Interactive 3D Flow Visualization Techniques Utilizing Integral Curves and Surfaces

Tony McLoughlin

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

2011

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed	....	(candidate)
Date	....	

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed	....	(candidate)
Date	....	

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed	....	(candidate)
Date	....	

Abstract

This thesis presents several contributions to the field of flow visualization. Flow visualization methodology can be classified into four main areas: direct, geometric, texture-based and feature-based flow visualization. Our research focuses on the geometric category, utilizing integral curves and surfaces.

The content is split into three parts. Part I introduces the reader to the basics of the field of flow visualization. It provides an overview and describes the above classification of techniques before presenting an in-depth review of the state-of-the-art. In Part II we present our research, this is discussed in more detail below. Part III details the design and implementation of our application framework which was used to create our algorithms. This part also provides the concluding comments and discusses areas for future work.

Our research is also split up into two groups. The first group focuses on surface construction techniques. We present two algorithms, the first algorithm is the construction of stream- and pathsurfaces. These two types of surfaces represent the trajectories of particles seeded from curves in time-invariant and time-dependent flow fields respectively. The second algorithm provides a method for constructing streaksurfaces. Streaksurfaces are far more complex than stream- and pathsurfaces and incur a greater computational overhead. Streaksurfaces are analogous to dye injection from experimental flow visualization. Both of our surface algorithms are designed around quad primitives.

The second group of our research is focused on enhancing interactive seeding for integral curves. Integral curves are ubiquitous in flow visualization and are one of the most frequently used tools by computational fluid dynamics (CFD) experts. Again, we present two algorithms in this group. The first introduces the concept of *sensitivity field* which highlights regions in the domain in which a small change in seed position results in a large change in the integral curves. The sensitivity field aids the user in navigating to interesting flow phenomena and serves as an indication as to the stability of the seeding parameters for a given set of integral curves. The final technique we introduce aids the user in improving the visualization results of a given set of integral curves emanating from a seeding curve. We provide novel similarity measure based on the concept of generating a signature for a curve and utilizing a popular statistical measure (χ^2 test) to compute the similarity between the integral curves. This enables us to perform clustering on the integral curves to produce focus+context visualizations. It also allows us to perform filtering of the integral curves to leave an expressive sub-set of curves while still preserving the set of curves which best represent the flow behavior. The end result being a less cluttered visualization which still portrays the key information about the flow behavior.

Acknowledgements

Publications

This thesis is based on the following publications:

- *Over Two Decades of Integration-based, Geometric Flow Visualization.*
  Tony McLoughlin, Robert S. Laramee, Ronny Peikert, Frits H. Post and Min Chen, In Eurographics State of the Art Reports, pages 73-92, 30 March - 3 April, 2009, Munich, Germany.

- *Easy Integral Surfaces: A Fast, Quad-based Stream and Path Surface Algorithm.*
  Tony McLoughlin, Robert S. Laramee and Eugene Zhang, In Proceedings of Computer Graphics International (CGI'09), pages 67-76, May 26-29, 2009, Victoria, Canada.

- *Over Two Decades of Integration-based, Geometric Flow Visualization.*
  Tony McLoughlin, Robert S. Laramee, Ronny Peikert, Frits H. Post and Min Chen, In Computer Graphics Forum, Vol. 29, No. 6, 2010, pages 1807-1829

- *Constructing Streak Surfaces for 3D Unsteady Vector Fields.*
  Tony McLoughlin, Robert S. Laramee and Eugene Zhang, In Proceedings of Spring Conference on Computer Graphics (SCCG 2010), pages 25-32, May 13-15, 2010, Budermice, Slovakia.

- *Using Integral Surfaces to Visualize CFD Simulation Data.*
  Tony McLoughlin, Matthew Edmunds, Robert S. Laramee, Mark W. Jones, Guoning Chen and Eugene Zhang, In Proceedings of NAFEMS World Congress 2011, Boston, Massachusetts, USA

- *Visualization of Interactive Streamline and Pathline Seeding Parameter Sensitivity*
  Tony McLoughlin, Robert S. Laramee, Guoning Chen, Nelson Max and Harry Yeh. *Submitted: Under Review*

- *Similarity Measures for Enhancing Interactive Streamline Seeding.*
  Tony McLoughlin, Mark W. Jones, Robert S. Laramee and Charles D. Hansen. *Submitted: Under Review*

- *Design and Implementation of Interactive Flow Visualization Techniques.*
  Tony McLoughlin and Robert S. Laramee. Book Chapter in Computer Graphics. InTech. ISBN 979-953-307-617-0. 2012

Table of Contents

Part I

Background and Motivation

1 Introduction

"Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime."
-Unknown[1]

Contents

VISUALIZATION is a method of communicating data. Data can be thought of as the lowest level of abstraction from which information may be derived. Data on its own has no meaning, it must be processed in some way in order to be useful. Visualization algorithms take data and produce a visual representation that provides a mechanism for deriving information about the data. Obtaining understandable information about data allows us to then derive knowledge and gain an understanding of it. The use of visualization to present information is not a new idea – maps and scientific drawings have existed for hundreds of years. However, progression in computing power and computer graphics allows us to employ sophisticated visualization algorithms to derive information from data that was not previously possible. In Computer Science we can classify the visualization method dependent upon the data domain and application:

- **Scientific Visualization:** provides graphical representations of numerical data for qualitative and quantitative analysis. Scientific visualization focuses on data that represents samples of continuous functions of space and time. This data has a natural geometric

[1]A famous proverb believed to be of Chinese origin.

structure and may consist with multi-variate data (several physical quantities at each sample point).

- **Information Visualization:** is the study of the visual representation of discrete, non-numerical abstract data. Information visualization techniques are applied to data such as text from various sources of literature, files and lines of code in software system and relationships of networks forming the internet. Trees and graphs are a common candidate for information visualization algorithms.
- **Visual Analytics:** is the science of analytical reasoning facilitated by visual interfaces [TC05]. Visual analytics has some overlap with scientific and information visualization. There is currently no clear consensus on the boundaries between them. However, visual analytics can be broadly thought of as being concerned with sense-making and reasoning.

The work presented in this thesis is on the topic of Flow Visualization. Flow visualization is a branch within scientific visualization. A more in depth introduction to this vibrant field follows.

1.1 Flow Visualization

Flow visualization is the study of techniques to understand data which describes the behavior of fluids. Typically this data is generated through computational fluid dynamics (CFD) simulations. CFD simulations use numerical methods and algorithms to approximate the behavior of liquids and gases. Due to increases in computing power and advancements in CFD research, the use of simulations is becoming more commonplace. The use of simulations also means that a real-world prototype can be delayed until the later stages of an engineering project. This provides a more cost- and time-effective solution during the initial stages of research and design. The increased usage of CFD simulations (and hence, volume of data) drives the need for effective visualization solutions to understand the simulation output/data.

Now that we know where flow visualization data is often generated, the next logical step is to discuss the data. There are many attributes that may be output as the result of a CFD simulation. However, the most interesting one is velocity, which is a vector value. Velocity is the rate of change in the direction and position of a fluid element:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt} \tag{1.1}$$

Each sample within the domain, defines the velocity of the fluid at the sample position. If we were to drop a massless particle within the field, its path would be described by:

$$d\mathbf{x} = \mathbf{v}dt \tag{1.2}$$

which is expressed in integral form by:

$$\mathbf{x}(\tau, \mathbf{x}_0) = \mathbf{x}_0 + \int_0^{\tau} \mathbf{v}(t)dt \tag{1.3}$$

where $\mathbf{x}_0$ is the seed/start position of the massless particle and τ is the length of time for which the particle is traced. Equation 1.3 is a fundamental equation within the field of flow visualization. Solving this equation forms the basis for particle tracing which plays a part in many algorithms within flow visualization. CFD simulations produce complex output. Solving equation 1.3 analytically would be an extremely difficult task, due to the complexity of producing a continuous definition of a complex flow field. Instead, we rely on numerical methods to provide an approximate solution.

1.1.1 Numerical Integration

Many numerical techniques exist that can be utilized in order to solve Equation 1.3 with the simplest being Euler's method [PTVF02]:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}(\mathbf{x}_n)\Delta t \tag{1.4}$$

Euler's method is simple, to calculate the subsequent position we sample the field at the current position. This gives the direction of the next point, the distance travelled along this direction is given by the product of the magnitude of the sampled value and a user-defined step-size, Δt.

While Euler's method is simple, and fast due to sampling the vector field only once per integration step, it can quickly accumulate error. The error of Equation 1.3 is related to the step-size and is of the order $O(\Delta t^2)$ [Cd80]. Reducing the step-size reduces the error generated, however, a small step-size leads to a large number of particle positions being computed. These positions often directly translate to vertex positions when we generate curves and surfaces to visualize flow behavior. A large number of vertices increases memory requirements and puts a greater strain on the graphics hardware and in some cases may result in slower performance overall. Fortunately, more advanced (higher-order) numerical schemes have been developed that produce more accurate results. The family of Runge-Kutta methods are one such set of techniques. In contrast to the first-order Euler's method, Runge-Kutta methods perform more than one vector field evaluation. The second-order Runge-Kutta method (RK2) is given by:

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{v}(\mathbf{x}_n)\Delta t \\
\mathbf{k}_2 &= \mathbf{v}(\mathbf{x}_n + \frac{1}{2}\mathbf{k}_1) \\
\mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{k}_2
\end{aligned} \tag{1.5}$$

It is also known as the mid-point method [PTVF02]. Euler's method evaluates the vector field at the current position (the beginning) and applies the value across the entire interval. The mid-point method takes a small, intermediate Euler step to the mid-point of the interval. The vector field is then evaluated at this intermediate position. The evaluated value at the mid-point is then used for the entire interval starting from the initial position. The RK2 method has error of order $O(\Delta t^3)$ [Cd80]. Thus, for the cost of a single extra vector field evaluation allows us to use a larger integration step. Runge-Kutta methods also define higher order techniques such as the fourth-order Runge-Kutta method (RK4):

$$
\begin{aligned}
\mathbf{k}_1 &= \mathbf{v}(\mathbf{x}_n)\Delta t \\
\mathbf{k}_2 &= \mathbf{v}(\mathbf{x}_n + \frac{1}{2}\mathbf{k}_1) \\
\mathbf{k}_3 &= \mathbf{v}(\mathbf{x}_n + \frac{1}{2}\mathbf{k}_2) \\
\mathbf{k}_4 &= \mathbf{v}(\mathbf{x}_n + \mathbf{k}_3) \\
\mathbf{x}_{n+1} &= \mathbf{x}_n + \frac{1}{6}\Delta t(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)
\end{aligned}
\tag{1.6}
$$

The error for RK4 is of the order $O(\Delta t^5)$ [Cd80], this comes at the cost of three extra function evaluations compared to Euler's method. These integration methods have a fixed step-size. Another class of integration schemes utilize an adaptive step. Adaptive integrators, such as the Runge-Kutta-Fehlberg method [PTVF02] estimate the local error of a single step. This is done by using two integration methods for a single step, one of order p and the other with order $p-1$. The difference between both of these steps yields an approximation of the error. This error estimate can then be used to control the step-size. Looking at this naively, it appears that this is an expensive process due to performing two integration methods. However, the two integration methods are designed so that the lower-order method uses the same vector field evaluations as used by the higher-order method. Adaptive integrators may improve efficiency by allowing large steps in smooth regions and using smaller steps in regions where large variations arise over a short length.

1.1.2 Particle Tracing

Section 1.1 introduced particle tracing through Equation 1.3 and in Section 1.1.1 we described various numerical methods to solve this equation. However, in practice, more stages are involved in order to perform particle tracing on CFD simulation data. This is due to the discretely sampled CFD simulation grid. We now discuss this process in more detail. Note that when we mention a particle in a velocity field we refer to a massless particle (unless otherwise stated). This is an important aspect, a massless particle has no inertia and thus its velocity changes instantaneously with the velocity field. While some analytic data sets do exist for synthetic flow field examples, such as cases of Arnold-Beltrami-Childress flow [Hal05], the vast majority of flow data arises from simulation data and is given by a discrete set of velocity values. Due to the vector field being discretely sampled, more often than not we have to reconstruct the velocities between samples. This is done through the use of interpolation.

The particle tracing process for CFD simulation data comprises of the following steps [SvWHP94]:

- Point location
- Velocity interpolation
- Integration

The point location stage identifies which simulation grid cell the point lies in (this also tests whether the point actually lies within the simulation domain). The difficulty of this stage depends on the grid type. For a regular grid, simple modulo arithmetic computes the cell. Unstructured grids involve a more complex search and may utilize spatial partitioning structures and intersection tests. Once the cell has been found the field values are retrieved at its vertices. These are then used to interpolate the value at the point's position. Again, the interpolation implementation may depend upon the cell type. The next stage is the integration process and typically uses one of the techniques described in Section 1.1.1. Note that when using higher-order integration schemes, that the point location and interpolation phases need to be performed for every function evaluation that the chosen integration method uses. Thus, efficient integration is paramount when using irregular grids due to the more complex searching and interpolation processes. Particle tracing is a very important tool for flow visualization scientists and forms the back-bone of many methods. The curve- and surface-based methods described in the following section all utilize particle tracing.

1.1.3 Terminology

This section introduces some common, important flow visualization terminology. Vector fields which are used to describe the motion of a fluid are referred to as ***velocity fields*** or ***flow fields***. Velocity fields are described as being either ***steady*** (static) or ***unsteady*** (changing over time). There are a variety of techniques that are usually more suited to one temporal dimensionality over the other.

One of the most common techniques used to visualize steady flow fields is the streamline. A ***streamline*** is a curve that is everywhere tangent to the steady-state flow field. It depicts the trajectory of a massless particle within the flow. In 2D flows streamlines have the property of not being able to intersect each other. With careful placement streamlines form ***separatrices*** which partition the flow field into regions of distinct behavior.

Unsteady flow is generally more challenging than steady flow and generally results in larger simulation. A natural way of visualizing time-dependent phenomena is through animation, which explicitly shows the changes over time. However, the use of animation is not always ideal and sometimes not even an option. Animation sequences mean that the user has to spend more time watching the sequence. It is also possible for the user to miss short lived features in the sequence, and may have to re-watch several times. Also for certain presentation purposes (e.g., in a technical report), video may not be an option. This rules out the ability to use animation and would have to suffice with showing several individual frames from the sequence. Fortunately, there are visualization methods available that can depict the entire temporal domain without being restricted to using animation. Streaklines and pathlines, for instance, are computed from successive time steps together so that multiple time-steps may be displayed in a single static image. A ***pathline*** or particle trace is the trajectory that a massless particle takes in an unsteady fluid flow. A ***streakline*** is the line joining a set of particles that have all been seeded at the same spatial location (but at successive times). A streakline is analogous to dye-injection from experimental flow visualization. If seeded at the same location in a steady flow field streamlines, pathlines and streaklines are identical. A ***timeline*** is a line connecting a set of particles that are seeded from a curve.

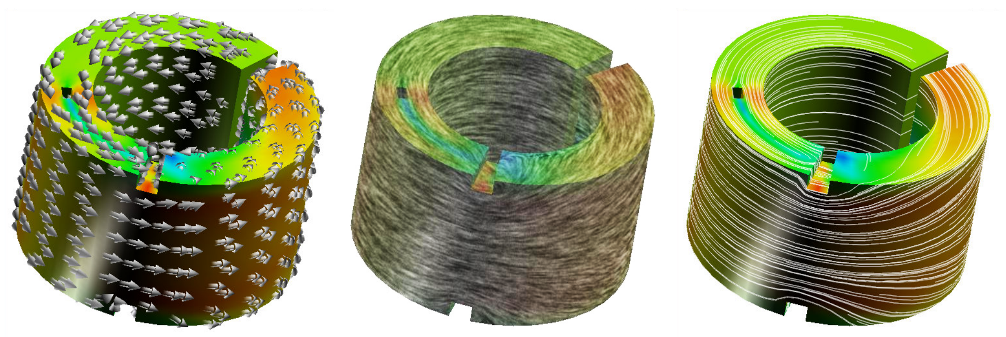


Figure 1.1: *A variety of flow visualization techniques to depict the flow on the surface of a ring. The left image is a direct visualization using arrow-glyphs. The center image utilizes a texture-based technique called line integral convolution (LIC) [CL93]. The right image is based on geometric objects. The lines shown are called streamlines and they show the tangential component of the vector field. The same data is used in all three images, only the visualization technique is different [Lar04].*

As their names imply, streamlines, pathlines, streaklines and timelines are based upon line primitives. Each of them also has a surface counterpart. A ***streamsurface*** is a surface that is everywhere tangent to the vector field, it is the locus of a set of streamlines from a shared seeding curve. Likewise, ***pathsurfaces*** and ***streaksurfaces*** are extensions of pathlines and streaklines are obtained by seeding from a curve instead of a point. A ***time-surface*** is the generalization of timelines, connecting particles that have been released from positions on a surface.

A ***critical point*** is a location in the velocity field where the velocity magnitude is zero. The behavior of the flow in the region around the critical point is used to classify its type. Some examples of critical points are ***sources***, ***sinks*** and ***saddle points***.

1.1.4 A Classification of Flow Visualization Approaches

A great variety of visualization techniques have been developed. The range of flow visualization techniques can be classified into four basic categories [PVH$^+$03]:

- Direct flow visualization
- Geometric flow visualization
- Dense, texture-based flow visualization
- Feature-based flow visualization

This classification is discussed in more detail in Chapter 2. Here, we provide common examples from these categories to give the reader an impression of common techniques from across the spectrum of flow visualization research. Figure 1.1 shows a collection of techniques from the direct, geometric and texture-based categories. They depict the flow upon the surface of a ring object. They use the same data but the resulting visualizations greatly differ. Figure 1.2

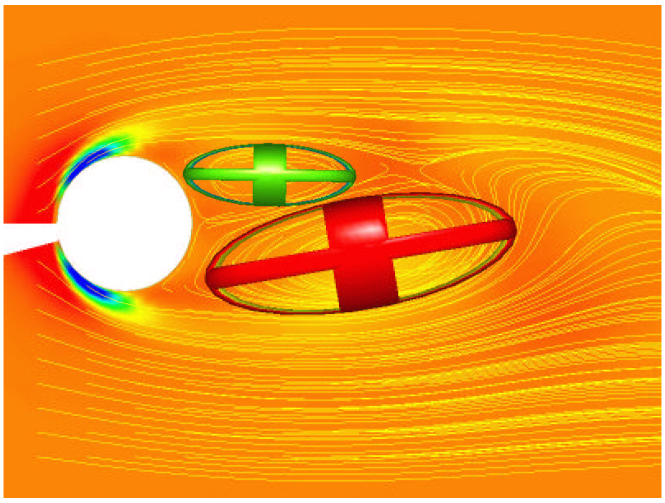


Figure 1.2: *An example of feature-based flow visualization. Vortices behind a tapered cylinder are highlighted using ellipsoids [SP99].*

shows an example of feature-based flow visualization. In this example the flow behind a tapered cylinder is studied. Two vortices (which are the features) are present. The vortices are highlighted using ellipsoids. Using the ellipsoids explicitly shows the vortices and prevents the user from having to manually analyze the streamline behavior. The body of research in this thesis is focused on surfaces for flow visualization and improving the user-experience when using line primitives. Thus, our work falls into the geometric-based category. A more detailed discussion of this classification scheme and of a further classification of the geometric-based techniques appears in Chapter 2.1.2.

1.1.5 Challenges of Geometric-based Flow Visualization

Flow simulations are computed using methods such as the Navier-Stokes equations [CSvS86] and are used to simulate experiments such as wind-tunnel tests on cars and airplanes. The visualization of these simulations poses many challenges, the most important of which we outline here.

Large Data Sets. A major technical issue arises from the sheer volume of data that may be generated from complex simulations. Velocity data comprises scalar values for each x, y, z velocity component at each sample point within the data domain. When coupled with several scalar data attributes and consisting of many time-steps, a large amount of data results. Advances in hardware lead to more computational power and the ability to process larger, more complex simulations with faster computation times. Therefore, flow visualization algorithms must be able to handle this large amount of data and present the results (ideally) at interactive frame rates in order to be most useful in the investigation and analysis of simulation data.

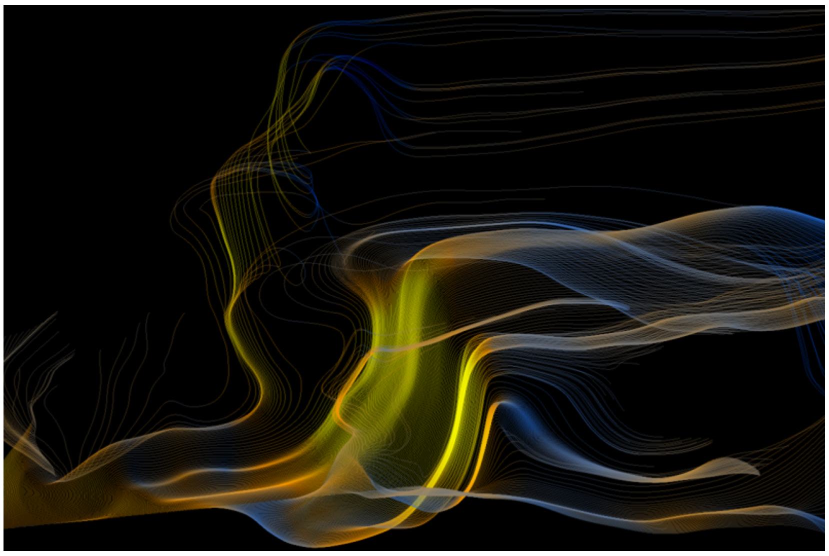


Figure 1.3: A set of streaklines exploiting the power of modern programmable GPUs for faster computation [DGKP09].

Interaction, Seeding and Placement. One of the main challenges specific to geometric flow visualization is the seeding strategy used to place the objects within the data domain. Geometric flow visualization techniques produce discrete objects whose shape, size, orientation, and position reflect the characteristics of the underlying velocity field. The position of the objects greatly affects the final visualization. Different features of the velocity field may be depicted depending on the final position and the spatial frequency of the objects in the data domain. It is critical that the resulting visualization captures the features of the velocity field, e.g., vortices, turbulence, sources, sinks and laminar flow, which the user is interested in. This aspect becomes an even greater challenge in the case of 3D where a balance of field coverage, occlusion, and visual complexity must be maintained. Time-dependent data also raises a challenge because the visualization then depends on when objects are seeded.

Computation Time and Irregular Grids. Another challenge stems from the computation time. Most of the visualizations compute a geometry that is tangential to the velocity field, e.g., the path a massless particle would take when placed into the flow. Computing such curves through 3D, unstructured grids is non-trivial. Thus much research has been devoted to this and similar topics. See Section 2.2.4. A recent trend to increase performance has been to move computations from the CPU and perform them on the graphics processing unit (GPU) [BSK$^+$07]. Although the resultant rendering looks the same as a CPU version (see Figure 1.3), this may offer a significant improvement in performance as the vector calculations are suited to the multiple execution units on the GPU – resulting in high performance parallel processing.

Perception. A central challenge in flow visualization (and visualization in general) relates to perceptual challenges in visualizing 3D and 4D velocity fields as well as multi-variate data sets.

If streamlines are used to visualize flow in 3D, too many lines causes clutter, visual complexity, and occlusion. If too few are rendered, important characteristics may not be visualized. Thus an optimal balance between coverage and perception must be achieved. Animated flow presents its own unique challenges to the perception of the user. For instance, it may not be intuitive what a user sees from a cloud of moving particles or a surface deforming to the local flow characteristics. It may also be difficult to discern the downstream direction in 3D space.

1.2 Motivation and Aims

The goal of flow visualization is to convey information about data which describes flow phenomena. This is usually obtained from CFD simulations. Interactive visualization techniques allow the user to explore the data in order to build their understanding of it and to focus on the regions they are most interested in. On the other hand, exploring data with no a priori knowledge can be time-consuming and unproductive. Manual exploration may also result in the user missing important behavior, thus the data may not be fully understood.

In order to increase its usefulness, it is also important for a visualization to portray the data in an informative way to the user. Through a review of the literature (see Chapter 2) we observe that the use of surfaces (as opposed to curves) aided the user by reducing visual clutter and improving depth perception in 3D flow fields. However, we also notice that, due to complexity of these algorithms and their limitations, there were few surface algorithms and that their use in visualization software was not common place – especially for time-dependent data.

These observations provide the motivation for this research. This work aims to investigate the problems with surface construction and provide an efficient algorithm that can be extended to time-dependent data. An investigation into improving interactive methods is also undertaken.

1.3 Contribution

The main contributions of the research are:

- A detailed overview of the current state-of-the-art in geometric, flow visualization methods and a novel classification of these methods [MLP$^+$09, MLP$^+$10].
- The development of a novel algorithm for the construction of stream surfaces and path surfaces [MLZ09, MEL$^+$11b] – surfaces that show the trace of a curve seeded in a time-independent or time-dependent vector field respectively.
- The development of a novel streaksurface algorithm [MLZ10, MEL$^+$11b]. Streaksurfaces emulate the behavior of dye injection from experimental flow visualization. They provide many unique challenges over streamsurface and pathsurfaces. Not the least of which is the computational expense involved. Streaksurfaces are a valuable tool for visualizing time-dependent simulations – particularly when their entire evolution is watched as an animated sequence.

- Visualizing the user-parameter space associated with manual streamline placement using seeding rakes [MEL$^+$11a]. Regions in which large changes to streamline geometry are highlighted. This brings several benefits such as guiding a user when placing streamline seeding rakes and the novel introduction of adjusting the inter-seed distance between streamlines along the seeding rake to ensure the flow behavior is captured in sufficient detail.
- A novel set of similarity measures for comparing streamlines [MJL11]. Our similarity metric combined with an efficient clustering algorithm lets the user the interactively customize the resulting visualization. A focus+context style approach allows the user to explore the behavior of a dense bundle of streamlines. Also, our filtering method allows us to reduce a large number of streamlines into a representative sub-set that preserves the main interesting flow characteristics. We also compare the performance of our similarity metric against the state-of-the-art from diffusion tensor imaging community and demonstrate performance gains of two orders of magnitude while providing similar quality.
- A comprehensive discussion of the design and implementation of our software framework [ML12]. A thorough discussion of design and implementation is usually omitted from research papers due to page length constraints. We provide details about the key systems and discuss the factors that influence our design.

1.4 Organization

The rest of the thesis is organized as follows: In Chapter 2 we present the state-of-the-art in geometric, integration-based flow visualization [MLP$^+$10]. A detailed overview of related research is presented. This extends the introduction to flow visualization given in Section 1.1.

In Chapter 3 we present a novel algorithm for the construction of streamsurfaces and pathsurfaces [MLZ09]. In Chapter 4 we discuss a novel streaksurface algorithm [MLZ10] for analysis of unsteady (time-dependent) flow simulations. Chapter 5 presents research into visualizing the user-parameter space associated with streamline seeding, guiding the user as to when and where to place the seeding rake throughout the entire spatio-temporal domain. Chapter 6 presents a novel set of similarity measures for comparing streamlines. The speed of our algorithm provides the user with a selection of tools in which to customize the resulting visualization.

Chapter 7 provides a detailed discussion about the design and implementation of the software and the user interface. Finally, Chapter 8 concludes the thesis and highlights areas of future work based on our work.

Effort has been made to design each chapter so that they can also be read individually without having to read the thesis in its entirety. Where required, in each chapter we have included an overview the most relevant related work to the chapter topic. This may include some of the literature discussed in the survey contained in Chapter 2, plus other work not from the field of flow visualization.

2

Over Two Decades of Integration-based, Geometric Flow Visualization

"*If you are going through hell, keep going.*"
-Sir Winston Churchill (1874–1965)[1]

Contents

[1]Former Prime Minister of the United Kingdom, known for his leadership of the United Kingdom during World War II.

2.1 Introduction

FLOW visualization is a classic branch of scientific visualization. Its applications cover a broad spectrum ranging from turbomachinery design to the modeling and simulation of weather systems. The goal of flow visualization is to present the behavior of simulation data in a meaningful manner from which important flow features and characteristics can be easily identified and analyzed.

Given the large variety of techniques currently utilized in visualization applications, selecting the most appropriate visualization technique for a given data set is a non-trivial task. Considerations have to be made taking into account the type of information the user wishes to extract from the visualization along with the spatial and temporal characteristics of the data set being analyzed. Different approaches have to be designed for different types of data. For example, visualization of 2D data is very different from visualizing 3D data. On top of this is the further complication of temporal dimensionality, with varying techniques more suited to steady flow compared to time-dependent flow fields and vice versa. To this end, many tools have been developed according to the differing needs of the users and the differing dimensionality of velocity field data.

2.1.1 Contributions

In light of these challenges and more than two decades of flow visualization research the main benefits and contributions of this chapter are:

- A review of the latest developments in geometry-based flow visualization research.
- The introduction of a novel classification scheme based on challenges including seeding. This scheme lends itself to an intuitive grouping of papers that are naturally related to each other. This allows the reader to easily extract the relevant literature without having to read the entire survey.
- A classification that highlights both unsolved problems in the area of geometric flow visualization and mature areas where many solutions have been provided.
- The most up-to-date presentation on this popular topic. The last time this topic was addressed in the literature was over six years ago [PVH$^+$03].
- The provision of a very concise introduction and overview in the area of vector field visualization for those who are new to the topic and wishing to carry out research in this area.

We have made a great effort not to provide simply an enumeration of related papers in integration-based, geometric flow visualization, but to compare different methods, related to one another and weigh their relative merits and weaknesses.

