gradient is used directly, but only if this value is smaller than zero:
4.1. SPACE-TIME LCS VISUALIZATION

Figure 4.4: Building blocks for space-time LCS visualization. Advection time for forward and reverse FTLE is $T = 4s$. (a) Space-time LIC qualitatively visualizes LCS dynamics: hyperbolic behavior is apparent. In addition, hyperbolicity is encoded by color saturation. A minimum FTLE value of 0.5 is used. (b) Intersection bands by clipping with complementary FTLE reduce occlusion but still provide context and convey structure of reverse FTLE. The minimum complementary FTLE value is 0.41. (c) Intersection bands by clipping with distance to intersection curves further reduces occlusion and provides the topological skeleton.

$$\text{hyperbolicity} = \begin{cases} 0 & \text{if } \det \nabla u \geq 0 \\ \det \nabla u & \text{else} \end{cases}.$$  

One can see how this technique not only reveals the presence of hyperbolicity but also allows for the interpretation of the hyperbolic regions around the intersection curves. To examine hyperbolicity more precisely a novel technique to visualize LCS dynamics is introduced in the next section.

4.1.3 Visualizing LCS Dynamics

The LCS in the space-time FTLE field of this approach are present as ridge surfaces and to fully capture the spatial variation of their dynamics, they lend themselves to dense texture-based visualization such as LIC. Since LCS lack intrinsic surface parametrization and need to be analyzed in overview scales as well as in local detail, image-space oriented approaches are predestined to visualize the space-time structure. The hybrid physical/device-space LIC approach by Weiskopf and Ertl is used here which is detailed in Section 1.4.5.

In the context of this visualization of LCS dynamics, the goal is to visualize the space-time direction of the vector field. Hence, space-time vectors are normalized during LIC computation to obtain LIC line patterns of uniform length for
optimal perception. In contrast to traditional spatial LIC, visual encoding of velocity magnitude is retained in the form of surface orientation in space-time. For example, small angles between surface normal and the time axis indicate high speed.

Figure 4.4a) exemplifies the method again on the same data set. It is apparent how this technique conveys the time-dependent dynamics within LCS. Combining it with the saturation-based visualization of hyperbolicity (Section 4.1.2) supports the identification of hyperbolic intersection curves and still provides the LCS dynamics context. Since LCS are often convoluted, they typically exhibit many intersections that are commonly occluded. This problem is addressed in the next section. At the same time, this approach explicitly addresses the analysis of the intersection curves.

### 4.1.4 LCS Intersection Bands

Even in the simple example shown so far, it is obvious that occlusion tends to be a problem in space-time visualization of LCS. To address this and to provide a method for analyzing intersection curves of LCS at the same time, two complementary approaches were developed that have proven valuable in experiments, with both approaches restricting the presented visualization to bands around the LCS intersection curves.

As discussed in Section 4.1.1, a common approach is to filter FTLE ridges by prescribing a minimum FTLE value. This way, the visualization is restricted to important LCS, i.e., those representing strong separation. This filter is applied to ridges in both forward and reverse FTLE fields. If an additional minimum value is prescribed for the complementary FTLE, i.e., the reverse in case of forward FTLE ridges and the forward in case of reverse FTLE ridges, one typically restricts the visualization to bands around the intersection curves, shown in Figure 4.4b). This technique has the advantage that the profile of the complementary FTLE field is conveyed, allowing qualitative interpretation of the interplay of LCS. Furthermore, it often features additional bands that do not exhibit LCS intersections. They are generated if FTLE ridges are located in regions of high complementary FTLE. These additional bands are still of interest: the respective regions exhibit both high forward and reverse-time FTLE. Additionally, these bands may connect to other bands that feature intersection curves and hence convey the overall organization of the LCS.

A drawback of this approach, however, is that the bands may get too narrow for appropriate LIC visualization or too wide to sufficiently reduce occlusion. Therefore, an alternative is proposed which restricts LCS to the neighborhood of their intersection curves. First, to avoid numerical issues, regions are omitted where the LCS intersect at small angle. Furthermore, a minimum length of the intersection curves is required to obtain significant visualizations. The remaining intersection curves are then used for distance computation, leading
to a distance field on the LCS that is then used for clipping. Figure 4.4c) shows an example: the dynamics of the LCS is well depicted by LIC and at the same time occlusion is reduced, allowing for the analysis of the intersection curves with respect to LCS dynamics and hyperbolicity. Since the resulting bands can still be too narrow due to perspective foreshortening, depth-corrected width of the bands is supported as described in Section 4.2.2.

To sum up, these clipping approaches result in visualizations that can be seen as an extended topological skeleton of time-dependent flow. Note that equal thresholds are used for forward and reverse-time FTLE ridge filtering as well as for complementary FTLE band clipping, in order to ensure a consistent visualization. Finally, there is a non-disputed similarity to the saddle connectors of Theisel et al. [TWHS03]. However, the approach of this work resides in space-time, whereas saddle connectors visualize 3D steady vector fields.

4.2 Implementation

This section details the implementation of the different building blocks of the technique as well as modifications to existing approaches. The pipeline shown in Figure 4.3 gives an overview of the steps and provides information about the data that are exchanged between different stages of the pipeline.

The software was written in C++. DirectX 9.0 and HLSL for shader programming were used in combination to create the final output images.

4.2.1 Preprocessing

Several steps in this technique are performed in a preprocessing phase. The original data set is given as a series of time steps of a 2D vector field. To create the stationary space-time 3D vector field, dimension lifting is applied, i.e., the time series of the 2D vector field are stacked and the time dimension is treated as additional third dimension. This space-time vector field is used to compute the 3D space-time FTLE field for forward and reverse time direction. Using this FTLE field, ridge surfaces are extracted. A detailed description of the ridge extraction method is given by Sadlo and Peikert [SP07]. The ridge surface meshes from forward and reverse-time FTLE are intersected to obtain the intersection curves. Once the geometry of all intersection curves is obtained, a distance field is computed that holds the distance of ridge surface vertices to the nearest intersection curve.

The next step is to compute vertex-based normals which are used for shading in the interactive visualization. During this process, normals are reoriented if necessary; however, since ridge surfaces are not necessarily orientable, this may not succeed for all normals. Remaining inconsistencies for the normals are treated during interactive visualization using a shader program. Finally, the
space-time flow vectors are sampled at the vertex locations of the ridge surface mesh. This resampling is independent of the FTLE sampling grid, allowing for acceleration methods [GGTH07, SP07, HSW11]. Distance values, normals, resampled flow vectors, and additional scalars like FTLE values, hyperbolicity, and the minor eigenvalue of the Hessian (see Section 4.1.1) are attached to the ridge surface mesh that is then passed to the interactive visualization stage.

4.2.2 Interactive Visualization

The core of this interactive visualization is based on hybrid physical/device-space LIC of Weiskopf and Ertl (see Section 1.4.5) to create line-like texture on the ridge surfaces. During rendering of the space-time ridge surfaces, Phong illumination is applied to enhance visibility and perception of the geometry. Since the ridge surfaces may be non-orientable, local normal vectors must be consistently oriented in order to avoid shading artifacts. Therefore, normal orientation is made consistent during fragment processing using the dot product between the normal and view vector. This prevents inconsistent shading due to normal interpolation; however, ridge surfaces may still appear rippled. This happens because of FTLE aliasing effects at strong and sharp ridges, where very high FTLE gradients are present. To compensate for this, normals are corrected to be perpendicular to the space-time vector field and hence to its LCS during fragment processing.

Occlusion is handled by attaching additional data (regular FTLE, complementary FTLE, and distance to nearest intersection curve) obtained during the pre-processing stage (see Section 4.2.1) to each vertex of the ridge surface mesh. This additional data is then uploaded as texture coordinates to the GPU. Fragments that do not meet the filtering criteria are discarded. All thresholds used in this process are adjustable in real-time by the user. In addition to user controlled clipping, the width of the LCS intersection bands is adjusted based on distance to the camera position. This results in intersection bands with constant image-space width, which reduces occlusion of intersection bands that are close to the camera. At the same time, intersection bands that are farther away are enlarged, which improves visibility of the LIC pattern.

4.3 Results

The presented methods are evaluated by applying them to different data sets. The first two data sets are synthetic, whereas the third is created by CFD simulation, and the fourth is obtained by measurements of ocean currents. The implementation was tested on a PC with an Intel Core Quad CPU (2.4 GHz), 4 GB of RAM and an NVIDIA GeForce 275 GPU with 896 MB of dedicated graphics memory. Each of the presented data sets is visualized at interactive frame rates. The implementation is based on the approach of Weiskopf and Ertl (see
4.3. RESULTS

Section 1.4.5), therefore, it shows the same performance behavior. Navigation and orientation in space-time is aided by a bounding box of the domain. This bounding box is color-coded—the time dimension is indicated by a blue axis while the two spatial dimensions have a red and green axis, respectively. The last time step of the space-time region of interest is located at the back end of the bounding box which shows the FTLE field as a color-coded texture. In this texture, FTLE values are mapped to saturation, with full saturation mapping to the highest FTLE value. There, the same color-coding is used as for the space-time ridge surfaces.

4.3.1 Oscillating Gyre Saddle

The synthetic vector field used as an example in this section is taken from Sadlo and Weiskopf [SW10]. It exhibits a non-linear saddle shown in Figure 4.5a) that oscillates between the locations \((0.25; 0.25)\) and \((-0.25; -0.25)\) at a period of \(\tau = 4\). Resulting visualizations are presented in Figures 4.1, 4.2, and 4.4. To sum up, a strongly hyperbolic intersection curve and several non-hyperbolic ones are visualized in Figure 4.2 (right). This is consistent with the Eulerian picture 4.5a), which shows a distinguished saddle behavior at its center. As mentioned by Sadlo and Weiskopf [SW10], there are other ridges due to shear processes. These are less important for mixing and cannot give rise to hyperbolic trajectories, i.e., their LIC patterns do not show hyperbolic behavior. Please note that FTLE ridges in these examples are filtered to show the strongest and largest LCS only.
Figure 4.6: Quad gyre example. The advection time for forward and reverse FTLE is $T = 7.5\, s$. (a) Full visualization of forward and reverse LCS. A minimum FTLE threshold of 0.4 is used. (b) Visualization restricted to complementary FTLE bands. Minor artifacts appear due to aliasing effects of forward and reverse FTLE. The minimum complementary FTLE value is 0.19. (c) Restriction to distance-based LCS intersection bands reveals the topological space-time skeleton.

### 4.3.2 Quad Gyre

The double gyre example was introduced by Shadden et al. [SLM05] to examine FTLE and LCS, and to compare them to vector field topology. It consists of two vortical regions separated by a straight separatrix that connects two saddle-type critical points: one oscillating horizontally at the upper edge and the other synchronously oscillating horizontally along the lower edge. This is a prominent example where the vector field topology result substantially differs from LCS. This data set is temporally periodic. A larger range of field is used to avoid boundary issues, resulting in four gyres.

As proposed by Shadden et al., the configuration $\epsilon = 1/4$, $\omega = \pi/5$, and $A = 1/10$ is used. Figure 4.5b) shows a plot at $t = 0$ for these parameters. Space-time ridges are heavily occluded when rendering the quad gyre without clipping, as shown in Figure 4.6a). Please note that the $y = 0$ plane represents an LCS in both forward and reverse direction, which results in z-fighting. Nevertheless, the LIC line pattern is consistent in that region due to the image-based LIC technique.

Reducing occlusion by clipping with the complementary FTLE removes parts of the ridge surfaces, while preserving the context of the bands; this is shown in Figure 4.6b). Note, for example, that the red bands are connected at the upper edge of the domain and hence are part of the same LCS. If the space-time ridge surfaces are clipped by distance to their intersection curves, as shown
4.3.3 Buoyant Plumes

The third data set was obtained by a CFD simulation of buoyant 2D flow. A square container was modeled with a small heated region at its bottom wall and a small cooled region at its top wall. Figure 4.7 illustrates the flow. Two plumes develop: a hot one rising to the top and a cold one moving in reverse direction to the bottom. Then, they collide at the center and give rise to two new plumes traveling horizontally toward the side walls. As they approach the walls, they both split and produce plumes traveling in vertical direction. From that point on, the regular behavior is replaced by increasingly turbulent flow behavior. Figure 4.8a) shows the visualization of both forward and reverse-time FTLE ridges. There is no clipping applied for this image, but saturation already guides to the hyperbolic regions, however, many of them are occluded. In Figure 4.8b), the distance-based LCS intersection bands nicely visualize the hyperbolic mechanisms. One can see how the two plumes approach each other and merge, then divide and later give rise to turbulent flow. Finally, several strong hyperbolic regions are identified toward the end of the
Figure 4.8: Buoyant plumes example. The advection time for forward and reverse FTLE is $T = 1.5 \text{s}$. (a) Full visualization of forward and reverse LCS. The dynamics of the two plumes is apparent in the first part of the examined time interval. A minimum FTLE threshold of 0.87 is used. (b) LCS intersection bands clipped by distance, revealing the skeleton.

The multitude of hyperbolic regions confirms the observation of strong buoyant mixing. The high intricacy and topological complexity of turbulent buoyant flow is reflected by this visualization.

4.3.4 OSCAR

Ocean Surface Currents Analyses Real-time (OSCAR) [BL02] is a project to calculate ocean surface velocities from satellite data. The OSCAR product is a direct computation of global surface currents using satellite sea surface height, wind, and temperature. The OSCAR analyses have been used extensively in climate studies, such as for ocean heat storage and phytoplankton blooms. The presented technique is applied to the Gulf Stream at the east coast of North America. The focus lies on a strong hyperbolic LCS system involved in mixing, as shown in Figure 4.9a). As expected, the visualization technique reveals a complex Lagrangian skeleton of turbulence [MHP+07]. The LIC patterns allow a direct and qualitative inspection of the LCS with respect to hyperbolic mechanisms and mixing. Whereas many regions in the OSCAR data set exhibited inferior hyperbolic behavior, it is prominent in the selected region. Again, the LCS intersection bands dramatically reduce occlusion while still conveying topological structure and hyperbolic dynamics, as shown in Figure 4.9b).
Figure 4.9: OSCAR example. The advection time for forward and reverse FTLE is $T = 25$ days. (a) Full visualization of forward and reverse LCS. A minimum FTLE threshold of $9 \times 10^{-7}$ is used. Flow around several intersection curves shows strong hyperbolic behavior. The map shows the Atlantic ocean and the east coast of North America. It gives a frame of reference and exemplifies the prevalent mixing due to the gulf stream. Please note that this map shows water temperature mapped to colors and was generated outside of the investigated time interval. (b) LCS intersection bands clipped by distance.
Following the LIC line patterns along the temporal axis directly conveys the action of the flow in terms of mixing, i.e., thinning and folding.
The previous chapter demonstrated that feature-based visualization increases the abstraction level to guide the user’s attention to interesting or important parts of vector fields. In this chapter, again, topological structures are used to analyze vector fields. There are, however, several important differences between these chapters—first, the physical background changes from flow fields obtained by CFD computations to magnetic fields where no inherent flow is present. Due to this change, the attention turns to flux as opposed to flow, since important physical aspects can be observed based on the existence and the magnitude of flux. Furthermore, whereas the previous chapter visualizes unsteady vector fields, this chapter is restricted to steady fields.

Magnetic fields are a subclass of vector fields with some restrictions compared to generic vector fields. A magnetic field is divergence-free; therefore, its field lines are either closed or of infinite length. Inspired by traditional vector field topology, the goal of our approach [BSW+12] was to create a visualization of the topology of magnetic flux. However, established methods from vector field topology cannot be applied in a straightforward manner because the vector field (i.e., the magnetic flux density) may be infinite at singularity points. The most prominent example of such a singularity is the magnetic dipole, which is the central object of investigation in this chapter. Here, the focus lies on static 2D magnetic fields created by sets of dipoles. Since there are no magnetic monopoles, the first non-trivial term in any field expansion is represented by a dipole. Furthermore, elementary particles like electrons are points (without physical extent) that carry the magnetic moment of a dipole. Hence, the dipole representation is highly relevant for magnetic fields.

One natural way of visualizing a magnetic field is to use field lines. Although field lines are directed and a magnetic flux is present, there is no transport of any matter involved per se. The global structure of the magnetic flux between dipoles is of special interest, i.e., the connectivity of dipoles via magnetic field lines. Here, the fact is exploited that the topology of magnetic fields is reduced to only two types of critical points: dipoles and saddles. It will be shown that magnetic flux through two dipoles is found in a region that is always bounded by two saddles. Although similar to Morse-Smale cells, these regions are also
Figure 5.1: Comparison of (a) traditional topology and (b) flux topology. Dipoles are represented as a two-colored rectangles with a red north pole and a blue south pole. With flux topology, magnetic rings ① or chains ② are easier to identify.

bounded by dipoles instead of sources and sinks. The interdependence of the magnetic field is utilized and its corresponding vector potential to locate these regions. The key feature presented in this chapter is the definition of distinguished field lines that connect two dipoles and are used to visualize the topology of magnetic flux. This flux topology is based on a new topological construct called dipole connectrix, or shorter connectrix. Given the task of finding dipoles that interact with each other, e.g., are forming rings or chains of magnetic flux, it is of course possible to use traditional topology. However, to accomplish this, the user has to keep track of several topological curves at once. By using connectrices, this task is reduced to following a single line. A comparison of traditional vector field topology and flux topology is given in Figure 5.1.

As an application of this visualization technique, results from the simulation of systems of single domain magnetic nanoparticles are presented. The simplest example of such a system to which this visualization has great potential is a magnetic fluid, consisting of magnetic particles with an average size of 7–9 nm, suspended in a non-magnetic carrier liquid. Recent experiments [KDK+06] have shown how complex the microstructure of two-dimensional layers of a ferrofluid can be. Additional insight into these systems was obtained in theory and by computer simulation [KCH08, PDKH09]. However, none of these approaches can directly characterize the field distribution in the sample.

Another possibility is to blend magnetic and elastic properties within a single material, by embedding magnetic nanoparticles into an elastic polymer matrix. Materials that are designed as such are called magnetic gels, or ferrogels, and can serve as the basis for various potential applications, ranging from artificial muscles, actuators, and micro machines to biomimetic energy-transducing
devices. A manifestation of magneto-elastic coupling can be observed in the deformation of a macroscopic ferrogel body in a uniform or gradient magnetic field [Zri00, RDT10]. However, any application of these materials is based on the profound knowledge of their microstructure and on the ability to control and design them on various levels. There are only few theories that can treat the gel on the mesoscopic level, e.g., the work by Stolbov et al. [SRB11] and references therein, and simulations [WC11] aimed at the understanding of the gel microstructure. That is why, here, any additional knowledge of the magnetic field is of high relevance for further development of theoretical models.

5.1 Dual Topology of Magnetic Fields

This section summarizes the physical background that is needed for this chapter. Then, the formulation of dual vector field topology is presented for the class of vector fields which is of interest. This duality is the basis for the flux topology that is introduced in Section 5.2.

5.1.1 Physics of Magnetostatics

A comprehensive introduction to magnetostatics in particular, and classical electrodynamics in general is given in the textbook by Jackson [Jac75]. In this chapter, a setting of magnetostatics is assumed, i.e., any magnetic effects that may be additionally introduced from dynamics are ignored. Such a scenario is relevant for typical setups with steady-state behavior. The key observation is that there are no magnetic monopoles (in contrast to electrostatics with its electric monopole). Therefore, magnetic dipoles serve as the main building blocks for establishing magnetic fields. A single dipole is described by its magnetic moment

\[ \vec{m} = \frac{1}{2c} \int \vec{x}' \times \vec{J}(\vec{x}') \, d^3x' \]  

(5.1)

with the current distribution \( \vec{J} \) and the speed of light \( c \).

It is common practice to use the vector potential \( \vec{A} \) to describe the magnetic field. The term “magnetic field” is often used to refer to the magnetic flux density \( \vec{B} \), which is related to \( \vec{A} \) as follows:

\[ \vec{B} = \text{curl} \, \vec{A}. \]

Therefore, magnetism may be described using \( \vec{A} \) or \( \vec{B} \). In fact, many computations in physics are based on the vector potential. In particular, computations often use series expansions of the vector potential, similar to Taylor expansions of functions. The mathematical background is based on the expansion
by vector spherical harmonics [Jac75]. This kind of expansion reads for $i$-th component of the vector potential $\vec{A}$:

$$A_i(\vec{x}) = \frac{1}{c||\vec{x}||} \int J_i(\vec{x'}) \, d^3x' + \frac{1}{c||\vec{x}||^3} \int J_i(\vec{x'})\vec{x'} \, d^3x' + \cdots$$

Here, $\vec{J}$ is the current distribution that gives rise to the magnetic field. If only the first term of the expansion is used, this leads to

$$\vec{A}(\vec{x}) = \frac{\vec{m} \times \vec{x}}{||\vec{x}||^3},$$

where $\vec{m}$ is the magnetic moment of a magnetic dipole (see Equation 5.1). Put differently, the expansion of $\vec{A}$ up to the first non-vanishing term yields a magnetic dipole. Therefore, dipoles are highly relevant as, at least approximated, representation of any magnetic field; the more localized the current distribution, the better the approximation.

Finally, the magnetic flux density corresponding to $\vec{A}$ of the magnetic dipole reads

$$\vec{B}(\vec{x}) = \frac{3\vec{m} (\vec{n} \cdot \vec{m}) - \vec{m}}{||\vec{x}||^3},$$

where $\vec{n} = \frac{\vec{x}}{||\vec{x}||}$.

### 5.1.2 Dual Vector Field Topology

First, to describe the flux topology of 2D magnetic fields, the 3D vectors $\vec{A}$ and $\vec{B}$ are reformulated for the restriction to 2D. The 2D field is assumed to be defined on the $x$-$y$ plane. For this, the 3D field has to meet two requirements. First, the $z$-component of the flux density has to vanish everywhere, i.e., $B_z \equiv 0$. Second, $\vec{B}$ should be independent from the $z$-position, i.e., it should be shift-invariant along the $z$-direction.

These two requirements lead to the following constraints for the vector potential $\vec{A}$. First, $\vec{A}$ can be modeled as vectors that only have a $z$-component; their $x$- and $y$-components vanish. Second, the vector potential is independent from the $z$-position. With these constraints, the magnetic flux density is obtained:

$$\vec{B} = \text{curl} \, \vec{A} = \left( \begin{array}{c} \frac{\partial A_z}{\partial y} \\ -\frac{\partial A_z}{\partial x} \\ 0 \end{array} \right) \equiv \left( \begin{array}{c} \frac{\partial A}{\partial y} \\ -\frac{\partial A}{\partial x} \end{array} \right) = \text{curl}_2 \, A,$$
which indeed is a 2D field. In the above equation, the vector potential is rewritten by just using its \( z \)-component, with \( A_z = A \). Furthermore, the analog of the curl operator in 2D, \( \text{curl}_2 \), was introduced. The effect of \( \text{curl}_2 \) can be expressed as

\[
\vec{B} = P \nabla A, \quad P = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.
\]

Geometrically speaking, \( \text{curl}_2 \) computes the gradient of the scalar potential \( A \) and then rotates the gradient vector by \(-\frac{\pi}{2}\). This geometric observation is utilized in the following construction of the magnetic flux topology. \( \vec{B} \) and \( \nabla A \) are dual vector fields—in this sense, the traditional topology of \( \vec{B} \) can be considered as the dual of the traditional topology of \( \nabla A \).

The effect of \( P \) on the topology of the \( \vec{B} \) and \( \nabla A \) fields is examined to derive the relation between primal and dual topology. Since the magnetic flux density \( \vec{B} \) is divergence-free, it can only contain centers and saddles (and periodic orbits, which are not of interest to us). In turn, the rotation-free \( \nabla A \) field can contain only sources, sinks, and saddles. The first observation is that critical points are not affected by \( P \), i.e., critical points of one field are found at the same location in its dual field because \( P \) does not change vector magnitude:

\[
||\vec{x}|| = \sqrt{x^2 + y^2} = \sqrt{y^2 + (-x)^2} = ||P\vec{x}||.
\]

The examination of the effect of \( P \) is continued with respect to centers, sources, and sinks. These topological constructs are isotropic, i.e., they are rotation-invariant. However, \( P \) converts sources with their respective vector field

\[
\vec{u}(\vec{x}) = \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix} \vec{x}, \quad a > 0
\]

into counter-clockwise centers with

\[
\vec{u}'(\vec{x}) = P \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix} \vec{x} = \begin{pmatrix} 0 & a \\ -a & 0 \end{pmatrix} \vec{x}.
\]

Changing the sign of \( a \) results in a sink that converts likewise to a clockwise center when applying \( P \). This effect is illustrated in Figure 5.2a). As a result, dipoles in \( \vec{B} \), which can be interpreted as the composition of centers of opposite orientation at infinitesimally close distance, find their counterpart as infinitesimally close pairs of a source and sink in \( \nabla A \). Finally, the effect of \( P \) is examined for saddles. To begin, it will be shown that \( P \) does not change the determinant of any matrix \( M \):
5.2. FLUX TOPOLOGY

Figure 5.2: (a) Centers are converted to sources or sinks (depending on their orientation) under the action of $P$, since every vector is rotated by $-\frac{\pi}{2}$. (b) Saddles are rotated by $-\frac{\pi}{4}$ under the action of $P$.

$$\det M = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - cb$$
$$\det PM = \det \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = -cb + ad.$$ 

Since $\det M$ is invariant under $P$, the condition for saddles of a negative determinant is not affected by $P$, which means that saddles in one field persist in the dual field. Now, the orientation of a saddle is considered when applying $P$. For the sake of simplicity, the investigation is limited to axis-aligned saddles, 

$$\bar{u}(\bar{x}) = \begin{pmatrix} -a \\ 0 \\ 0 \\ b \end{pmatrix} \bar{x}, \quad a, b > 0$$

and apply $P$:

$$\bar{u}'(\bar{x}) = P \begin{pmatrix} -a \\ 0 \\ 0 \\ b \end{pmatrix} \bar{x} = \begin{pmatrix} 0 & b \\ a & 0 \end{pmatrix} \bar{x},$$

which results in eigenvalues $\lambda_{1,2} = \pm \sqrt{ab}$ of $\nabla \bar{u}'$.

Hence, saddles with $a = b$ of one field are rotated by $-\frac{\pi}{4}$ in the dual field, as illustrated in Figure 5.2b). For arbitrary values of $a, b$, the dual saddle is deformed, however, this is not of importance.

5.2 Flux Topology

In this section, the new flux topology is introduced that describes if, and how much, flux is present between dipoles.

5.2.1 Connection Regions

The discussion starts with dipoles that are oriented in opposite direction and create magnetic fields as illustrated in Figure 5.3a). There is no magnetic field
Figure 5.3: (a) In this configuration, there is no magnetic field line that connects the two dipoles. All magnetic field lines are separated by the separatrices (green). (b) Dipoles are oriented into the same direction. The region delineated by the separatrices (green) is called a dipole flux connection region.

Figure 5.4: (a) Two dipoles are oriented in the same direction, which results in magnetic flux between both dipoles heading from left to right. (b) The two centers of a dipole, which are infinitly close together, are shifted apart for illustration purposes only. (c) The two centers of a dipole within $\vec{B}$ are replaced by a source and a sink in the corresponding $\nabla A$ field according to the dual topology rules.

line passing through both dipoles (the separatrices converge to the saddle in infinite time). However, of special interest is the configuration with a consistent orientation as illustrated in Figure 5.3b). Here, a region is found that consists of all magnetic field lines that connect one dipole with the other one. This region is called dipole flux connection region, or shorter, connection region. The connection region is bounded by the separatrices that start at the two saddles located between the dipoles and run through both dipoles.

The magnetic field lines around a dipole form closed curves with the centers of these curves approaching the dipole center infinitely close. If more than one dipole is involved, some of the field lines can pass through other dipoles, as illustrated in Figure 5.4. For the sake of simplicity of illustration, the infinitesimal distance between the two centers of a dipole is expanded. Every configuration of two dipoles connected by a magnetic field line can be converted into a topologically equivalent configuration as shown in Figure 5.4a)
Figure 5.5: (a) A Morse-Smale cell (green) can be found in the $\nabla A$ field between connected dipoles. (b) Returning to the $\vec{B}$ field, a respective connection region (yellow) is found between the two dipoles.

and consecutively into the configuration shown in Figure 5.4b). The next step is to switch from the $\vec{B}$ field to the dual $\nabla A$ field, as shown in Figure 5.4c). As discussed in Section 5.1.2, the two centers of a dipole are now replaced by a source and a sink. In addition, the two saddles located between the dipoles are rotated (and possibly deformed).

The next step is to consider the field lines of the $\nabla A$ field, which are shown in Figure 5.5a). Please note that the source and sink pairs represent again dipoles in the $\nabla A$ field. In general, i.e., in non-degenerate cases, a Morse-Smale cell forms in $\nabla A$ between the original dipoles in $\vec{B}$. The final step in this reasoning switches back to the $\vec{B}$ field, shown in Figure 5.5b). Now, the Morse-Smale cell of the dual $\nabla A$ field converts to the previously mentioned connection region. Hence, there is always a connection region between two dipoles that share a magnetic field line. Also, such a connection region will always be defined by the two dipoles and consequently by the two corresponding saddles.

5.2.2 Connectrices

Once a connection region is found, the magnetic flux can be visualized through this region in a topological manner, i.e., its topology (connectivity of dipoles) is preserved and its morphology is neglected. To do this, a mathematical definition of connectrices is provided. To start, the set of all dipoles $D$ is declared. A field line connecting to dipole $d_1 \in D$ either extends to infinity or connects to another dipole $d_2 \in D$, where $d_1 = d_2$ is allowed. Please note that a field line is stopped when it reaches a dipole. To proceed, $F_{d_1,d_2}$ is defined as the set of all field lines that connect to both $d_1$ and $d_2$ and $d_1 \neq d_2$. $F_{d_1,d_2}$ is the connection region. Now two field lines $f_i, f_j$ are defined to be equivalent if $f_i, f_j \in F_{d_1,d_2}$. 
Hence, any one field line $f_i \in F_{d_1,d_2}$ is topologically equivalent to $F_{d_1,d_2}$. Such a field line $f$ is called the dipole connectrix, or shorter, connectrix of the connection region $F_{d_1,d_2}$.

The method that computes the visual representation of a connectrix is shown in Section 5.3.2. The magnetic flux within the connection region is visualized by the thickness of the line representation of the connectrix, and the orientation of the flux using a transition from red (north) to blue (south). The magnetic flux of such a region could be computed by integrating the magnetic flux density along any curve that connects the separatrices of the two respective saddles and stays within the connection region. As will be shown in Section 5.3.3, such an integration process can be avoided and a much simpler approach can be used to compute the magnetic flux.

### 5.2.3 Complete Topology

The presented method also supports domains with boundaries, e.g., limited rectangular domains that might serve as a window or come from simulations with boundaries. Within such a domain, saddle-type critical points are found using a sampling grid. The critical point extraction is the only step in this approach where a discretized version of the field is required. The remaining process can act solely on the original data given as a set of point dipoles. If a saddle is located outside of the sampling grid, it cannot be taken into account for detecting connection regions. Therefore, a connection region that would be defined by such a saddle is not found, which would lead to missing connectrices.