2.1.2 Classification

One of the main challenges of a survey is classifying these approaches and presenting them in a meaningful order. There are four general categories into which vector-field visualization approaches can be divided: *direct, dense texture-based, geometric, and feature-based*. This paper focuses on the geometric approaches to flow visualization, which has received little coverage in previous surveys [LHD$^+$04, PVH$^+$03, LHZP07]. A large volume of research work has been undertaken in geometry-based vector field visualization. We use four tiers of categorization. Our top level of classification groups the literature by the dimensionality of the object used in the resulting visualization, i.e., curves, surfaces and volumes. We then subdivide the literature further according to the spatial dimensionality of the data domain, i.e, 2D velocity fields, velocity fields on surfaces and in 3D volumes. Temporal dimensionality is also used to group papers together, i.e., steady vs. unsteady flow. And lastly, those papers belonging to the same sub-class appear chronologically (See Table 2.1). Classifying the literature in this way facilitates comparison of similar papers with one another. It also highlights unaddressed challenges and problems for which a range of solutions exist. We give a brief overview and comparison of the four main categories before analyzing the geometric approaches in more detail.

2.1.3 Direct, Texture-based, and Feature-based Flow Visualization

Direct techniques are the most primitive methods of flow visualization. Typical examples involve placing an arrow glyph at each sample point in the domain to represent the vector data or mapping color according to local velocity magnitude. Direct techniques are simple to implement and computationally inexpensive. They allow for immediate investigation of the flow field. However, direct techniques may suffer from visual complexity and imagery that lacks in visual coherency. They also suffer from serious occlusion problems when applied to 3D data sets.

Dense, texture-based techniques, as the name implies, exploit textures to form a representation of the flow. The general approach uses texture (generally a filtered noise pattern) which is smeared and stretched according to the local properties of the velocity field. Texture-based approaches provide a dense visualization result, provide lots of detail, and capture many flow characteristics even in areas of intricate flow such as vortices, sources, and sinks. Texture-based methods generally cover the entire domain. They also share some of the same weakness of 3D domain representation as direct methods and are generally more suited to 2D or surfaces. A thorough investigation of texture-based flow visualization is presented by Laramee et al. [LHD$^+$04].

Feature-based algorithms focus the visualization on selected features of the data such as vortices or topological information rather than the entire data set. This may result in a large reduction of the required data and thus these techniques are suited to large data sets that may consist of many time-steps. Since they generally perform a search of the domain, these techniques require considerably more processing before visualization. A survey of feature-based approaches is presented by Post et al. [PVH$^+$03].

Integration-based Geometric Object	Curves				Surfaces	Volumes
	2D	On surfaces	3D Particle Tracing	3D Rendering and Placement		
Steady Data Field	🟥 [TB96]	🟩 [vW92]$_p$	🟨 [BS87]	🟩 [HP93]	🟩 [Hul92]	🟩 [SVL91]
	🟥 [JL97a]	🟩 [vW93a]$_p$	🟨 [RBM87]	🟩 [ZSH96]$_p$	🟩 [vW93b]	🟩 [MBC93]
	🟥 [JL97b]	🟥 [MHHI98]	🟨 [Bun89]	🟩 [FG98]	🟩 [BHR$^+$94]	🟩 [XZC04]
	🟥 [JL01]	🟥 [SLCZ09]	🟨 [BMP$^+$90]	🟥 [MTHG03]	🟩 [LMG97]	
	🟥 [VKP00]	🟥 [RPP$^+$09]	🟨 [KM92]	🟩 [LWSH04]	🟨 [WJE00]	
	🟥 [LJL04]		🟨 [USM96]	🟩 [MPSS05]	🟨 [SBH$^+$01]	
	🟥 [MAD05]		🟨 [LPSW96]	🟩 [LGD$^+$05]	🟩 [GTS$^+$04]	
	🟥 [LM06]		🟨 [SvWHP97]	🟩 [LH05]	🟩 [LGSH06]	
	🟥 [LHS08]		🟨 [SdBPM98]	🟥 [YKP05]	🟩 [PS09]	
			🟨 [SRBE99]	🟥 [CCK07]	🟩 [BWF$^+$10]	
			🟨 [NJ99]	🟥 [LS07]		
			🟨 [VP04]			
Unsteady Data Field	🟥 [JL00]		🟨 [Lan93]	🟩 [BL92]	🟩 [STWE07]	🟩 [BLM95]
			🟨 [Lan94]	🟩 [WS05]	🟩 [GKT$^+$08]	
			🟨 [KL95]	🟥 [HE06]	🟩 [vFWS$^+$08]	
			🟨 [KL96]		🟩 [MLZ09]	
			🟨 [TGE97]		🟩 [KGJ09]	
			🟨 [TGE98]		🟩 [BFTW09]	
			🟨 [TE99]		🟩 [FBTW10]	
			🟨 [SGvR$^+$03]		🟩 [HGB$^+$10]	
			🟨 [KKKW05]			
			🟨 [BSK$^+$07]			

Table 2.1: *An overview and classification of integration-based geometric methods in flow visualization along the x-axis. Research is grouped based on the temporal dimensionality along the y-axis. Each group is then split into techniques that are applicable to steady or unsteady flow. Finally the entries are grouped into chronological order. Each entry is also colored according to the main challenge, as outlined in Section 1.1.5, that they address. The color coding scheme used is* 🟥 *red for seeding strategies,* 🟩 *green for techniques addressing perceptual challenges and* 🟨 *yellow for methods aimed at improving application performance. The subscript "p" indicates visualization using particles. This table provides an overview of research and highlights unsolved problems as well as challenges for which a range of solutions have been provided.*

2.1.4 Integration-based, Geometric Flow Visualization

Geometric methods define sets of seeding points from which trajectories (streamlines or pathlines) are computed. Trajectories are then used for building geometric objects, in contrast to other methods where they are used for filtering or advecting textures or for topological analysis.

Geometric approaches compute discrete objects within the data domain. Velocity, $\mathbf{v} = \frac{d\mathbf{x}}{dt}$, is a derivative quantity. If we imagine tracking a massless particle through a velocity field, the displacement of such a point can be described by:

$$d\mathbf{x} = \mathbf{v} \cdot dt \tag{2.1}$$

where $\mathbf{x}$ is the position of the point, t is the time and $\mathbf{v}$ is the velocity field. The analytical solution is approximated using a numerical integration method. Thus geometry-based techniques are also known as integration-based and characterize the flow field with their geometry. It is a non-trivial task to automatically distribute the objects such that all of the important features of the velocity field are captured in the resulting visualization.

Two main aspects of geometric flow visualization dominated research for a decade. In the first decade, the focus was on particle tracing, i.e., the numerical computation of trajectories for various types of data discretization. In the second decade, interest shifted to particle seeding strategies. Geometric visualization techniques are suited to all spatial and temporal dimensions. However, without careful use they are susceptible to visual clutter and occlusion problems. These problems mainly arise from poor seeding strategies and thus considerable effort has been put into researching seeding strategies that provide clear, detailed visualizations. We start out our survey of the literature with the point-based seeding algorithms in 2D velocity field domains.

Table 2.1 provides a concise overview of the literature grouped according to our classification scheme. Literature is divided up based upon both the dimensionality of the geometry-based objects in the domain and the dimensionality of the data domain itself. Organizing the literature in this way points out the mature areas where many solutions are offered and those areas still rich with unsolved problems.

2.2 Integral Curve Objects in 2D

All of the algorithms in this section use points to place streamlines in the vector field domain. Data is usually obtained from flow simulations. Most of the research here focuses on automatic seeding. However, interactive seeding strategies are possible, as demonstrated in[BL92][SGvR+03][BSK+07].

2.2.1 Streamlines in 2D Steady-State Flow Fields

Here we group those methods restricted to two-dimensional, steady state domains.

Streamline placement for 2D flow fields greatly affects the final image(s) produced by visualization applications. Streamlines that are seeded in arbitrary locations may provide an unsatisfactory result. Critical features in the flow field may be missed if there are regions containing only a sparse amount of streamlines. Conversely, where there is a large number of

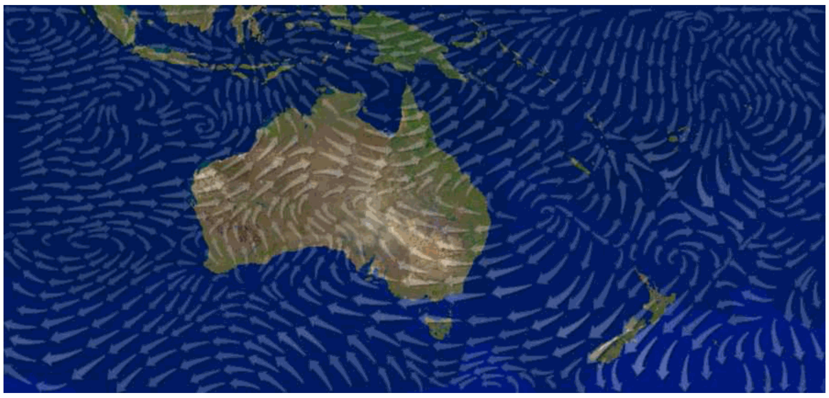


Figure 2.1: *Arrows showing the wind direction and magnitude over Australia. The arrows are placed along streamlines generated using the image-guided placement technique of Turk and Banks [TB96]. Image courtesy of Greg Turk.*

streamlines in a localized region, a cluttered image may result making it difficult to distinguish flow behavior.

An image-guided streamline placement algorithm was introduced by Turk and Banks in 1996 [TB96]. One of the goals of this algorithm is to produce visualizations similar to hand-drawn illustrations found in textbooks. Prior seeding algorithms were simply based on regular grids, random sampling or interactive seeding [BL92]. The seeding of the streamlines is influenced by the resultant image. The goal is to obtain a uniformly dense streamline coverage. It is formulated as an optimization problem where the objective is to minimize the variation of a low-pass filtered (blurred) image. Starting from a random initial streamline seed, the problem is solved by iteratively performing one of the operations *move* (displace a seed), *insert, lengthen, shorten* and *combine* (connect two streamlines with sufficiently close end points) on the set of streamlines. Figure 2.1 shows one result.

A follow-up technique is presented by Jobard and Lefer [JL97a]. The motivation is to introduce a new streamline seeding strategy that was computationally efficient and less costly than the previous streamline seeding strategy [TB96] and allows the user to control the density of the displayed streamlines. The authors introduce two user-controlled parameters d_{sep} and d_{test}. These parameters are used to control the distance between adjacent streamlines. Existing streamlines are used to seed new streamlines and candidate seed points are chosen that are at a distance $d = d_{sep}$, from a streamline. All candidate points of one streamline are used before moving on to the next streamline. This process stops when no further data points are generated. The d_{test} parameter is a proportion of d_{sep}. d_{test} is used to control the closest distance that streamlines are allowed to one another. Sufficient coverage (i.e., a minimum density) is ensured by seeding streamlines at a distance d_{sep} from one another. The method was also combined with texture advection techniques [JL97b] for animating steady flow fields.

Jobard and Lefer [JL01] introduce a novel algorithm that produces images of a vector field with multiple simultaneous densities of streamlines. The paper builds upon the previous technique [JL97a]. The multi-resolution property is ideal for vector field exploration as it allows for sparse streamline placement for a quick overview of the vector field, the streamline density can then be increased to allow for a more detailed investigation into areas of interest. They also demonstrate the use of the technique for zooming and enrichment.

Lefer et al. provide a novel technique for producing variable-speed animations [LJL04]. This method encodes the motion information in a so-called motion map and a color table is utilized to animate the streamlines. Once streamlines have been computed they remain valid for all frames due to the steady vector field, so the challenge of animating the streamlines is simplified to a coloring the streamlines appropriately. The algorithm begins by creating a dense set of streamlines that cover every single pixel. Animating the streamlines is achieved by shifting the color table entries so that the color pattern appears to travel down the streamlines. The local velocity magnitude is taken into account when computing how fast to move the color pattern.

Verma et al. present a novel method of streamline placement that focuses on capturing flow patterns in the vicinity of critical points [VKP00]. Templates are defined for types of critical points that may be present in 2D flow fields. The algorithm begins by determining the location and type of critical points in the field. Verma et al. use Voronoi partitioning around the critical points that contain regions that exhibit similar flow behavior. A random Poisson disk seeding strategy is finally used to populate streamlines in sparse regions. Seeding in regions of critical points first ensures that they are covered by a sufficient amount of streamlines and that these streamlines have a longer length. In this implementation FAST [BMP$^+$90] is used to compute critical point locations and determine their nature, Voronoi diagrams are computed using *triangle* [She96].

The work of Mebarki et al. [MAD05] builds upon previous research by Turk and Banks [TB96] and Jobard and Lefer [JL97a]. The results produced are of comparable quality to the work of Turk and Banks [TB96] while being produced faster [MAD05]. The algorithm uses a farthest seeding point strategy. Roughly speaking, when a new streamline is created, the point furthest away from all current streamlines is used as the seed point for the subsequent streamline. Using a farthest point seeding strategy ensures that long streamlines are produced. To determine the farthest point, the points of the streamlines are inserted in a 2D Delauney triangulation. The incident triangles of a newly integrated point are used to generate a minimal circumdiameter. Any diameter that is above the desired spacing distance and below a saturation level is pushed onto a priority queue that is sorted by length. The top circle is then popped out of the queue, the center may then be used as the seeding point for the next streamline.

The work of Liu et al. [LM06] introduces another evenly-spaced streamline algorithm. It builds upon the work of Jobard and Lefer [JL97a] and Mebarki et al. [MAD05]. Cubic Hermite polynomial interpolation is used to create fewer evenly-spaced streamline samples in the neighborhood of each previous streamline in order to reduce the amount of distance checking. Placement quality is enhanced by double queues to favor long streamlines (this minimizes discontinuities). The presented method is faster than that of Jobard and Lefer [JL97a]. In addition, it incorporates the detection of streamline loops.

Li et al. [LHS08] introduced a novel method for streamline placement, which is different from its predecessors in that its goal is to generate the fewest number of streamlines possible while still capturing the most important flow features of the vector fields (see Figure 2.2). The images produced use a small amount of streamlines and are intended to be similar to hand-drawn diagrams. This is achieved by taking advantage of spatial coherence and by using distance fields to determine the similarity between streamlines. New streamlines are only created when they represent flow characteristics that are not already shown by neighboring streamlines. This way repetitive flow patterns are omitted. Similarity between streamlines is measured locally by the directional difference between the original vector at each grid point and an approximate vector derived from nearby streamlines, and globally by the accumulation of local dissimilarity at every integrated point along the streamline path.

Reflection: The two most notable contributions to streamline seeding in planar flows are by Turk and Banks [TB96] and Jobard and Lefer [JL97a]. Turk and Banks are the first to introduce the notion of a user-controlled spatial frequency for streamlines, while Jobard and Lefer accelerate the idea to fast rendering times. The contributions that follow are all variations on these two themes. Overall, we believe the challenge of streamline seeding in 2D steady flow to be a solved problem.

2.2.2 Integral curves in a 2D, Time-Varying Domain

Jobard and Lefer extend their evenly-spaced streamline technique [JL97a] to unsteady flow [JL00]. Streamlines across several time-steps represent the global nature of the flow at each step and give insight into the evolution of the vector field over time. However, simply generating a set of streamlines at each step and cycling between them leads to an incoherent animation. The authors present a set of parameters that are used to choose a suitable set of streamlines for the next time-step using the current set of streamlines as a basis. A so-called feed forward method is used which selects an appropriate subset of streamlines from the subsequent time-step that correlate with the current set. A technique is employed that quantitatively evaluates the corresponding criterion between streamlines. The best candidates are then used for the next time-step. Several methods are used to improve the animation quality, such as giving priority

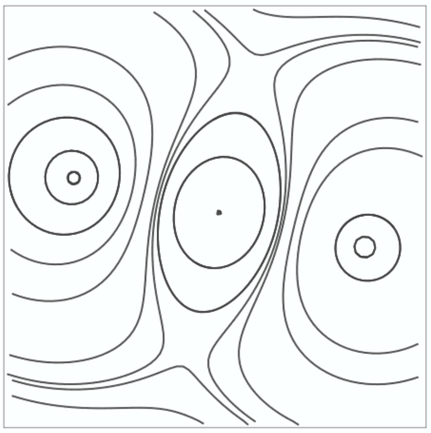


__Figure 2.2:__ A representative set of streamlines generated by the Illustrative Streamline Placement algorithm [LHS08]. Image courtesy of Han-Wei Shen.

to circular streamlines and adding tapering effects to the streamlines. A cyclical texture is also applied and this is animated to indicate the downstream direction of the flow on the streamlines.

We have not discovered any literature describing the solution to explicit pathline or streakline seeding algorithms for 2D, unsteady flow. This is still an open challenge.

2.2.3 Streamline Seeding on Surfaces

In practice, most vector field domains consist of either 2 or 3 spatial dimensions. Some approaches are more suitable for one spatial dimension over the others. Typically, as we move from 2D to 3D, the complexity of algorithms increase. This is due, in part, to the effort required to minimize visual clutter and occlusion and, to the extra complexity of another spatial dimension.

A novel visualization scheme based upon a particle system is introduced by Van Wijk [vW92]. The particles can be seeded from a variety of geometric objects. The most obvious objects are: points, lines, circles, rectangles and spheres. The sources also have a temporal attribute too, the particles can be injected at discrete time pulses or as a continuous stream. A continuous point source will result in streamlines being created by the particles and a stream surface will be created if a curve-based continuous source is used. The particles have a normal, which allows the lighting equations to be used in order to apply shading and provide greater depth cues. A Gaussian filter is used to smooth the visualization, softening aliasing and strobing artifacts.

Van Wijk [vW93a] builds upon his previous work in 1992 [vW92]. Here an improved shading model is used to reduce the aliasing and strobing artifacts that were found in his previous work. A more detailed discussion of the seeding objects is also presented detailing the flexibility of using the surface particles to emulate an array of visualization techniques such as streamlines, stream surfaces and stream tubes. We classified the work of Van Wijk [vW92, vW93a] into point-based objects on a surface domain because the focus of the research is on how to effectively render particles on stream surfaces.

Mao et al. [MHHI98] present an evenly-spaced streamlines technique for curvilinear grids. They expand upon the work of Turk and Banks [TB96] by applying the seeding strategy to parameterized surfaces. This algorithm takes the vectors from the 3D surface and maps them to computational space. An extended 2D image-guided algorithm is then applied and streamlines of a desired density are generated. The streamlines are then mapped back onto the 3D curvilinear surface. However, curvilinear grids cells can vary significantly in size. This means that streamlines distributed evenly in computational space won't necessarily be evenly spaced when they are mapped back to physical space. This challenge is overcome by altering the computational-space streamline density. The streamline density is locally adapted to the inverse of the grid density in physical space [MHHI98]. This is achieved by using Poisson ellipse sampling which distributes a set of rectangular windows in computational space.

Spencer et al. [SLCZ09] extend the 2D evenly-spaced streamlines technique by Jobard and Lefer to surfaces, Figure 2.3 shows evenly-spaced streamlines on the boundary surface of a gas engine simulation. It starts by projecting the vector data onto the image plane, by rendering a so-called velocity image into the frame buffer. Performing the streamline computations in image-space effectively reduces the complexity of the 3D problem into a 2D one. This

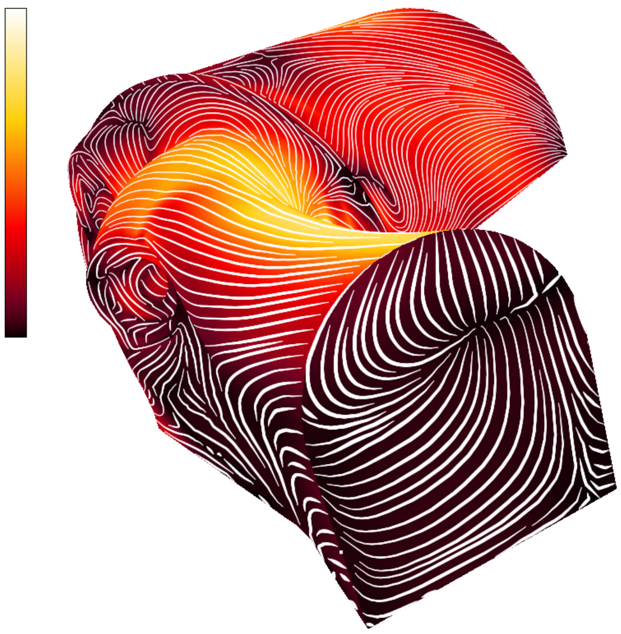


Figure 2.3: Evenly-spaced streamlines on the boundary surface of a gas engine simulation. Perspective foreshortening is utilized and the density of streamlines further away from the viewpoint is increased [SLCZ09].

also has the advantage of a simpler implementation using the programmable portions of the graphics pipeline and inherently takes advantage of the hardware interpolators. Another major performance benefit arises from the z-buffer and frustum culling. These discard any occluded fragments and clipped geometry, this prevents unnecessary streamline computations on areas that aren't visible to the user. A combination of seeding strategies is employed. A grid traversal strategy, checking each cell as a candidate seeding position is used in combination with seeding based on current streamlines, like Jobard and Lefer's algorithm [JL97a]. This ensures that all visible regions of the geometry are processed.

More recently Rosanwo et al. [RPP+09] provide a solution to streamline seeding based on dual streamlines. *Dual streamlines* run orthogonal to the vector field instead of tangential. Two sets of streamlines are maintained, a set of tangential streamlines, S, and a set of dual streamlines, D. This is similar to the technique used by Mebarki et al. [MAD05] where the largest voids are found in order to place a new seed. Streamlines are seeded in a similar way using the dual streamline segments. As this method only uses the arc-length of distance metric it may be efficiently applied to curved surfaces where other distance metrics such as geodesic distance are more computationally expensive and/or are hard to apply correctly. This algorithm requires that a suitable starting set of streamlines are used in order to be efficient and to ensure complete domain coverage and critical points. This is achieved by computing the topological skeleton of the dual field as the initial dual streamline set. In cases where the vector field contains no topology randomly seeded dual streamlines are seeded. This method shows a slower growth in computation time when increasing the streamline density compared to algorithms by Verma et al. [VKP00] and Mebarki et al. [MAD05].

Reflection: An important solution to the challenge of streamline seeding on surfaces comes from Spencer et al. [SLCZ09]. This is the only solution of its kind that handles general surfaces and unstructured, adaptive resolution meshes. It is also a fast algorithm that supports exploration through user-interaction. Pathline and streakline seeding on surfaces remains an open challenge however.

2.2.4 Efficient Particle Tracing in 3D

This subsection summarizes particle tracing strategies in 3D space. The volume cell types used in simulations vary. Depending upon the model used to generate them. The simplest grid type is a Cartesian grid. Curvilinear grids, commonly used in flow simulations, contain the same cell type but the grid is usually distorted (usually curving) so that it fits around a geometry. Unstructured data may contain several different cell types, tetrahedra and hexahedra are commonly used. Unstructured grids are generally more challenging than structured grids and some algorithms are specifically aimed at particle tracing solutions on them. Computations on unstructured grids may either be performed in physical-space or computational-space on a per-cell basis. Physical-space uses the velocity field and grid as output from the simulation. Computational-space transforms a grid cell to make it axis-aligned and unit length, and adjusts the velocities accordingly. Computational-space methods are used to simplify certain operations such as point-location.

This collection of papers focuses on computational methods, addressing the challenge of providing fast, accurate results that can be utilized by other visualization methods to improve their performance. This is in contrast to the other methods that directly provide novel visualizations. The forerunners to these techniques along with some of their applications can be found in [BS87, RBM87, Bun89].

2.2.4.1 3D Particle Tracing in Steady Vector Fields.

PLOT3D [BS87, WBPE90] is a command line driven program for displaying results of CFD simulations on structured and unstructured grids. Besides a wide range of graphics functionality, e.g., hidden line and hidden surface techniques, PLOT3D offers 2D and 3D streamlines of the velocity field, the vorticity field (vortex lines), and the wall shear stress field (skin friction lines). The software was designed to run on supercomputers, e.g., for computing movies, but also on the first graphics terminals and workstations with hardware supported viewing transformations. PLOT3D was the precursor of FAST (Flow Analysis Software Toolkit) [BMP$^+$90], a modular redesign which added a GUI and distributed processing. Visual2 [GH90] and Visual3 [HG91] were packages written by Haimes and Giles for the visualization of 2D and 3D flow fields. They provided interactive X-windows based visualization of steady or unsteady flow fields given on unstructured grids. Visual3 was later adapted to network computing and renamed to pV3 [Hai94]. The techniques for vector fields available in Visual3 include streamlines and variants such as ribbons and tufts (or streamlets).

Ueng et al. [USM96] present an efficient method of streamline construction in unstructured grids. This method uses calculations performed in computational-space to reduce computational cost of the streamline generation. To perform the calculation in computational-space the physical-space coordinates of a cell and its corresponding vector data must be transformed into canonical coordinates. A cell searching strategy similar to that used in [KL96] takes advantage of the canonical coordinates to simplify and speed up the operation. A specialized Runge-Kutta integrator is also presented for use in the canonical coordinate system which and offers improved computation times compared to the second- and fourth-order Runge-Kutta integrators that perform in physical space.

The techniques used for **streamtube** and **streamribbon** construction are also described by Ueng at al. [USM96]. Streamtubes are created by placing circular curves, oriented normal to the flow, at the streamline integration points. The circular cross sections are then connected to form an enclosed tube object. The radius of the streamtube illustrates the local cross flow divergence and is calculated at each streamline point, i.e., when the circular glyphs are created. Streamribbons are created using the streamline for one edge and then using a constant length normal vector generate the position of the opposite edge. The constant length normal rotates around the initial streamline in order to depict local flow vorticity.

UFLOW is a system introduced by Lodha et al. [LPSW96] to analyze the changes resulting from different integrators and step-sizes used for computing streamlines. A pair of streamlines are interactively seeded by the user and each streamline is generated using a different integrator or integration step-size. It is also possible to create a single streamline and then trace it backwards from its end point and compare it to the initial streamline (See Figure 2.4).

Sadarjoen et al. present a comparison of several algorithms used for particle tracing on 3D curvilinear grids [SvWHP97]. The particle tracing process is broken down, with a brief description, into basic components: *point-location*, locating which cell a point is in, *interpolation*, and *integration*. A more thorough discussion and comparison of *physical-space* and *computational-space* algorithms then ensues. Results for the implemented algorithms are also given showing that *physical-space* computation algorithms generally perform better than their *computational-space* counterpart.

Sadarjoen et al. [SdBPM98] present a 6-tetrahedra decomposition method for σ-transformed grids. σ-transformed grids are structured hexahedral curvilinear grids in which the x and y dimensions differ by 2-3 orders of magnitude from the z dimension, thus resulting in very thin cells. The method presented here is more accurate and allows for faster operations to be performed than the more common 5-tetrahedra decomposition that is usually employed [SvWHP97]. Decomposing the hexahedral cells into 6 tetrahedra prevents a center tetrahedron covering the center of the cell and thus makes point location much easier. This method also reduces the chances of infinite loops between two cells when using the 5-decomposition approach [SdBPM98].

Schulz et al. [SRBE99] present a set of flow visualization techniques that are tailored for PowerFLOW, a lattice-based CFD simulator. The grids in these simulations are multi-resolution Cartesian grids, where finer voxels are used in areas of interesting flow or boundary surface geometry. Particle tracing and collision detection are discussed, demonstrating the need for collision detection between the particle and the object surface. When a collision occurs the particle tracing for that line may either terminate or follow a path along the object boundary. The system has several seeding types, that can be interactively manipulated, ranging from rakes, planes and cubes.

Nielson et al. introduce efficient methods for computing tangent curves for three-dimensional

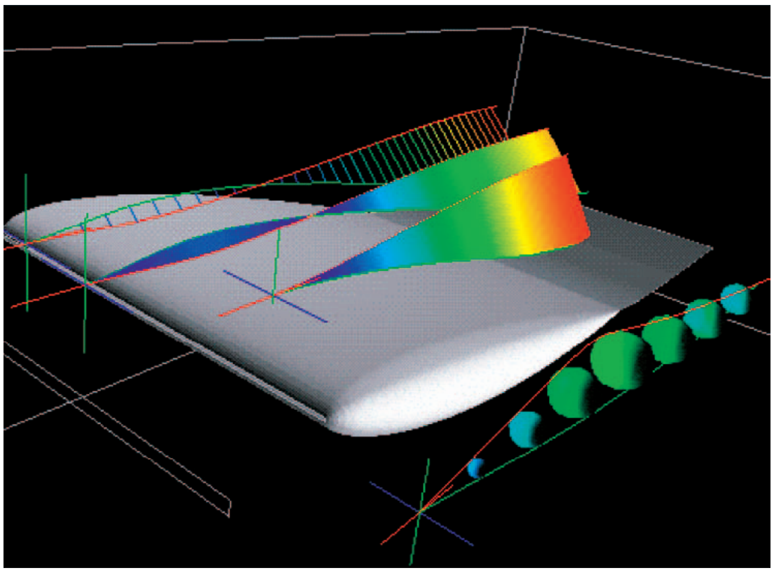


Figure 2.4: *Comparing streamlines of two datasets simulated using different turbulence models. The streamlines are compared using line glyphs, strip envelopes, and sphere glyphs to highlight the differences between them. Image Courtesy of Alex Pang [VP04].*

flow fields [NJ99]. This technique is an extension to 3D of their previous research [NJS$^+$97]. The techniques are designed to be used on tetrahedral grids. Incremental methods are used for stepping along the analytic solution of the streamline ODE and as a result produce exact results. Techniques for both Cartesian and barycentric coordinates are presented, allowing the user to use the tools for the coordinate system that is most suited to the current application. Several cases are defined based on the types of the eigenvalues found at a particular point, these in turn are used for the calculation of a tangent curve through a tetrahedron. Results are presented that compare the accuracy of the presented algorithms compared to Euler and fourth-order Runge-Kutta integrators.