This problem is addressed by introducing boundary flux indicators that visualize where, and how much, flux crosses the borders of the data set or the boundary of the sampling grid. Boundary flux indicators are constructed as follows: to begin, boundary switch points [WTHS04] of $\vec{B}$ are located, i.e., points where $\vec{B}$ is tangential to the boundary. In this case, these points represent extrema of $A$ along the boundary, which allows a simplified extraction by scanning the vector potential $A$ of the outermost cells of the sampling grid. The boundary switch points serve two purposes: First, they adopt the role of saddles when extracting connection regions—they are starting points for “virtual separatrices” in $\vec{B}$, which are constructed only to delimit a connection region that would otherwise be left open (e.g., connectrix 2 in Figure 5.6 would be missing). Second, they form a set $S_{\text{bsp}}$, whereas end points of separatrices in $\vec{B}$ that leave the sampling grid form a set $S_s$. These two sets form $S = S_s \cup S_{\text{bsp}}$. For two adjacent elements $e_{1,2} \in S$, a border segment is created, i.e., a segment on the boundary of the sampling grid delimited by $e_1$ and $e_2$. For each border segment, semi-connectrices are constructed—connectrices that are attached only to one dipole and end at their corresponding border segment. Semi-connectrix
5.3. Connectrix Algorithm

The algorithm that constructs connectrices is split in two parts. The first step, presented in Section 5.3.1, finds all connection regions in the data set. In the second step, one connectrix is created for each connection region (Section 5.3.2).

5.3.1 Finding Connection Regions

To find connection regions, all saddles have to be found as well as corresponding separatrices in the magnetic field. Separatrices are started at saddles and
Figure 5.7: Identifying connection regions: start with neighboring separatrices (1a) and (1b) of the left dipole along with saddles at (2a) and (2b). To identify the second dipole, follow a field line $f_B$ in $\vec{B}$ starting at the left dipole.

traced until they end at a dipole or leave the sampling grid. Depending on which pole of a dipole is hit by the separatrix, the separatrix is defined to be of type “north” or “south”. Once all separatrices of the $\vec{B}$ field are created, the necessary data is at hand to identify connection regions. These regions are defined by two dipoles, two corresponding saddles, and are bounded by the separatrices that connect them as illustrated in Figure 5.3b). Separatrices ending at the first dipole have to be of opposite type (either “north” or “south”) than the two separatrices ending at the second dipole. A connectrix cannot exist if the dipoles are facing each other in a topologically equivalent configuration (as shown in Figure 5.3a)). In addition, self-connectrices are not allowed, i.e., connectrices that start and end at the same dipole. This all leads to the algorithm described below.

As a preliminary step, all separatrices at a dipole that are of the same type (either “north” or “south”) are sorted according to the potential $A$. The potential of a separatrix is obtained from the corresponding saddle position. This sorting allows one to choose “neighboring” separatrices with respect to the potential, which is essential to the algorithm.

1. For each dipole $d$, consider only separatrices of type “north”, which form the set $S_d$.
2. Follow all pairs of neighboring separatrices $s_{1,2} \in S_d$; check that $s_{1,2}$ connect to different saddles.
3. Start a field line $f_B$ in $\vec{B}$ at $d$ between $s_{1,2}$ and trace away from $d$ (backwards if necessary) until a dipole $d'$ is reached.
4. A connection region is found iff $d' \neq d$.
5. Process separatrices of type “south” of $d$ accordingly.
5.3. CONNECTRIX ALGORITHM

Figure 5.8: Constructing a connectrix: field line tracing is performed along $\nabla A$ into the same direction as the halfway vector $\vec{h}$ at ① until $a_{avg}$ is found in $A$ at ②. Tracing forwards and backwards in $\vec{B}$ constructs the connectrix for this connection region.

This algorithm is depicted in Figure 5.7 with a representative example scene. Please note that the field line $f_B$ is already a valid connectrix according to the previously given definition. However, in order to create a good representation of the connectrix, additional steps are performed as described in the following section.

5.3.2 Constructing Connectrices

The second part of this method constructs a representative connectrix for each connection region. The main task is to find an appropriate seed point within this region that generates the connectrix by field line integration in $\vec{B}$ in both directions until both dipoles are reached. The idea is to choose the field line that corresponds to the mean value of the potential $A$ of the respective connection region—in the sense of a “mean” position. The following approach was implemented to find a seed point (a special case of numerical root finding) and to construct a connectrix, as illustrated in Figure 5.8.

1. Evaluate $A$ at the two saddle positions and compute the average $a_{avg}$. This represents the mean potential within the connection region.
2. Choose one of the two saddles arbitrarily and perform field line tracing in the $\nabla A$ field, heading into the connection region. Please note that this is not immediately possible, since $\nabla A = 0$ at a saddle position. Therefore, follow the half-way vector of the two separatrices that meet at the chosen saddle position. A small step is used, typically $1/10^{th}$ of the cell size of the sampling grid. Perform field line tracing in $\nabla A$ to advance
inside the connection region.

3. At each step of the tracing process in the $\nabla A$ field, access the value of $A$ at the current tracing location. Once the value of the $A$ field crosses the average value $a_{\text{avg}}$, the seed point for the connectrix is found by bisection.

4. From that point, perform forward and backward integration in $\vec{B}$ until a dipole is reached, yielding the connectrix.

In step 2, one can choose between the two saddles of the connection region as a starting point for the tracing process in the $\nabla A$ field. The resulting seed points for the connectrix are different depending on the chosen saddle; however, both seed points will be located on the same field line of the $\vec{B}$ field due to the duality of the $\nabla A$ and $\vec{B}$ fields and because of a unique $a_{\text{avg}}$. This is due to the fact that $A$ is a stream function for $\vec{B}$, i.e., contours in $A$ represent field lines in $\vec{B}$. Therefore, the final visualization for the connectrix is independent of the chosen saddle point.

5.3.3 Visualizing Magnetic Flux

In addition to encoding the topology of the flux between dipoles, the flux magnitude is visualized by varying the line width of the connectrices. This emphasizes connectrices of dipoles that create a strong magnetic flux. The magnetic flux found in a connection region is defined as the result of integrating the magnetic flux density $\vec{B}$ along any curve within this region that connects the separatrices of the two saddles. However, this integration can be avoided: it is only necessary to evaluate $A$ at the two saddle locations. The difference of these two potentials equals the magnetic flux through the connection region. Since the potential $A$ works analogous to a stream function, integration of flux over a curve can be replaced by a difference in the potentials of its end points.

5.3.4 Implementation

The implementation uses a uniform grid to resample the $A$ and $\vec{B}$ field defined by the given set of dipoles. Critical points are detected according to the approach described by Helman and Hesselink [HH91]. Both separatrices and connectrices are constructed using field line integration that is performed using an adaptive step-size, fourth-order Runge-Kutta algorithm. The integration process is stopped when entering a cell that contains a dipole or when leaving the sample grid. Therefore, the sampling grid is chosen in such a way so that cells are small enough to contain at most one dipole. The prototype was implemented in C# using the XNA library on a system with an Intel Core i7 CPU running at 3.4 GHz and an NVIDIA GeForce 580 GPU. Magnetic field lines are visualized with an HLSL shader implementation of LIC [CL93].
5.4 Application

To demonstrate the usefulness of this method, it was applied to simulation and visualization problems from the domain of soft-matter sciences [KCH08]. All data sets are initially represented as collections of dipoles with varying position, orientation, and magnitude of magnetic moment. To improve the visibility of topological structures, the visualization of the magnetic field $\vec{B}$ is omitted.

5.4.1 Monolayer

In Figure 5.9, a snapshot is presented which was obtained for a ferroparticle monolayer consisting of 128 dipoles. In this case, the interparticle interaction is strong enough for the magnetic particles to form various small clusters. The prevalent topological structures in the visualization of the magnetic flux are chains and rings, readily classifying the clusters into these two groups (see, e.g., center of Figure 5.9). Moreover, this visualization gives quantitative insight in their magnetic properties, e.g., distinguished chains and loops in the visualization represent clusters with strong magnetic interaction. In this way, the technique provides a powerful approach to the analysis of ferroparticle monolayers, e.g., to differentiate between randomly aligned particles and stable clusters. This is of particular importance for understanding the microstructure of ferrofluids in confinement.

5.4.2 Ferrogel

With the series of images shown in Figure 5.10 and Figure 5.11, the time evolution spanning 1 000 time steps of a magnetic gel simulation is presented. The gel is constructed with an initially square lattice whose connectivity is indicated by black lines. The sample is composed of 4 104 dipoles and additional non-magnetic particles visualized as black dots. In this simulation, the gel is exposed to a strong external magnetic field that is aligned from left to right in the images. As the properties of the gel primarily depend on dipole–dipole interaction, this external field is omitted in the visualization.

In the first snapshot, the signature of a square lattice is still visible. Here, lattice segments that are aligned with the external field are part of chains of pronounced magnetic flux. In contrast, particles on segments perpendicular to the field are rarely part of a common connectrix. They are rather part of long, weak connectrices between neighboring lattice segments, or part of weak loop-like structures. The overall shape of the magnetic ferrogel still reflects the initial lattice because the strong horizontal chains have a repulsive effect on each other. The formation of horizontal connectrix chains proceeds during the later time steps. As these structures grow in strength of magnetic flux, the ferrogel is lo-
Figure 5.9: Monolayer data set. Ring and chain structures can be readily identified in the flux topology visualization. Computing the flux topology takes 14.74 seconds.

cally contracted, displacing the non-magnetic particles into saw-tooth shaped configurations. At the same time, the overall shape of the gel sample undergoes a transition into a diamond-shaped configuration, as the dipoles with energetically less advantageous configuration rearrange. In the final snapshot, the majority of the lattice segments assume a diagonal orientation, with long chains of dipoles that are co-aligned with the external magnetic field.
Figure 5.10: From left to right: time steps 1, 500, and 900 of a ferrogel simulation. Upper row: whole data set; lower row: zoomed-in views of the regions denoted by black rectangles. For all images, the same scaling for the flux magnitude was used. Non-magnetic particles are shown as black dots. Computing the flux topology takes 256, 270, and 264 seconds, respectively. The black rectangle in the lower right image refers to Figure 5.11.

Figure 5.11: This view corresponds to the black rectangle in the lower right image of Figure 5.10. Flux is scaled down by one order of magnitude to reduce visual clutter.
Part III

Data-Domain Visualization
Continuous Scatterplots

Feature-based vector field visualization techniques described in the previous part of this thesis can be a very effective means to analyze two- or three-dimensional fields. However, today’s simulations not only produce large data sets, but also complex ones—usually including several data dimensions. In such cases, the visualization methods presented in previous chapters might not be sufficient to effectively analyze those data sets. Having only those methods at hand, correlations between data dimensions may remain hidden, since only a few dimensions can be evaluated at the same time.

In this chapter, the undisputed visualization power of scatterplots is utilized and extended to display the type of data typically generated in computational sciences. This method fits in the recent trend of combining statistical visualization methods like scatterplots with scientific visualization methods like volume or flow visualization.

The term continuous scatterplot was defined in our publication [BW08b] as follows. First, input data is no longer a collection of discrete data points but a field of data values defined on a continuous domain. Typically, the domain is 2D or 3D and has intrinsic spatial embedding; the dimensionality of the domain increases by one if time dependency is included. Data representation often relies on a grid with respective interpolation or approximation schemes. Please note that the data field defined on the continuous domain is not necessarily continuous, i.e., even non-continuous functions can be visualized. The second aspect of a continuous scatterplot is that the output is continuous: instead of a collection of discrete points, a continuous density is drawn on the diagram, providing a continuous frequency description of two data dimensions.

This model of continuous scatterplots and respective computations is generic in the sense that both the spatial domain of the input field and the dimension of the scatterplot can be chosen freely. In practice, the dimensionality of the input-field domain is determined by the data input, which mostly is restricted to dimensions 2–4. From a perceptual point of view, useful output dimensionalities are 1 (i.e., continuous histogram) and 2 (i.e., scatterplot in its original sense). Although there is previous work on 3D discrete scatterplots, their perceptual effectiveness is unclear. To visualize higher-dimensional data,
2D continuous scatterplots can be combined to respective scatterplot matrices. Alternatively, higher-dimensional data can be visualized with parallel coordinates [ID90]. For the case of scientific visualization, parallel coordinates can be computed in a continuous way as well—based on the ideas presented in this section. These continuous parallel coordinates were introduced by Heinrich and Weiskopf [HW09].

6.1 Mathematical Model

This section presents the mathematical model of generic continuous scatterplots. First, required terminology is defined before the generic model is described. Later parts of this section derive specific results for special cases like 2D scatterplots or 1D histograms. In this context, these continuous versions are compared with their well-known discrete counterparts.

6.1.1 Generic Model

The computation of continuous scatterplots needs two different domains: the \( n \)-dimensional domain on which the input field is defined, and the \( m \)-dimensional domain of the scatterplot onto which the input field is mapped. The first one is denoted as spatial domain because it typically describes 2D or 3D spatial positions. Despite this terminology, the spatial domain may also include a time dimension; in fact, any continuous field domain is supported. The second kind of domain is denoted as data domain because it represents the multi-attribute data values of the input field. Typical examples are \( m = 1 \), which corresponds to a histogram for one data component, or \( m = 2 \), which corresponds to a 2D scatterplot. The input field to be visualized can be represented mathematically by a map from the spatial domain to the data domain: \( \tau : \mathbb{R}^n \rightarrow \mathbb{R}^m \). The map \( \tau \) can represent all typical scientific data, including 3D scalar fields, 3D vector fields, or multi-attribute fields. The problem of constructing a continuous scatterplot can then be formulated as finding a density function \( \sigma \) defined on the data domain:

\[
\sigma : \mathbb{R}^m \rightarrow \mathbb{R}, \quad \xi \mapsto \sigma(\xi),
\]

which represents continuous frequency and depends on \( \tau \). In fact, the mathematical basis of the continuous scatterplot is an operator that maps the function \( \tau \) to the function \( \sigma \).

To construct this operator, a continuous description is derived by starting from well-known discrete scatterplots and considering the limit process for infinitely dense data points. This approach is similar to deriving continuum mechanics from systems of discrete mass points (see, e.g., in Goldstein [Gol80, Ch. 12]). Figure 6.1 illustrates the discrete particle model for the example...
$n = m = 2$. The derivation is based on two assumptions: First, points in the spatial domain are given according to some kind of density description (typically, uniform density), and second, the mapping $\tau$ does not change the number of points. The second assumption is identical to mass conservation if mass points are considered. The limit process for infinitely dense particles leads to a replacement of particle mass by mass density.

According to the first assumption, the mass (i.e., sampling) density $s$ is known in the spatial domain, with $s : \mathbb{R}^n \to \mathbb{R}, x \mapsto s(x)$. The mass $M$ of a covered volume $V \subset \mathbb{R}^n$ is $M = \int_V s(x) d^n x$. Similarly, the density $\sigma$ in the data domain can be integrated to compute the mass of the covered volume $\Phi \subset \mathbb{R}^m$. Please note that this notation uses Latin characters for quantities related to the spatial domain and Greek characters for quantities related to the data domain; lower case letters denote scalar or vector values, uppercase letters denote volumes.