Verma and Pang present methods for comparing streamlines and streamribbons [VP04] (Figure 2.4), and some of their methods are loosely based on those that appear in the UFLOW system [LPSW96]. Large CFD simulations are generally run on supercomputers, however the applications used to visualize these simulations are generally run on workstations. Some of these simulations have to be approximated with smaller data sets to make their use on workstations more feasible. Different datasets are compared simultaneously, with the second dataset being a subsampled version of the first dataset. A metric for measuring difference is needed and here the Euclidean distance between associated streamline points is used. Associated points are connected by lines, giving a ladder effect, which aids the visual representation of the differences between the streamlines. Strip envelopes, which fill in the ladder sections and spheres are also used to depict the difference when comparing a pair of streamlines.

Reflection: We consider the challenge of particle tracing to be solved for the case of steady-state structured grids only. This is because many streamlines can be traced interactively for steady-state fields. The same cannot be said for large unstructured grids however. Tracing many streamlines (hundreds) tends to be non-interactive. This is still and unsolved problem.

2.2.4.2 3D Particle Tracing in Unsteady Vector Fields.

Lane introduces a system for using streaklines (refer to Section 2.2.2 for the visualization of unsteady flows) [Lan93]. Lane presents the numerical background for particle integration over many time-steps as well as integration over simulations that involve a moving grid, a feature demonstrated in very few systems. The two datasets visualized are in excess of 15GB and 64GB including both their grids and solutions. Seeding points are positioned manually. Lane shows that only two time-steps need to be loaded at once to perform integration for one step, thus enabling this technique to be applied to large datasets. The tools in this application build upon similar systems such as the Virtual Wind Tunnel [BL92] (see also Section 2.2.6).

UFAT [Lan94] is a system that is used to generate streaklines on datasets with a large number of time-steps. One of the major challenges of unsteady flow is the size of the datasets that may be produced. The size makes them difficult to store in memory. This is also true when a time-dependent grid is used. Examples of moving grids are shown, such as engine cylinder simulations with a moving piston, and turbine simulations with rotating blades. A second-order Runge-Kutta with adaptive step size is used to advect the particles through the flow field. The system stores the streaklines at each time on disk so that they can be recovered without re-computation and used to create an animation. This system builds upon work done on systems such as the Virtual Wind Tunnel [BL92], pV3 [Hai94] and FAST [BMP$^+$90].

Kenwright and Lane [KL95, KL96] present methods that increase the efficiency of particle tracing for simulations on curvilinear grids. Many simulations output the vector data on curvilinear grids. In this case, particle tracing can be calculated in physical space, i.e., on the curvilinear grid in its original state, or in computational space, which transforms the curvilinear grids coordinates into Cartesian space. Calculations in computational space are easier to perform but tend to be less accurate due to the vector field transformation using approximated Jacobian matrices. Physical space computation is more accurate but point location, can be an expensive operation if done naively (e.g., a brute force linear search in every cell). The authors overcome this barrier by implementing a more efficient point-location strategy for tetrahedral grids.

Teitzel et al. [TGE97] describe an analysis of integration methods used in scientific visualization. The integration methods investigated are both adaptive and non-adaptive Runge-Kutta integrators of orders 2, 3 and 4. A robust integration scheme is found by establishing the link in numerical errors between the integration method and the linear interpolation of the vector field values between the discretely sampled grid points. Their approach is shown to be more efficient than that of [BLM95] and [KL95]. The authors also describe implicit integration methods for use in stiff problems (areas of strong shear or vorticity).

Teitzel et al. [TGE98] introduce a particle tracing method for sparse grids built upon their previous work [TGE97]. The main difficulty in this task is the interpolation operation to find the vector values along an integral path. On a full grid the tri-linear interpolation is done as a local operation. To help with the efficiency the authors have used an array to store the contributing coefficients of the sparse grid as values can be accessed directly. Functions are added to calculate the contributing samples and to accumulate them over the different levels of the sparse grid. The flow is visualized with color-coded streak balls, streak tubes and streak bands (or ribbons). **Streak balls** follow the same path as streaklines, however, the spacing between objects depicts acceleration and the size of the ball depicts local flow convergence and divergence. **Streak tubes** (a derivative of stream tubes) use a closed-curve seeding object resulting in a tube that follows a streakline path. The diameter of the tube depicts flow divergence and convergence. A **streak band** uses a short line segment as a seeding object. This results in a ribbon when traced through unsteady flow whose twisting depicts the vorticity (or swirling motion) of the flow.

Teitzel et al. [TE99] also introduce an improved method to accelerate particle tracing on sparse grids and introduce particle tracing on curvilinear sparse grids. An adaptive evaluation of the sparse grids is implemented. This is achieved by omitting contribution coefficients with a norm below a given error criterion during the interpolation process. The combination technique is also used to improve the efficiency. Streaklines, streak balls and streak tetrahedra are used to visualize flow on curvilinear sparse grids. **Streak tetrahedra** attempt to combine the advantages of streaklines, ribbons, tubes, and balls. The displacement of the tetrahedra along a streakline path depicts acceleration, rotation depicts vorticity, and volume reflects convergence and divergence.

Reflection: We consider the challenge of particle tracing in 3D, unsteady vector fields to still be a challenge for the case of unstructured grids. Tracing many integral curves is generally still not interactive from a performance point of view.

2.2.5 Streamline Rendering and Placement in a 3D Steady-State Domain

This section surveys streamlines used to visualize 3D vector fields. Here, the challenges are perceptual. Rendering too many field lines results in clutter, complexity, occlusion, and other perceptual problems. Rendering too few field lines may lead to missing important characteristics of the data. Conveyance of depth and spatial orientation are also challenges.

In 1993, Hin and Post introduce a method for depicting turbulent flow using a particle system [HP93]. Turbulence is a common feature of flow fields, however, there are relatively few techniques that are specifically focused on this flow feature. Turbulence is modeled on Reynolds' decomposition [Rey95], which expresses turbulence of flow into mean flow and fluctuation, where the fluctuation represents local turbulent motion. This was implemented using a stochastic process whereby a compound velocity was composed of the mean velocity and a random perturbation generated using random-walk models. Tracing the random-walk particles over many steps leads to an effect representing turbulent behavior. The seeding of particles is based on a uniform Cartesian grid aligned with the domain boundary.

Zöckler et al. introduce a method of illuminating streamlines [ZSH96]. Graphics APIs such as OpenGL support hardware acceleration for lighting when applied to surface primitives. OpenGL uses the Phong reflection model which typically uses the orientation of the surface (i.e., its normal) with respect to the light direction and the viewing angle. However, there is no native support for the lighting of line primitives in these libraries, due to the fact that line primitives have no unique normal vector.

From the set of possible normal vectors, the method chooses the ones that maximize diffuse and specular reflection, respectively. For this, two products $t_1 = \mathbf{L} \cdot \mathbf{T}$ and $t_2 = \mathbf{V} \cdot \mathbf{T}$ are computed from the light, tangent and view unit vectors $\mathbf{L}, \mathbf{T}$ and $\mathbf{V}$ on the vertices. By using specially constructed textures and t_1 and t_2 as texture coordinates, diffuse and specular terms are obtained per pixel.

A streamline placement algorithm is also introduced. For the placement technique a stochastic seeding algorithm is applied. The degree of interest in each cell is defined on some scalar value (i.e., velocity magnitude). An equalization strategy is then employed to distribute the seed points more homogeneously. See Weinkauf et al [WT02, WHN$^+$03] for applications of this seeding strategy.

Mattausch et al. [MTHG03] combine the illuminated streamlines technique of [ZSH96] with an extension of the evenly-spaced streamlines seeding strategy of Jobard and Lefer [JL97a] to 3D. With the 2D version of evenly-spaced streamlines presented by Jobard and Lefer [JL97a] the d_{sep} parameter is used in connection to the current streamline point for the candidate streamline seed points. In 2D there are only two possible positions for this new candidate seed position (one on either side of the streamline). When this is extended to 3D there are an infinite number of positions around a line at an orthogonal distance of d_{sep}. The authors simplify the extension to 3D by defining six points around a streamline that may be used for the candidate seed point generation.

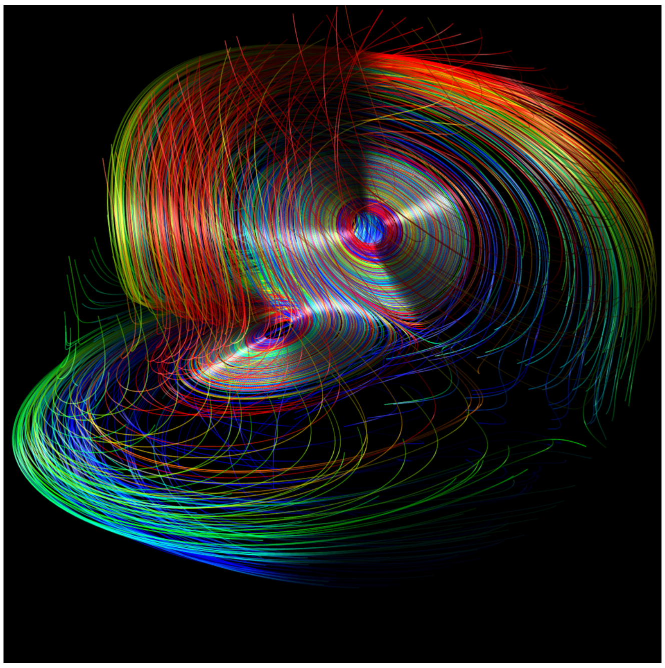


Figure 2.5: *A Lorenz attractor visualized using streamlines. The streamlines are illuminated using [MPSS05] by Mallo et al.*

Mallo et al. present an improved illuminated lines technique [MPSS05]. This method builds upon the previous illuminated lines by Zöckler et al. [ZSH96] and the cylinder averaging technique presented by Schussman and Ma [SM04]. This method calculates the diffuse and specular components of lighting from the infinitesimal facets of a cylinder. The authors take advantage of programmable GPUs and implement shader programs. This technique improves upon Zöckler et al's technique which used maximal reflection due to the fact that the maximal reflection technique produces bi-directional lighting. The cylinder averaging technique does not produce bi-directional lighting and thus provides clearer orientation and depth information without having to use a strong specular component. Figure 2.5 shows an example of illuminated streamlines.

Fuhrmann and Gröller [FG98] present a technique for virtual environments that aims to reduce perceptual problems in visualizing 3D data such as occlusion and visual clutter. The concept of a **dashtube** is introduced. A dashtube is an animated, opacity mapped streamline. The dashtubes are seeded using a straightforward extension of the evenly-spaced streamlines algorithm [JL97a] to 3D. For simplicity the tube portions are set to either being fully opaque or fully transparent. The opacity mapping is achieved using textures with animation taking place in texture space to improve efficiency and ease of implementation. This method, like most texture based algorithms, can suffer from aliasing problems. The authors present two methods for resolving this. The first method is a variation of well known mip-mapping, which instead of filtering the mip-maps, produces sub-maps. The second method uses a texture with bands of varying sizes for different sized regions on the streamline (regions further away from the user appear smaller). The authors also present focus and context techniques: magic lenses and magic boxes. The region within the lens contains a higher density of dashtubes and allows the user to investigate selected areas in more detail. The magic box shows a discrete volume which forms the focus and works on the same principle as the lens while allowing the user to change viewing position and orientation.

Laramee and Hauser present a set of geometric visualization techniques including the introduction of two novel approaches: the streamcomet and a fast animating technique [LH05]. These techniques are demonstrated in the context of CFD simulation data. Oriented streamlines improve upon standard streamlines by depicting the downstream direction of the flow in a static image. Animation of streamlines is achieved by a stipple pattern. The streamcomet is a metaphor that offers a large amount of flexibility and interaction from the user. A streamcomet is comprised of a head section and a tail section.

Ye et al. [YKP05] present a method for streamline placement in 3D flow domains. This paper addresses the common goals of streamline placement, namely, the generation of uncluttered visualizations, and sufficient coverage of the domain to ensure that all important features are captured by the visualization. Conceptually, this algorithm can be viewed as an extension of Verma et al's method [VKP00] to 3D.

This approach scans the vector field for critical points and extracts them, identifying important areas of interest. Different seeding templates are defined a priori and positioned around the vicinity of critical points. This approach also contains an operation which detects the proximity of one critical point to another. A proximity map is then used to merge the two most appropriate templates. Poisson sphere seeding is used to add streamlines to regions of low streamline density. Filtering of the streamlines is then used to remove redundant streamlines and to avoid visual clutter. The filtering process is multi-staged and considers both geometrical and spatial properties. First the streamlines with short lengths and small winding angles are removed. The next step considers the similarity of the remaining streamlines. The streamlines are ranked in order of winding angle. The distance between endpoints and centroids of streamlines with similar winding angles are then considered. If the distance is below a predefined threshold then one of them is filtered out.

Chen et al. [CCK07] present a novel method for the placement of streamlines. Unlike many other streamlines placement methods this technique does not rely solely on density placement or feature extraction. Streamline generation methods relying on a density measure may contain redundant streamlines. Strategies based on the extraction of critical points in the field require binary filtering of data based on whether or not they describe a feature. This approach is based on a similarity method which compares candidate streamlines based on their shape and direction as well as their Euclidean distance from one another.

Li et al. [LS07] present a streamline placement strategy for 3D vector fields. The motivation is drawn from the fact that streamlines that are generally well-organized in 3D space may still produce a cluttered visualization when projected to the screen. This is the only approach of its kind – where an image-based seeding strategy is used for 3D flow visualization. The approach presented here places the streamline seeds in image-space and then unprojects them back onto object-space. The first stage of this algorithm is to randomly select a seed point on the image plane for the initial streamline. This position is then unprojected back into object space. Switching between image-space and object-space is made possible by exploiting a depth map. Once the initial seed has been placed, the streamline is integrated and placed into a queue. The oldest streamline is then removed from the queue and used to generate a new seed position for another streamline. There are two candidate positions for the seeds, one on either

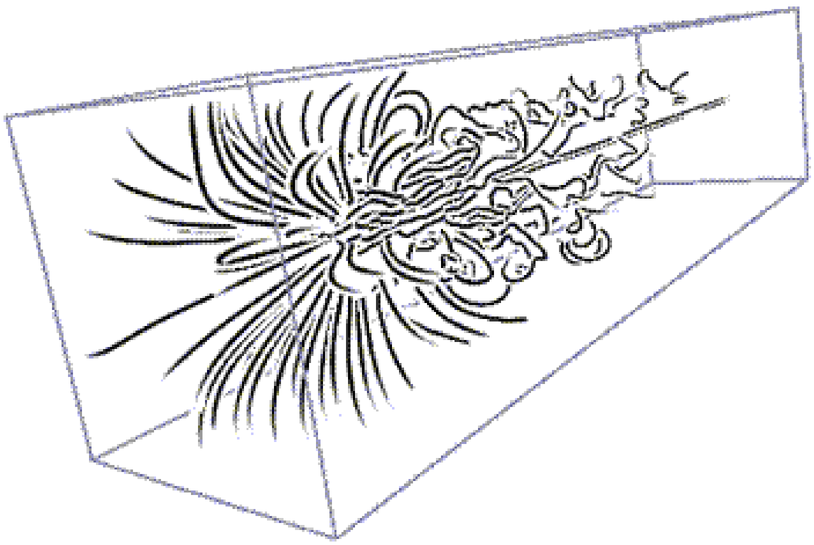


Figure 2.6: *Streamlines seeded in a 3D domain using an image-based technique [LS07]. Image courtesy of Han-Wei Shen.*

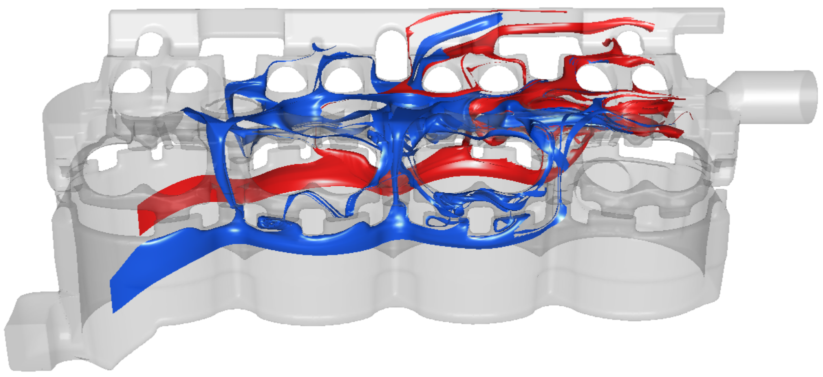


Figure 2.7: *Stream surfaces showing the flow through an engine cooling jacket. Either side of the surface is colored differently to easily identify the orientation of the surface [LGD$^+$05].*

side of the streamline. The new streamline is integrated until it is within a threshold, d_{sep}, from other streamlines, it is then placed in the queue. This process is then repeated. Complications arise when 3D streamlines in object-space then overlap in image-space. Halos are one of the tools used to address this problem.

Interactive, seeding strategies have been used in various modern, real-world applications including the investigation and visualization of engine simulation data [Lar02, LWSH04, LGD$^+$05] (See Figure 2.7).

Reflection: One of the major milestones to perception in 3D vector fields came from Zöckler et al [ZSH96]. They were the first to introduce an enhanced lighting and illumination model for 3D streamlines. However, we still consider perception of 3D vector fields to be an open problem with various unsolved challenges. For example no user-study evaluation of illuminated streamlines exists.

2.2.6 Integral Curve Placement in a 3D, Time-varying Domain

The following research focuses on point-based seeding in unsteady, 3D vector fields. Bryson and Levit[BL92] introduce the Virtual Wind Tunnel. The tunnel is a virtual environment for the exploration of vector fields. It utilizes a mounted head-tracked stereoscopic display. This serves two main purposes: The stereoscopic display provides depth information to the user and the head-tracking allows the user to change their view point within the application by physically changing the position and orientation of their head. This system also allows the user to interact and manipulate objects (such as seed positions) in the system through the use of a glove with input based on gestures from the user. Visualization techniques that are used include tufts, streaklines, particle paths and streamlines. Performance issues arise from the use of large data sets, which result in high bandwidth and memory requirements. This problem is exaggerated more by the head-tracking feature, the application needs to maintain a minimum execution performance rate (a minimum of 10fps is recommended) to prevent the user from losing co-ordination within the virtual environment.

Wiebel and Scheuermann present two methods for static visualization of unsteady flow [WS05]. This is opposed to using animation which is much more commonly used to visualize unsteady flow. The first method involves bundles of streaklines and pathlines that pass through one point in space (the eyelet) at different times. A group of pathlines or streaklines passing through the eyelet point (at different times) form the basis of a tangent surface. This method is similar to the technique proposed by Hultquist [Hul92], however in the cases of convergence, a line trace is not terminated, it is just simply ignored for the purpose of surface construction. In the case of divergence a test is made to see if there are any pathlines that are currently being ignored and if so and they are in the correct place they are then used again. If no appropriate pathline exists then a new line must be traced from the eyelet. It is not adequate to simply interpolate a new position between two pathline for a seed point, this is because this new seed point won't necessarily pass through the eyelet. Regions of high activity are of more interest for investigation in general and iso-surfaces are used to separate regions of high activity from regions of (nearly) steady flow. This effectiveness of this visualization technique depends greatly upon the placement of the eyelets within the flow field. Positioning the eyelet is based on sharp edges or corners of objects in the simulation, vortices, critical points, and regions of high activity i.e., rapidly fluctuating flow direction.

Helgeland and Elbroth present a hybrid geometric and texture based method for visualizing unsteady vector fields [HE06]. The seed positions for the field lines are computed as a pre-processing step. A random initial seed position is used to prevent visual artifacts that may arise when using a uniform distribution of seed points. The seeding algorithm is based upon the evenly-spaces streamline strategy introduced by Jobard and Lefer [JL97a]. As a seed point is placed it is advected both upstream and downstream a certain distance. If the field lines don't maintain a minimum distance, d_i, from all other field lines, the seed point is removed. The final set of seed points are stored in a 3D texture. Particle advection is implemented using a fourth-order Runge-Kutta integrator. Particles are added at inflow boundaries using the same scheme as the initial seeding strategy. During the course of the visualization, particles may cluster together producing both regions of high particle density and regions of low particle density. To prevent this particles are removed in regions of high density and new particles are injected into sparse regions of particles. A texture-based approach is then used to generate the field lines.

Reflection: Overall, there has been very little work in seeding of integral curves in 3D, unsteady flow fields. We consider this an open problem. Challenges related to both interactive computation time and perception remain. Also, there is no general consensus on an optimal seeding strategy in 3D.

2.3 Surface-based Integral Objects

This section describes geometric methods involving surface-based integral objects. Increasing the seeding object dimensionality increases the dimensionality of the resulting integral object. Surfaces have the added benefit of providing greater perceptual information over line primitives, as shading provides better depth cues. Surfaces also suffer to a lesser extent from visual complexity when compared to line primitives as many lines can be replaced by a single surface, providing more spatial coherency. We note that a significant amount of related work has also been carried out in the applied mathematics community. See Krauskopf et al. [KOD$^+$05] for an overview.

2.3.1 Surface-based Objects in 3D Steady-State Domain

In 1992, Hultquist introduced a novel stream surface construction algorithm [Hul92]. Streamlines are seeded from a curve and are advanced through the vector field. The sampling frequency is updated at the integration step if necessary. This is achieved using distance tests for neighboring streamline front points. For convergent flow the distance between neighboring points reduces and conversely for divergent flow. In the case of divergent flow a new streamline is seeded when the points exceed a pre-determined distance. In the case of convergent flow, the advancement of a streamline may be terminated if neighboring streamlines come too close. These operations help control the density of the points of the advancing front and maintain a sampling frequency that accurately reconstructs the vector field. The streamline points are used for the stream surface mesh. A locally-greedy tiling strategy is used to tile the mesh with triangles to construct the surface. The stream surfaces may also split apart in order to visualize flow around highly divergent areas such as the flow around an object boundary. The stream surfaces are seeded using an interactive seeding rake.

In contrast to the local method of stream surface presented by Hultquist [Hul92], Van Wijk presents a global approach for stream surface generation [vW93b]. A continuous function $f(x,y,z)$ is placed on the boundaries of the data set. A scalar field is then computed throughout the domain by streamlines placed at all grid boundary points and propagating the value of f along the streamline. An iso-surface of this so-called stream function can then be extracted to construct the stream surface. One drawback of this approach is that it only generates stream surfaces that intersect the domain boundary.

Scheuermann et al. present a method of stream surface construction on tetrahedral grids [SBH$^+$01] that builds upon previous work introduced by Hultquist [Hul92]. This method advances the surface through the grid one tetrahedron at a time and calculates where the surface intersects with the tetrahedron. When the surface passes through the tetrahedron the end points are traced as streamlines. For each point on a streamline, a line is added connecting it to its counter-point. These are then clipped against the faces of the tetrahedron cell and the result is the surface within the cell. Due to the nature of this method, i.e., using the underlying grid in the surface construction process, this method is inherently compatible with multi-resolution grids and thus benefits from the increased grid resolution in interesting flow regions, such as near object boundaries within the flow field.

Brill et al. [BHR$^+$94] introduce the concept of a streamball and apply them to the visualization of steady and unsteady flow fields. Streamballs are defined by a set of discrete points in the vector field based on the metaballs of Wyvill et al. [WMW86]. A ***streamball*** follows the same path of a streamline, however acceleration and deceleration are depicted by the amount of displacement between neighboring spheres. Other local properties of the flow can be mapped to the radius of the sphere.

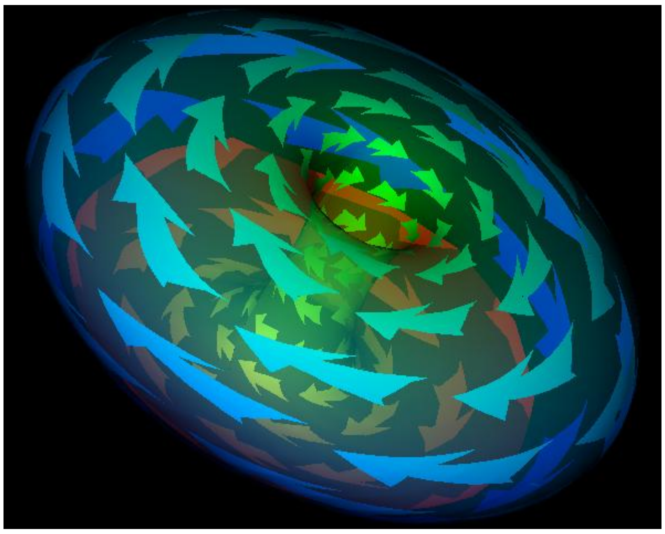


Figure 2.8: *Stream-arrows textured to a streamsurface. The portions outside the arrows are semi-transparent reducing the occlusion by the surface [Löf98]. Image courtesy of Helwig Hauser.*

Using discrete streamball placement it is possible to construct streamlines and pathlines by ensuring that the center points for each streamball are close enough so that they blend together and form an implicit surface [BHR$^+$94]. This technique can also be applied to stream surface construction by advancing a set of streamballs that are seeded along a curve. This produces a smooth surface where the streamballs merge automatically in areas of convergence and split in areas of flow divergence.

Westermann et al. [WJE00] present a level-set method for the visualization of flow fields. Vector-field data is converted into a scalar level-set representation. These level-sets are used to create implicit time surfaces. This approach is similar to the implicit stream surfaces method of van Wijk [vW93b]. Once we have this scalar representation the surfaces can then be computed using an iso-surface extraction technique.

Garth et al. [GTS$^+$04] introduce a stream surface technique that handles areas of intricate flow more accurately than previous techniques such as the method presented by Hultquist [Hul92]. The algorithm is demonstrated in the context of vortex structures and is based upon an advancing front with the insertion and deletion of points at each integration step in order to maintain sufficient resolution for the construction of an accurate stream surface. In order to handle more complex flow regions, a high-order integrator is used as well as streamline integration being based upon arc length as opposed to parameter length, as used in Hultquist's method [Hul92]. This results in an improved triangulation for the stream surface mesh, i.e., triangles are more regular throughout the mesh. For the insertion of a new streamline the authors also introduce a new rule based upon the angle between triples of neighboring points on the stream surface front.

Löffelmann et al. [LMG97] introduce an extension called stream-arrows for the enhancement of stream surfaces. ***Stream-arrows*** are partitions that are removed from the surface in order to convey inner flow structure within the surface. The removal of the partition also help to reduce occlusion as the area behind the removed portion is visible. In the original stream-arrows algorithm the arrows were placed on the stream surface using regular tiling. However, this technique may provide unsatisfactory results in regions of convergence and divergence, as the arrows may become too small or too large. The hierarchical extension overcome this shortfall by generating a stack of stream-arrow textures, where each texture contains a unique resolution of arrows. The most appropriate texture is then chosen according to the size of the stream surface, this ensures that the stream-arrows all keep a similar size.

Laramee et al. [LGSH06] present a hybrid method by applying texture advection to stream surfaces. This hybrid technique enables the visualization to convey more information about its inner flow structure. This technique has been used on iso-surfaces [LSH04] but the textures can mislead the user. If an iso-surface is generated using velocity magnitude, there is no guarantee that the surface will be everywhere tangent to the flow. This means there may be a component of the flow vector that is, at least in part, orthogonal to the iso-surface and thus the advection may be misleading. Stream surfaces don't suffer from this weakness as they are, by definition, always aligned with the flow field.

Peikert and Sadlo[PS09] present a hybrid seeding and construction method for topologically relevant stream surfaces. The topology of the velocity field is used to compute the seeding curve of the most expressive stream surfaces. Seeding for cases such as periodic orbits and critical points is described to ensure that the stream surface is not multiply covered. Cases of open and closed seeding curves are described. A quad-based construction method is used to compute the stream surfaces. Quadrilateral cells are added at the front of the surface enclosed by streamlines and orthogonal curves. Sinks within the velocity field are detected when the orthogonal edge segments become too small. These edges are flagged as inactive. Integration terminates when there are no more active edges. Saddle points are handled by tearing the surface resulting in a closed surface front becoming an open one and an already open front splitting into separate portions.

Reflection: A milestone in this category was that of Hultquist who introduced the first stream surfaces for 3D-steady flow [Hul92]. The work that follows improves that work in terms of performance and perception. A few of the open challenges here relate to performance and perception. How to minimize occlusion while maximizing coverage is a big challenge. Interactive computation of stream surfaces for unstructured grids still remains to be seen.

2.3.2 Surface-based Objects in 3D Time-Dependent Domain

Schafhitzel et al. [STWE07] introduce a point-based stream surface construction and rendering method. This method is also applicable to unsteady flow fields, for which it generates path surfaces. This method was implemented to exploit graphics hardware acceleration and therefore all data structures used lend themselves to being stored in textures. Like the method presented by Hultquist [Hul92] this algorithm includes operations to adjust the density of the surface front. By adding or removing points this method updates the sampling frequency and allows for accurate surface construction in flow exhibiting local convergent or divergent behavior. A method and condition for splitting the surface are also implemented which are similar to the technique Hultquist [Hul92] used in his algorithm.

In order to render the surface, particles are distributed with a sufficient density (to cover the image space represented by the surface) so that point sprites can be used, this results in a closed surface. Surface normals can be estimated for each particle and this allows the surface to benefit from shading and its associated advantages, i.e., greater depth cues etc. A surface-based LIC algorithm was also applied in order to provide internal visual structure for the surfaces.