If $V$ and $\Phi$ are related by $\tau(V) = \Phi$, mass conservation under the transformation $\tau$ implies

$$
M = \int_V s(x) d^n x = \int_{\Phi=\tau(V)} \sigma(\xi) d^m \xi,
$$

which determines the unknown density function $\sigma$ for a given input density $s$ because Equation 6.2 has to hold for any volume $V$ in the spatial domain. Rewriting Equation 6.2 leads to the alternative formulation

$$
\int_{\Phi} \sigma(\xi) d^m \xi = \int_{\tau^{-1}(\Phi)} s(x) d^n x,
$$

which equally determines $\sigma$ because Equation 6.3 has to hold for any volume $\Phi$ in the data domain. The inverse map $\tau^{-1}(\Phi)$ is well defined even if $\tau$ is
not invertible because, here, this is done on maps of sets, not of single function values.

Please note that \( \sigma \) is only indirectly defined in Equations 6.2 or 6.3 via the effect of integration. For the generic case of scatterplot computation, this indirect definition of \( \sigma \) is required so that not only regular functions are supported but also distributions (generalized functions) like Dirac \( \delta \) distributions. An introduction to distributions and functional analysis is given in the textbook by Griffel [Gri02]. In fact, the earlier definition of \( \sigma \) as a regular function in Equation 6.1 is extended to allow for distributions as well. Later, it will be shown that \( \delta \) distributions are useful to build a relationship between continuous and discrete scatterplots, and to allow for scatterplots and histograms of (partly) constant functions. The indirect formulation of Equations 6.2 and 6.3 can be rewritten to directly compute \( \sigma \) for some special, yet important cases.

6.1.2 Case \( m = n \)

To begin, the special case of equal dimension \( m = n \) is considered in order to compute \( \sigma \). Since \( \sigma \) is directly based on \( \tau \), this discussion is split into several parts that cover different possible subcases depending on the properties of \( \tau \). First, it is assumed that \( \tau \) is differentiable. Here, both subcases are considered, where \( \tau \) is a diffeomorphism or no diffeomorphism. Later, this discussion is extended to non-differentiable \( \tau \).

Assuming that \( \tau \) is a diffeomorphism from \( \mathbb{R}^n \to \mathbb{R}^m \), the integration variable \( \xi \) can be substituted by \( x \) in Equation 6.2 according to the transformation theorem for integrals:

\[
\int_{\tau(V)} \sigma(\xi) \, d^{m=n} \xi = \int_V \sigma(\tau(x)) |\det(D\tau)(x)| \, d^n x = \int_V s(x) \, d^n x , \tag{6.4}
\]

where \( D\tau \) denotes the derivative of the map \( \tau \), i.e., the \( n \times n \) Jacobi matrix. Note that the determinant is a volume measure, representing the volume spanned by the partial derivatives of \( \tau \). Since the second equality of Equation 6.4 has to hold for any \( V \subset \mathbb{R}^n \), the integrands need to be equal, which leads to

\[
\sigma(\xi) = \frac{s(\tau^{-1}(\xi))}{|\det(D\tau)(\tau^{-1}(\xi))|} . \tag{6.5}
\]

Figure 6.2 illustrates the density mapping according to Equation 6.5 for a 1D example \( m = n = 1 \). Geometrically, the density ratio \( \sigma/s \) in a small neighborhood is given by the ratio of the covered lengths, \( V/\tau(V) \), which in turn is computed by the reciprocal of the slope of \( \tau \). Please note that a similar approach is followed for the inversion of cumulative distribution functions in order to map probability distributions or derive histogram equalization.
Figure 6.2: Mapping of intervals from the spatial domain $x$ to the data domain $\xi$. The ratio of the original and mapped intervals, which is given by the reciprocal of the slope of $\tau$ (slope is indicated by the straight line), is a measure for the change of density from the spatial to the data domain.

Figure 6.3: Continuous histogram (left) of a piecewise constant function (right). The histogram is rotated by 90 degrees to visualize data correspondence between the function plot and the histogram. $\delta$ peaks are indicated by arrows; the length of a $\delta$ peak corresponds to the width of the respective interval in the function plot.
When \( \det(D\tau) = 0 \), then \( \tau \) is not a diffeomorphism and Equation 6.5 cannot be used to define \( \sigma \). Such a case can occur, e.g., when \( \tau \) contains regions of constant value, extremal points, or other points with vanishing \( \det(D\tau) \). To begin, the case of constant value is considered. Figure 6.3 illustrates the example of a piecewise constant function for dimensionality \( m = n = 1 \). In the spatial domain, a piecewise constant function can be modeled as \( \tau(x) = \sum_i \tau_i \chi_i(x) \), with the characteristic function

\[
\chi_i(x) = \begin{cases} 
1 & \text{if } x \in \text{Cell}(i) \\
0 & \text{else} 
\end{cases}
\]

It is assumed that the spatial domain is partitioned into cells. Then, \( \tau \) has constant value \( \tau_i \) within cell \( i \). Assuming constant density in the spatial domain, \( s = 1 \), the density in the data domain is:

\[
\sigma(\xi) = \sum_i \delta(\xi - \tau_i) \text{Size}(\text{Cell}(i)), 
\]

with Dirac delta \( \delta \). The correctness of Equation 6.6 can be verified by plugging Equation 6.6 into the mass-conservation Equation 6.3:

\[
\int_{\Phi} \sigma(\xi) \, d^n\xi = \sum_i \int_{\Phi} \delta(\xi - \tau_i) \text{Size}(\text{Cell}(i)) \, d^n\xi = \sum_{\text{for all } i \text{ where } \tau_i \in \Phi} \text{Size}(\text{Cell}(i)) = \int_{\tau^{-1}(\Phi)} 1 \, d^n x.
\]

Since \( s = 1 \), Equation 6.3 is correctly met. Note that \( s = 1 \) was chosen to simplify notation; an analogous proof would work for any choice of \( s \), assuming that Equation 6.6 includes \( s \).

A different approach is taken for the other problematic case where \( \det(D\tau) = 0 \) at isolated parts. To be more precise, in this case, the determinant of the derivative vanishes at isolated null-sets (null-sets with respect to integration in the spatial domain). Here, a two-step approach is performed. First, regions where \( \det(D\tau)(x) = 0 \) are identified. These regions are denoted \( \Gamma = \{ x \in \mathbb{R}^n | \det(D\tau)(x) = 0 \} \). Second, the null-set \( \Gamma \) and its image under the map \( \tau \) are removed from the computation of density in Equation 6.5, i.e., \( \sigma \) is not defined at those locations. Since integration over null-sets always yields 0, those isolated locations can be removed without affecting the conservation-of-mass model.

The same approach is taken if the underlying data set is not continuous, i.e., \( \tau \) is not differentiable (e.g., between cells). This results again in null-sets, which can be removed from the computation of the density. Finally, in theory, \( \tau \) might be non-diffeomorphic on non-nullsets. However, this case is not considered, since any realistic data set will meet the requirement that \( \tau \) is non-differentiable or has vanishing \( \det(D\tau) \) at the most at isolated null-sets. In
Figure 6.4: General example of a continuous histogram (left) for the function $\tau$ (right). The spatial domain of $\tau$ is partitioned into intervals with non-vanishing derivative; intervals are color-coded (right image). Respective densities $\sigma$ are shown in the left image (same color coding). The black dashed curve shows the sum of the intermediate densities $\sigma$, which should be overlaid with the $\delta$ peak (green) corresponding to the constant part of $\tau$.

In particular, any grid-based data set with piecewise cell-oriented interpolation (which is most common in scientific visualization), meets this requirement.

To summarize the special case of $m = n$, $\sigma$ can be computed by the following schematic algorithm.

1. For an extended volume (i.e., not a null-set) where $\tau$ is constant, a $\delta$ peak is associated that is weighted by the size of the volume.

2. Isolated null-sets of non-diffeomorphism are identified and used to partition $V$ into volumes where $\det(D\tau) \neq 0$. The null-sets themselves are removed from the computations, and intermediate $\sigma$ are computed for each element of the partition.

3. The intermediate results from steps one and two are added. This is valid because the defining Equation 6.3 is linear.

Figure 6.4 illustrates the schematic algorithm for an example of a continuous histogram with $m = n = 1$ and with constant density $s$ in the spatial domain.

This construction of continuous histograms allows the reproduction of discrete histograms. Please note that discrete input data has no attached spatial domain in traditional statistical analysis. Discrete data points are identified with arbitrarily positioned cells in the spatial domain; all cells are chosen with equal size. Then, the continuous histogram consists of $\delta$ peaks at the data point values (see construction in Equation 6.6), i.e., $\delta$ peaks in the continuous histogram correspond to entries in the discrete histogram. Typical bucketing of discrete histograms can be achieved by integrating the continuous histogram with box functions that represent the buckets.
6.1. MATHEMATICAL MODEL

Figure 6.5: Projection issue for the case \( m < n \). A small interval (blue) in the data domain around \( \zeta_0 \) (right) corresponds to a bent strip (outlined in red) around the isocontour (solid center line) with isovalue \( \zeta_0 \) in the spatial domain (left). The bent strip is the union of the normal spaces for all points on the isocontour. The blue arrows in the left image mark the normal space at point \( \mathbf{x} \).

6.1.3 Case \( m < n \)

Another common case is \( m < n \), which arises for example for a 2D scatterplot that shows two scalar attributes of a 3D volume data set. Here, the dimension of the spatial domain is reduced when \( \tau \) maps to the data domain. Therefore, the transformation theorem for integration, which was used in the previous subsection, does not apply. In particular, \( \det(D\tau) \) does not exist, which means that the same approach cannot be used as in Equation 6.4.

Figure 6.5 illustrates the geometry of the underlying problem for the case \( n = 2 \) and \( m = 1 \): a single point in the data domain corresponds to an infinite set of points in the spatial domain. In this example, the infinite set is the isoline within the spatial domain that corresponds to the isovalue in the data domain. This imbalance in dimensionality does no longer permit the transformation of differentials as in the previous subsection.

As in the generic discussion of Section 6.1.1, two related volumes are considered in the data and spatial domains: \( \Phi \subset \mathbb{R}^m \) and \( V = \tau^{-1}(\Phi) \subset \mathbb{R}^n \). To be more specific, only a small neighborhood \( \Phi \) around a point \( \zeta_0 \in \mathbb{R}^m \) is considered. To overcome the dimensionality problem, \( V \) is split into two parts: first, the inverse image of the point \( \zeta_0 \) (i.e., a generalized isocontour), and second, the perpendicular space around the inverse image of \( \mathbf{x}_0 \). \( \tau_{\text{normal}}^{-1}(p) \) is denoted as the space that is normal to \( \tau^{-1}(\zeta_0) \) at a point \( p \in \tau^{-1}(\zeta_0) \) and that is also contained within \( \tau^{-1}(\Phi) \). Here, \( \tau \) is assumed to be a smooth non-constant function, and therefore, the isocontour is smooth as well and the normal space is well defined. If \( \tau \) is not smooth, then the spatial domain is split into piecewise smooth regions and the method is applied in a piecewise manner (this
approach does not apply to completely non-smooth functions). If \( \tau \) is (partly) constant, such a constant volume is separated out of the computation similar to the discussion in Section 6.1.2, leading to \( \delta \) contributions. In the regular case, by construction, the normal space has the same dimension \( m \) as the data domain, whereas the isocontour has dimension \( n - m \). Figure 6.5 illustrates the isocontour and the normal space. Now, the integration in Equation 6.3 can be split into isocontour and normal parts, similar to the approach of the co-area computation [Fed96]:

\[
\int \sigma (\xi) \, d^m \xi = \int s(\chi) \, d^n \chi = \int \left( \int s(\hat{x}) \, d^m \hat{x} \right) \, d^{(n-m)} \hat{\chi}. \tag{6.7}
\]

Within the normal space, the same approach can be used as in Section 6.1.2 because dimensionalities coincide and a computation based on derivatives is possible. Denoting the intermediate contribution to the density at points \( \hat{x} \) by \( \tilde{\sigma}_{\hat{x}} \), the inner integration is re-labeled in the right-hand side of Equation 6.7 according to:

\[
\int_{\tau^{-1}_{\text{normal}}(\hat{x})} s(\hat{x}) \, d^m \hat{x} = \int_{\Phi} \tilde{\sigma}_{\hat{x}}(\tilde{\xi}) \, d^m \tilde{\xi}.
\]

Assuming a diffeomorphism in the normal space, Equation 6.5 can be adopted to obtain:

\[
\tilde{\sigma}_{\hat{x}}(\tilde{\xi}) = \frac{s(\hat{x})}{|\text{Vol}(D\tau)(\hat{x})|},
\]

where \( \hat{x} \) and \( \tilde{\xi} \) are related by \( \tau(\hat{x}) = \tilde{\xi} \). Here, \( |\text{Vol}(D\tau)| \) replaces the determinant \( |\text{det}(D\tau)| \) in Equation 6.5. The volume measure \( |\text{Vol}(D\tau)| \) is defined as the volume spanned by the partial derivatives of \( \tau \) restricted to variations of parameters in the normal space \( \tau^{-1}_{\text{normal}}(\hat{x}) \). Figure 6.5 (left) illustrates the reciprocal volume measure \( 1/|\text{Vol}(D\tau)| \) within the spatial domain. The volume measure is explicitly computed in Section 6.1.4 for the special case \( m = 2, n = 3 \), and below in this subsection for the case \( m = 1, n = 3 \). For the final overall density, \( \tilde{\sigma}_{\hat{x}} \) is integrated along the complete isocontour:

\[
\sigma(\xi_0) = \int_{\tau^{-1}(\xi_0)} \frac{s(\hat{x})}{|\text{Vol}(D\tau)(\hat{x})|} \, d^{(n-m)} \hat{\chi}, \tag{6.8}
\]

which completes the generic discussion of the case \( m < n \).

Here, a comparison with the work on isosurface statistics by Carr et al. [CDD06] is included because histograms similar to Carr et al. can be produced by using \( m = 1 \) and \( n = 3 \). The work from Carr et al. focuses on analyzing isosurface behavior, whereas continuous scatterplots target visual data analysis in the full domain. This is the reason for different definitions of histograms. Carr et al. define their histogram as the volume of the inverse
image, i.e., the area of the respective isosurface. Using the notation introduced in Section 6.1.1, their histogram would read $\sigma(\xi_0) = \int_{\tau^{-1}(\xi_0)} 1 \, d^{(n-m)}\hat{x}$ instead of the computation in Equation 6.8. Both approaches use the size of the isosurface (here, via integration over $\tau^{-1}(\xi_0)$).

One (minor) difference is that continuous scatterplots support a space-variant input density $s$. The major difference, however, is that in this thesis $1/|\text{Vol}(D\tau)(\hat{x})|$ is taken into account, whereas Carr et al. do not. For $m = 1$ and $n = 3$, $|\text{Vol}(D\tau)(\hat{x})|$ is the magnitude of the gradient at $\hat{x}$. Put differently, the neighborhood of values in the data domain and how they are affected by derivatives of $\tau$ is considered by continuous scatterplots, whereas Carr et al. use a point mapping from the data domain to the spatial domain. In this sense, the approach presented there is related to, but no identical with, Legendre transformations that take into account derivatives (see the geometric interpretation of the Legendre transformation in [CH62, pp. 32–39] and its use for Hamiltonian and Hamilton-Jacobi mechanics described by Goldstein [Gol80]). Therefore, this definition of continuous scatterplots takes into account the behavior of $\tau$ in its full neighborhood; only in this way, it is possible to represent the transformation of sampling density. Scheidegger et al. [SSD+08] have independently derived the same weighting factor of $1/|\nabla\tau(\hat{x})|$ when revisiting Carr et al. [CDD06]; i.e., for the case of isosurface histograms, both approaches lead to the same result.