Von Funck et al. present a novel technique for smoke surface construction that provides nearly interactive frame-rates by avoiding the expensive mesh re-triangulation at every time step [vFWS$^+$08]. This method exploits irregular triangles and maps the opacity of the triangle to its size and shape. This provides a fair approximation of the optical model for smoke resulting in surfaces that give a smoke-like effect. A number of enhancements are also given, showing how simple modifications to the core algorithm can be used to simulate smoke injection from nozzles and wool tufts attached to the boundary surface of geometries within the flow domain.

Garth et al. present a novel stream and path surface technique focusing on accuracy [GKT$^+$08]. This method defines refinement criteria that are based upon the order of continuity of the surface lines. New points are added when the curves need to be refined. When discontinuities of first and second order are encountered, the portions of the surface either side of the discontinuity are treated independently of each other, like the surface tearing as handled by Hultquist [Hul92]. This method has no mechanism for removing points from the surface, where sampling is more than sufficient. They trade the cost of the extra integrations for the cost of performing the tests for redundant points and their removal. The meshlines are stored as polynomials and the mesh is discretized after the entire advection stage.

McLoughlin et al. [MLZ09] present a simplified stream and path surface construction technique (Figure 2.9). This method is closely related to Hultquist's technique [Hul92]. This technique makes use of simpler data structures – a 2D array, compared to the tracer and ribbon structures used by Hultquist. This simpler approach is made possible by the use of quad primitives for the surface construction. Quads lend themselves naturally to a 2D array, which forms an implicit parameterization of the surface. However, memory may be wasted as elements of the 2D array may be empty, containing no geometry information, but maintain the parameterization property of the surface. Insertion and deletion of vertices is performed to maintain a sufficient sampling of the vector field. This is achieved by processing the mesh quad by quad, dividing and merging quads when a change in resolution is required. Shear flow is also handled by an adaptive step-size integration technique along the quad edge to ensure the quads are more regular.

Krishnan et al. [KGJ09] present a novel streak and time surface algorithm (See Figure 2.10). This technique guarantees a C^1 continuous curve for the integral curves due to the use of an adaptive step-size 5th-order Runge-Kutta outputting a sequence of fourth-order polynomials. This allows for interim points to be computed easily and provides more accurate results than gained from a simple piecewise linear interpolation between sample points. Three basic operations are defined for the surface adaptation process, these are *edge split*, *edge flip* and *edge collapse*. An edge split ensures that no edge on a triangle is longer than a prescribed threshold. A new vertex is inserted and this is used to create a new integral curve. Egde flipping locally re-

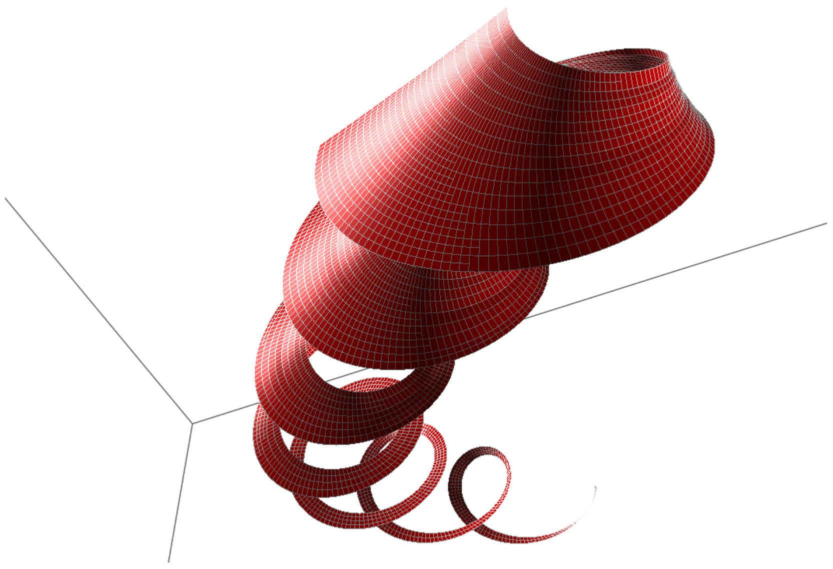


Figure 2.9: *A stream surface visualizing a tornado simulation [MLZ09]. Surfaces reduce visual complexity compared to line primitives.*

fines an area to maximize the minimum angles within the triangles such that triangles are more regular. Edge collapse removes edges from the mesh in regions where the density of triangles is too high. This prevents the unnecessary propagation of curves in future computations. The algorithm is demonstrated on large unsteady unstructured grid simulations, which inherently consume much processing effort. It is shown that this technique lends itself to parallelism.

Bürger et al. [BFTW09] present two streak surface techniques implemented on the GPU. These techniques provides interactive rates throughout the surface construction. The first technique is based on quads. Each quadrilateral patch contains four vertices. The same vertex is stored (and propagated) multiple times. Refinement of patches is achieved by splitting the longest edge of the quadrilateral and the edge opposite it. This may result in discontinuities within the mesh. A two-pass rendering operation ensures that the quads form a smooth surface during the rendering phase. The first pass creates a *depth imprint* of the enlarged quadrilateral patches in respect to the position of the viewer. On the second pass a biased depth test is used on the depth imprint to ensure that only patch samples close to the surface are used. A smooth transition for shading, coloring etc. is done by using a Gaussian kernel at each path centroid to weight the attributes which are finally accumulated using additive blending and normalization.

The second technique more closely follows the more common mesh approaches. Duplicate vertices are not stored explicitly. This is handled by memory layout in the vertex buffer. Surface refinement is performed in three stages: timeline refinement, connectivity update, streakline refinement. During timeline refinement particles may be inserted, spawning new streaklines, or particles may be removed. Connectivity is updated by each particle on a timeline searching along neighboring timelines for the closest match based on a uniqueness criterion. In the streakline refinement phase a test for the maximum Euclidean distance is performed for neigh-

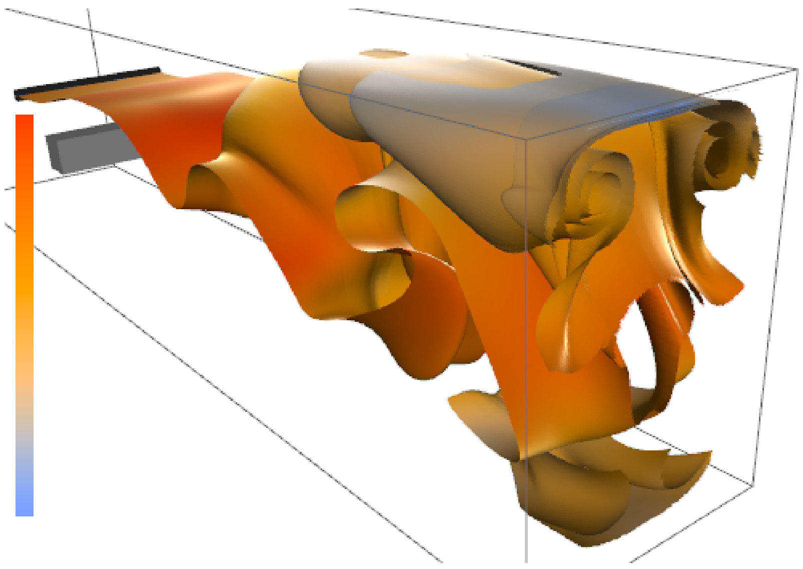


Figure 2.10: *A streaksurface depicting a flow behind a square cylinder simulation. Streaksurfaces are well suited for visualizing the complex structures occurring in unsteady flow [MLZ10].*

boring timelines. If the distance is above or below given thresholds, an *entire* streakline is added or removed respectively.

Reflection: Constructing and rendering of integral surfaces in 3D unsteady flow is clearly an unsolved problem with various challenges remaining including performance and perception. Current solutions do not handle shear flow very well. Also, the combination of large datasets and interaction still poses challenges. This category of techniques is currently and active area of research.

2.4 Volume Integral Objects

This section describes geometric methods involving 2D surface or planar-based seeding objects. Once more, increasing the dimensionality of the seeding object increases the dimensionality of the integral object. The result is a geometric object that sweeps a volume. The volume of these objects can be used to depict flow characteristics such flow convergence and divergence.

2.4.1 Volumetric Integral in a 3D Steady-State Domain

Schroeder et al. introduce the *Stream Polygon*[SVL91]. A ***stream polygon*** is a regular n-sided polygon that is oriented normal to the vector field. The stream polygon can be used by placing a new polygon for each point of a streamline or it may be swept along the streamline to form a

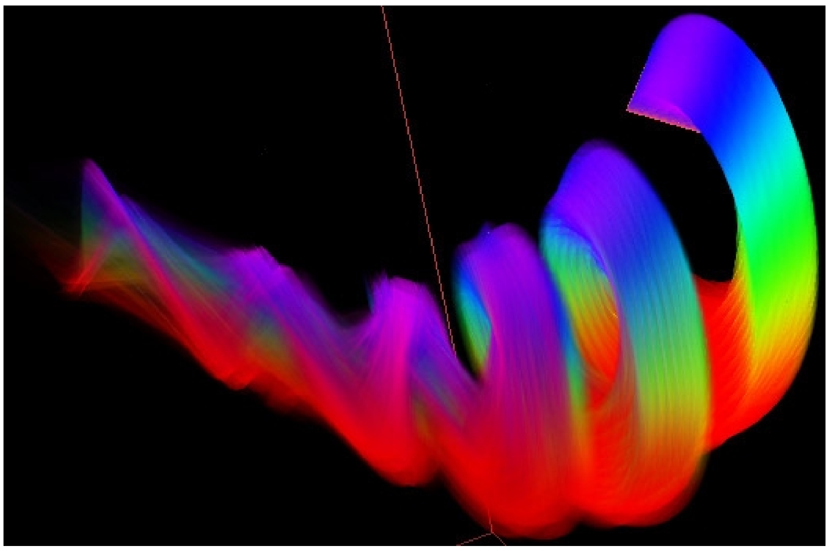


Figure 2.11: *A flow volume created using the tornado dataset. Image courtesy of Roger Crawfis [MBC93].*

tube. The polygon is deformed according to the local flow properties. Rotation of the polygon reflects the local vorticity of the flow field. There are no constraints on the polygon maintaining a rigid body structure, therefore deformation of the polygon is used to illustrate the local strain of the flow field.

Max et al. introduce flow volumes [MBC93]. A ***flow volume*** (Figure2.11) is the volumetric equivalent of a streamline. This method draws inspiration from experimental flow visualization, using a tracer material released into a fluid flow. As the trace propagates through the flow it forms into a flow volume. The flow volume is divided into a set of tetrahedra, the projection of which are divided into triangles. Color and opacity are computed for each tetrahedra using the density emitter model of Sabella [Sab88]. The contributing pixel values can be composited in an arbitrary manner, thus negating the need for a complex sorting algorithm for the volumetric cells. This is suitable as it produces a reasonable approximation of the tracer material effect that the authors are aiming for. An interactive seeding object is used which is always oriented normal to the local flow field and allows the user to change attributes of the seed object such as: position, color and opacity of the smoke and the number of sides for the seeding polygon.

Xue et al. introduce implicit flow volumes [XZC04]. This idea builds upon flow volumes introduced by Max et al. [MBC93] and the implicit stream surface technique presented by Van Wijk [vW93b]. Two techniques are presented for the rendering of the implicit flow volume, a slice-based 3D texture mapping and interval volume rendering. The first approach renders the flow field directly without the inflow mapping to a scalar field, as used by Van Wijk [vW93b]. Volume shaders can be used to change the appearance and representation of the flow volume. This method allows for high levels of interactivity and fine texture detail in all regions of the flow volume (See Figure 2.12). The second approach utilizes a flow mapping that produces a scalar field, the flow volume created from the interval volume enclosed between two iso-

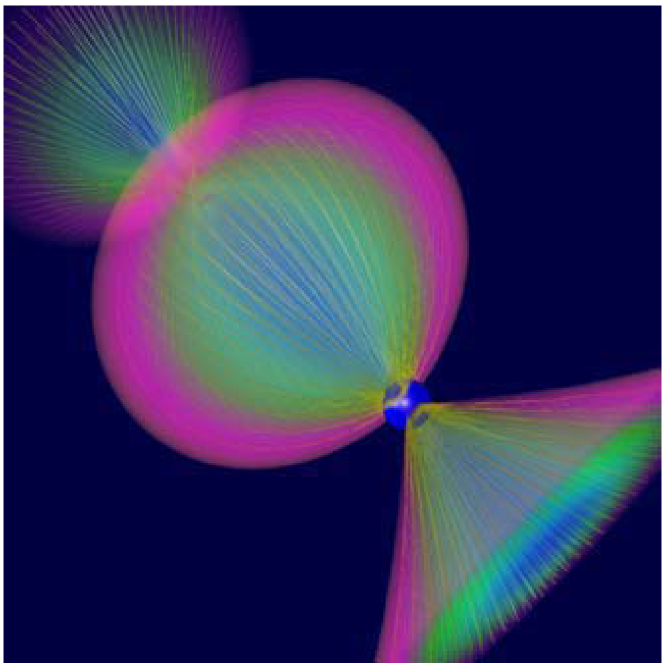


Figure 2.12: *An implicit flow volume based on the technique of Xue et al. [XZC04]. Image courtesy of Roger Crawfis.*

surfaces. The rendering of the volume is then achieved by using a tetrahedron-based technique.

2.4.2 Volumetric Integral in a 3D Time-Dependent Domain

Becker et al. extend the flow volume technique [MBC93] for the use with unsteady vector fields [BLM95]. Flow volumes in steady flow fields are created using a set of streamlines seeded from a polygon oriented normal to the local flow. To extend this to unsteady flow Becker et al. construct the flow volumes using streaklines. As in the steady case, the volume is divided up into tetrahedra and volume rendered using hardware. Using streaklines instead of streamlines introduces several complications to the initial flow volume strategy. In the steady case, only the end points of each streamline are advected and a new layer is added to the end of the flow volume in the downstream direction during each integration step. However, when streaklines are used, every point on the streakline must be advected (not just the end points). This may result in the flow volume geometry changing over time. The subdivision strategy used in [MBC93] was performed only at the end of the flow volume, but the changing geometry here requires a subdivision strategy that operates anywhere in the volume. A subdivision created in a previous time step may be unnecessary in future time steps. Subdivision in time is also required, if all particles within a given layer exceed a given distance threshold from the previous layer, a new layer is inserted between them. The reverse is also applicable with the possibility that a layer may be removed.

Reflection: Work on volume integral objects is clearly less mature in comparison to research using curve and surface-based solutions. Also, no individual contribution has triggered a chain of follow-up approaches offering enhancements to the original. Both computational and perceptual challenges remain in this area. Seeding is also a challenge that has not been addressed.

2.5 Discussion and Conclusions

A variety of techniques have been discussed, each with their own relative merits and shortcomings. As clearly illustrated in this literature, there is no single visualization tool which provides optimal results for all given phenomena. The most appropriate method is dependent upon several factors such as the data dimensionality (both spatial and temporal). The size of the simulation output and the goal of the user are also factors, e.g., is the visualization to be used for detailed investigation in specific regions, is the visualization intended for fast exploration of the vector field or for high quality presentation purposes.

In the context of geometric approaches, a large volume of effort has been placed on streamlines. Streamlines are an effective tool and coupled with an effective seeding strategy may produce some insightful visualizations. The success of streamlines comes partly from their ease of implementation and the quality of the results produced. Streamline enhancements tend to fall in one of two categories, particle tracing or seeding algorithms. The particle tracing algorithms also have their own subset of classifications, with their contributions differentiating them from other tracing algorithms. Most effort has been undertaken on providing ever faster tracing, while other methods have focused tracing on specific grid types, while others on accuracy, producing exact results rather than approximations when using a numerical integration method. Particle tracing methods were a popular area of research in the 1990's with comparatively very little work been undertaken recently. This may be due to the efficiency of current methods, making it more difficult to obtain further significant gains in performance.

Seeding strategies have been heavily researched and are still an area of active research. Many algorithms are based on providing aesthetic, insightful visualizations in image-space. The focus of these papers tends to be on producing uncluttered visualizations that avoid bombarding the user with visual overload. The majority of seeding algorithms have been targeted at 2D domains with only a handful being extended or specifically aimed at higher spatial dimensions. We believe that seeding in 3D domains is still a fruitful area of research and seeding strategies may be extended to providing efficient algorithms for unsteady flow fields.

During the composition of this survey we identified an interesting trend. As the dimensionality of the integral object increases the volume of research decreases, this is evident from Table 2.1. We believe that this is due to the added complexity of creating higher dimensional geometric objects and ensuring that they maintain an accurate representation of the underlying vector field. We have already stated the advantages of the use of surfaces and volumes over the use of line primitives, but these advantages are countered by the difficulty of creating a suitable mesh for a surface or volume. Stream surfaces are a very useful tool for flow visualization, however, research and their implementation in industrial applications is limited. While several algorithms exist for stream surface construction, their complexity is often a barrier for a developer. We believe better construction methods that are both efficient and simple are possible.

The extension to unsteady flow fields is also an attractive prospect with relatively few papers explicitly showing a time-dependent implementation using surfaces. We also note that there are very few strategies that focus on automatic placement of surface structures in the flow field. We expect this to be a very active area of research in the coming years.

Many of the techniques here are used in commercial applications. The array of tools is too vast for all of them to be combined into a single package and so the most appropriate subset must be chosen. This will be more successful if the application designer works closely with a CFD engineer who has expert knowledge of the simulations they will be creating and the phenomena they wish to investigate.

Some of the key areas we identified needing additional work are:

- Higher dimensional (both spatial and temporal) data domain seeding strategies.
- Uncertainty visualization tools for geometric techniques.
- Comparative visualization tools for geometric techniques.
- Improved surface and volume construction methods.
- Surfaces for visualizing unsteady flow fields.
- Automatic seeding for surfaces and volumes.
- Interactive, real-time visualization of unsteady flow fields. Specialized vector field compression techniques must be used to reduce the bottleneck of loading new time steps into the GPU.
- GPU-based methods on unstructured grids.

A large amount of success has been gained for 2D vector fields and more recent techniques are offering a less significant improvement over current methods. Three-dimensional vector fields have also attained a high level of progress. However due to the added challenges, there is still room for significant improvement in many areas. Similarly, many problems remain unsolved when visualizing unsteady flow and the barriers are constantly pushed as simulations are increasing the size of their output and ever more efficient methods are required to address this expansion.

Part II

Research

3

Easy Integral Surfaces

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the universe trying to build bigger and better idiots. So far, the universe is winning..."
-Rich Cook (1944–)[1]

Contents

STREAMLINES and pathlines are ubiquitous in flow visualization applications. They are a well understood, intuitive tool that produces insightful visualizations. However, these curve-based primitives can suffer greatly from visual complexity, and finding an optimal seeding strategy that produces an uncluttered but detailed visualization is non-trivial. They are also restricted in the characteristics that they represent. For example, a simple tangent line does not exhibit the downstream direction of the underlying vector field.

Stream surfaces are an extension of the streamline. Stream surfaces, surfaces everywhere tangent to the flow, are a viable solution for the visualization of 3D vector fields. Firstly they do

[1]A fantasy author from the United States. He is best known for his Wizardry series of books.

not suffer from the visual complexity the same way seeding many streamlines can. Secondly, depth cues can be easily added using shading. The 3D orientation of the surface can clearly be perceived. Path surfaces are the extension of stream surfaces to unsteady flow. They are integral surfaces (computed from integration) everywhere tangent to a time-dependent vector field. Path surfaces provide the same benefits over pathlines as stream surfaces over streamlines.

However, despite the aforementioned benefits integral surfaces provide for the visualization of 3D vector fields, they are seldom used in commercial applications or for research purposes. We hypothesize that this is due to the implementation complexity, and thus inaccessibility, of previous approaches to surface construction.

Simplicity of an algorithm often directly affects its popularity and impact. For example, we believe that much of the success behind the Marching Cubes Algorithm [LC87] stems from the fact that the technique is based on simple, local analysis of cube primitives, thus making the technique highly accessible. This means it can easily be incorporated into commercial software and it can handle large data sets. It is this kind of accessibility that we strive for here.

In general, previous algorithms are based upon complex data structures and an elaborate mesh triangulation, that rely on several sets of co-ordinates (physical-space, computational-space and parametric-space). This, coupled with the cost of maintaining an optimal triangulation strategy depreciates the attraction of using stream and path surfaces. A re-triangulation may also be necessary, if a global tiling strategy is used, whenever the surface is extended. In contrast, our approach is based on a small set of simple, local operations performed on quad primitives and requires no global re-meshing strategy. Quad-based meshes have become increasingly popular in recent years, and their application to stream and path surfaces seems natural when we consider the anisotropy of flow fields. Quad-based meshes allow for easy extension. When the mesh is extended the previous connectivity remains unchanged and new quads are appended onto the front of the surface. Using a global triangulation would result in the connectivity of the entire mesh having to be recomputed.

The main contribution of this chapter is the introduction of a novel algorithm that is significantly easier to model and implement than existing methods with the following benefits:

- An algorithm that is fast because it is based on only a few local operations applied to quad primitives. It requires a simple 2D array and no surface parameterization.
- An algorithm that is suitable for large data sets because of the local nature of the processing.
- An algorithm that supports a number of enhancements that stem naturally from the technique's simplicity including: easy timelines, easy timeribbons, easy stream arrows and easy evenly-spaced streamlines.
- We introduce a novel interaction tool called a stream surface painter designed specifically to address the perceptual problems associated with visualizing 3D flow.

However, in order to achieve these benefits, the same challenges must be addressed as with previous solutions, namely how to construct a smooth and accurate surface in flow characterized by high convergence, divergence and curvature. Although tangent surface construction is

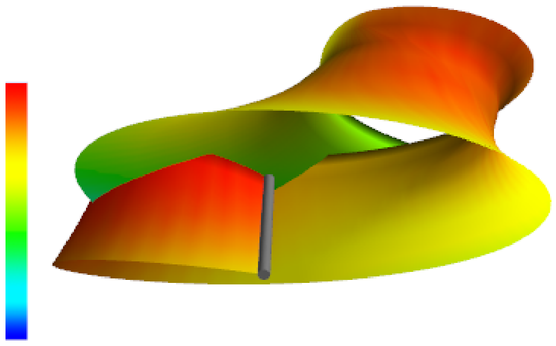


Figure 3.1: *A quad-based path surface, color coded according to velocity magnitude, along the core of the tornado from the tornado simulation. This figure shows our algorithm is capable of visualizing a field of high local rotation and twisting behavior. The surface also demonstrates a unique property of pathsurfaces – the ability to intersect themselves.*

fundamentally a vector field sampling problem, this is the first algorithm that formulates the construction of such surfaces explicitly as a sampling problem.

The choice of quad-based meshes is partially inspired by the increase in their popularity within the meshing community [ACSD$^+$03] [TACSD06], and by the following observations:

- A quad-based mesh contains edges aligned with the directions of the flow. This results in a more natural placement of the primitives and allows a user to distinguish the flow behavior from the mesh itself, without the need for additional processing. This cannot be said about their triangular counterparts.
- Quad meshes make for a high-quality representation of the flow.
- The lines, to some extent, mimic lines that artists themselves might draw when creating depictions of the flow.
- Quads can provide a more compact representation of surface geometry in terms of both space and processing time, e.g., the topology of the mesh does not require explicit storage.

The rest of the chapter is organized as follows: Section 3.1 provides a discussion of the previous work related to stream and path surfaces in the context of our algorithm. Section 3.2 describes the computational model and the local tests that are performed on the quad primitives. Section 3.3 shows that several known enhancements to integral surface-based visualizations stem naturally from the quad-based model described in Section 3.2, namely, easy timelines, easy timeribbons, easy stream arrows, easy evenly-spaced flow lines and the stream surface painter. Section 3.4 presents some results of the algorithm including its application to a large hurricane simulation. Section 3.5 concludes the chapter and identifies potential directions of future work.

3.1 Related Work

There have been relatively few proposed solutions to stream and path surface construction, especially when compared to the volume of effort concentrated on texture-based flow visualization [LHD$^+$04]. Well known work is presented by Hultquist [Hul90, Hul92]. Hultquist's algorithm first involves advancing a front in order to generate streamribbons. During streamribbon front advancement, candidate triangles are generated at each iteration. The goal of this approach is to create a globally-minimal tiling. One disadvantage of creating a globally-minimal tiling is that no triangles can be created until all of the integration of the curves has been completed. Van Wijk [vW93b] introduced a method for implicitly generating stream surfaces. This is achieved by defining a continuous function, $f(x,y,z)$, on the boundaries of a flow data set. This function is then extended to the interior of the domain. The iso-contour of the function $f(x,y,z)$ is then used to generate the stream surface. Scheuermann et al. [SBH$^+$01] present a method for the construction of stream surfaces on tetrahedral grids.

Garth et al. [GTS$^+$04] presented an improved method for stream surface construction in areas of intricate flow based on an extension of Hultquist's algorithm. The improved triangulation results from using higher order integration schemes combined with arc length parameterization.

More recently, Schafhitzel et al. introduce an alternative stream surface construction algorithm based on points [STWE07]. However, this approach introduces the new complication of how to generate a smooth and continuous surface from a set of discrete point primitives. Their GPU implementation also places limits on the size of the data sets that can be visualized. With the exception of Schafhitzel et al.'s approach, all previous algorithms are based on complex data structures and surface mesh triangulations. Consequently, they are difficult to implement and use in practice. In contrast, our approach is based on simple, local operations on quad primitives.

Enhancements to stream surfaces are also a fruitful area of research. Laramee et al. [LGSH06] presented an application in which texture advection was applied to stream surfaces to increase the number of characteristics of the flow that could be visualized simultaneously. Stream surfaces are also used to segment a vector field into regions of similar behavior [MBS$^+$04]. This is achieved by growing the surfaces starting near critical points in the vector field. Löffelmann et al. presented stream arrows [LMG97, LMGP97], an enhancement to stream surfaces that removes arrow-shaped holes out of the surface in order to reduce occlusion when the stream surface(s) overlap by allowing the user to see through the displaced sections. This method also serves the purposes of indicating the downstream flow direction by means of the orientation of the arrow. We demonstrate a range of enhancements on top of our core algorithm in Section 3.3.

The problem of quad-dominant remeshing, i.e., constructing a quad-dominant mesh from an input mesh, has been a well-studied problem in computer graphics. A key observation is that a nice quad-mesh can be generated if the orientations of the mesh elements follow the principle curvature directions [ACSD$^+$03]. This observation has led to a number of efficient remeshing algorithms that are based on streamline tracing [ACSD$^+$03, MK04, DKG05]. Ray et al. [RLL$^+$06] note that better meshes can be generated if the elements are guided by a 4-RoSy field. Dong et al. [DBG$^+$06] perform quad remeshing using spectral analysis, which

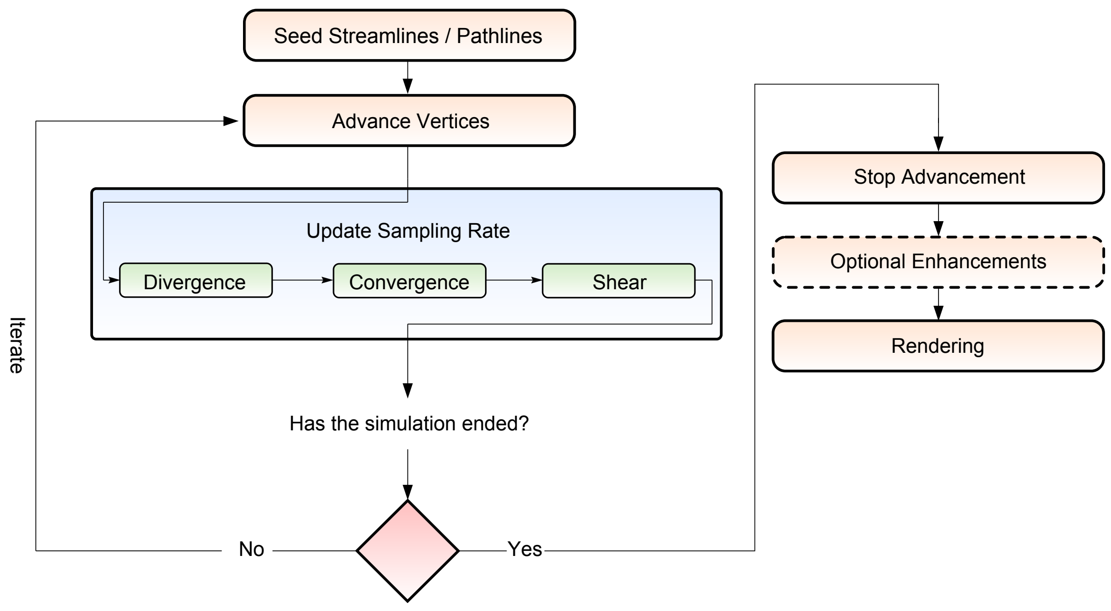


Figure 3.2: *An overview of the easy integral surfaces algorithm.*

produces quad meshes that in general do not align with the curvature directions [PZ07]. Felix et al. [KNP07] present a quad-remeshing technique that guarantees no T-junctions. However, this method requires the generation of a surface parameterization on the input mesh, which is not suited for our interactive approach.

3.2 Easy Integral Surfaces

Our algorithm consists of a series of operations illustrated in Figure 3.2:

1. We start out by seeding a curve using an interactive seeding rake (Section 3.2.1).
2. Then the stream (or path) surface front is advanced according to the vector field (Section 3.2.1).
3. The next step is to update the sampling rate of the advancing surface front in order to handle divergent, convergent, and rotational flow (Sections 3.2.2 and 3.2.3).
4. Terminating conditions such as leaving the domain are tested before returning to step 2 (Section 3.2.4).
5. User-controlled optional enhancements are enabled and the scene is rendered according to the current rendering parameters (Section 3.3).

What follows is a description of each stage of our algorithm along with a description of the technical challenges at each point.

3.2.1 Seeding and Advancing the Front

We use an interactive seeding curve so that the user can generate a variety of integral surfaces at run time. This also allows the user to quickly investigate any areas of interest within the flow domain. The seeding curve can be placed at any position and the user is able to adjust the length in order to create wider surfaces and analyze more of the vector field in a single visualization. The initial distance between seeds is $\frac{1}{2}d_{sample}$ i.e., $\frac{1}{2}$ the distance between data sample points. However the sampling rate can be adjusted, determining the number of flow lines that are initially seeded along the curve. By flow lines we mean streamlines and pathlines.

We construct an integral surface from quad primitives. There are two important distances to consider when constructing a new quad: (1) d_{sep} - the distance between neighboring flow line points that correspond to the same integration time t and (2) d_a - the advancement distance. Our goal is to generate quads that result in smooth and accurate surfaces. In practice, this is obtained by determining the appropriate lengths of d_{sep} and d_a so that we maintain an appropriate sampling rate of the underlying vector field. The sampling rate we choose in all cases throughout our algorithm is guided by the Nyquist Limit, namely, the sampling frequency must be (at least) twice that of the underlying data frequency for accurate reconstruction. Thus we choose an initial d_{sep}: $d_{sep} < \frac{1}{2}d_{sample}$. Similarly for d_a, we start a new quad when the flow line integration distance exceeds $\frac{1}{2}$ the between data samples. We use a second order Runge-Kutta integrator in our implementation, which provides a good balance between accuracy and computational speed.

Using quads, the advancement of the front is simplified. If triangles are used [Hul90, Hul92, GTS$^+$04] then an optimal tiling strategy should be included in order to prevent long, thin triangles. The approach of Hultquist [Hul90, Hul92] requires three data structures, two tracer structures and a ribbon structure. The two tracer structures are used to generate the sequence of points that define two flow lines S^0 and S^1. Each tracer stores the context required by Hultquist's algorithm to advance a particle through a sampled vector field. The ribbon structure is necessary to connect S^0 and S^1. The position of each point is recorded in physical space, computational space, and the surface parametric space. In contrast our algorithm requires a 2D array of points in physical space only. The two dimensions of the array correlate to the S and t parameters of the surface, thus the mesh topology is stored implicitly within the data structure and does not require explicit computation and storage. To find a quad from a given point, we access its neighboring points in the 2D array. When we encounter four valid points we have a quad. We use the term *valid point* due to the fact that we provide flags indicating the state of a particular vertex for which some cases may be skipped during this search process. These flags and how they are used are discussed later.

3.2.2 Divergence and Convergence

Divergence is a common phenomenon of flow. When flow diverges the distance between points of neighboring flow lines, d_{sep}, increases. As d_{sep} increases the data sampling rate is reduced. If we were to construct a surface just using these points, the surface could potentially miss critical features of the flow and thus be an inaccurate representation of the underlying field. To overcome this a new flow line is seeded whenever the distance between a pair of flow

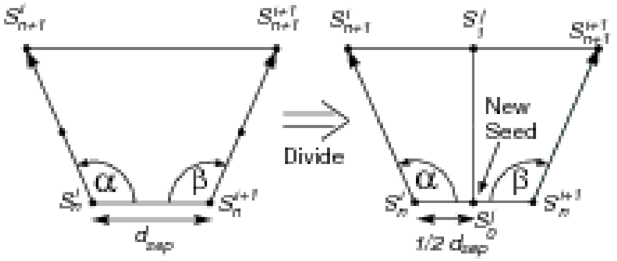


Figure 3.3: *Divergence is monitored by the testing the value of d_{sep}. When $d_{sep} > \frac{1}{2}d_{sample}$ and $\alpha > 90^\circ$ and $\beta > 90^\circ$ the quad is divided. A new flow line seedpoint S^j_0 is introduced as a result of the subdivision.*

lines exceeds a threshold. At each stage of the front advancement, the separating edge length between corresponding flow lines is monitored for accurate sampling frequency. As soon as $d_{sep} > \frac{1}{2}d_{sample}$ and $\alpha > 90^\circ$ and $\beta > 90^\circ$ we simply divide the quad as in Figure 3.3. This results in two quad primitives and a new streamline seeding point. The seed point for the new streamline is calculated by interpolating between the distance halfway between S^i_n and S^{i+1}_n. Using quads results in a much simpler mesh, see Figure 3.4.

In contrast, the approach of Hultquist [Hul90, Hul92] requires new tracer and ribbon structures to be stored in the linked list which represents the front. Also previous methods describe *how* new particles are inserted in order to handle convergence (and divergence) but not *when*. By formulating the algorithm as a sampling problem explicitly, the answer as to when new particles should be inserted (or removed) from the flow becomes clear.

In order to add new streamlines into our 2D array and maintain the topology information, we have to move columns over in order to accommodate the new streamline. However, this results in elements within these newly inserted columns that don't contain a mesh vertex. We handle this by having a flag that indicates whether a point exists at a particular array element or whether it contains no vertex information. Elements flagged as empty are simply skipped

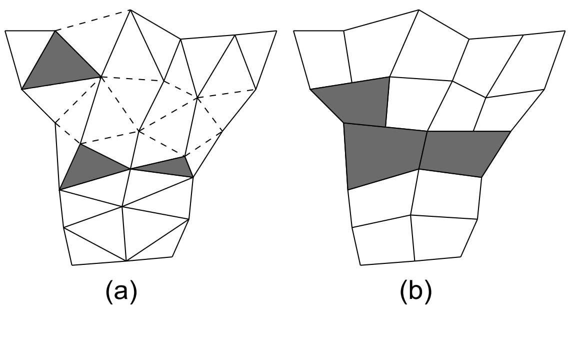


Figure 3.4: *A comparison between the divergence operations of (a) Hultquist's algorithm and (b) Easy Stream Surfaces divergence operation. The gray colored sections indicate the mesh transition.*

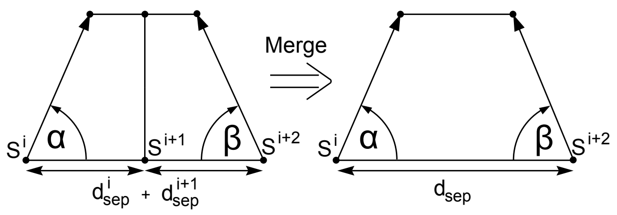


Figure 3.5: Convergence is detected when $d^i_{sep} + d^{i+1}_{sep} < \frac{1}{2}d_{sample}$ and $\alpha < 90°$ and $\beta < 90°$. It is handled by terminating the middle flow line. Two quad primitives are merged into a single quad.

over during the rendering process and by the sampling rate operations. Instead, their next valid neighbor is used (as these are the correct neighbors in physical-space). Note that there are more flags defined for the elements, these are discussed where they are appropriate in future sections.

A vector field may also locally converge. Adjacent flow lines begin to approach each other causing the distance between the corresponding points of S^0 and S^1 to decrease. This can lead to many points being generated in a small vicinity on the stream surface. This results in an area being oversampled and, very small quads being produced for the surface mesh. We monitor for cases of flow convergence at each stage of the surface front advancement. As soon as $d^i_{sep} + d^{i+1}_{sep} < \frac{1}{2}d_{sample}$ and $\alpha < 90°$ and $\beta < 90°$ we simply terminate streamline S^{i+1}, the flow line between S^i and S^{i+2}. The result is a quad merging (Figure 3.5). Hultquist's algorithm [Hul90, Hul92] requires the insertion of three new triangles which make the transition between the leading edges of two ribbons and the single leading edge of a new replacement. We point out that, once again, quad meshes result in a simpler mesh, see Figure 3.6. Note, no description of *when* adjoining ribbons are merged into one is given (only how).

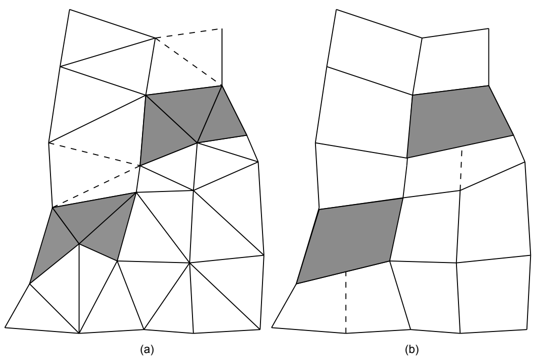


Figure 3.6: A comparison between the merging operations of (a) Hultquist's algorithm and (b) Easy Stream Surfaces merging operation. The gray colored sections indicate the transition using the convergence operation. In (a) three new triangles must be created, while in (b) no extra primitives have to be inserted in order to create a smooth transition between the strip widths.

Quad meshes can introduce T-junctions which may cause gaps. For divergence, no gaps are produced as S_0^j is chosen to lie on the edge from S_n^j to S_n^{j+1}. However, during convergence, gaps may occur. This can be handled in two different ways. One is by dragging S^{i+1} onto edge $S^i - S^{i+2}$ for the adjacent quads. Another solution is to simply patch the gap with a triangle. We also point out that the vast majority of cases are divergent.

3.2.3 Curvature

After advancing the front by one quad, we monitor the flow for high curvature. Under curving flow we want the edges of our quad primitive to maintain their basic shape. One of the elements of our algorithm is ensuring that after each integration step the local advancing front remains generally orthogonal to the downstream direction. This maintains a quad mesh with desirable properties. Curvature within the flow field challenges the goal of maintaining edge shape. When flow lines are integrated through curving flow, the flow lines on the inside of the curvature follow a shorter path and the flow line front does not remain orthogonal to the flow direction, see Figure 3.8. This results in the underlying quads becoming sheared and warped. This gets progressively worse. Also the more rapid the curvature the more severe the shearing of the quad primitives.

To overcome this we apply a method suggested by Darmofal and Haimes [DH92] and presented in detail by Kenwright and Lane [KL95]. We use an adaptive step size integrator that measures the angle between velocity vectors and the quad base at each stage of the front advancement in areas of flow curvature. To produce an optimal result, the integration step size must be a function of the length on the adjacent streamline. We look at the interior angles of the quads in order to adjust the quads edge lengths as in Figure 3.7, which shows the case of a quad under right-rotational flow ($\alpha < 90°$ and $\beta > 90°$). We adjust the step-size between S_n^{i+1} and S_{n+1}^{i+1} such that the quad can maintain its shape (and accuracy) under rotational flow. In this case d_{i+1} is shortened proportional to the angle α using the relation $d_{i+1} = \frac{\alpha}{90} d_i$, where d_i is the step-size between S_n^{i+1} and S_{n+1}^{i+1}, d_{i+1} is the step-size between S_n^{i+1} and S_{n+1}^{i+1}. In order to avoid a jagged surface front under flow rotation we process rows of quads successively as a unit with common flow behavior. We need to ensure that the flow lines are adjusted in the correct order. Our stream and path surface front advancement uses two passes as shown in Figure 3.9. In the first pass we process the divergent and convergent quads and simply identify the quads under

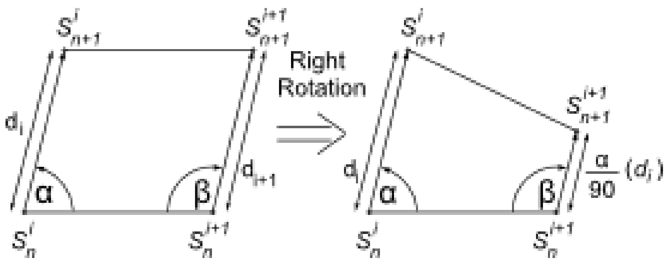


Figure 3.7: *Given a vector field under local right rotation, where $\alpha < 90°$ and $\beta > 90°$ the integration step-size between S_n^{i+1} and S_{n+1}^{i+1} is decreased by a factor proportional to the amount of rotation. Likewise for the case of left rotational flow.*

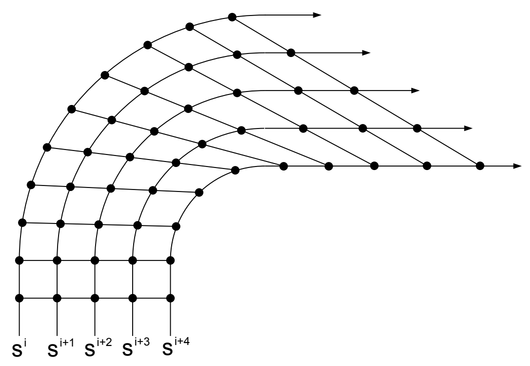


Figure 3.8: *Curvature in the flow field deforms the orthogonal tangent surface front and results in sheared quad primitives for the underlying surface mesh.*

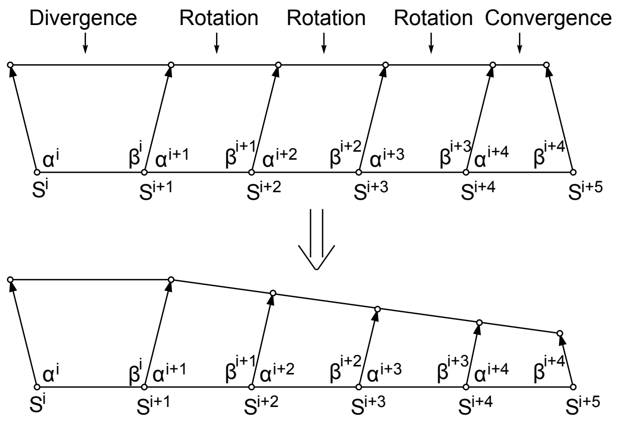


Figure 3.9: *In order to generate a smooth front of quads under rotation we group them and process them from left-to-right for the case of right rotation. Similarly for the case of left rotation.*

rotation. This results in a grouping of quads according to common flow behavior, e.g, a region of right-rotating flow.

For a strip of right-rotating quads we process them by marching from left-to-right order adjusting the edge lengths as shown in Figure 3.9. For left-rotating flow we perform the mirror opposite operation. We then iterate along each group adjusting the step-size of the latest segment. If flow starts to diverge we get two major groupings of flow lines: one group curving in each direction. The grouping method produces reasonable results with two fronts that expand out, one for each direction of rotation. In the extreme case of coinciding convergent and rotating quads, e.g., $\alpha << 90^\circ$, $\beta > 90^\circ$ and $d^i_{sep} + d^{i+1}_{sep} < \frac{1}{2} d_{sample}$ under right rotational flow, we first perform a merge followed by a rotation operation. Similarly for extreme cases of coinciding divergent and rotating quads. We note that no description of how to handle rotating flow is provided in previous literature [Hul90, Hul92, GTS+04].

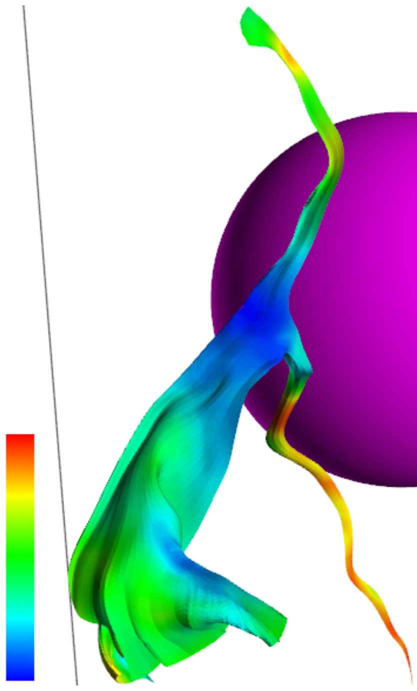


Figure 3.10: *When an object boundary is encountered the surface splits. The surface is torn and the separate portions are advanced independently of each other. Color is mapped to velocity magnitude.*

3.2.4 Splitting and Termination

If an object is encountered in the flow field, the surface can split into two sections that advance in separate directions. This is handled by terminating the flow lines contained in the surface that encounters a boundary (or other terminating condition). The sections on either side of terminated flow lines are then calculated independently. This prevents the whole surface advancement being halted unnecessarily and ensures that the separating portions maintain their accuracy and a well structured mesh. Figure 3.10 shows a stream surface splitting and the two sections advancing independently.

To handle splitting we introduce a new flag for the vertices. This indicates that a split has occurred. If we encounter a split flag while searching for the neighboring vertices, we simply halt the search for the vertices as we are on the edge of the surface. We then move onto the next valid point and search for its neighbors that comprise a quad. This allows multiple sections of the surface to be computed independently of each other.

Surface advancement is terminated when one of a series of termination conditions is met. The termination conditions are: the entire surface front leaves the vector field domain, the distance that the surface has travelled exceeds a preset geodesic length, or an area of zero velocity is encountered, i.e., a solid object within the flow field or a critical point.

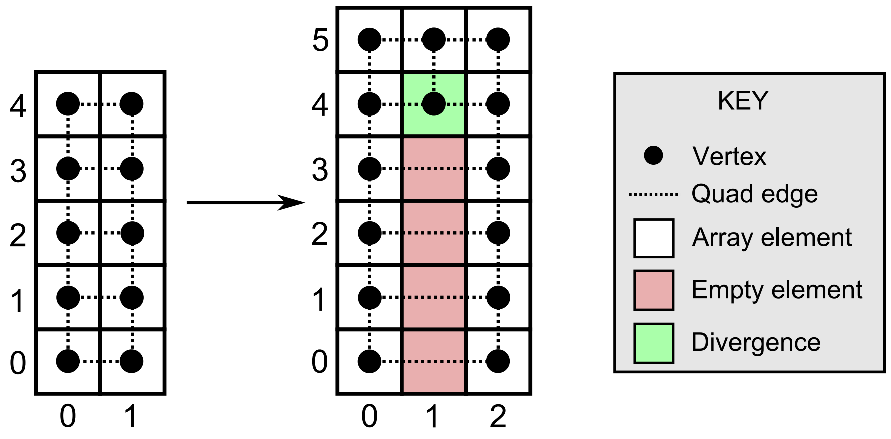


Figure 3.11: *This figure shows the change in memory layout after a divergence operation. It shows a minimal example starting with a pair of integral curves. The memory layout at two consecutive stages is shown – before and after a quad split. The rows and columns reflect the rows and columns in the 2D array used to store the mesh data. When the new integral curve is inserted between the pair, the array elements before the newly inserted seed point are marked as empty (the red boxes). The dotted lines show the mesh topology. A quad is formed by searching for neighboring vertices along the rows and columns of the array. If an element that is flagged empty is encountered it is skipped over and the search continues. This is shown in the right image where the elements [0,0], [0,1], [1,1], [1,0] are used to form the first quad. Elements marked as divergent (the green element in the figure) are skipped over when searching for vertices to form the quad front.*

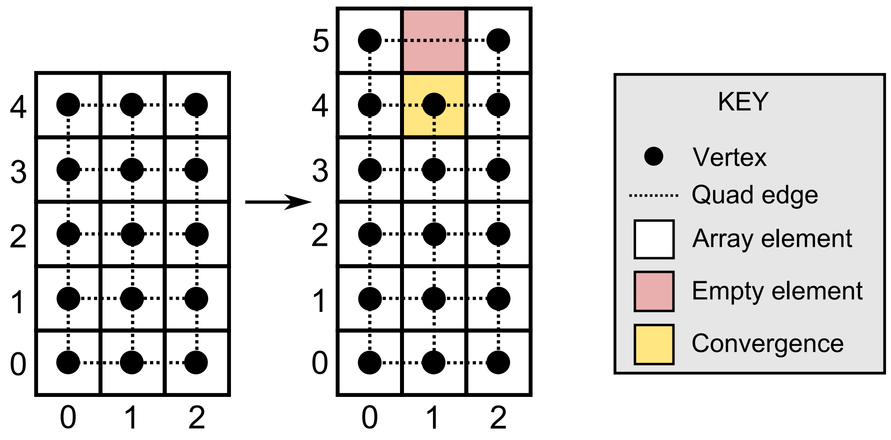


Figure 3.12: *This figure shows the change in memory layout after a convergence operation. The memory layout at the stages prior and after the convergence operation are shown. A convergence operation stops the advancement of a streamline. Subsequent elements in that the array column corresponding to the terminated streamline are marked as empty elements.*

3.2.5 Memory Layout

This section provides more details about the implementation and the data structure used in our algorithm. We demonstrate minimum working examples of how divergence, convergence and splitting are handled. Note that the shear operation does not affect the layout of the data structure – only the values of the vertices stored within it. We use a 2D array to store the mesh vertex information. The columns in the array correspond to integral curves.

Figure 3.11 shows a divergence case where the two integral curves are neighboring in the data array. A pair of integral curves is traced, forming a single strip of quads. When the front is advanced a quad split then occurs, the two images show the array layout before and after the split. The new integral curve is inserted further along the surface away from the seeding rake. In order to keep the array elements in sync with the integral curve points we mark elements in the array before the seed position as empty (the red elements in Figure 3.11). A quad is formed by connecting four neighboring vertices in the array, ignoring the empty elements and the divergent element when it would result in the creating the quad front edge. Note that empty elements can only be skipped across columns.

Figure 3.12 shows the a divergence case where the two integral curves are neighboring in the data array. Three integral curves are traced, creating a pair of quad strips. When the criteria for convergence are met, the central integral curve is terminated and the front pair of quads are collapsed into a single quad. The collapse occurs due to the updated array layout. Subsequent elements in the array column that corresponds to the terminated streamline are marked as empty elements. Thus, when the vertex neighbor search is performed to construct the quads, the empty elements are skipped over and the outer streamline vertices form the single quad. In order to improve performance, should the streamline represented in columns 0 and 2 of the array diverge at a later time and trigger a quad split, we can use the subsequent points in column 1 of the array that would have otherwise been marked as empty.

Updating the data structure as a result of the surface front splitting is handled in a similar

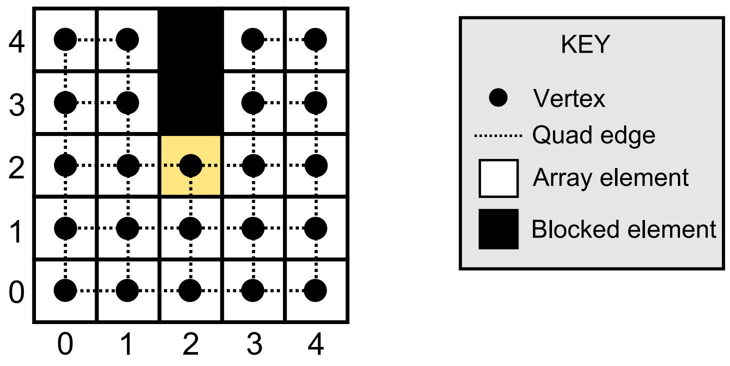


Figure 3.13: *Updating the data structure after the integral surface has split is handled in a similar fashion to convergence. However, instead of marking the subsequent array elements for the terminated streamline as empty, we use another flag marking them as a blocked element. During the quad construction, a blocked element can not be traversed. This ensures the surface front behaves as independent portions.*

way to convergence. When the surface encounters an obstacle or region of very low velocity magnitude, a streamline is terminated (or more than one). In the convergence case we set each of the subsequent elements of the terminated streamline to empty elements. In the splitting case we assign them another flag which we call blocked. A blocked element is similar to the empty element in that it does not contain any vertex information. However, it differs in regards to preventing the traversal of that element during the quad construction. It also prevents the traversal of the shear operation. This enforces the portions either side of the split remain independent of each other. See Figure 3.13.

3.3 Enhancements

A simpler surface generation scheme is not the only benefit of our algorithm. In this section we demonstrate that several enhancements stem naturally from its data structure.

3.3.1 Surface Painter

The first enhancement we discuss is called the surface painter. The surface painter is used to adjust the geodesic length of the surface (from the seeding rake to the surface front). Using a slider the user is able to interactively "paint" the surface in the downstream (or upstream) direction of the flow. This allows the user to easily observe how the surface evolves. It also addresses the challenge of occlusion resulting from overlapping portions of the surface, see Figure 3.14. With the interactive surface painter control the viewer can "rewind" the surface thereby revealing what would otherwise be occluded portions of the surface. The user can simply paint the surface automatically and view an animation of the surface front evolving step-by-step. A video demonstration of the surface painter is provided [MLZ].

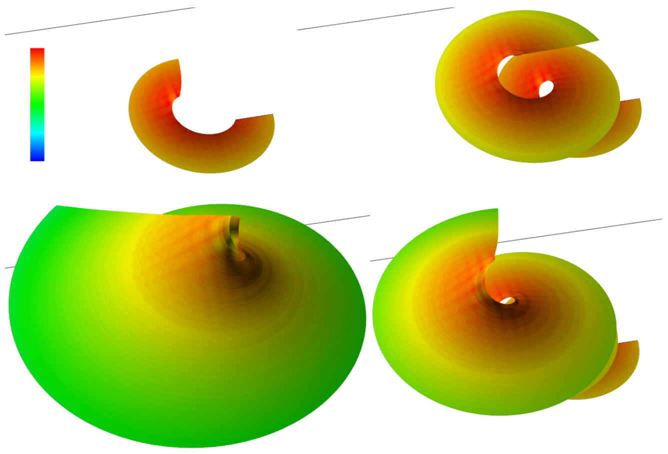


Figure 3.14: *The stream surface painter shows the evolution of the construction of the stream surface. Starting from the top-left image and commencing clockwise the stream surface painter has been set to render 25%, 50%, 75% and 100% of the stream surface respectively. The stream surface is colored according to the vector field magnitude.*

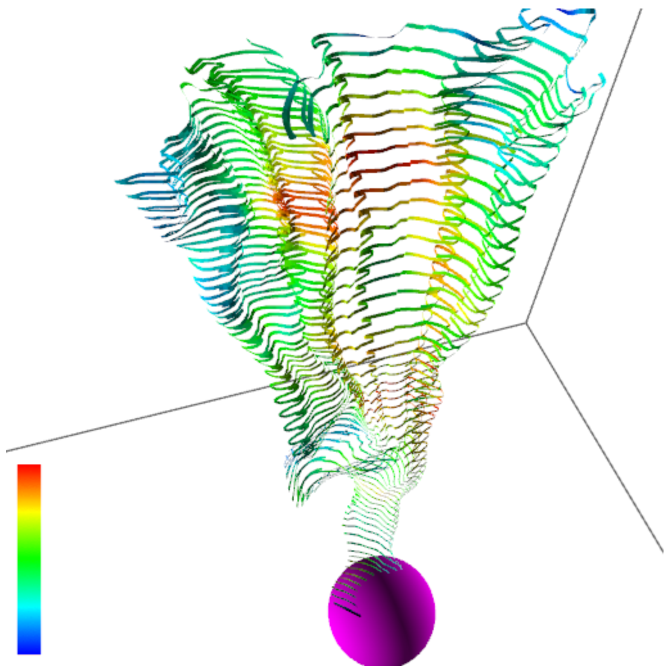


Figure 3.15: *This image shows timeribbons rendered as shaded strips of quads to visualize a smoke plume simulation. Color is mapped to local velocity magnitude.*

Although such an interaction would be possible with previous methods [GTS$^+$04, Hul90, Hul92] it would require a mesh parameterization. Previous methods require a parameterization of both the displacement of points along the streamlines ($s \in [0,1]$) and the advected images of the seed rake along the downstream displacements ($t \in [0,N]$). An explicit parametric description of the surface is unnecessary with our method.

3.3.2 Easy Timelines and Easy Timeribbons

A timeline is the line formed by connecting a series of massless particles in the flow seeded at the same instant of time (but at different locations). Although they are often mentioned as a very useful vector field visualization technique (because they show the convergent and divergent behavior of the flow), they are not commonly featured in a typical visualization software application. This may be due to complex implementation challenges. Timelines can easily be implemented from our algorithm by using the stream surface front at each integration stage and simply connecting these points.

Two minor adjustments have to be made to the surface construction algorithm in order to accommodate timelines. We use a flow line integrator whose step-size is proportional to velocity magnitude (as opposed to using an integrator with a uniform step-size). We also need to disable the rotation operations, described in Section 3.2.3.

The result are timelines that show the range of speed at which different regions of the flow travel. The advancing timelines benefit from the changes in sampling frequency in accordance with the divergence and convergence operations. Animating the timelines is a simple process that may produce insightful visualizations of the evolution of the flow [MLZ]. We extend timelines using quads as opposed to using line primitives, the result are a novel geometric visualization called *timeribbons*. Just as streamribbons extend streamlines to show curling flow behavior, timeribbons extend timelines to show oscillating (or wave-like) flow behavior, see Figure 3.15. We simply use a row of quads and render every n^{th} row. This forms ribbons that are easier to observe in a 3D scene by providing the perceptual benefits of using polygonal primitives over lines, e.g., shading. A video demonstration of animated timelines and timeribbons created by our application is provided [MLZ]. Timelines and timeribbons would be possible to incorporate into previous algorithms [GTS+04, Hul90, Hul92]. A surface parameterization, $S \in [0,1]$, $t \in [0, T_{max}]$, is required whereas timelines and timeribbons are a natural byproduct of our approach.

3.3.3 Easy Stream and Path Arrows

Stream arrows are arrow-separated portions of a stream surface. They enhance the basic stream surface visualization in (a minimum of) three ways: (1) The direction of the arrows shows the downstream direction of the flow. (2) The arrows (or their complement with respect to the stream surface) can be made transparent thus allowing the viewer to see through portions of the surface. This alleviates the occlusion problems caused by overlapping portions of the geometry. (3) They enrich wide stream surfaces with internal visual structure. Löffelmann et al. [LMG97, LMGP97] present two algorithms for the placement of stream arrows on stream surfaces. They involve the use of surface data structures that enable fast searching of the geometry in order to find a suitable place to map texture-based arrows.