A formal, mathematical advantage of this model in Equations 6.2 and 6.3 is its generic applicability to any dimension of the spatial and data domains. In particular, densities $s$ and $\sigma$ are automatically adapted to the respective integration dimensions $n$ and $m$—if actual physical units of mass density were used, they would have SI units (Système International d’unités) [kg/m$^n$] and [kg/m$^m$], respectively. Moreover, even (partly or completely) constant functions $\tau$ are supported in this model; in this case, $\delta$ distributions occur in $\sigma$. In contrast, the formal derivation of Equation (5) by Carr et al. [CDD06] is problematic because the integration measure in their expression $\int_{f^{-1}(h)} 1 \, dx$ is not explicitly specified but, from the context of their Equation (3), should be $d$-dimensional. Typically, $f^{-1}(h)$ is an isosurface, which is a null-set, and therefore the integral vanishes. Alternatively, the integration measure could be adapted to the dimensionality of $f^{-1}(h)$. Then, their integration in Equation (5) would be fine as long as the dimensionality of $f^{-1}(h)$ is constant, i.e., problems would occur when the function is partly constant, leading to a mix of 2D isosurfaces and 3D isovolumes. Scheidegger et al. [SSD+08] resolve these problems by restricting the computation of isosurface histograms to 3D scalar fields with non-vanishing gradient, implying integration on 2D isosurfaces. The above issues with continuous distributions demonstrate the usefulness of the presented density-based definition of generic continuous scatterplots.
6.1.4 Case $m = 2$ and $n = 3$

This subsection addresses a special case of the above subsection: $m = 2$ and $n = 3$. This case is important in practical applications because typical data is given on a 3D spatial domain and analyzed by 2D scatterplots. The other important practical application is the computation of 1D histograms for data on 3D spatial domains; this application was covered at the end of the previous subsection.

For simplicity of discussion, it is assumed that $\tau$ is a smooth non-constant function so that for any choice of coordinates in the data domain, $\xi = (\xi_1, \xi_2)$, two smooth isosurfaces corresponding to $\hat{\xi}_1$ and $\hat{\xi}_2$ are obtained (for (partly) constant or non-smooth $\tau$, a special treatment similar to Section 6.1.2 is required). Furthermore, $\tau$ is assumed to be non-degenerated so that the intersection of the two isosurfaces yields 1D curves. Figure 6.6 illustrates the geometry of the scenario. Here, a zoomed-in view is shown; therefore, the smooth isosurfaces appear planar. In this case, Equation 6.8 reads

$$\sigma(\hat{\xi}_0) = \int_{\tau^{-1}((\hat{\xi}_1,\hat{\xi}_2))} \frac{s(\hat{x})}{|\text{Vol}(D\tau)(\hat{x})|} \, d\hat{x}, \quad (6.9)$$

where the integration is along the 1D intersection of the two isosurfaces. The 2D area $|\text{Vol}(D\tau)|$ in Equation 6.9 is spanned by the gradients $\partial \hat{\xi}_1 / \partial x$ and $\partial \hat{\xi}_2 / \partial x$. Figure 6.6 illustrates the respective reciprocal volume $1/|\text{Vol}(D\tau)|$ carved out around the two isosurfaces. By using vector computations, the vol-
ume measure is computed as the cross product of the two gradients:

\[
|\text{Vol}(D\tau)| = \left\| \frac{\partial \xi_1}{\partial x} \times \frac{\partial \xi_2}{\partial x} \right\|.
\] (6.10)

In summary, the density \(\sigma\) is obtained by integration along the 1D intersection curves of the two isosurfaces, weighted by the reciprocal of the magnitude of the cross product of the two respective gradients.

### 6.1.5 Case \(m > n\)

This subsection briefly discusses the remaining uncovered case \(m > n\) in order to complete the description of cases. From a visualization point of view, this case is not very useful because the dimensionality of the scatterplot is higher than the dimensionality of the spatial domain, which adds visual complexity instead of reducing it.

The map \(\tau\) from the spatial domain \(\mathbb{R}^n\) to the data domain \(\mathbb{R}^m\) leads to a coverage of the data domain by an \(n\)-dimensional subset. For example, a 1D spatial data set would typically result in a 1D curve within a 2D scatterplot, i.e., the support for the density \(\sigma\) would be that curve. The density distribution \(\sigma\) can be computed by applying the mapping from Section 6.1.2 within the support of \(\sigma\), considering this support as an \(n\)-D manifold, and by allowing for \(\delta\) distributions to obtain finite values when integrating over null-sets in the data domain.

### 6.2 Scatterplot Algorithm for Tetrahedral Meshes

In practice, the most important examples of continuous scatterplots are the cases \(n = 3, m = 1\) (i.e., continuous histogram) and \(n = 3, m = 2\) (continuous 2D scatterplot). Both cases work on a 3D spatial domain, which is common for scientific data. Even for time-dependent 4D data, visualization is often restricted to showing 3D time slices. The case of continuous histograms can be implemented similar to Carr et al. [CDD06]; the only difference is the additional weighting by the reciprocal of the gradient magnitude and by the original density \(s\). The extension of Scheidegger et al. [SSD+08] already includes the weighting by the reciprocal of the gradient magnitude and, thus, their implementation could be directly adopted. Therefore, this section focuses on the other case—the construction of continuous 2D scatterplots.

According to Equations 6.9 and 6.10, the intersection curve of two isosurfaces as well as the two gradients along the intersection curve need to be determined and combined by integration along the curve. The result of this computation depends on the functional behavior of the data field \(\tau\). Typically, volumetric
Figure 6.7: This illustration shows how the distance is measured that is used to determine the density. At point $P$, a depth value that corresponds to the density must be computed. This is done by calculating the distance between $P$ and $P'$ in the spatial domain of the tetrahedron. The point $P'$ is the intersection point between the face opposite to $P$ and the $(\xi_1, \xi_2)$ isoline through $P$. (Please note that $P'$ does not coincide with a vertex, except for degenerated cases.)

data is given on a grid, and $\tau$ is reconstructed by piecewise cell-based interpolation within the grid. The focus lies on tetrahedral grids because they are naturally equipped with linear (barycentric) interpolation.

Since the overall density $\sigma$ is based on the linear model of Equations 6.2 and 6.3, $\sigma$ can be constructed by linear superposition of the contributions from tetrahedral cells. Therefore, the remaining question is how to compute $\sigma$ for a single tetrahedron. Here, the linearity of barycentric interpolation simplifies the computation substantially because of the following reasons: first, isosurfaces within a tetrahedron are planes. Therefore, the intersection between two isosurfaces is a straight line (in the non-degenerate case). Second, the gradient within the cell is constant. Thus, the volume measure from Equation 6.10 is constant as well. In conclusion, $\sigma$ is obtained by computing the length of the intersection of the two isosurfaces and dividing that value by the constant volume measure. Here, a constant density $s$ is assumed in the spatial domain.

The projected tetrahedra algorithm by Shirley and Tuchman [ST90] is adopted to compute the isosurface intersection. The original algorithm is designed for volume rendering of scalar fields on tetrahedral grids by projecting tetrahedra onto the image plane. However, the image plane is located within the spatial domain, whereas 2D scatterplots need to project the tetrahedra to the data domain. This projection is achieved by interpreting $(\xi_1, \xi_2)$ as coordinates for orthographic projection. The Shirley-Tuchman algorithm partitions the image footprint of the tetrahedron into a collection of up to four triangles, depending on the viewing direction. Within each triangle, parameters are interpolated linearly. The same kind of triangle partitioning is used for scatterplots. Here, the linearly interpolated parameter is the geometric depth of the tetrahedron in the spatial domain, computed at the corresponding data values $(\xi_1, \xi_2)$. Figure 6.7
illustrates the computation of depth. Linear interpolation of depth within the triangle is correct because the underlying 3D geometry is linear as well. The final $\sigma$ value is obtained by dividing depth by the volume measure from Equation 6.10.

The algorithm consists of the following steps:

1. Classify the tetrahedron based on its silhouette in the data domain. This step yields up to four triangles.
2. Attach data values $(\xi_1, \xi_2)$ as 2D geometric coordinates to the triangle vertices.
3. Determine the volume measure according to Equation 6.10.
4. Compute the Euclidean distance between frontface and backface of the tetrahedron at the vertices. Attach this distance divided by the volume measure as texture coordinate to the vertices.
5. Render triangles. The volume-weighted distance is interpolated during scanline conversion and yields the density at the current fragment. Output the result to the framebuffer.

If a data value is constant within the tetrahedron, the corresponding $\sigma$ is no longer a regular function, but a $\delta$ distribution. Such $\delta$ distributions are approximated by assigning a very large value (a constant defined within the implementation, e.g., the maximum number which can be represented as a floating point variable). In this way, even degenerate cases can be handled.

The overall density $\sigma$ is obtained by rendering all tetrahedra with additive blending. Since additive blending is commutative, triangle sorting is not necessary prior to rendering. Finally, color mapping is applied by using a color lookup table in order to generate the final result.

The above algorithm works for any tetrahedral grid. For non-simplicial grids, cells are decomposed into tetrahedra before rendering. In the common case of hexahedral cells, decomposition in five tetrahedra per cell is employed. Since any 3D grid can be approximated by triangulation, any grid-based data set can be processed.

The projected tetrahedra algorithm lends itself to acceleration by graphics hardware because rasterization of triangles and blending are efficiently supported by graphics hardware. Similarly, the 2D scatterplot algorithm can be implemented using graphics hardware. Steps 1–4 of the above algorithm are performed on the CPU, similar to traditional implementations of the Shirley-Tuchman algorithm. The results from all tetrahedra are combined by additive blending within a render-target texture. For appropriate blending quality, the format for the frame-buffer and respective textures is 32-bit floating-point. The render-target texture is used as input to another render process that applies the color table (implemented as 1D dependent texture) to $\sigma$ to generate the final image. The results of this approach are presented in Section 6.4.1.
6.3 Parallelized Computation of Continuous Scatterplots

The approach presented in Section 6.2 is implemented on the CPU, which, depending on data set size, leads to computation times well beyond interactive frame rates. The time to compute a scatterplot may take up to several minutes—too long to efficiently use continuous scatterplots in a visualization system. To overcome this issue, a modification of the original algorithm was implemented that enables the parallelized construction of continuous scatterplots on the GPU. To perfectly suit the architecture of modern GPUs, tetrahedra are processed by CUDA compute threads.

In order to increase the computational performance of continuous scatterplots, CUDA is used to execute a modified version of the original continuous scatterplot algorithm on the GPU. Directly porting the algorithm from CPU to GPU would be highly inefficient, due to the different architecture and programming model of GPUs. On the GPU, complicated data structures should be avoided and execution paths should not branch in order to achieve optimal performance. The modifications that are necessary to compute continuous scatterplots on the GPU are explained in the following paragraphs. The modified algorithm to compute continuous scatterplots is split into two parts: First, a preprocessing step is performed on the CPU. After this step, the actual computations that are necessary to render the continuous scatterplot are performed on the GPU.

To begin, it is necessary to introduce the notion of classes for tetrahedra. The original classification of tetrahedra was presented by Shirley and Tuchman [ST90]. However, for this approach, a simplification of this classification is used that was proposed by Wylie et al. [WMFC02]. In their work, only classes one and two are used for classification, since the remaining classes are only degenerate cases of these first two classes. In Figure 6.8, the two main classes are shown for reference.

To compute a continuous scatterplot, the density of each tetrahedron needs to be computed—which in turn requires a separate handling of tetrahedra based on their class. To achieve optimal performance for density computations on the GPU, the input tetrahedra are sorted in two lists which only contain tetrahedra of one of the two classes. This is necessary because on the GPU, diverging threads in a CUDA warp force the whole warp to serially execute each branch which results in significant computational overhead. Please note that sorting is a preprocessing step which is necessary only once per data set. After this step, the actual computations for the continuous scatterplot are performed. Depending on the resolution of the data set, not all tetrahedra can be uploaded to the GPU at once. If this is the case, the list of input tetrahedra is split into data subsets that fit into GPU memory. A serialized processing of these data subsets is trivial since individual tetrahedra are rendered using additive blending—therefore, the resulting projected tetrahedra are accumulated...
6.4 Application

In this section, examples of both discrete and continuous scatterplots are provided for different visualization examples. Please note that constant input density \( s = 1 \) is used for all examples. The scatterplot functionality is part of a multi-attribute visualization tool that also supports multiple coordinated views, brushing and linking, and volume visualization. For continuous scatterplots, the implementation described in the previous section was used. For discrete scatterplots, a similar GPU-based implementation was applied; here, points of the scatterplot are rendered via point sprites.

Please note that in contrast to the previous chapters, vector fields are not vi-
visualized directly with continuous scatterplots. Instead, multi-attribute fields are analyzed, i.e., vectors contained in the data may be decomposed to their individual data dimensions. An example of this approach can be seen in Section 6.4.1, where the “Tornado” data set is analyzed by mapping the $z$-component of the velocity data to the vertical axis of the scatterplot.

### 6.4.1 Qualitative Results

The first example in Figure 6.9 shows the “Tornado” data set. This data set is commonly used in flow visualization as a benchmark data set. It represents the 3D velocity field of air flow of a simplified tornado. Data is given on a uniform grid of resolution $128^3$. The two resulting scatterplot variants are compared in the upper part of Figure 6.9. For this data set, several data dimensions are available for visualization in the scatterplot. In Figure 6.9, the magnitude of the velocity is mapped to the horizontal axis and the velocity in $z$-direction to the vertical axis. In this way, different features of the “Tornado” can be extracted, e.g., the inner part of the vortex region (Figure 6.9b)) or the outer boundary of the vortical structure (Figure 6.9c)). Furthermore, brushing and linking is demonstrated for the “Tornado” data set. During this process, features in the scatterplot are identified and selected using a selection rectangle. In the volume visualization, voxels are highlighted if they correspond to the selected area defined in the scatterplot. Two different selections were made and their results are shown in the lower part of Figure 6.9.

The next example is the IEEE Visualization 2004 Contest data set “Hurricane Isabel”, depicted in Figure 6.10. This data set is shown in two different resolutions—a down-sampled version with a size of $128 \times 128 \times 30$ and the original data set with a size of $500 \times 500 \times 100$. Air temperature is mapped to the horizontal axis, whereas air pressure is mapped to the vertical axis. This example shows that continuous scatterplots are structurally independent of the resolution of the data set. In particular, the discrete scatterplot of the low-resolution data set induces misleading structures (i.e., the slanted, nearly vertically aligned clusters of points), which are not part of the data but due to the low sampling resolution. Moreover, while the visual result of a discrete scatterplot depends on the size of the individual points, continuous scatterplots are parameter-free.