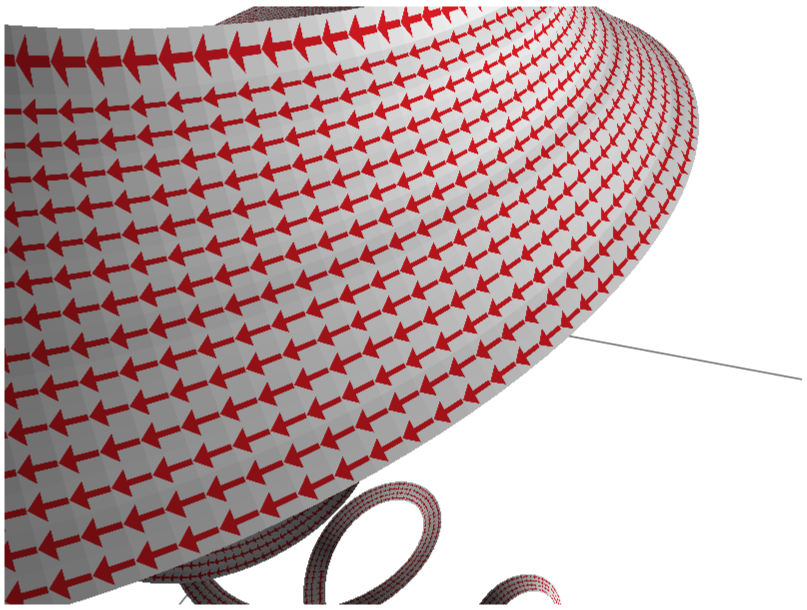


Figure 3.16: *An arrow texture can simply be mapped to the quad primitives to show the downstream direction of flow. The arrows can then be animated by scrolling the texture [MLZ].*

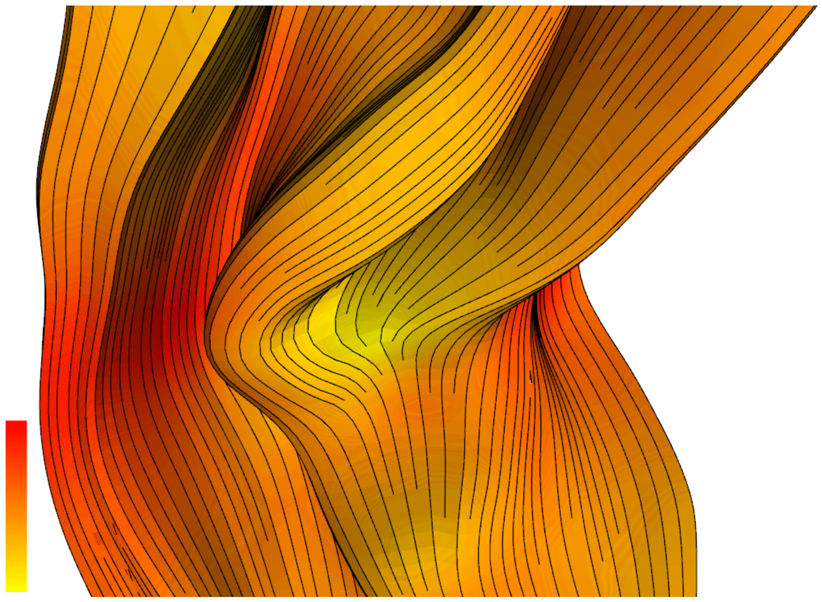


Figure 3.17: *We can create a hybrid visualization of surfaces and evenly-spaced flow lines. Color is mapped to velocity magnitude.*

Our mesh construction algorithm allows us to readily map an arrow texture onto the surface. The regular pattern generated by the quad primitives is ideal for tiling a texture: it requires no additional data structures and no search process as in previous approaches. It is also straight forward to add a user option that allows for the textures to cover several quads, thus increasing the size of the textured arrows in line with user preference and suitability of the current rendering. See Figure 3.16 for example results.

Animation of the arrows, can be added. This provides another insightful enhancement in which to visualize the flow characteristics of the surface. A video demonstration of animated stream arrows is provided [MLZ]. Stream and path arrows are more difficult to place if a triangular mesh is used.

3.3.4 Easy Evenly-Spaced Flow Lines

A range of algorithms have been proposed for the creation of evenly-spaced streamlines [JL97a, LS07, LM06, TB96]. Evenly-spaced streamlines or pathlines clearly capture the features of the flow field by distributing the streamlines or pathlines evenly on the surface. They also produce illustrative visualizations similar to those found in textbook depictions of hand-drawn flow. Note that evenly-spaced seeds do not necessarily result in evenly-spaced streamlines. Evenly-spaced flow lines are a simple by-product of our surface construction. The merge and divide operations ensure that $d_{sep} \approx \frac{1}{2}d_{sample}$. In fact, the d_{sep} parameter used throughout this chapter is very similar to the d_{sep} parameter used by Jobard and Lefer [JL97a]. Figure 3.17 shows how our easy integral surface algorithm can be used for the depiction of evenly-spaced streamlines. Evenly-spaced streamlines are not such an obvious by-product with triangular meshes.

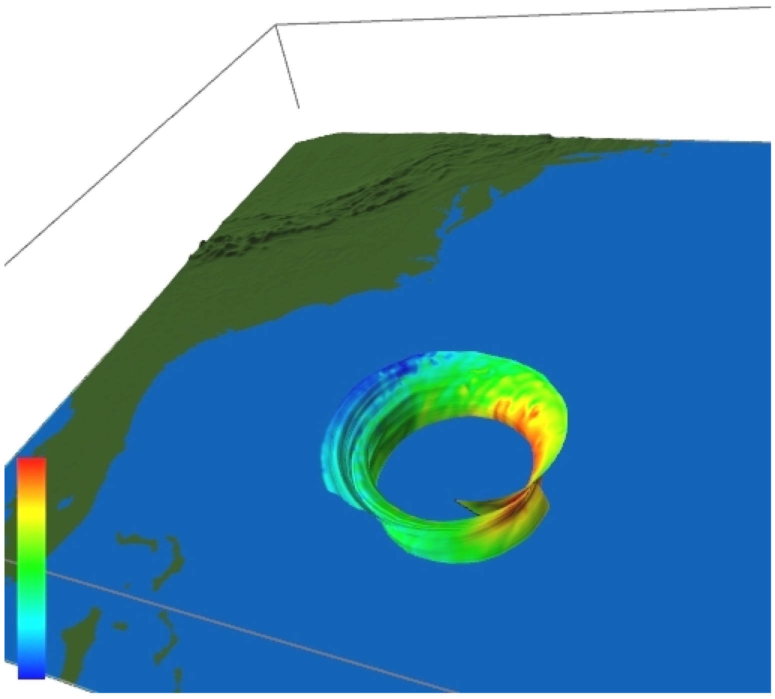


Figure 3.18: A stream surface depicting the eye of Hurricane Isabel.

3.4 Results

In practice our criterion for right rotation, $\alpha > 90^\circ$ and $\beta > 90^\circ$, proves too rigid. In the implementation we use a more relaxed constraint: $(\alpha > 90^\circ + \varepsilon_{rotation})$ and $(\beta < 90^\circ - \varepsilon_{rotation})$ where $\varepsilon_{rotation}$ is a user-defined threshold. We found a value of $\varepsilon_{rotation} = 3^\circ$ yields good results. In extreme cases of divergence, if $d_{sep} >> \frac{1}{2}d_{sample}$, i.e., $d_{sep} = n \cdot d_{sample}$ after advancing the front we can divide the quad and insert multiple new seeding points at a rate of $2n$, where $n = \frac{d_{sep}}{d_{sample}}$.

We also note the potential wasted memory in the 2D array. Elements marked as empty or split contain no explicit information with respect to the surface geometry. However, a mesh containing 1,000,000 vertices uses less then 16MB of memory. Compared to the large amounts of RAM in a typical PC nowadays (usually in the order of GB) that this is a minor issue in order to pursue an much simpler implementation.

Figure 3.18 shows stream surfaces used to visualize the Hurricane Isabel data set. One seeding curve has been placed in order to depict the eye of the storm. The algorithm was run on the original $500^2 \times 100$ resolution simulation. The local processing nature of our algorithm enables fast processing with large data sets. Figure 3.1 shows a path surface generated on the tornado data set of size 128^3. This stream surface was seeded close to the core of the tornado. The easy integral surface algorithm was implemented in C++ on a PC with a 2.66Ghz Intel Core 2 Duo processor with 4GB RAM and a 256MB nVidia GeForce 8600GT graphics card. We

	Stream surface size		
	10k quads	50k quads	100k quads
Euler	0.01s	0.04s	0.08s
RK2	0.02s	0.06s	0.12s

Table 3.1: *This table shows a range of stream surface construction times measured in seconds using our algorithm. Both an Euler and a second order Runge Kutta integrator are compared.*

tested 3 different sizes of stream surface, consisting of 10,000, 50,000 and 100,000 quads. Two different integrators were compared for each size of stream surface, namely an Euler integrator and a second-order Runge Kutta integrator. The results of these tests are presented in Table 3.1, three different size surfaces were created for each integrator, consisting of 10,000, 50,000 and 100,000 quads. The algorithm is fast even though it does not rely on any special-purpose GPU programming. In our review of the previous literature, the last report of performance times for polygonal stream surface construction was Hultquist [Hul90] in 1992, thus making performance time comparisons with previous work non-trivial. We note that the quad mesh lends itself to easy mesh simplification as a post-processing step for large meshes with smooth regions. See Daniels et al [DSSC08] for examples.

3.5 Conclusion

We present a novel integral surface construction algorithm whose simplicity readily allows for implementation and incorporation into visualization applications. The techniques benefits from the advances that quad-based approaches. The algorithm is fast and handles large high-resolution datasets due to the local nature of its processing. The algorithm comprises of operations that monitor local flow for characteristics such as rotation, convergence and divergence and does not require a parameterization of the surface. Our formulation is given explicitly in terms of data sampling thus making decisions regarding when and where to insert or delete new flow lines clear. The enhancements presented are easy stream and path arrows, easy timelines, easy timeribbons and easy evenly-spaced flow lines. We also presented the surface painter. All enhancements stem more naturally from quad meshes as opposed to triangular meshes. The implementation is CPU-based and uses OpenGL 1.2, thus making it widely portable. The availability of an efficient, simple-to-implement integral surface construction algorithm, which is immediately open to useful enhancements could, we believe, have a significant impact on the adoption of tangent surfaces as a visualization tool.

4 Constructing Streaksurfaces for the Visualization of 3D Unsteady Vector Fields

"*Spending your life waiting for the messiah to come save the world is like waiting around for the straight piece to come in Tetris...even if it comes, by that time you've accumulated a mountain of **** so high that you're ****** no matter what you do.*"
-Unknown[1]

Contents

STREAKLINES are a visualization technique commonly used to depict complex, time-varying phenomena. A streakline is the line joining a series of massless particles passing through a common point at distinct successive times. A streakline corresponds to the use of dye injection from a fixed point source in laboratory flow visualization.

Streak surfaces are the culmination of continuously seeding a curve from a fixed spatial location over time. However, despite the advantages and insight that streak surfaces enable

[1]A humorous quote I first encountered in a user's signature in the forums at www.gamedev.net. This website has provided valuable knowledge throughout my PhD.

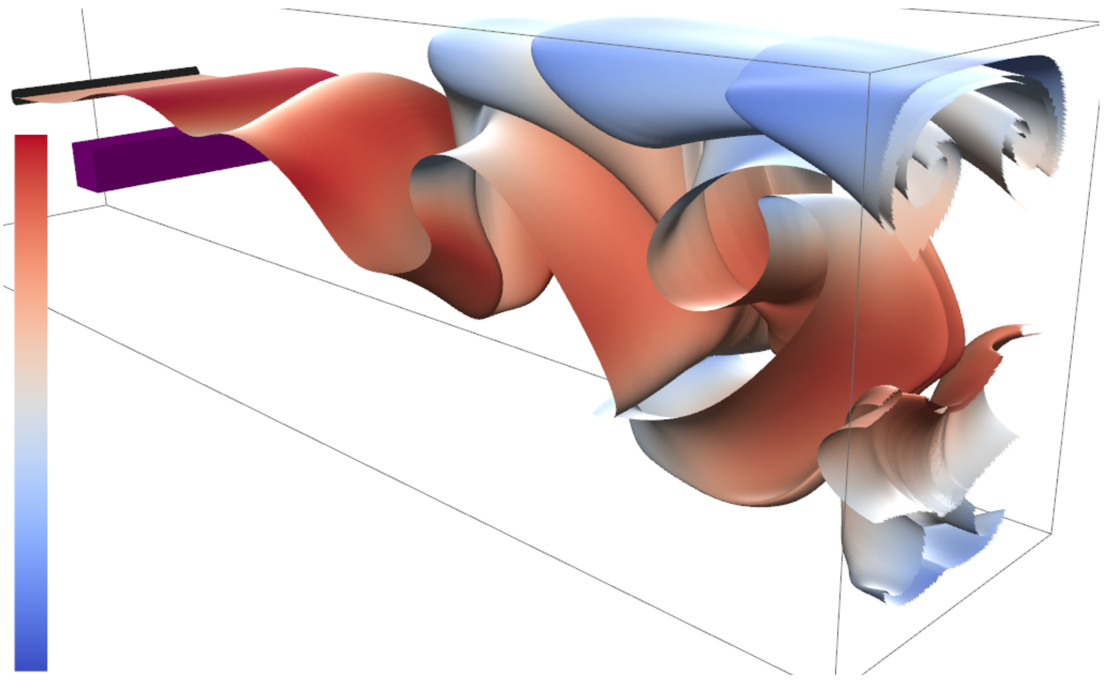


Figure 4.1: *A complete streak surface depicting flow past a cuboid computed and rendered at approximately 3 frames-per-second. This image shows complex regions illustrating interesting flow structures. Color is mapped to velocity magnitude. The full animation is given in the supplementary video.*

when investigating flow fields, they are not included into visualization applications. This is due to the computational complexity involved in generating their dynamic meshes. The computational cost of constructing streak surfaces stems from factors including the large number of integrations required. In contrast to stream and path surfaces, *every* vertex in the mesh must be integrated at *each* time step. For large meshes this results in a large number of computations per time step. Another major source of computational expense arises from computing the connectivity of the mesh vertices. Divergence, convergence and shear can occur anywhere in the surface at any time, not just the surface front. Due to the time-dependent nature of a streak surface a re-triangulation of the surface may be necessary after every integration step in order to avoid long, thin triangles resulting from divergence, convergence, and shear. These factors generally make the inclusion of streak surfaces into visualization packages prohibitively expensive. Previous streak surfaces algorithms are either (1) GPU-based: very fast but place more restrictions on mesh size and precision than a CPU-based version or (2) CPU-based but not fast enough to support interactive exploration of the flow.

We present a novel CPU-based streak surface algorithm (Figure 4.1) that can both run at interactive frame rates and places less restriction on mesh size and precision than a gpu-based implementation. The main contributions of this chapter are:

- The introduction of a novel streak surface algorithm that can offer both interactive frame rates and high precision.

- An algorithm that is suitable for use on large out-of-core data sets and relatively simple to implement as a result of the local operations performed on quads. Thus, it is suitable for inclusion in any visualization system.
- A streak surface model and implementation for the explicit treatment of shear flow.

We provide platform-independent, CPU-based pseudo-code (see Appendix B) in order to facilitate implementation into any visualization application. We provide user-controlled parameters that allow the user to trade off between performance and accuracy. This enables the user to switch between modes for quick investigation of the flow domain and high-accuracy representation for presentation and analysis.

The use of quad meshes has increased in recent years, with many algorithms aimed at quad-based re-meshing or simplification of quad-meshes. Some benefits of quad-based meshes are demonstrated by Alliez et al. [ACSD$^+$03] and Tong et al. [TACSD06].

However in order to generate such an algorithm several challenges must be overcome such as maintaining a continuous, accurate, quad mesh under flow convergence, divergence and shear. In Chapter 3 we demonstrate a strategy for creating stream- and pathsurfaces using quadrangular meshes. This chapter presents an extension of that algorithm to streaksurfaces. The rest of the chapter is organized as follows: Section 2 provides a discussion of previous work related to flow surface construction algorithms. Section 3 describes the computational model of our algorithm. A detailed discussion of the implementation is provided in Section 4. Section 5 presents an evaluation of the algorithm showing it applied to various simulation data sets. Finally, Section 6 concludes the chapter and identifies areas of future research.

4.1 Related Work

See Chapter 2 for a complete overview of geometric visualization techniques. Much effort has been focused on the construction of *stream surfaces* – surfaces everywhere tangent to a steady-state vector field. Hultquist introduced a method based on an advancing front [Hul92]. The sampling rate is adjusted by the insertion or removal of streamlines. In contrast to the local method of stream surface presented by Hultquist [Hul92], Van Wijk presents a global approach for stream surface generation [vW93b]. A continuous function $f(x,y,z)$ is placed on the boundaries of the data set. An iso-surface extraction technique can then be used to construct the stream surface. Scheuermann et al. devised a method where the underlying tetrahedral grid is used in the construction of the stream surface [SBH$^+$01]. Garth et al. [GTS$^+$04] present a method for the construction of stream surfaces in areas of complex flow. This is based upon the advancing front method introduced by Hultquist [Hul92]. Laramee et al. [LGSH06] combined texture advection with stream surfaces to provide a more detailed visualization by showing the inner flow structure of the surface. All of these previous methods are restricted to steady-state flow.

A point-based method for *stream* surface and *path* surface construction was introduced by Schafhitzel et el. [STWE07]. Insertion and removal of points are handled similarly to Hultquist's method [Hul92] to maintain sufficient density of points in order to create an enclosed surface when they are rendered using point-sprites.

Garth et al. present an improved *stream* and *path* surface construction algorithm focusing on high accuracy [GKT$^+$08]. This method decouples the surface integration and the surface rendering process. The surface construction comprises of advecting a set of timelines through the flow field. Connecting curves representing timelines are then computed. These curves are subject to given predicates in order to refine them where necessary.

McLoughlin et al. [MLZ09] demonstrate a simplified stream surface construction method using quad primitives. The refinement of the surface front is performed on a quad-by-quad basis. Quads may be split or merged to maintain sufficient sampling of the vector field. Shearing is handled by analyzing the surface front and processing groups of quads to make them more regular.

Schneider et al. [SWS09] present a method of stream surface construction using a higher-order interpolation scheme. This method provides very smooth surfaces of fourth-order accuracy.

It is important to note that all of the above approaches focus on stream surfaces and/or path surfaces, whereas the presented work focuses on streak surfaces.

Von Funck et al. [vFWS$^+$08] describes the construction of *smoke* surfaces. Smoke surface generation involves coupling the opacity of the triangles that comprise the mesh to their size and shape. The more the interior angles of the triangle deviate from 60° the more transparent the surface becomes. Mesh re-triangulation is avoided and this produces a surface approximating the smoke optical model. However, areas of complex flow become transparent by definition, thus interesting features may not be visible.

Krishnan et al. [KGJ09] present a novel streak and time surface algorithm. This technique guarantees a C^1 continuous curve for the integral curves. Three basic operations are defined for the surface adaptation process, these are *edge split*, *edge flip* and *edge collapse*. An edge split ensures that no edge on a triangle is longer than a prescribed threshold. Edge flipping locally refines an area to maximize the minimum angles within the triangles such that triangles are more regular. An edge collapse removes edges from the mesh in regions where the density of triangles is too high. They present a CPU-based implementation for unstructured meshes that runs on the order of hours, thus it does not support interactive visualization of the flow.

Bürger et al. [BFTW09] present two streak surface techniques implemented on the GPU. The first technique is based on quads. Each quadrilateral patch contains four vertices. The same vertex is stored (and propagated) multiple times. Refinement of patches is achieved by splitting the longest edge of the quadrilateral and the edge opposite it. This may result in discontinuities within the mesh. This is resolved during the rendering phase where the patch vertices are extended along the line that passes through the centroid and the vertex. A two-pass rendering operation ensures that the quads form a smooth surface during the rendering phase. Both the quad and triangle-based versions of the algorithm are very fast, e.g., several frames per second. However their GPU-implementation places unnecessary limits on both the size of the streak surface and the data set: All of which need to fit into GPU memory. Also, shear flow is not handled.

We present the first streak surface algorithm that combines the properties of speed, quality and size. It runs at fast frame rates supporting exploration of the flow and large data sets that do not have to fit into memory. It is also the first streak surface method to *explicitly* treat shear flow.

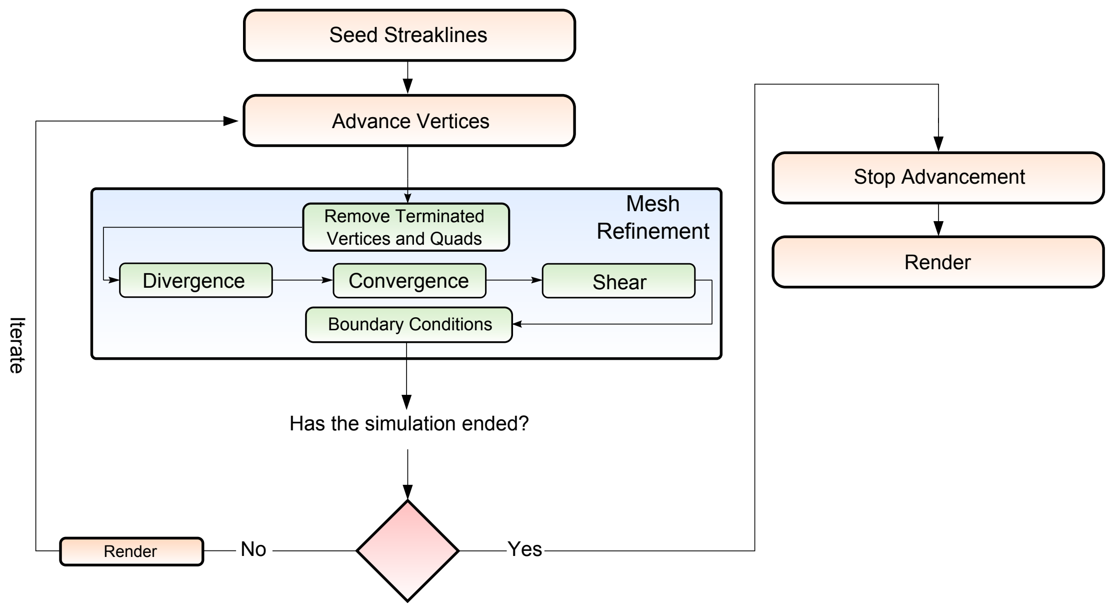


Figure 4.2: *An overview of the streak surface construction algorithm. The overall algorithm pipeline is similar to the pipeline from Chapter 3. However, due to the greater complexity of streaksurfaces many of the stages are completely re-designed.*

4.2 Computational Model

Our algorithm consists of a series of operations as illustrated in Figure 4.2:

1. Seed a curve using an interactive rake.
2. Update the streak surface by integrating every mesh vertex.
3. Refine the surface according to local deformation, by inserting/removing vertices/quads and updating the mesh connectivity.
4. Update the sampling rate of the vector field, by inserting and/or removing mesh vertices and joining them together using a quad-based topology.
5. Test boundary conditions such as object boundary intersection and zero velocity.
6. Render the surface.
7. This process iterates until the surface exits the space-time domain. At which point integration is terminated and the final surface is rendered.

Local operations are performed on quads and their neighbors which maintain sufficient sampling of the vector field. Vertex insertion is introduced when the Euclidean distance of a quad's edge is too long. Conversely a vertex is removed if neighboring points are too close. Mesh connectivity is then resolved ensuring all quad primitives are generally regular and that the surface is smooth. What follows is a description of each stage of our algorithm along with a description of the technical challenges and how they are addressed.

4.2.1 Streamlines, Pathlines, Streaklines, and Timelines

Trajectories of a vector field are streamlines. They are solutions to:

$$\frac{d\mathbf{x}}{ds} = \mathbf{v} \text{ or } \frac{dx_i}{ds} = \mathbf{v}_i(x_1, x_2, x_3, t) \tag{4.1}$$

Where s is a streamline parameter. Integral equation (4.1) results in streamlines *at the instant t*. Intuitively streamlines correspond to the path a massless particle traverses in steady flow. If we vary the temporal parameter, t, then we obtain *pathlines*: the path a single particle travels in unsteady flow.

The term streakline is used to denote the curve traced by a massless substance that is injected into the flow at a fixed point continuously over time. At time t, a streakline passing through a fixed point $\mathbf{y}$, defines a curve from $\mathbf{y}$ to $\mathbf{x}(\mathbf{y},t)$. The position of a particle coincides with this curve if it passes through $\mathbf{y}$ between time ϕ and t. The equation of a streakline at time t can be given by:

$$\mathbf{x} = \mathbf{x}[\zeta(\mathbf{y}, s), t] \tag{4.2}$$

where ζ is the initial coordinates of the point and where the parameter s lies in the interval $\phi \leq s \leq t$ [Ari90]. Intuitively a streakline is the line joining all vertices passing through a position $\mathbf{x}$ at successive times. Dye injection/tracing is a very common method in laboratory flow visualization and streaklines allow for a direct comparison to this technique. A *timeline* is a line joining a series of vertices all released into the flow at the same time. Note that for steady vector fields, streamlines, pathlines, and streaklines are the same.

4.2.2 Interactive Streak Surface Seeding and Advancement

We use an interactive seeding curve that allows the user to generate a variety of streak surfaces at run time and explore the whole domain. The user interactively controls the position and orientation of the seeding curve as well as its length and the number of streaklines emanating from it. The default separating distance, d_{sep}, between streakline seeds is $\frac{1}{2}d_{sample}$ i.e., half the distance between neighboring data sample points on the underlying grid. This conforms to the Nyquist limit, namely, the sampling frequency must be (at least) twice that of the underlying data frequency for accurate reconstruction. This formulation is advantageous in that it can automatically adapt to changes in the data resolution i.e., adaptive resolution sampling. The

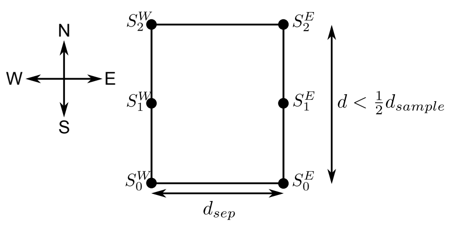


Figure 4.3: *Line segments joining points along streaklines can be used to form quads. Edge lengths are $\leq \frac{1}{2}d_{sample}$. North is the direction pointing downstream from the seeding curve.*

seeding distance between points may be adjusted by the user if a more dense sampling is required or if faster performance is a priority.

As streakline points are integrated, we compute their distance from the seeding rake. As soon as a point, S_i, exceeds a distance of $\frac{1}{2}d_{sample}$ from the seeding rake, points S_i^W and S_i^E are joined to form a quad (see Figure 4.3). This allows us to adhere to the Nyquist limit by setting the advancement length to be half (or less) of the distance between the sample points of the underlying grid. This simple scheme allows the connectivity of the quad mesh elements to be composed of the corresponding points between adjacent streaklines, i.e., the points S_i^W, S_i^E, S_{i+1}^E and S_{i+1}^W, where i denotes the i^{th} point on the streakline (see Figure 4.3). We use the convention that north is the direction pointing downstream from the seeding curve.

As the streaklines elongate, their shape reflects the underlying characteristics of the unsteady flow. During their evolution, they encounter areas of shear, divergent and convergent flows. These areas require special handling.

4.2.3 Divergence

Divergence is a common characteristic in flow phenomena. When visualizing an area of divergent flow with a surface technique, the surface expands. Divergence in this case is defined when distance between the underlying flow lines used to construct the surface increases. By flow lines we mean timelines and streaklines. Streak surfaces present unique challenges for cases of divergent flow. Streak surfaces are dynamic in their entirety and therefore, divergence may occur anywhere within the surface. It is not restricted solely to the surface front.

Not only may the distance between adjacent streaklines increase, but the distance between neighboring points on the same streakline may increase as in Figure 4.4. In our framework, no distinction must be made between neighboring streakline or timelines due to the quad's symmetry. Our algorithm processes the quad mesh in a local quad-by-quad fashion. The edge lengths that define the quads are tested at each time-step to ensure that the streak surface maintains an optimal sampling frequency, i.e., to prevent under-sampling. To handle this, we simply divide the quad based on edge lengths longer than d_{sep}.

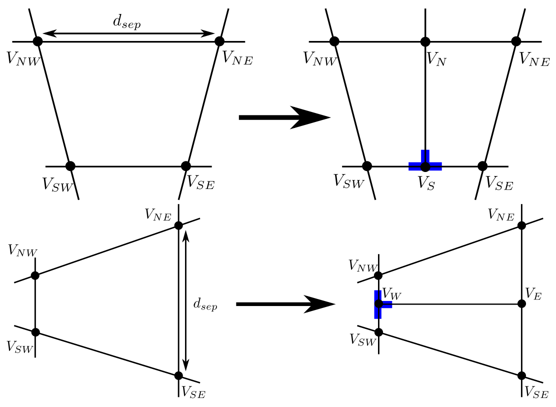


Figure 4.4: *Our algorithm examines each quad edge. When $|V_{NE} - V_{NW}| > d_{sep}$ a quad is sub-divided. The same holds for when any quad edge length exceeds d_{sep}.*

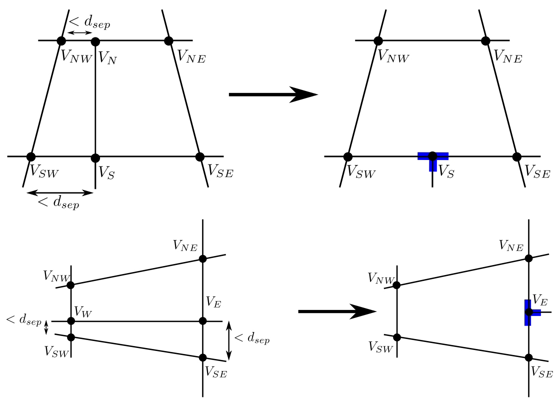


Figure 4.5: (Top) When $|V_{SW} - V_S| < d_{test}$ and $|V_{NW} - V_N| < d_{test}$, we collapse the two quads into a single one. (Bottom) When $|V_W - V_{SW}| < d_{test}$ and $|V_E - V_{SE}| < d_{test}$, we collapse the two quads into a single one. $d_{test} = d_{sep} - \varepsilon_{conv}$, where ε_{conv} is $\frac{d_{sep}}{2}$.

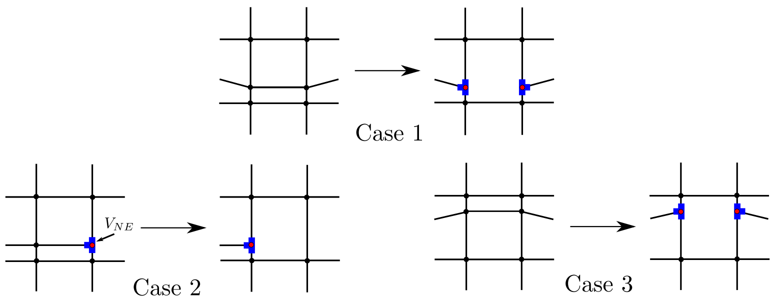


Figure 4.6: This figure shows changes in topology due to convergence. For each case the left hand side illustrates the state of the topology before the convergence operation has been applied. The right hand side shows the updated topology after the merge has taken place.

4.2.4 Convergence

Convergence occurs when the distance between flow lines decreases. As a consequence regions with a high density of points occur in the surface mesh. A high concentration of vertices in a local area may result in unnecessary oversampling. To prevent this we look at pairs of adjacent vertices and test the edge lengths we test for very thin quads, collapsing a pair of quads into a single one as in Figure 4.5.

Multiple ways of handling convergence are possible. Typically the removal of vertices is performed. Advancing front-based methods terminate an individual streamline/pathline and form a ribbon comprising of the neighbors of the terminated trace line. Garth et al. [GKT$^+$08] propose not removing these vertices, claiming that the cost of handling convergent cases offsets the cost of integrating these additional points. There is also the possibility that the particular region of the surface may diverge again in future resulting in points being re-introduced, this may be avoided, or at the very least alleviated, if convergent points are not removed.

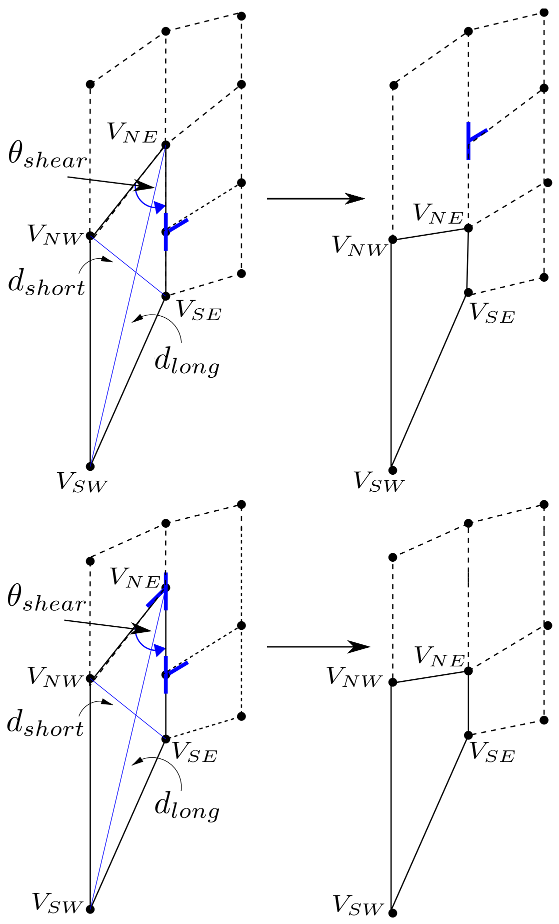


Figure 4.7: *When a sheared quad is encountered a change in mesh topology results. In this example, the sheared quad has a T-Junction on its east edge, E_E. The Vertex, V_{NE}, is now updated to produce a more orthogonal quad. In the top row a T-junction is added to the north quad. $\theta_{shear} = 5^\circ$*

When converging the quads together, careful attention must be paid to the changes in mesh topology. The special cases are shown in Figure 4.6. The left-hand side of each case shows the mesh topology before mesh convergence is applied while the right-hand side shows the converged quads. In the implementation we first test to see which topological case the quad is in before updating the topology. More implementation details are given in Section 4.3 and in the supplementary material.

4.2.5 Shear Flow

Shear in the flow field presents difficult challenges when constructing a surface representation. Shear flow is problematic due to warping of the quad primitives. We define a pair of simple tests to determine the deformation of the quad. If the test indicates the quad is malformed we

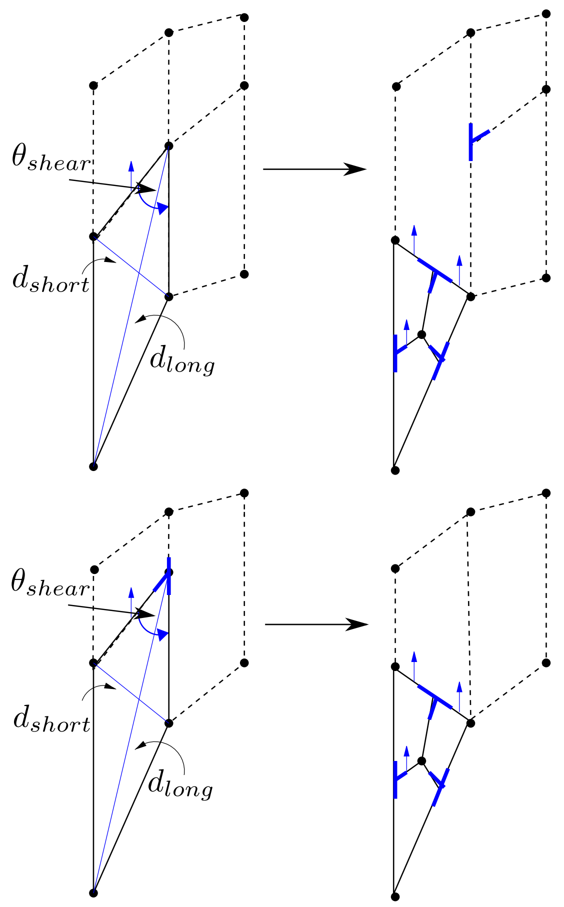


Figure 4.8: *When a sheared quad is encountered a change in mesh topology results. In this example, the sheared quad has no T-junction on the east edge. The updated topology forms an intermediate triangle in the mesh. In order to maintain a quad-based topology we decompose the triangle into three quads using a method adapted from Alliez et al. [ACSD$^+$03].*

then alter the local topology of the mesh to produce a more regular mesh while maintaining its accuracy. For the first test we compute the diagonals of the quad. The ratio $\frac{d_{short}}{d_{long}}$ can be used to quantify the amount of local shear. A perfect quad has a ratio of 1:1. If $\frac{d_{short}}{d_{long}} > \varepsilon_{shear}$ we then move onto the second test. The second test checks the subtending angle V_{NE}. If this angle is below the threshold θ_{shear} we then consider the quad as malformed and update the mesh connectivity ($\varepsilon_{shear} = 0.3$ and $\theta_{shear} = 5°$).

Figures 4.7 and 4.8 show how we handle shear flow. In Figure 4.7, the simple case, a T-junction is located on the edge E_E of the sheared quad (left column). When $\frac{d_{short}}{d_{long}} < \varepsilon_{shear}$ and $\theta_{shear} < \varepsilon_{\theta-shear}$ we re-connect the north edge, E_N, to the T-junction on E_E (right column). In the top row a new T-junction is introduced. In the bottom row, two T-junctions snap together.

In the second case, Figure 4.8, the left column shows the sheared quad before any update is made to the mesh topology while the right column depicts the updated topology. If $\frac{d_{short}}{d_{long}} <$

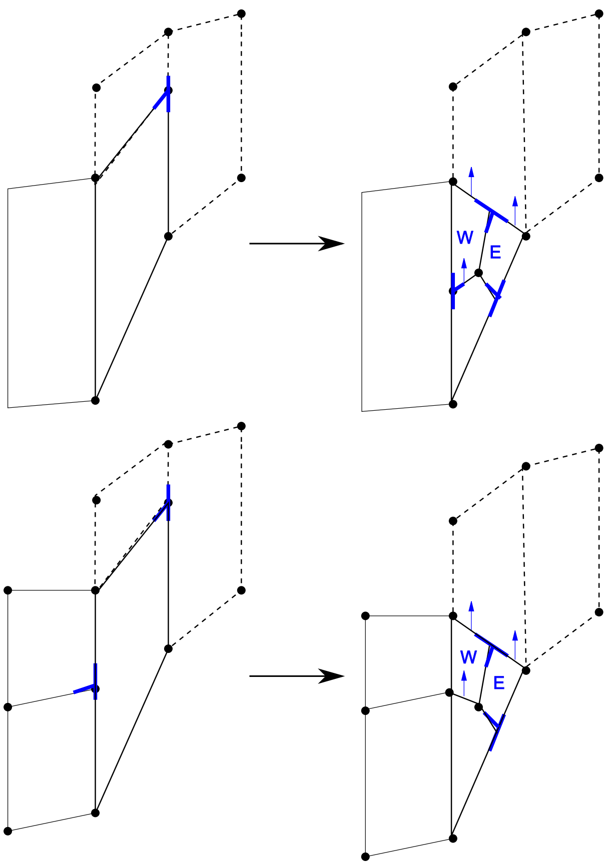


Figure 4.9: *Left column: sheared quad, Right column: Updating the topology. When we subdivide the intermediate triangle, we test for existing T-junctions along its edges. If no T-junction is present, as in the top row, we insert a new vertex and interpolate it's position along the triangle edge. In the bottom row there is a T-junction present on the west edge of the triangle. In this case we simply use this vertex and it's position.*

ε_{shear} and $\theta_{shear} < \varepsilon_{\theta-shear}$, we re-connect the sheared quads north-east vertex to the eastern neighbor's southwest corner. This forms an intermediate triangle. The triangle is then subdivided into three quads – a method inspired by Alliez et al [ACSD$^+$03]. In the top row, a new T-junction is created. In the bottom row, a T-junction is removed. The north direction of the new quad is depicted in Figure 4.8. This approach has the advantage that the original streakline points are re-used to handle the shear. Vertices are not moved, the changes lie in the mesh topology.

There are also two possible cases when we subdivide the intermediate triangle. For each edge of the original quad we test to see whether there is a T-junction present. If there is no T-junction we insert a new vertex on that edge and interpolate its position at the mid-point of the two edge vertices, this case is depicted in terms of the west neighbor of the original sheared quad in the top row of Figure 4.9.

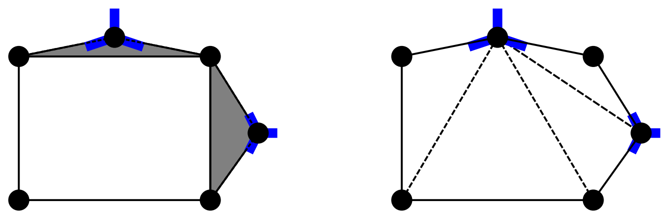


Figure 4.10: *The image on the left shows a quad that contains T-junctions that do not lie on the quad's edge. The grey areas represent where cracks would be encountered in the mesh. The right image illustrates how our implementation handles this. A triangle fan is used, whose base is one of the T-Junction vertices. The triangle fan then proceeds in a clockwise manner connecting to the available vertices. This configuration allows for any combination of T-junctions.*

4.2.6 Surface Discontinuities

It should be noted that handling divergence and convergence as in Figures 4.4 and 4.5 can cause T-junctions – a consequence of using quad meshes. T-junctions can be handled in a number of ways. Firstly, the vertices at the T-junction can be snapped to the edge causing the junction. For example, dragging V_S in Figure 4.5 (top) to edge E_S. Secondly, cracks appearing at T-junctions can simply be patched with triangles. The triangles are ignored during the integration and topology update phases. They can be re-applied afterwards in a separate pass. A third option is to only insert entire streaklines or timelines. This is recommended by Becker et al. [BLM95]. The third option is costly in terms of performance. The first option involves moving points that have been previously calculated by the integration scheme and thus introduces error. This error may also accumulate over time.

In order to handle surface discontinuities we replace a quad that contains one or more T-junctions with a triangle-fan. The triangle fan uses one of the T-junctions as its central vertex. It then proceeds clockwise adding vertices. We test each edge to see if it contains a T-junction, if one exists its vertex is used in the triangle fan, if no T-junction exists we move onto the next quad vertex. Figure 4.10 illustrates this. If we don't consider T-Junctions when rendering, discontinuities may appear in the mesh. This happens when the T-Junction vertex does not lie exactly on the quad edge to which it belongs. The triangle fan is ignored during the integration and used for rendering only.

4.3 Implementation

Our implementation involves three key objects: vertex, quad and T-junction objects. In this section we discuss these objects and how they relate to each other. We also provide the pseudo-code necessary for implementing the divergence, convergence and shear operations in Appendix B.

4.3.1 Mesh, Vertex, and Quad Objects

Mesh vertices contain three floats, each representing the x, y and z spatial components. They are stored in a central list. The ordering of the vertices may be arbitrary. The ability to compute vertices in any sequence allows us to add new vertices at the back of the list without re-ordering. The list also allows the removal of vertices without moving all subsequent elements.

Our quad objects are used to maintain the surface topology. They consist of four pointers to mesh vertices (which define the quad in the physical domain) and four pointers to neighboring quads, one for each direction: NORTH, EAST, SOUTH and WEST (where a NULL pointer indicates no neighbor). They also contain pointers to T-junction objects. A quad has a maximum of one junction and two quad neighbors per edge. If a pair of T-junctions is added on opposite edges of the quad, we simply split the quad. Quad objects are stored in a vector. Like the mesh vertex array, quads can be stored in any order. All divergence, convergence and shear operations and rendering are performed on a per-quad basis by simply iterating over the vector. The order of the operations is important. Quads are tested in the following order (1) divergence, (2) convergence and (3) shear (as in Fig 4.2). Also only a single operation is performed on a quad in a single pass through the surface in order to simplify implementation: either divergence, convergence, shear or no operation. A convergence or shear operation may sometimes result in a divergence operation in the next pass.

4.3.2 T-Junction Objects

T-junction objects are used to handle the transition in mesh resolution and a T-junction occurs when a single quad's edge is neighbored by two quads. This can occur in cases of convergence and divergence. T-junction objects contain a pointer to the extra neighbor of the quad and a pointer to the extra vertex, e.g., Q_W in Figure 4.11. When a quad that contains a junction splits we use the T-Junction vertex to ensure that the split quad shares the relevant vertices with its neighbors. This also serves to prevent the unnecessary insertion of extra vertices. In

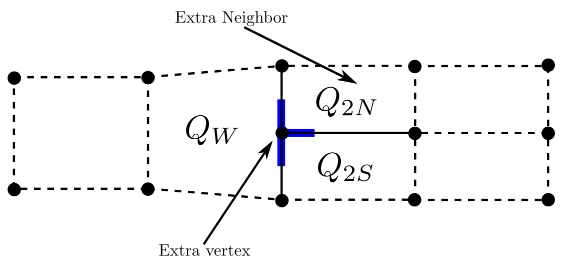


Figure 4.11: *A T-Junction object. T-Junctions occur at a transition in mesh resolution caused by the divergence and convergence operations. The quad, Q_W stores the T-Junction object. The T-Junction object contains a pointer to the T-junction vertex, which can be used when this quad splits. This prevents duplicate vertices being inserted into the mesh. It also contains a pointer to the extra quad neighbor. The extra neighbor pointer is used when Q_{2N} splits, the newly inserted quad will point to Q_{2N}. Storing this explicitly prevents a search within the local region of the mesh, thus speeding up the computation.*

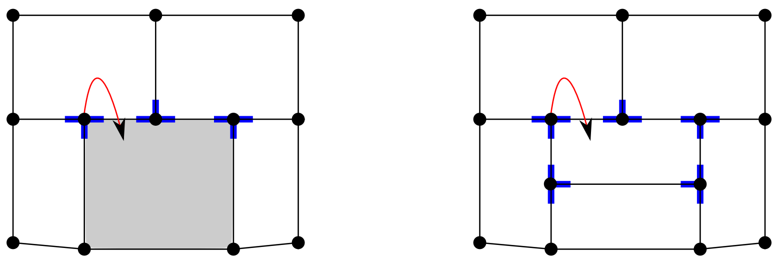


Figure 4.12: *A subtle divergence case. We test whether the quad to the north is pointing to this quad or if it has a T-Junction that is pointing to this quad. We then update that pointer to point to the newly inserted quad. If this is not done the north quad will point to the incorrect quad.*

the case where we add a T-junction object to a quad which already contains a T-junction on the opposite edge, we simply split the quad using the two T-junction points. T-Junctions are allocated dynamically when needed. They are removed when the quad is removed or when it splits.

4.3.3 Updating Mesh Topology

Appendix B provides a detailed discussion of the changes to topology that arise due to divergence, convergence and shear. We provide detailed diagrams and pseudo-code in order to facilitate implementation by others. Figure 4.12 depicts a subtle case where care must be taken when updating the mesh connectivity. This ensures that each quad is pointing to the correct neighbors.

4.4 Results

The reader is encouraged to view the accompanying video. In Figure 4.1, a complex streak surface is constructed. This surface shows interesting flow structures that would be difficult to see if only streaklines were rendered. Surfaces also aid in identifying the transition of turbulence be showing stretching and folding. Our system constructs and renders this streak surface at approximately 3 frames per second (fps).

Figure 4.13 (top) depicts a complete streaksurface exhibiting a multitude of flow characteristics. This surface contains regions of divergence, convergence and shear as well as the splitting behavior when an object is encountered. This surface is computed and rendered at roughly 2 fps. Color is mapped to velocity magnitude in all our examples unless stated otherwise. Figure 4.13 (bottom) depicts another semi-transparent streak surface generated from the simulation of flow past a cuboid. The result demonstrates the tearing of the surface into two independent regions when an object boundary is encountered. Splitting is detected when the velocity magnitude of an internal streakline drops below a given threshold. When the surface tears, the separated wavefront are advanced independently. Figure 4.14 shows the streaksurface mesh of a surface generated from the simulation of flow behind a square cylinder. These images show that under strong deformation the quadrangular mesh is well-behaved. The handling

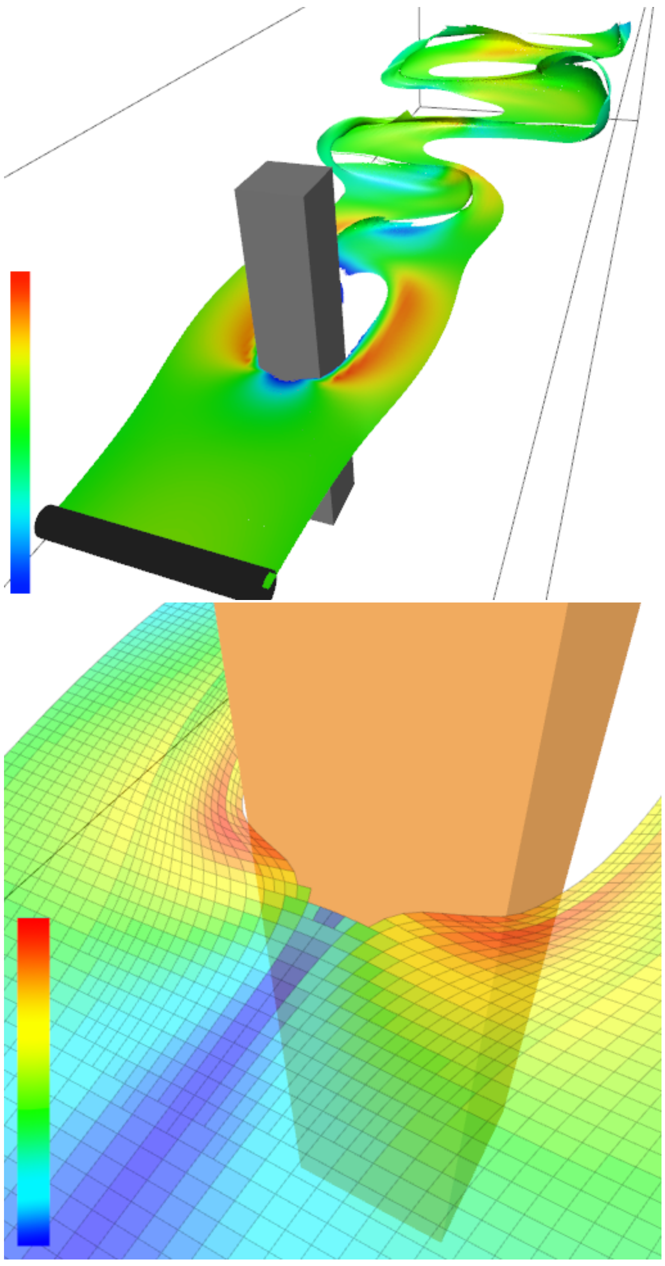


Figure 4.13: *(Top) A streak surface of the simulation of flow past a square cylinder. The image shows a late stage into the simulation and shows the complete surface exhibiting divergence, convergence, shear and splitting behavior. (Bottom) When an object boundary is encountered the surface tears and moves around the boundary. This surface encounters a cuboid. As the surface splits the separate portions are constructed independently of each other. Colour is mapped to the local deformation of the quad primitives (how much they deviate from being a regular quad.)*

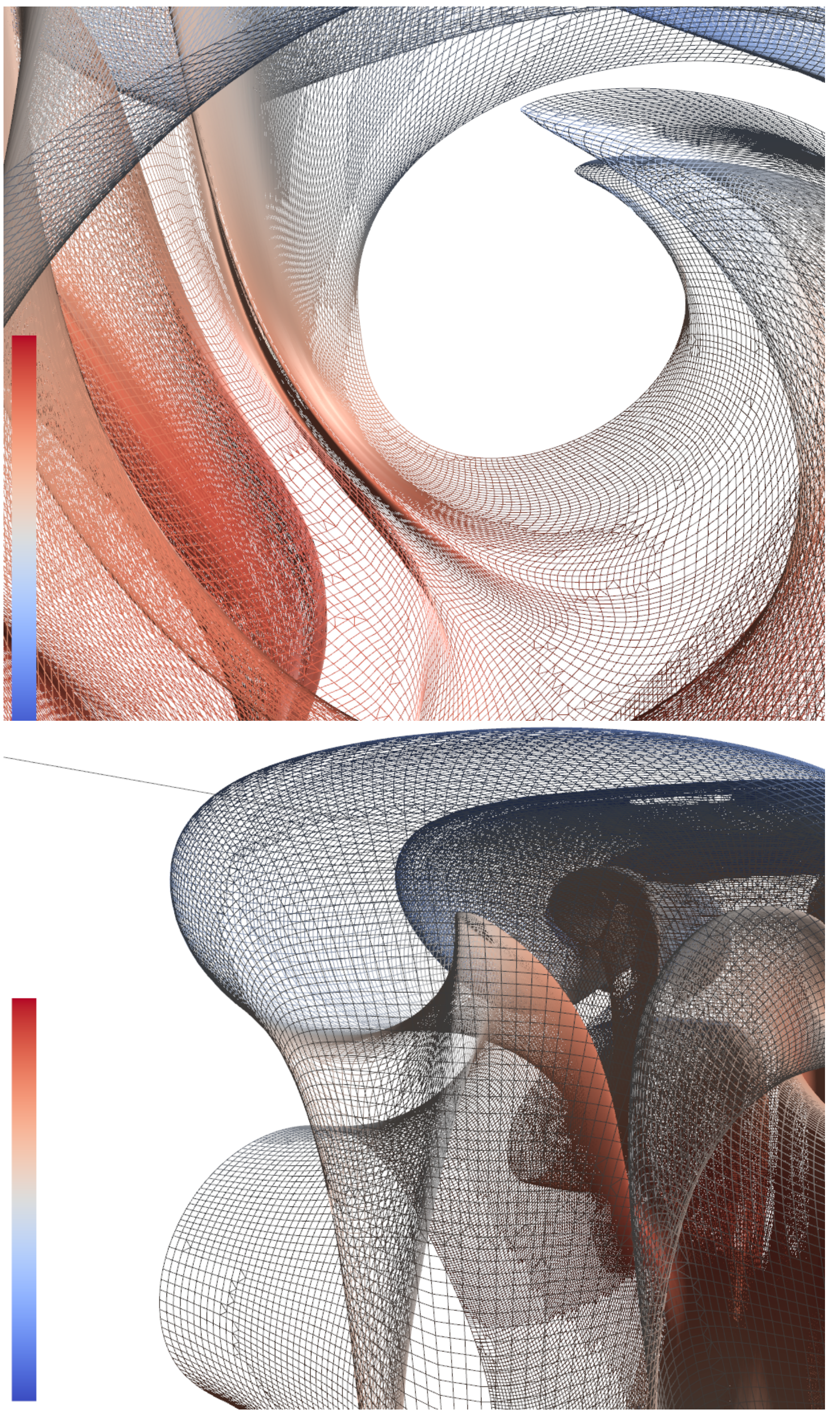


Figure 4.14: *Even under strong deformation, the mesh remains well-structured and produces a smooth surface. These images also demonstrates the use of triangle fans to render quads that contain t-junctions.*

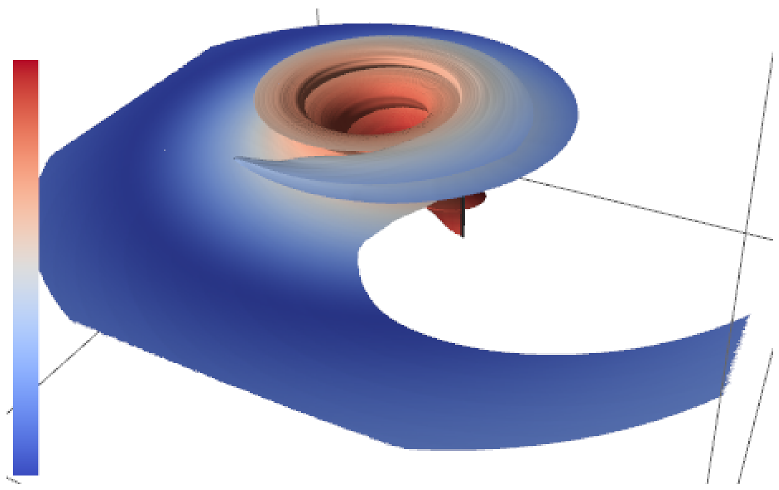


Figure 4.15: *A streaksurface depicting a tornado simulation. The surface encounters large amounts of shear that accumulates as the simulation progresses.*

of t-junctions is also demonstrated. A triangle fan is used to prevent any surface discontinuities appearing as cracks within the mesh. Figure 4.15 shows streak surfaces generated on a time-dependent tornado simulation. The tornado exhibits large regions of shear flow and is ideal to test the robustness of our shear-handling method. Our implementation renders this at 10 fps.

Large Data Sets Our system supports out of core memory management in order to handle large data sets. We adopt a method similar to Bürger et al [BSK$^+$07]. We store as many time-steps as possible in main memory. We then employ a sliding window. When performing the particle advection, we interpolate between a pair of time-steps. The management of time-steps is performed in a separate thread. The main thread is dedicated to the computation of the surface. This provides a benefit on multi-core systems as the blocking I/O call does not prevent the rendering and construction of the surface, while the relevant time-steps are loaded. This enables simultaneous streaming of data and the construction of the surface. Care must be taken to lock the portion of working memory in order to prevent incorrect values from being used in the surface computation.

Figure 4.16 depicts a streak sheet on the full resolution (500x500x100x48) Hurricane Isabel simulation. A streak sheet is created by simply terminating the insertion of new points at the seeding rake after a period of time. In this case the sheet was released so that it was captured by the eye of the hurricane. It then traces the hurricane's path. Figure 4.17 depicts a streak surface of an Ion Front Instability simulation. This is a high resolution turbulent data set with a resolution of 600x248x248 and consists of 200 time-steps.