The third example in Figure 6.11 shows scatterplots that support the user-guided specification of 2D transfer functions for volume rendering [KKH02]. The scatterplots visualize the “Blunt Fin” data set, which is given on an unstructured grid derived from a curvilinear grid of resolution $40 \times 32 \times 32$. Since this data set contains only one scalar value representing the velocity of the flow, the second data dimension is obtained by computing the magnitude of the gradient of the scalar velocity (a more detailed description is given in Section 1.6). Although both types of scatterplots show arc-like patterns, differ-
Figure 6.9: In the upper part, both types of scatterplots are shown for the “Tornado” data set. The discrete scatterplot is shown in (a), the continuous version in (b). The lower part shows three volume visualizations of the data set. The lower-left image (c) shows the tornado visualized by a representative isosurface of velocity magnitude. The image in the middle (d) shows highlighted voxels (yellow) that were marked in the continuous scatterplot. This highlighting corresponds to the upper-right selection rectangle in the continuous scatterplot. The other selection rectangle in the lower-mid part of the continuous scatterplot highlights different voxels, as shown in the lower-right volume-visualization image (e). In image (e), highlighted voxels (yellow) and the velocity magnitude are simultaneously visualized by rather transparent volume rendering in order to show selected features at different depths. Therefore, we can see that different voxels than in (d) are highlighted, especially not the ones in the center of the tornado.
Figure 6.10: Comparison of discrete (a, c) and continuous (b, d) scatterplots for the “Hurricane Isabel” data set. The upper row shows scatterplots for the low-resolution version of size $128 \times 128 \times 30$. The middle row shows the scatterplots for the data set in its original size of $500 \times 500 \times 100$. The left image in the lower row shows highlighted voxels (yellow) that were marked in the upper left scatterplot by the non-rectangular selection area. The highlighted voxels cover the bottom slice of the volume, showing that the prominent structures of the discrete scatterplot are simply related to the low resolution in $z$-direction. The lower right image shows the selection of all negative air pressure samples (selected by the large white box in the upper left scatterplot). The highlighted voxels lie exclusively in the center of the hurricane.
Figure 6.11: The discrete scatterplot of the “Blunt Fin” data set is shown in (a), whereas (b) is the continuous version. Both types of scatterplots visualize the scalar data value along the horizontal axis and the magnitude of the gradient along the vertical axis. Choosing these data dimensions, material and boundary identification is possible by finding arc-like structures.

ences between discrete and continuous scatterplots are clearly visible. Unlike the discrete scatterplot, the continuous version provides a dense visualization that allows spotting of interesting features more easily than in the discrete representation. The discrete scatterplot just uses the data at the grid points and ignores the underlying grid structure, whereas the continuous scatterplot takes into account the varying size and shape of grid cells by computing gradients within cells. Therefore, differences between discrete and continuous scatterplots may be particularly pronounced for unstructured or curvilinear grids compared to uniform grids with their constant cell size.

For all three examples, continuous scatterplots show better visual quality than discrete scatterplots. In particular, discrete scatterplots tend to miss visual information in plots; those visual gaps require extra mental work by the user in order to close those gaps. In addition, some features are glossed over or are completely missing in discrete scatterplots. In contrast, continuous scatterplots provide guaranteed coverage of the relevant parts of the scatterplot domain and, thus, cannot miss important features.

6.4.2 Performance Results

The main goal of Section 6.3 was to speed up the computation of continuous scatterplots. In this section, the original CPU implementation is compared with the GPU implementation. Both approaches are comparable since exactly the same data is processed, and the same result is produced. For the time measurements a computer was used with an Intel CPU running at 2.4 GHz and an NVIDIA GeForce 8800 GTX graphics card. All continuous scatterplots were created for a viewport size of 1024 × 768 pixels. The measurements were per-
Figure 6.12: Comparison of CPU and GPU implementation for three data sets. The time which is needed to compute a continuous scatterplot is measured in seconds with identical parameters for both approaches.

formed for three different data sets and the results are shown in Figure 6.12. The first data set is the “Bucky Ball”, an artificial volume data set with a resolution of $32^3$. The data set itself contains scalar values, which are mapped to the horizontal axis of the continuous scatterplot. The second data dimension is derived according to the approach described in Section 1.6. The “Bucky Ball” data set is very small, resulting in short computation times for the continuous scatterplot. The CPU implementation needs 1.25 s to finish, whereas the GPU version takes only 0.03 s. The preprocessing step for the GPU version needs 0.86 s for this data set. Taking this into account, the GPU version is 1.4 times faster. However, once the pre-sorting is done, the GPU version is approximately 41 times faster.

The second data set is “Hurricane Isabel”, where several data dimensions are available at a resolution of $100 \times 100 \times 20$. As in Section 6.4, air temperature is mapped to the horizontal axis and air pressure to the vertical axis. For this data set, the CPU version takes 7.88 s to compute a continuous scatterplot. The GPU version needs 0.08 s to perform the same task, after a precomputation step which takes 5.11 s. Taking the precomputation step not into account, the GPU version is approximately 98 times faster than the CPU version.

The third and last data set is a CT scan of a human tooth. This data set has a resolution of $64 \times 64 \times 80$. Again, the scalar values of the data set are mapped to the horizontal axis, whereas the magnitude of the gradient of these scalar values is mapped to the vertical axis. For this data set the CPU version needs 13.15 s to produce the final result. The GPU version needs 8.46 s for the pre-computation step and 0.32 s for the actual computation of the continuous scatterplot. Without considering the precomputation step, the GPU version is approximately 41 times faster than the CPU version.
When taking the preprocessing step into account, the algorithm performed on the GPU with CUDA is up to 35 percent faster than the CPU implementation. This shows that exploiting the massively parallel execution possibilities of modern GPUs pay off for even small data sets. Considering that a user of continuous scatterplots commonly has to recomputate the final image several times to finish her tasks, the speed-up is even greater once the preprocessing step is performed. After this step, the computation of a continuous scatterplot is performed with interactive frame rates.
Adaptive Rendering of Continuous Scatterplots

The practical implementation of 2D continuous scatterplots presented in Chapter 6 is subject to a few limitations. The first limitation is related to the interpolation method used for computing the density in the scatterplot domain. The original implementation of continuous scatterplots follows the idea of projected tetrahedra [ST90]. Within a tetrahedron, barycentric interpolation is applied—this linear interpolation leads to a simplified computation of scatterplot density. For regular grids, the hexahedral cells are decomposed into five tetrahedra to compute a continuous scatterplot. The drawback of this approach, however, is that native trilinear interpolation of regular grids is not supported. The second limitation of the original continuous scatterplot approach is related to the time needed for computing the overall density. Splitting a regular grid into tetrahedra introduces additional overhead; finding the density for average-sized data sets (e.g., in the size range of $128^3$) is very time-consuming and may require several seconds (cf. Table 6.12) or even minutes for larger data sets. Since the original scatterplot algorithm is based on projected tetrahedra, its run-time behavior is similar and does not scale well for large data sets given on regular grids, even when executed on GPUs using the technique shown in Chapter 6.3.

To overcome the above drawbacks of the tetrahedra-based scatterplot computation, an additional approach [BW09] was created which does not, in contrast to the one presented in Chapter 6.3, rely solely on parallel execution on GPUs to increase rendering performance. The new approach makes use of ideas of adaptive grid subdivision and hierarchical octree structures to efficiently approximate the scatterplot image. In this way, continuous scatterplots can be computed for arbitrary interpolation or reconstruction functions applied to the data set on the spatial grid. In addition, when compared to the original algorithm for computing a continuous scatterplot, the new approach is simpler and easier to implement. The new method natively supports regular grids, i.e., triangulation is no longer necessary. This is one of the reasons why the time needed to compute the density for a continuous scatterplot is greatly reduced. Furthermore, the new approach is adaptive, and the approximation error introduced when estimating the density contribution can be controlled by a single parameter. This parameter enables the user to balance computation time
and scatterplot quality. The direct support for regular grids, which are most popular in scientific visualization, and the high rendering speed, allows one to seamlessly integrate continuous scatterplots into typical interactive visualization systems.

## 7.1 Mathematical Approximation Model

This section adds the new approximation model for a fast computation of scatterplots. As shown in Section 6.1.4, a continuous scatterplot needs to render a density function $\sigma$ defined on the data domain. The problem is that this requires a complicated integration of a potentially varying integrand $s(x)/|\text{Vol}(D \tau(x))|$. Moreover, the integration domain $\tau^{-1}((\xi_1, \xi_2))$ is the intersection of two isosurfaces within the 3D spatial domain (corresponding to isovvalues $\xi_1$ and $\xi_2$). Both issues can be directly resolved for the special case of barycentric interpolation in tetrahedral cells, as exploited in Section 6.2. However, for general interpolation or reconstruction functions $\tau$, the computation of Equations 6.9 and 6.10 is non-trivial and might not even have an analytic solution. Therefore, the following approximation strategy is applied. This approximation starts with the observation that generic continuous scatterplots can be derived from an abstract version of mass conservation described in Section 6.1.1 and revisited here:

$$M = \int_V s(x) \, d^n x = \int_{\Phi = \tau(V)} \sigma(\xi) \, d^m \xi. \quad (7.1)$$

Here, $M$ describes the “mass” of virtual material in either the spatial domain (left integral) or the data domain (right integral). The term $V$ describes any volume in the spatial domain, and $\Phi = \tau(V)$ is the corresponding volume in the data domain. For the special case of 2D scatterplots of 3D data sets, $n = 3$ and $m = 2$.

Equation 7.1 is used to approximate the density $\sigma$ by assuming constant distributions of densities $s$ and $\sigma$ inside those volumes:

$$M \approx s \, V \approx \sigma \, \Phi. \quad (7.2)$$

This leads to

$$\sigma \approx \frac{s \, V}{\Phi}. \quad (7.3)$$

Typically, $s$ is constant for the whole data set and, thus, can be assumed to be 1. Then, Equation 7.1 is reduced to

$$\sigma \approx \frac{V}{\Phi}. \quad (7.4)$$

With this equation, the density can be directly computed in the data domain. The problem is reduced to determining the volumes $V$ and $\Phi$. The approach is
\( n = 3 \)

\( m = 2 \)

Figure 7.1: Illustration of the approximation of \( \Phi \). Projecting a hexahedron from the spatial domain (left) to the data domain (right) results in eight points located in the data domain. The stippled lines indicate the shape that is constructed to represent \( \Phi \): an axis-aligned rectangle or the convex hull of the eight points.

to consider a volume \( V \) in the spatial domain (i.e., some kind of discretization of 3D space) and apply the transformation \( \tau \) to the volume \( V \), which yields the volume \( \Phi \) in the data domain. An illustration of this projection step is shown in Figure 7.1. Please note that a simplified notation is used in which \( V \) and \( \Phi \) denote the actual geometry of a region or its corresponding volume or area, depending on the context.

This approach lends itself to a control of the approximation error. Depending on the extent of \( \Phi \) in the data domain, a direct measure is available for the maximum error in the data domain (with respect to the error of the projected footprint). If the extent of \( \Phi \) exceeds a given error threshold, then \( V \) needs to be reduced to keep the approximation error within the specified error bounds. Once the approximation error requirement is met, the density contribution of the data values from \( V \) can be approximated by calculating the ratio \( V / \Phi \). The only remaining issue is to compute the size of \( V \) and \( \Phi \).

### 7.2 Algorithm

Two variants of an algorithm were developed that approximate the density in the data domain. Both variants are based on the idea expressed in the previous section: subdivision in the spatial domain controls the approximation error of the density in the data domain. This is done in two steps:

1. the volume \( V \) is projected to the data domain,
2. the size of the corresponding volume \( \Phi \) is estimated.

These two steps are repeated until the extents of \( \Phi \) are below a user-specified threshold. In contrast to the original continuous scatterplot approach (shown
in Chapter 6.2), both algorithmic variants do not rely on dividing the spatial domain into tetrahedra. Instead, the regular grid of the data set is used to form hexahedra, each of them storing eight multivariate data samples at the corners. As in the original continuous scatterplot approach, the overall density $\sigma$ in the data domain can be found by linear superposition of all cell contributions. Therefore, this algorithm can construct $\sigma$ by considering one cell after another.

The two versions of the algorithm differ in the way how the size of the volume $\Phi$ in the data domain is computed. The first version uses a convex-hull approach to calculate the volume of $\Phi$ accurately, whereas the second version approximates $\Phi$ by an axis-aligned rectangle that encompasses the exact shape of $\Phi$.

### 7.2.1 Subdivision

Following the idea described in Section 7.1, the first step is to project a small volume $V$ to the data domain. The spatial volume $V$ is constructed by creating hexahedra within the regular grid, attaching eight multivariate input data samples to the corners of each hexahedron. Projecting the eight data samples to the data domain results in eight point locations that determine the shape of $\Phi$. Now, the size of $\Phi$ is calculated by finding the convex hull of the eight points. The extents of this convex hull can be computed easily.

The subdivision process is triggered when the extent of the convex hull exceeds a user-given limit in the data domain. Possible criteria to measure the extent of $\Phi$ include the area of $\Phi$ or the maximum extent of $\Phi$ in the $\xi_1$ and $\xi_2$ dimensions. For this implementation, the latter option is used in order to guarantee that the maximum length of $F$ is bounded. When the subdivision criterion triggers a subdivision step, the current hexahedron is split into eight new hexahedra in a regular fashion. Regular subdivision makes it easy to determine the size of the subdivided volumes $V$: $V$ covers a relative volume of $2^n2^n2^n$ if $n$ is the subdivision level.

For each of the new hexahedra, the attached data values are recomputed using trilinear interpolation. Please note that generic interpolation or reconstruction methods can be applied to find those data values, replacing the trilinear reconstruction filter. The process of projecting the data values of the new hexahedra to the data domain is repeated recursively until the threshold is no longer exceeded. In this case, the resulting convex hull is rendered as a filled polygon with constant density $V/\Phi$, using additive blending. This algorithm is outlined in Figure 7.2 as pseudo code.

An even faster approach to approximate the size of $\Phi$ uses an axis-aligned rectangle in the data domain that forms a tight fit around the eight projected points. Since only the lower-leftmost and upper-rightmost points have to be found, the computational effort is reduced when compared to finding the con-
// main loop:
for each Cell in data set
{
    Process (Cell);
}

// function Process (IN: CurrentCell)
{
    project CurrentCell to data domain;
    if (Size (ProjectedCell) > threshold)
    {
        // split into eight new cells
        // generic interpolation possible!
        Split CurrentCell;
        for each NewCell do // recursion
            Process (NewCell);
    }
    else // size of Phi small enough
    {
        // draw either convex hull
        // or axis-aligned rectangle
        create triangles for CurrentCell;
    }
}
7.2. ALGORITHM

// precomputation step:
BuildOctree;

// Invoke rendering:
TraverseOctree (OctreeRoot);

// function TraverseOctree (IN: OctreeNode)
{
  if (OctreeNode.SizeOfPhi > threshold)
  {
    if (OctreeNode has children)
    {
      for all children // recursion
        TraverseOctree (child)
    }
    else // reached a leaf
    {
      // perform subdivision
      Process (OctreeNode.Cell)
    }
  }
  else // size of Phi small enough
  {
    create triangles for OctreeNode.Cell;
  }
}

Figure 7.3: Pseudo code that traverses the octree. Once a leaf is reached, the same subdivision is used as in Fig. 7.2.

vex hull and computing its extents. In addition, the rendering is based on simple rectangles instead of more complex filled polygons. Otherwise, this algorithmic variant is identical to the first one.

7.2.2 Octree Hierarchy

Until now, all cells of the original data set are taken into account equally, regardless of their contribution to the final density in the data domain. However, many data sets contain large homogeneous regions. To reduce the amount of cells that have to be considered, an additional octree hierarchy is applied that quickly processes those homogeneous regions in the input data set. Since the computational overhead of the octree should be as small as possible, subdivision is done in a regular fashion only (i.e., cells of the same level in the octree all have the same size). For each node of the octree, the smallest and largest data values of each data dimension are stored. With these values, the extents of the previously mentioned axis-aligned rectangle are known. While traversing the octree, these extents are simply compared with the user-given threshold.
Descending further down along the octree is done as long as the extents exceed the threshold. Once the octree is traversed to the lowest level, each leaf contains a single hexahedron of the original data set. If the size of \( \Phi \) of that hexahedron still exceeds the threshold, the hexahedron is subdivided as described in Section 7.2.1. For this approach, filled rectangles are rendered with constant density as a representation of \( \Phi \). The summary of this algorithm is presented as pseudo code in Figure 7.3.

### 7.3 Generalization

The basic idea of this approach can be extended to other kinds of grids than regular grids. Alternative grids may lead to modifications of the computation of \( V \) or \( \Phi \), while the basic idea of subdivision remains. For example, generic structured grids, such as curvilinear grids, have the same topology as regular grids. Therefore, the structure and shape of \( \Phi \) are not affected by this kind of generalization. Only the computation of \( V \) needs to incorporate a more complicated volume formula for deformed hexahedral cells. For other kinds of primitive cells—such as tetrahedra or prisms—the shape of \( \Phi \) might differ from the hexahedral case. However, the concept of axis-aligned bounding volumes or convex bounding cells in the data domain is not affected. In other words, the main issue with more complex grid structures is not the subdivision process, but to construct an adequate replacement of the octree hierarchy that is put on top of the original data set.

Another kind of generalization replaces the trilinear interpolant. The only required change is that, during subdivision, the more general reconstruction function is invoked to compute the new data values at newly inserted grid points. If the generic reconstruction function is convex, then the convex-hull approach still computes an accurate volume \( \Phi \). A function is denoted as convex when the function values stay within the min-max interval of the input values (i.e., the data values at grid points). This is the case for trilinear interpolation and for the example of on-the-fly gradients used for 2D transfer functions (see example in Section 7.5). Even if the generic reconstruction function is not convex, this approach may still provide a good approximation as long as the function values do not reach too far outside the min-max interval of the input values.

### 7.4 Implementation

The implementation to compute the density in the data domain is written in C++ and executed on the CPU. In order to keep the implementation simple, the code is running single-threaded. The approach that uses the convex hull to compute the size of \( \Phi \) employs the algorithm described by Jarvis [Jar73].
The octree needed for acceleration of this algorithm is built in a preprocessing step and kept in main memory for all following traversal steps. All generated triangles are rendered with OpenGL using triangle strips. The final result is stored in a floating point texture—by doing this, the continuous scatterplot must be recomputed only if the user changes the viewing parameters. For other user inputs, e.g., brushing areas of interest, the texture is drawn that stores the continuous scatterplot.

### 7.5 Results

There are two aspects that are of main interest: scatterplot quality and computational speed. The quality of a scatterplot created by the proposed approximation algorithms directly depends on the user-specified threshold. Raising the threshold increases the approximation error, but decreases the time necessary to compute a continuous scatterplot and vice versa. Please note that this threshold is intuitively specified in terms of pixels in the scatterplot—it defines the maximum extents of a projected cell in the data domain.

The scatterplots for the following analysis were created on a personal computer equipped with an Intel CPU with 2.4 GHz and 4 GB of RAM. The computer’s GPU is an NVIDIA GeForce 8800 GTX with 768 MB of texture memory. All scatterplot images have a resolution of 1024 × 768.

The method is first applied to the “Tornado” data set which has dimensions $128^3$. This data set contains simulated wind speeds stored as 3D vectors. The horizontal axis shows the magnitude of the velocity, whereas the vertical axis is mapped to the $z$-dimension of the wind speed to gain the same results as in Section 6.4.1. The resulting images are shown in Figure 7.4. The maximum approximation error is reduced for each image from left to right which results in increased image quality as expected.

As a second example, the CT scan of a human tooth is analyzed. The spatial extent of this data set is $128 \times 128 \times 160$. The second data dimension is derived as described in Section 1.6. Computing the gradients on-the-fly for the second data dimension is an example of a generic non-linear reconstruction function that can be used with this approach. First, a series of continuous scatterplots is shown which are created with the subdivision approach using a convex hull to represent $\Phi$ in the data domain. To examine the effect of the user-specified threshold, the maximum approximation error is decreased step by step. Figure 7.5 (upper row) shows this series of scatterplots. Despite the significant differences in the threshold, the continuous scatterplots differ only marginally. As it turns out, density values within a cell do not vary strongly, therefore the approximation that uses constant density does not deviate too far from the true values. Differences to the original continuous scatterplot can be explained with the improved interpolation method.
Figure 7.4: These continuous scatterplots were created with the octree approach for the “Tornado” data set. The effect of different thresholds is shown. A coarse approximation is used in (a) with a threshold that allows $\Phi$ to extend up to 100 pixels in each dimension, (b) uses a decreased threshold of 50 pixels, and (c) is drawn with a threshold of 25 pixels.

Figure 7.5: These continuous scatterplots were created with the subdivision approach for the “Tooth” data set. Pictures (a), (b), and (c) show results of the subdivision approach using a convex hull to represent $\Phi$ in the data domain. In (a), $\Phi$ is allowed to span up to 200 pixels in each dimension, (b) is created with a threshold of 100 pixels, whereas (c) uses a threshold of 50 pixels. Pictures (d), (e), and (f) show the same data set, but this time the subdivision approach uses axis-aligned rectangles to represent $\Phi$. The same thresholds as for the upper row are applied.
Figure 7.6: These continuous scatterplots were created with the octree approach for the “Engine” data set. The effect of different thresholds is shown. A coarse approximation is used in (a) with a threshold that allows $\Phi$ to extend up to 200 pixels in each dimension, (b) uses a decreased threshold of 100 pixels, and (c) shows a good approximation since the threshold is lowered to 50 pixels.

The same subdivision approach can be used in combination with axis-aligned rectangles to approximate $\Phi$. In Figure 7.5, this series is shown in the lower row. In contrast to the previous example, the effect of the threshold is clearly visible. Using a high threshold and therefore allowing a high approximation error results in a coarse scatterplot. When compared with the subdivision approach that uses the convex hull, this approach yields scatterplots of lower quality since the approximation of $\Phi$ tends to deviate much more from the correct solution. On the other hand, scatterplots computed with this approach are created faster, due to simpler computations.

An additional data set was analyzed which was created with a CT scan of an engine block. The spatial extent of this data set is $256 \times 256 \times 110$. As for the first data set, the second data dimension is generated by computing the magnitude of the gradient of the data samples.

With this “Engine” data set, a third series of continuous scatterplots was created. Here, an octree is used in combination with axis-aligned rectangles to approximate the density contribution of a cell in the data domain. The resulting pictures are shown in Figure 7.6. Due to the octree hierarchy, a speed-up of up to two orders of magnitude is achieved compared to the original continuous scatterplot approach. The octree is created in a precomputation step which needs less than one second to prepare for the “Engine” data set. Since memory consumption of the octree is very low, the overhead introduced by the tree structure can be neglected. Depending on the given error threshold, this approach allows one to draw very coarse representations of the scatterplot, since an arbitrary number of cells can be combined in higher levels of the hierarchy. Therefore, this approach is completely independent from the input resolution.
of the data set. The computational speed directly depends on the error threshold; however, faster computation leads to lower scatterplot quality.

In order to quantify the effects of the threshold with regard to scatterplot quality, error plots are shown in Figure 7.7. The $L_2$ norm was used to measure the error between scatterplots. In order to obtain an $L_2$ norm that can be interpreted easily, the density values of the scatterplots are normed. By doing this, the scale of the scatterplot changes in such a way, that a density value of one corresponds to the arithmetic mean of all density values. Since an analytic solution of a continuous scatterplot is not available, the convergence behavior is analyzed by comparison to a numeric solution with a low error threshold of only 10 pixels. Both error plots for the subdivision approach show a similar behavior: first, the $L_2$ norm stays on a constant plateau before it drops with decreasing error threshold. High error thresholds do not trigger the subdivision process, therefore the same values are returned by the $L_2$ norm. For lower error thresholds, the subdivision approach converges in an expected way. Please note that the error values are at a very low level at all times (below 0.2 percent for the “Tooth” data set and 0.1 percent for the “Engine” data set), since the convex hull provides a good approximation. The error plots for the hierarchical octree approach do not have an upper bound as the error plots for the subdivision approach. This is explained with the fact that the octree hierarchy allows very coarse representations for the size of $\Phi$. However, the same behavior as for the subdivision approach can be observed for lower thresholds: the error converges similarly to zero for low error thresholds. However, the absolute scale of the $L_2$ values is higher due to the coarser approximation by axis-aligned rectangles.

Continuous scatterplots are designed to be included in existing or future inter-
Table 7.1: Time in seconds needed to compute a scatterplot depending on the chosen approach and error threshold. For conventional and continuous scatterplots, there is no threshold that can be set, therefore, only one result is recorded. “Convex Hull” is the subdivision approach using the convex hull to represent $\Phi$. “Octree” is the hierarchical approach that uses axis-aligned rectangles in combination with an octree. Different thresholds were used for the performance measurements; these thresholds are listed in the top row.

<table>
<thead>
<tr>
<th></th>
<th>Tooth</th>
<th>Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Conventional Scatterplot</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Continuous Scatterplot</td>
<td>39.8</td>
</tr>
<tr>
<td></td>
<td>Convex Hull</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Octree</td>
<td>-</td>
</tr>
</tbody>
</table>

active visualization systems. Therefore, their run-time behavior with regard to computation time is of interest as well. Table 7.1 lists time measurements of different methods that compute a continuous scatterplot of the “Tooth” and “Engine” data set. There are some interesting conclusions that can be drawn from these measurements. First of all, the original continuous scatterplot approach is not very user-friendly, since it needs a very long time to compute. This is overcome by using the approaches presented in this chapter. However, the user has to take care that the approximation-error threshold is appropriate. If this is not the case, the presented methods may even need more time to compute a continuous scatterplot than in the original approach. On the other hand, if a good trade-off between accuracy and speed is found, these algorithms compute a continuous scatterplot much faster while the scatterplot is still approximated fairly well.
The preceding chapters presented solutions to speed up the computation of continuous scatterplots. Parallel processing of grid cells shown in Section 6.3 achieves this goal using specialized GPU hardware, whereas the subdivision algorithm of Chapter 7 employs a different computation scheme to reduce necessary computations. Our alternative approach [HBW11], presented in this chapter, is similar to the latter, as it offers a trade-off between output quality and rendering performance, depending on the needs of the user. In this way, these two algorithms create approximations of increasing quality with each iteration.

In this chapter, a forward-mapping technique inspired by splatting for direct volume rendering is employed to compute the influence of samples to the final image and successively approximate the true representation of the plot. High frame rates are the expected outcome for interactive data exploration using continuous scatterplots. In this case, splatted continuous plots allow trading accuracy for rendering performance by resampling the data at a lower sampling rate. This may be implemented by skipping some of the input data points or by resampling at a few, freely chosen positions in the spatial domain. In contrast, for very small data sets, additional samples may be distributed over the spatial domain to improve image quality. A key observation is that the continuous plots are based on samples that are completely independent from the number and positions of the grid points of the data set.

To this end, a novel closed-form analytic description of the splatted footprint of Gaussian input kernels is presented for scatterplots. A new progressive refinement algorithm is introduced that allows obtaining initial results extremely fast and hence complements a very useful property of continuous data plots: for many data sets, a small subsample of the full data may provide a good approximation to the final density distribution, making progressive refinement the ideal algorithm for rendering.

This new construction algorithm relies on splatting. The splatting method by Westover et al. [Wes89] is adopted, modified, and extended. The original method was introduced as a forward-mapping algorithm for direct volume rendering. In contrast to ray tracing techniques where pixel intensities are
computed by mapping the image plane to the data space, volumetric splatting computes the contribution of samples from the data space to the image plane. Since ray integration within a sample’s reconstruction kernel is independent of its density, the integral can be precomputed for a given view direction. The resulting image plane footprint of the kernel is then used to compute the projected image of all data space samples and only has to be recomputed if the viewing direction changes. For continuous plots, however, the viewing transformation depends on the data, such that footprints have to be computed for every sample individually.

An approach similar to splatting densities has recently been presented by Feng et. al. [FKLT10] for the visualization of uncertain data samples. In their work, a probability density function is estimated using normally distributed, uncorrelated kernels. Hence, all samples in the scatterplot are represented by scaled Gaussian footprints. In contrast, the model presented in this work uses gradient information contained in the data to transform the density distribution of scatterplots. Furthermore, densities are derived analytically to allow for the precomputation of splat footprints.

### 8.1 Footprint Computation

This section describes the computation of density footprints for reconstruction kernels in the original spatial domain of the data and the respective mapping to scatterplots. Please note that terminology from measure theory is used and thereby weighted, non-normalized densities are used (as opposed to the probability density which is typical for statistics literature).

As introduced in Section 6.1.1, the two main ingredients for the mathematical model of the statistical plots are:

- **Density function** $s(\vec{x}) : \mathbb{R}^3 \to \mathbb{R}$, $\vec{x} \mapsto s(\vec{x})$, which describes the scalar-valued density over the spatial domain of the data set, i.e., the importance of the respective spatial part of the data.

- **Map** $\tau : \mathbb{R}^3 \longrightarrow \mathbb{R}^n$, $\vec{x} \mapsto \tau(\vec{x})$, which represents the multivariate data set defined on the same spatial domain as above.

The mathematical problem setting can now be formulated as follows: what is the transformed density function in the data domain? This density function explicitly depends on the density $s$ in the spatial domain and is implicitly affected by the map $\tau$, which connects from the spatial domain to the data domains. Figure 8.1 illustrates the domains and the example of mapping a typical splatting kernel.

In the following, splats are first defined in the spatial domain and the map $\tau$ is discussed in the context of sampling at the center points of splats. Then, the discussion is restricted to the transformation of a single splat template,
aplying this model to continuous 2D scatterplots.

### 8.1.1 Spatial Domain and Data-Set Function

Following the splatting approach for direct volume visualization [Wes89], the density in the spatial domain is represented by a weighted sum of kernels at various spatial locations. Then, the overall density on $\mathbb{R}^3$ is given by

$$ s_{\text{overall}}(\vec{x}) = \sum_i w_i s_i(\vec{x}) $$

with scalar-valued weights $w_i$ and kernels $s_i$. Similar to splatting in volume rendering, the kernels $s_i$ are assumed to be derived from a single template function that is just shifted to different positions. Furthermore, such a template is usually assumed to be spherically symmetric due to reasons of isotropy.

The main idea of splatting is that the template is transformed in a preprocessing step to form a splat. The transformation of the overall density $s_{\text{overall}}$ is then computed by overlaying the splats with the same weights $w_i$. This approach requires that transformation and weighted summation are commutative, which is true for continuous scatterplots because they are computed by a linear operator.

The sample positions $\vec{x}_i$ may be arbitrarily chosen. In particular, they are independent from positions of grid points of the data set. Typically, the sample positions $\vec{x}_i$ are evenly distributed in space, e.g., by putting them on a regular sampling grid or by applying low-discrepancy point sets (see Section 8.3).
The usual case of constant overall density $s_{\text{overall}}$ can be implemented by even distribution of sample positions and constant weights $w_i$.

The kernel $s_i$ is centered at the point $\vec{x}_i$ with relative “radius” $k$. The Gaussian kernel is popular in volume rendering due to its fast fall-off behavior with increasing distance from the kernel center. Therefore, it is used here as well.

### 8.1.2 Scatterplot Domain

The density function described by the kernel $s(\vec{x})$ in the spatial domain is now transformed to the corresponding density function $\sigma(\vec{\xi})$ at position $\vec{\xi} = (\xi_1, \xi_2)^T$ in the 2D data domain. $\sigma(\vec{\xi})$ is also called footprint of the splat or, in short, just splat or footprint.

The footprint can be computed according to Equation 6.9—since $\tau$ is linear, the partial derivatives are constant and the volume measure can be moved outside the integral. Heinrich et al. [HBW11] describe in detail how such a footprint of any sample can be transformed to a generic template footprint using a scaling matrix multiplied by a matrix for rotation around the $z$-axis with a specific angle.

Although the splat computation for scatterplots resembles the one for volume rendering, there is an important difference: the scatterplot-domain splat additionally depends on $\tau$, which is not the case in volume rendering.

### 8.2 Sampling and Progressive Refinement

The previous section provided the basis for rendering a single splat in the continuous statistical plot. The overall plot is obtained by applying this process to many different splats that densely cover the spatial domain, leading to an appropriate representation of the overall density by accumulating the splats with additive blending. Besides the data values, the partial derivatives of the data are reconstructed at the splat sample so that the position, size, and orientation of the splat can be determined. In this way, the generic footprint template is transformed according to Section 8.1. Using the inverse transformation, a splat is rendered as a rectangle with the correct texture coordinates and density values. The generic template footprints for scatterplots are discretized in a 2D texture during pre-processing. The 2D standard Gaussian $e^{-\|\vec{x}\|^2}$ is sampled on the uniform grid of the texture, which then serves as a lookup table during rendering.

The splatting technique is further improved by extending it to progressive rendering. Due to the linear superposition of splats, progressively sampled intermediate images can be combined by linear superposition as well. More specifically, a sequence of several, independent continuous plots of low sam-
pling resolution are generated that are then accumulated in a separate image (e.g., an offline rendering target on the GPU). Mass conservation, as described in Section 6.1.1, is guaranteed for the transformed densities by compositing single images $I_1$ and $I_2$ using the over operator [PD84] and a fixed value for $\alpha$:

$$I = \alpha I_1 + (1 - \alpha)I_2 \quad \text{with} \quad 0 \leq \alpha \leq 1.$$ 

If both intermediate image observe mass conservation (i.e., each has the same overall mass $M$ as accumulated from the densities or all pixels), then the blended image is guaranteed to have identical mass as well because mass is also subject to alpha blending.

The footprints further depend on the choice of the smoothing parameter $k$, which is a measure for the “radius” or bandwidth of the Gaussian kernel in the spatial domain. This parameter is transported to the data domain, where it affects the extents of the footprints to be drawn. On the one hand, large kernels provide a better coverage of, and higher overdraw on, the reconstructed spatial area and thus allow reducing the sampling resolution. On the other hand, large splats introduce a large error to the overall density, resulting in potentially highly blurred images. Analogously, small kernels produce a more accurate, but potentially non-smooth sampling of the density distribution.

### 8.3 Results

In this section, the results obtained with splatting are compared with discrete and previous non-splatting continuous plots with respect to rendering performance and visual quality. First, results of all of the three rendering techniques are compared with respect to their visual appearance. Then, some results are shown which were obtained by using resampling and progressive refinement. The relation of splat size and sampling resolution is also investigated with respect to rendering performance and image coarseness. All measurements were produced with an implementation based on C++ and GLSL. The implementation was tested on a PC with an Intel Core 2 Quad CPU running at 2.4 GHz and an NVIDIA GeForce 8800 GTX graphics card.

As discussed in Section 8.2, increasing splat size allows reducing the sampling resolution, while the level of coarseness of the resulting image is approximately maintained. The relationship between these parameters is illustrated in Figure 8.2. Here, every plot was rendered without progressive refinement but with different sampling resolutions and reconstruction kernel sizes. As expected, the images become smoother with increasing number of samples and with increasing splat size. Furthermore, coarseness is approximately maintained when doubling the number of samples with half of the kernel size and vice versa.

Figure 8.3 shows a performance analysis for different splat sizes and sampling
8.3. RESULTS

<table>
<thead>
<tr>
<th>Samples</th>
<th>Discrete</th>
<th>$k = 0.2$</th>
<th>$k = 0.4$</th>
<th>$k = 0.8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{11}$</td>
<td><img src="image1" alt="Image" /></td>
<td><img src="image2" alt="Image" /></td>
<td><img src="image3" alt="Image" /></td>
<td><img src="image4" alt="Image" /></td>
</tr>
<tr>
<td>$2^{12}$</td>
<td><img src="image5" alt="Image" /></td>
<td><img src="image6" alt="Image" /></td>
<td><img src="image7" alt="Image" /></td>
<td><img src="image8" alt="Image" /></td>
</tr>
<tr>
<td>$2^{13}$</td>
<td><img src="image9" alt="Image" /></td>
<td><img src="image10" alt="Image" /></td>
<td><img src="image11" alt="Image" /></td>
<td><img src="image12" alt="Image" /></td>
</tr>
</tbody>
</table>

Figure 8.2: The effect of the sampling resolution and the kernel size parameter with respect to the overall coarseness of the plot. In the matrix shown above, the number of samples increases from top to bottom while the splat size increases from left to right. In both cases, the corresponding images become smoother. At the same time, however, increasing splat sizes result in less accurate plots, as can be seen in the rightmost column. Here, the blur introduced by larger splats makes the image appear “wider”. The bottom row shows the $L_2$ error of densities from the splatted plots with respect to a traditional, discrete scatterplot rendered with $10^7$ samples. While the convergence behavior is similar for all columns, the total error decreases with increasing smoothing factor $k$. 
Figure 8.3: Rendering performance for splatted continuous scatterplots of the “Bucky Ball” dataset depending on splat size and sampling frequency. From the plot, a linear dependency between sampling resolution and frames per second can be concluded. Note that, for comparison, the performance of traditional discrete scatterplots and of continuous scatterplots rendered with the projected tetrahedra algorithm are included. The traditional scatterplots are limited to 60 frames per second by the frame refresh rate of the test-hardware. In contrast, the frame rate of continuous scatterplots lies well below one frame per second.
rates, compared with traditional discrete scatterplots and continuous scatterplots using the original projected tetrahedra algorithm. From the measurement data, a linear dependency between sampling rate, splat size, and rendering performance can be concluded (note the \( \log_2 \) scale of the \( x \) axis). Although performance decreases with the number of splats, the progressive refinement algorithm introduced in the previous section can still be used to achieve interactive frame rates, as the total number of splats is divided into successive rendering frames. As a consequence, the algorithm scales well with data set size and can easily be adopted for streaming data, as only a fixed number of samples is required for the individual rendering steps. This also includes large simulation data or time-dependent data, where visualization may be required in real-time. Finally, efficient rendering of statistical plots naturally facilitates the implementation of stacked displays or small multiples, where many instances of the same plotting technique are used to visualize different parts, projections, or subareas of the data set. For example, the scatterplot matrix may be extended using progressive refinement.
The visualization of vector fields is an established, however still ongoing field of research. The goal of this thesis was to develop techniques that allow the visualization and analysis of various kinds of vector fields that encode directional information, being it flow or another property. Depending on the nature of particular fields, different technical or fundamental problems had to be solved to achieve this goal.

For highly complex and memory-consuming vector fields, a cluster environment was employed to create an interactive visualization tool as detailed in Chapter 2. With each compute node that is integrated into the cluster environment, the available memory increases directly, however, the expected speed-up does only scale well for a small number of nodes, but with decreasing effectiveness for a higher number of cluster nodes. More precisely, the performance gain decreases with every added cluster node due to additional communication overhead. Further research is necessary to reduce the communication overhead to allow many cluster nodes being involved in the visualization process. Additionally, the parallel rendering method could be extended to the projection of the surface geometry instead of replicating the mesh on each node. This would allow visualizing extremely large surface meshes.

For the analysis of vector fields, a visualization tool not only needs to be interactive, it also must be able to guide the user and the visualization metaphors that are used must be helpful and efficient. For a vector field, not only the direction of the flow is of interest, but also the magnitude which can be visualized with various visual cues, with animation being among them. In order to employ results of cognitive psychology research, Chapter 3 presents a technique that was developed to visualize vector fields in a dense way using animation. In contrast to existing methods, this technique is able to tune its pattern frequencies to achieve the optimum for the human visual system. Recently, Yeh et al. published a technique to visualize stream lines using repeated asymmetric patterns [YLL12] based on the same idea of using patterns that are orthogonal to the stream lines that are visualized. Although these methods do not lend themselves directly to an extension to true 3D fields, Schulze et al. [SRGT12] present a solution based on “as-perpendicular-as-possible sur-
faces”, which can be seen as the counterpart of ortho-vis in 3D. What remains to be done is an extensive user study to confirm the usefulness and effectiveness of the proposed method for 2D and 2.5D fields. Additionally, the focus of ortho-vis lies on low-level, local motion perception. Therefore, the relationship between global motion perception and the effectiveness of conveying flow structures remains an open question. Further investigations are also needed to make sure that the visual signatures introduced by the temporal filtering process give the correct impression of the visualized vector field.

The second part of this thesis takes a different approach on the visualization of vector fields, as features of interest are used to simplify the resulting images—visual clutter and complexity is reduced by using a topological approach to visualization. Such “simplification” methods are gaining importance more and more, since data sets are challenging not only in terms of memory size, but also with respect to complexity which manifests itself, e.g., in the form of turbulence—a phenomenon which is extraordinarily hard to visualize in an easily understandable way, and especially hard to find are cues that help the user analyze the inner workings of turbulence. The technique described in Chapter 4 combines existing techniques—LIC on curved surfaces (described in Chapter 1.4.5), and previous work by Sadlo and Weiskopf [SW10]—to enhance the visualization of LCS. This combination of techniques gives simple representations for data sets that are not very turbulent. Perception problems can, however, arise for complicated flow fields with turbulent regions. The structure of turbulent flows is highly complex by its nature, therefore, future work could address techniques that reduce the complexity of the visualization by finding ways to visualize only the essence of such complex data. Other future work is the extension to 3D time-dependent vector fields, i.e., space-time visualization in four dimensions which includes the intersection curves of LCS and the surfaces they span over time.

Guiding the user with visualization metaphors that are designed to simplify the analysis of vector fields was also the goal for the visualization of magnetic flux in magnetostatic fields presented in Chapter 5. Available techniques from classical vector field topology had to be extended since they were not directly applicable for this specialized scenario. Here, a topological construct, the connectrix, was introduced that is designed to visualize regions that are connected with each other with respect to magnetic flux—as opposed to classical topology which uses separatrices to visualize where regions of different flow behavior are located. Relevant results for application domain collaborators were obtained with this method. An open question left for future work is the extension to three dimensions. The main challenge here will be that the scalar-valued potential \( A \) will have to be extended to a 3-component vector field potential.

The third and final part of this thesis introduces continuous scatterplots, a statistical visualization method that was created to analyze scientific data sets. As
opposed to traditional scatterplots, this method allows one to work with data
defined on a continuous domain based on a generic mathematical model. This
mathematical model maps an arbitrary density value defined on an \( n \)-D input
data set to \( m \)-D scatterplots. Not only does this model provide a solid and re-
liable basis for many variants of frequency plots of continuous data, but it also
allows one to assess the errors introduced by previous discrete frequency plots,
which can be viewed as examples of numerical approximation of continuous
scatterplots. Therefore, continuous scatterplots lead to the same basic visual
mapping as traditional histograms, scatterplots, or other frequency plots. In
this way, they add one missing piece to the general approach of applying sta-
tistical and information visualization methods to scientific data. Furthermore,
the generic model presented in this chapter has value of its own in any scien-
tific discipline that strives for unification and simplification.

Several ways for the implementation of continuous scatterplots are explored
that either broaden the possibilities in terms of interpolation or reconstruction
methods that can be used to compute a continuous scatterplot, or decrease the
computational cost to create such a plot. To achieve the latter, speeding up
the computation of continuous scatterplots is attained with several different
computation schemes that employ user-controlled approximation methods to
reduce the time to compute a continuous scatterplot. Finally, hardware ac-
celeration is used to reach the same goal utilizing the parallel architecture of
GPUs. Remaining future work is the extension to higher-dimensional spatial
domains, such as time-dependent 3D data sets.

The mathematical model of continuous scatterplots provided new possibilities
for follow-up research in this direction, leading to a several related papers, e.g.,
“Continuous Parallel Coordinates” by Heinrich and Weiskopf [HW09], and
“Discontinuities in Continuous Scatterplots” by Lehmann and Theisel [LT10].
These examples originate from the visualization community, however, re-
searchers of other communities work on related topics. To name an example
from computer vision, Dowson et al. [DKB08] construct a continuous model
to obtain the joint distribution of image pairs. Related to this work is the paper
by Kadir and Brady [KB05] that addresses the problem of estimating statistics
in regions of interest by applying continuous density estimates.

The implementations of continuous scatterplots presented in the third part of
this thesis are not the only possible approaches—due to the generic mathe-
matical basis, the technique presented in Chapter 6 is not only unrestricted
with respect to the dimensionality of the data that it handles, it is also open
to various implementation approaches. Although several methods have been
developed to compute continuous scatterplots, e.g., as presented in Chapters 7
and 8, improvements with respect to computational performance or integra-
tion quality are still possible. Therefore, it becomes clear that the true value
of this approach is not technology-based or hardware-based—something that
may be outclassed sooner or later by future technology—the main contribu-
tions are found in the theoretical foundation presented in this thesis.

During the course of my thesis, I have encountered problems that are related mostly, but not only to flow visualization. It became clear that even software that is written to create a proof-of-concept application needs a profound software engineering approach to avoid problems that are related to organically grown software. However, extensive application of software engineering principles prolong the development process of proof-of-concept software too much—a good balance between the extremes is necessary to produce software of as-high-as-possible quality without losing too much time. For this reason, the visualization techniques described in Chapter 6 and 8 were implemented using the MegaMol framework developed by Sebastian Grottel. This framework encourages and supports high code reusability as well as a modularized approach to software engineering.

For the remaining visualization techniques of this thesis, individual tools were developed using the most up-to-date software and hardware technologies that were deemed best suitable to solve the technical problems related to the respective research project at this time. Although these tools are technically different, they allow the interactive analysis of vector fields from different points of view—perception-oriented with the methods presented in Part I, feature-based analysis with techniques of Part II, and, lastly, focusing on multi-attribute, data-based analysis in Part III.

Despite their technical differences, these proof-of-concept tools can be seen as modules that could be integrated in a larger visualization system. Depending on user requirements, this would allow, e.g., to visualize large, complex vector fields using the cluster environment described in Part I, and combine it with topological methods presented in Part II to handle occlusion and ease the analysis of such a vector field.

Closing the link to Section 1.1, the contributions to the challenges mentioned there are critically evaluated. Every method presented in this thesis adds some details to the overall picture. Some techniques are more generally applicable, whereas others are more application specific. A critical sum-up based on the remarks of this section is given in Figure 9.1, which is adopted from Section 1.8 and modified to reflect the additional aspect of generality and application specificity.

On a final note, and as mentioned in the beginning of this chapter, the visualization and analysis of vector fields is a still ongoing field of research. However, the visualization community loses interest in techniques that concentrate merely on directly visualizing vector fields, as it is the case for the ones presented in Part I. What are the reasons for this development? Possibly, this is due to the high saturation of well working visualization methods. These methods have arrived at a very high level, leaving not much room for improvements. Because of this, and because of the need for methods that are
Figure 9.1: Visualization of the techniques presented in this thesis with respect to level of abstraction and, as a second aspect, their generality or application specificity.
able to handle even more complex data sets, the attention turns to techniques that emphasize features, like, e.g., the ones presented in Part II of this thesis. The importance of models for such techniques, as well as corresponding algorithms to compute visual representations, is expected to rise even more in the future.


[BSW\textsuperscript{12}+] S. Bachthaler, F. Sadlo, R. Weeber, S. Kantorovich, C. Holm, and


1991. 107


[SEP+01] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy,


logical Methods in Data Analysis and Visualization, pages 151–165. Springer, 2011. 79


Sven Bachthaler


Sven Bachthaler graduated in 2005 from the Universität Stuttgart, Germany, with the Dipl. Inf. (MSc) degree. In 2006, he started with the graduate program of computing science at Simon Fraser University, Canada. In 2007, he returned to Germany to continue working on his PhD degree at the Visualization Research Center, Universität Stuttgart (VISUS). His research interests include scientific visualization and computer graphics.