Performance Table 4.1 shows performance timings for a sample streak surface generated on the Tornado simulation. The table presents the performance timings for the integration phase and the subsequent updating of the topology. The results show that the integration phase comprises a large proportion of the computational effort – typically 40-80% of the computation

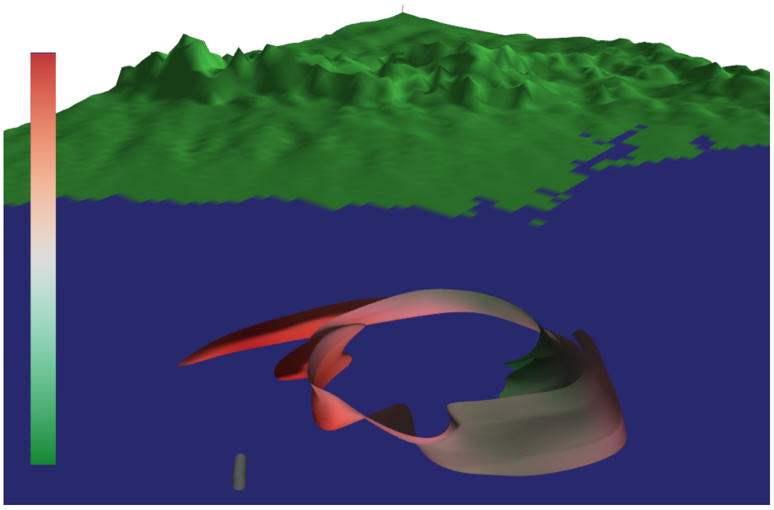


Figure 4.16: *This image depicts the surface generated from the full resolution (500x500x100x48) of the Hurricane Isabel simulation. In this image we stop seeding the surface after a period of time and advect the sheet. Here the sheet is caught by the eye of the Hurricane.*

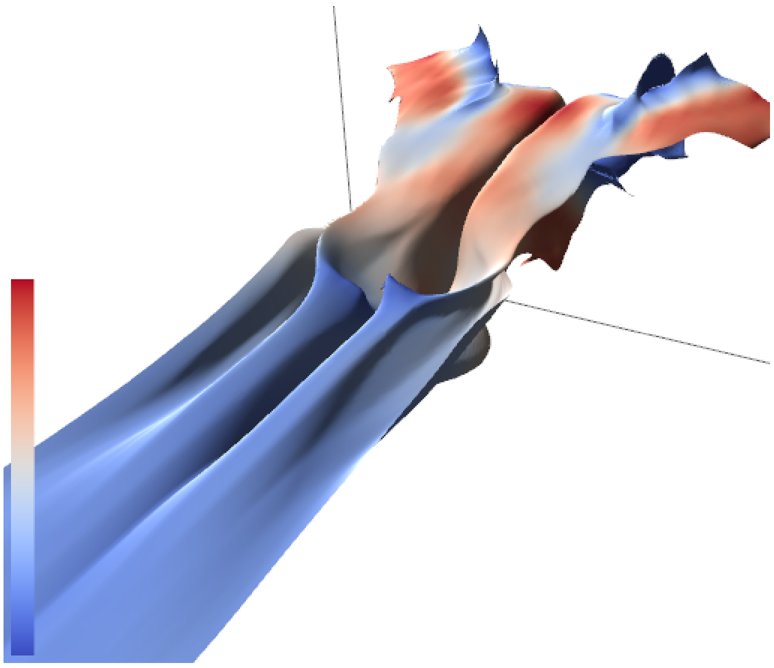


Figure 4.17: *A streak surface on the Ionization Front Instability simulation (from the IEEE Vis'08 contest). This is a turbulent data set with 200 timesteps at a resolution of 600x248x248.*

No. Vertices	RK2 Int.	Update Mesh: Divergence, Convergence, Shear	Total
10	16	15	31
20	46	15	61
50	109	32	141
80	174	34	208
100	203	47	250
200	406	93	499
500	1030	1310	2340

Table 4.1: *Performance times for our implementation, running on the Tornado simulation, given in milliseconds. We report both the time taken for the Runge Kutta 2 integration of the vertices and also the time taken to update the mesh topology after the integration phase. The results show that the integration process takes a large proportion of the computation time.*

time. The mesh vertex advection is a largely parallel component, however our implementation performs this operation within a single thread. A multi-threaded version of this stage would greatly reduce the cost of this stage and consequently speed up the algorithm. However even in its current single-threaded state the algorithm generally still performs at interactive rates. While not as fast as a GPU-based version [BFTW09] the mesh does not need to fit in graphics card memory, and thus handles larger streak surfaces. In principle, previous algorithms for unstructured data would run faster on structured data [KGJ09]. However, no implementation has been provided that demonstrates this. The bottom row in the table also shows a case of the updating of the mesh requiring more computational effort than the integration stage, due to a large number of divergent cases in the simulation data.

4.5 Conclusion

We present a novel interactive streak surface construction algorithm for the visualization of 3D unsteady flow. It combines the advantages of both speed and size with an efficient CPU-based, platform-independent implementation that places less restriction on memory and precision than a GPU-version. It is this quad-based approach from which the speed of the algorithm stems. It avoids expensive mesh re-triangulations. The local nature of the operations applied to the quad primitives enables the algorithm to be applied to large data sets (see accompanying video). The algorithm handles flow divergence, convergence, and shear. The algorithm also allows the surface to split when it meets object boundaries. The algorithm is demonstrated on a variety of data sets posing various challenges such as turbulence and large-scale simulations. A detailed CPU-based, platform-independent implementation is provided in the supplementary material to facilitate incorporation into visualization applications.

As future work we would like to treat the case of strongly deformed non-planar quads. Our model here does not treat them explicitly because we have not encountered this problem in our current simulation data sets.

5

Visualizing Interactive Streamline and Pathline Seeding Parameter Sensitivity

"*Continuous effort - not strength or intelligence - is the key to unlocking our potential.*"
-Sir Winston Churchill (1874–1965)[1]

Contents

[1]Former Prime Minister of the United Kingdom, known for his leadership of the United Kingdom during World War II.

VISUALIZATION of uncertainty is identified as one of the top future visualization problems [JEH$^+$04, LK07]. Uncertainty may be introduced in various stages of the visualization pipeline (Figure 5.1). Uncertainty may arise from numerical error and visualization parameter instability. For example, in the data acquisition phase, measurement instruments are associated with an inherent amount of error. Uncertainty is introduced in simulations by both the PDEs (partial differential equations) used in the computation and from the discrete grid-based mesh used to save the result. The mapping of data to visualization primitives may also introduce uncertainty. For example, the use of linear interpolation to reconstruct data in between samples introduces error. Streamline integrators accumulate error along the length of their curves. However, discrete primitives such as streamlines and iso-surfaces imply certainty.

Another source of uncertainty stems from user-input parameters. Small changes to input parameters can introduce large changes to the resulting visualization. In this chapter, we use the term *sensitivity* to describe the notion that a small change in input value can cause a large change to visualization output. This is related to the idea of uncertainty in the same sense that the user may not be fully aware of sensitive or unstable input values that could have a big effect on what they observe.

It is not clear to what extent the output of a flow visualization technique is affected by the combined effect of the uncertainties introduced in the various stages of the visualization pipeline. Without conveying these uncertainties, any flow visualization may be misleading. Interestingly, a lot of focus is placed on interaction in the visualization literature. However very little attention is paid to the uncertainty introduced by user-input parameters and their corresponding threshold values – the last stage of the visualization pipeline. Furthermore, interaction may be iterative, e.g., visualization results are repeatedly refined. Conceptually, this chapter advocates a visual map of user input sensitivity.

We focus on uncertainty introduced and associated with user-controlled input parameters for flow visualization. While there have been a number of papers examining uncertainty in flow visualization due to the data acquisition and visualization phases of the pipeline, relatively little research has been invested into visualizing the effects and uncertainty due to interaction or user-input to flow visualization. In many cases the user may be unaware of effects due to input parameter stability. Specifically, we analyze the effect that stream and pathline seed position and seeding rake orientation and position, in both space and time have upon the resulting set of integral lines. Inter-seed spacing along a seeding rake is also investigated. Our investigation includes unsteady flow, highlighting regions within the entire spatio-temporal domain that are sensitive to stream and pathline seeding. The goals of our investigation are to:

1. Investigate the sensitivity of flow visualization with respect to user-input parameters used for stream and pathline seeding position in both space and time.
2. Evaluate the stability of user-specified values for rake positioning and orientation.
3. Provide visualization tools to explore and investigate the behavior and sensitivity of stream and pathline seeding for different user-controlled input parameters.

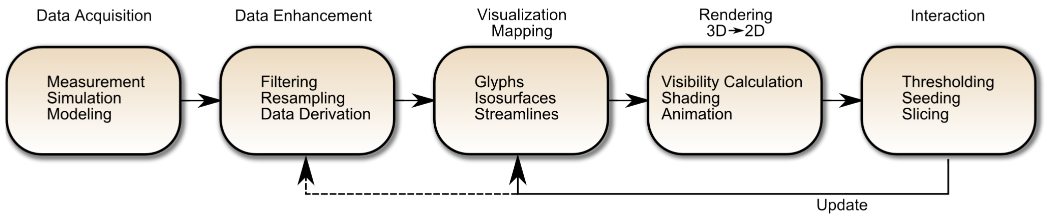


Figure 5.1: *The visualization pipeline. Each stage in the pipeline contains uncertainty. Our goal is to identify, quantify and visualize the uncertainty associated with user-input (right) typical for flow visualization and to present it to the user in a helpful and meaningful way.*

As part of our contribution we study and visualize the following:

- The sensitivity of stream and pathline seeding position in space and time and seeding rake position and orientation.
- A streamline similarity metric, including a comparison of various metrics, that can be used to quantify flow sensitivity, i.e., reflect areas where a small change in position results in large changes to streamline geometry.
- Novel techniques to visualize these user-input parameters and their associated sensitivity.
- The sensitivity of inter-seed distance between adjacent streamline seeds resulting in a novel method for distributing stream and pathline seeds along a rake.

We also demonstrate our visualizations to a domain expert from fluid mechanics and report their feedback. With existing flow visualization tools, discovering how user-input parameters and threshold values affect the visualization results requires what essentially amounts to a manual search through parameter space. The space may grow exponentially with each new user input option. A manual process is very time consuming and in some cases may even be for practical purposes impossible. Even if parameters can be modified interactively, users may get lost in their search through parameter space and the space-time domain. To provide insight into an algorithm's behavior requires specific visualization techniques for a range of user input.

The rest of the chapter is organized as follows: Section 5.1 describes previous work. Section 5.2 details and evaluates metrics for computing stream and pathline seeding sensitivity. Section 5.3 demonstrates our method applied to a range of user-input parameters including seeding position, rake position and rake orientation. Section 5.4 discusses the application of our method to adaptively spacing seeds along a seeding rake. Section 5.5 shows how seeding position can be visualized in unsteady flow, highlighting interesting regions in the spatio-temporal domain. In Section 5.6 we report the performance of our algorithm and provide domain expert feedback. Finally, the chapter is concluded in Section 5.7.

5.1 Related Work

Figure 5.1 illustrates the visualization pipeline. Uncertainty is accumulated at each stage. The goal of this work is to identify uncertainty associated with the parameters controlled by the user in the interaction stage (right).

5.1.1 Data Uncertainty and Visualization Mapping Uncertainty.

Wittenbrink et al. [WPL96] investigate the use of glyphs to visualize uncertainty. A variety of glyphs are presented and evaluated, typically mapping uncertainty to glyph attributes such as length, area, color and/or angle.

Lodha et al. [LPSW96] present a system for visualizing uncertainty called UFLOW. UFLOW is used to analyze the changes resulting from different integrators and step-sizes used for computing streamlines. Visualization of the differences between the streamlines is achieved using several methods such as glyphs that encode uncertainty through their shape, size and color. A novel application of mapping sound to uncertainty is presented by Lodha et al. [LWS96].

Cedilnik and Rheingans [CR00] simultaneously visualize the data and its uncertainty while minimizing the distraction that can occur when both are visualized together. A procedural texture incorporates a grid-like structure which is blended with the image. To represent the uncertainty the grid is deformed with the amount of deformation mapped to the uncertainty within a given region.

Rhodes et al. [RLBS03] visualize uncertainty in volume visualization. They extend the Marching Cubes algorithm so that an uncertainty value is associated with each sample. Uncertainty is mapped either to hue or to the opacity value of a stipple texture pattern on a secondary surface that envelops the iso-surface.

Pang et al. [PWL96] and Verma and Pang [VP04] present comparative visualization tools to analyze differences between two datasets. Streamlines and streamribbons are generated on two datasets, one being a sub-sampled version of the other. To compare streamlines, Euclidean distance between them is used. Visualization primitives can then be added to aid the user in seeing how a pair of streamlines differ.

Brown [Bro04] demonstrates the use of vibrations to visualize data uncertainty. Experiments using oscillations in vertex displacement, and changes in luminance and hue are investigated.

Botchen et al. [BWE05] present a method of visualizing uncertainty in flow fields. Texture advection is used to provide a dense visualization of the underlying flow field. Error in texture-based representations of flow is handled by using a convolution filter to smear out particle traces, modifying the spatial frequencies perpendicular to the particle traces.

Sanyal et al. [SZD$^+$10] depict uncertainty in numerical weather models using glyphs, ribbons and spaghetti plots. Praßni et al. [PRH10] highlight areas of ambiguity and allow the user correct potential misclassification of volume segmentation.

Potter et al. [PKRJ10] present an extension to the traditional box plot in order to portray uncertainty information.

5.1.2 User-input Uncertainty

Brecheisen et al. [BVPBtHR09] analyze the stability of user-input parameters used for Diffusion Tensor Imaging fiber tracking, specifically the terminating criteria for the fiber tracking. Often, the same set of fiber tracking parameters is applied to several different data sets. Thus, the user may be unaware of what effect the current parameter configuration has on the current data. They visualized the effects of two common termination criteria for fiber tracking, namely, the anisotropy threshold and the curvature threshold. This is the inspiration behind the current work. Berger et al. [BPFG11] introduce an interactive approach to continuous analysis of a sampled parameter space. Using this approach users are guided to potentially interesting parameter regions. Uncertainty in the prediction is depicted using 2D scatterplots and parallel coordinates.

To the authors knowledge the present work is the only work on the visualization of user-input parameter sensitivity specifically for stream and pathline seeding and rake positioning. In fact Brecheisen et al. highlight seeding as a direction for future work. The difference between our work and the above is that we are the first to focus on the actual seed positioning, whereas Brecheisen et al. focused on terminating criteria.

Other work on uncertainty in flow visualization has been focused on different stages in the visualization pipeline and not on the interaction stage, mainly on the data enhancement and visualization mapping stages. Chapter 2 covers the topic of streamline seeding and also uncertainty flow visualization in more detail.

5.1.3 Automatic Streamline Seeding

There have been a number of important papers on the topic of automatic streamline seeding for 3D data. See for example Spencer et al [SLCZ09] and Li et al. [LS07]. See McLoughlin et al. [MLP$^+$10] for a complete overview. However the focus of the work we present does not fall into this category. We classify these into the visualization mapping stage of the pipeline from Figure 5.1. In contrast, our work focuses on the last stage – interaction.

Another aspect we emphasize is the importance of manual seeding. CFD engineers require manual seeding. First, interpreting the results of fully-automatic streamline seeding strategies without a complete understanding of the underlying algorithm is very difficult. Second, in general, automatic seeding strategies do not take into account the full range of CFD attributes, e.g., pressure, temperature, kinetic energy etc. and are usually based solely on velocity. CFD engineers require seeding based on these other attributes. Third, not all features of interest are known a priori. Thus feature-based streamline seeding has limitations.

5.1.4 Finite-Time Lyapunov Exponent

The finite-time Lyapunov exponent (FTLE) [Hal01] has recently gained increasing attention [PPF$^+$10]. It quantifies the local of separation behavior of the flow. It is used to measure the rate of separation of infinitesimally close flow trajectories. It is computed from the set of all trajectories to produce a *flow map*, ϕ. Once the flow map has been computed, $\phi(\mathbf{x}_0, t_0, t)$ maps the position of the trajectory starting at time t_0 from position $\mathbf{x}_0$ for a finite-time, t. Using the flow map, the *Cauchy-Green deformation tensor field*, $\mathbf{C}^t_{t_0}$, is obtained by left-multiplying the flow map with its transpose [Mas99]:

$$\mathbf{C}^t_{t_0}(\mathbf{x}) = \left[\frac{\partial(\mathbf{x}_0, t_0, t)}{\partial \mathbf{x}_0}\right]^T \left[\frac{\partial(\mathbf{x}_0, t_0, t)}{\partial \mathbf{x}_0}\right] \tag{5.1}$$

From this, the FTLE is computed by:

$$\mathrm{FTLE}_{t_0}^{t}(\mathbf{x}_0) = \frac{1}{2(t-t_0)} \ln \lambda_{max}(\mathbf{C}_{t_0}^{t}(\mathbf{x})) \tag{5.2}$$

where $\lambda_{max}(\mathbf{M})$ is the maximum eigenvalue of matrix $\mathbf{M}$ [Hal01].

FTLE requires the choice of a temporal-window. The effect that a change in the temporal-window length has not been studied sufficiently [PPF+10]. A common use of FTLE is to extract *Lagrangian Coherent Structures* (LCS). LCS are extracted from an FTLE field by ridge extraction. This is dependent upon the definition of ridges themselves as well as upon the quality of the ridge extraction technique employed.

FTLE can be thought of as a property of the data, and could be classified in the data derivation stage of the visualization pipeline. From a visualization point of view, FTLE is used to extract Lagrangian Coherent Structures belonging to the visualization mapping stage of the pipeline (Figure 5.1). Our work focuses on the interaction stage and is not concerned with explicit extraction of flow features. The work presented here develops a visual map of streamline and pathline seeding parameter space.

5.2 A Method for Defining Streamline Seeding Sensitivity

We examine the effect that seeding position has on a streamline by analyzing and quantifying the change in curve geometry and direction when compared to another streamline seeded in close proximity. We describe spatial and temporal metrics for comparing the similarity of a pair of streamlines. Previous work on defining a similarity metric is introduced. This is followed by a description of the modifications we apply to the metric in order to incorporate an entire field of streamlines or pathlines. We discuss how the metric is used to create both spatial and temporal sensitivity fields – the key behind real-time navigation, exploration, and visualization of input parameter space. Figure 5.2 shows an overview of the workflow which is now discussed in detail.

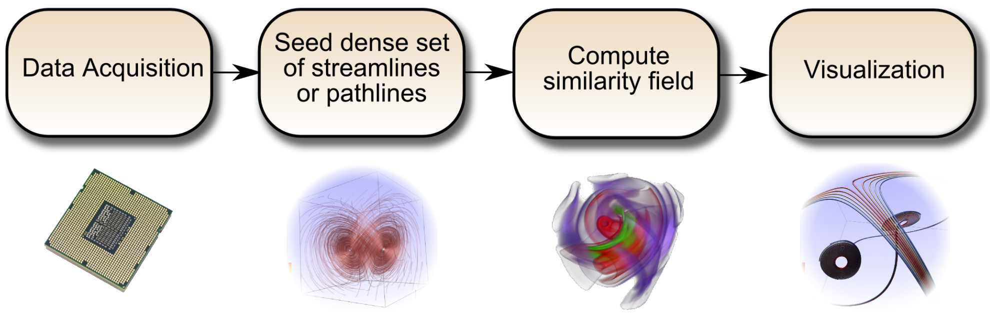


Figure 5.2: *An overview of our system workflow used to visualize stream and pathline seeding and rake parameter sensitivity. A dense set of streamlines or pathlines are computed on the simulation data. This set is utilized to construct the sensitivity field in both space and time. Various visualization tools are employed to render the parameter space based on the sensitivity field.*

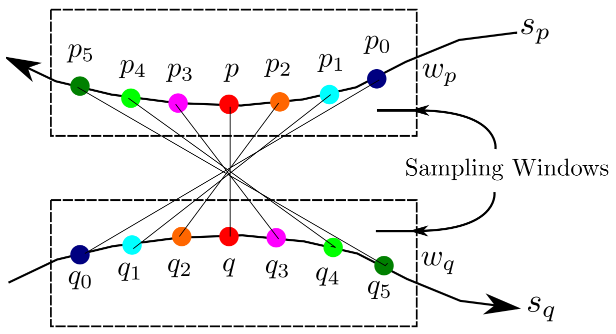


Figure 5.3: *Two sampling windows, w_p and w_q, placed on two streamlines, s_p and s_q. The sampling window incorporates the downstream direction and curvature in the computation of the similarity metric. The similarity between corresponding points along sub-lengths s_p and s_q is computed using Equation 5.3.*

5.2.1 A Previous Streamline Similarity Metric

A streamline similarity metric is introduced by Chen et al. [CCK07]. This approach (Figure 5.3) measures the similarity between local regions of streamlines in order to facilitate a streamline seeding strategy. For a given point, p, $p \in s_p$, the closest point, q, $q \in s_q$ is identified. Sampling windows are then placed over s_p and s_q. The sampling windows, w_p and w_q, sample n points on either side of p and q, about which they are centered (n is user defined). This yields two sets of evenly-spaced sample points, for s_p and s_q, $[p_0, ..., p_{n-1}]$ and $[q_0, ..., q_{n-1}]$. Note that these points retain the directional information of s_p and s_q. As we traverse from the 0^{th} point to the $(n-1)^{th}$ point we proceed downstream. See Figure 5.3. These two sets of points are then used to compute the local similarity between s_p and s_q. The similarity, d is defined by:

$$d(s_p, s_q) = ||\overrightarrow{p-q}|| + \alpha \frac{\sum_{k=0}^{n-1} \left| ||\overrightarrow{p_k - q_k}|| - ||\overrightarrow{p-q}|| \right|}{n} \tag{5.3}$$

Equation 5.3 measures two key attributes. The first major term measures the translational distance between s_p and s_q, while the second major term corresponds to the shape and orientation difference. A weighting coefficient α is placed on the second term to allow the user to place more emphasis on the difference in shape. Values between 1 and 3 for α are found to be the most useful [CCK07]. There are other possible similarity metrics, e.g., Hausdorff, for measuring curve similarity which we discuss in Section 3.4.

5.2.2 An Adapted Streamline Similarity Metric

For computing streamline similarity, we adopt the above metric. However, there are some key modifications in order to compare entire streamlines to each other, not just sub-sections. First, w_p and w_q include the entire length of s_p and s_q. Every $p \in s_p$ is associated with a point along s_q. The algorithm for locating the associated point is presented below. Secondly, p_0 and q_0 are

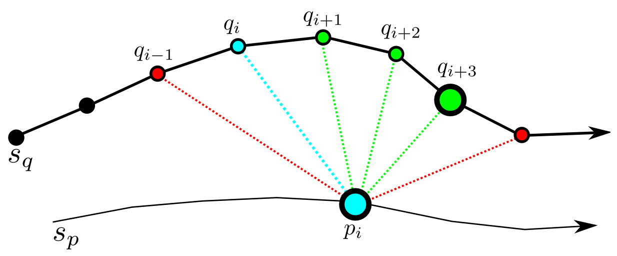


Figure 5.4: *Finding the associated point, q_n, on a neighboring streamline, s_q. The neighboring point is tested by computing the distance $d_i = ||\overrightarrow{p_i - q_i}||$, marked in blue on the streamlines. We traverse in the downstream direction if d_i decreases. This is highlighted by the green points. This is repeated until d_i increases (the red points) at which point we store the current point, q_{i+3}. The same process is performed upstream. These two points (q_i and q_{i+3} in the example) are then compared and the pair with the shortest distance (the large green point in the figure) is stored.*

always seeding points. When both lists are complete the similarity distance, d, between s_p and s_q is computed.

Note that this creates an asymmetry between the distance values. In general:

$$d(s_p, s_q) \neq d(s_q, s_p) \tag{5.4}$$

Associating Points between Streamlines and Shear: Given a point p_i, $p_i \in s_p$, finding the associated point, q_j, $q_j \in s_q$, involves a search. Rather than a brute-force search, we start out by exploiting the parameterization along s_p and s_q. For the i^{th} point along s_p, p_i, we start the search at the i^{th} point, q_i, and compute the Euclidean distance $d_i = ||\overrightarrow{q_i - p_i}||$. If s_q contains fewer than i points, we simply compare the final point, i.e., q_{n-1} if s_q has n points. We test points in the downstream direction of s_q (i.e., q_{i+1}). If $d_{i+1} < d_i$, i.e., if $||\overrightarrow{p_i - q_{i+1}}|| < d_i$, we then proceed to test q_{i+2}. This test is repeated until an increase in d is detected. Finally, we store the current point q_{min}. This is repeated in the upstream direction (starting from q_i). This results in two points. These may be the same point in the case that q_i is closer to p_i than q_{i+1} and q_{i-1}. We then determine which of these two points is closest to p_i. These points are then used in the similarity distance computation. This process is illustrated in Figure 5.4. The associated point is computed this way so that we compare corresponding local regions between s_p and s_q. This enables (1) the curvature, (2) translational distance, and (3) downstream direction to define the similarity between s_p and s_q.

Shear flow is another important property. This is highlighted in Figure 5.5. The user may be interested in factoring shear into the similarity metric. Thus we provide the user option to incorporate shear by using the corresponding points between s_p and s_q as resulting from their parameterization (Figure 5.5 *right*). If one streamline is longer than the other, the last point is used on the shorter streamline when its length has been traversed.

Incorporating shear into the similarity metric generally provides better results in our experiments. It also reduces computation time. Figure 5.5 compares two volume renderings of the sensitivity field without and with shear enabled. Enabling shear produces more well defined characteristics. Therefore, we enable this option by default. All visualizations use this unless noted otherwise. See Figure 5.7 for a visual comparison of the metrics.

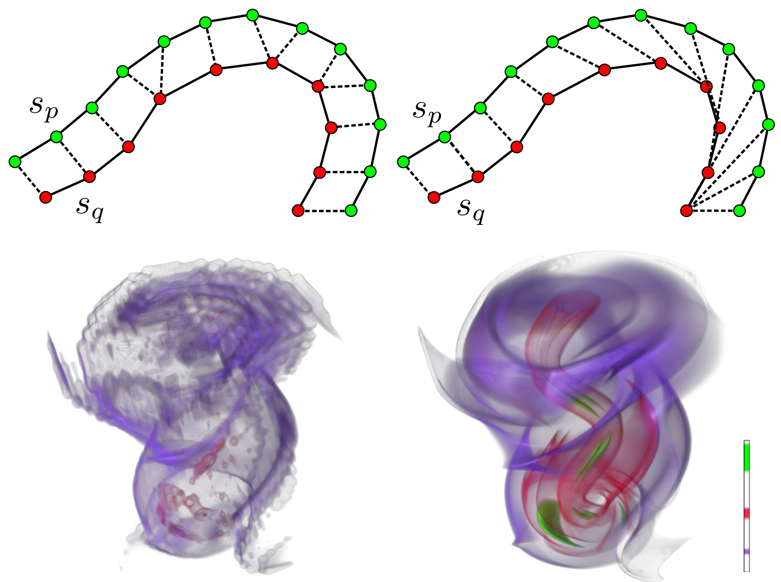


Figure 5.5: *(Top-Left) Connecting the associated points using our searching algorithm (Section 5.2.2). This produces a method of comparison of s_p and s_q based on their geometry. The mean of the length of the connecting line segments is related to the similarity distance. (Top-Right) Connecting the points along s_p and s_q based on the parameterization of the streamlines. These connections form an alternative comparison of s_p and s_q. As s_p and s_q approach the curve the connecting lines become sheared and don't link to the closest points between s_p and s_q measured by Euclidean distance. If s_q has fewer points we use the last point. (Bottom) The spatial sensitivity fields created from a tornado simulation. The two images were generated without and with the shear attribute respectively.*

5.2.3 Computing a Spatial Sensitivity Field

To facilitate interactive analysis, i.e., reporting parameter stability for a given set of seeding points, we provide this information in real time to the user. We pre-compute a spatial sensitivity field that utilizes the above similarity metric for a dense set of streamlines. We formulate streamline seeding sensitivity in terms of the derived sensitivity field defined over the entire spatial domain. We now provide a description for the computation of the spatial sensitivity field.

For each sample position, p_i, within the domain, $\mathbb{R}^3$, a streamline is computed. For every streamline, s_p, within $\mathbb{R}^3$, the similarity is computed by incorporating each of its neighbors (9 in 2D and 27 in 3D) (Figure 5.6). The total spatial similarity value is computed using Eq. 5.5.

$$D(p_0) = \frac{\sum_{i=1}^{n} d(s(p_0), s(p_i))}{n} \tag{5.5}$$

where, n is the number of neighbors and $d(s(p_0), s(p_i))$ is the similarity distance between a streamline seeded at p_0 and another seeded at p_i. Figure 5.6 illustrates this for a 2D field.

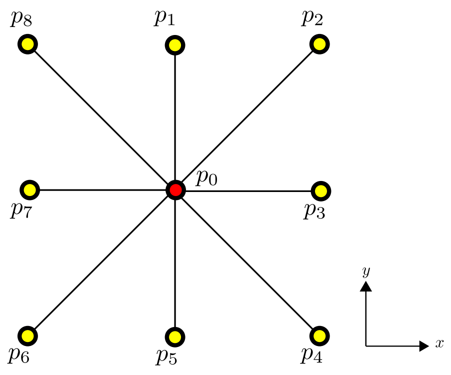


Figure 5.6: *A streamline, s, is seeded at every sample point, p_i, in the domain. In order to compute the sensitivity field for a given p_i, the similarity between $s(p_i)$ and its neighbors is computed. The mean of the similarity computations quantifies p_i (Eq. 5.5). This example is presented for a 2D field for simplicity.*

5.2.4 Candidate Metrics

In addition to the metric in Section 3.2, we have experimented with various metrics for quantifying integral curve similarity. For example, the closest point distance, d_c, could be used:

$$d_c(s_p, s_q) = \min_{p_k \in s_p, q_k \in s_q} ||p_k - q_k|| \tag{5.6}$$

However d_c ignores curvature and shear. The mean distance $d_m(s_p, s_q)$ of closest distances is another candidate:

$$\begin{aligned} d_M(s_p, s_q) &= \text{mean}(d_m(s_p, s_q), d_m(s_q, s_p)) \\ \text{where } d_m(s_p, s_q) &= \text{mean}_{p_k \in s_p} \min_{q_k \in s_q} ||p_k - q_k|| \end{aligned} \tag{5.7}$$

This does not quantify downstream direction or shear. The Hausdorff distance, d_H could be used:

$$\begin{aligned} d_H &= \max(d_h(s_p, s_q), d_h(s_q, s_p)) \\ \text{where } d_h(s_p, s_q) &= \max_{p_k \in s_p} \min_{q_k \in s_q} ||p_k - q_k|| \end{aligned} \tag{5.8}$$

However, this does not intuitively quantify downstream direction and is computationally expensive. Another candidate is the Fréchet distance [EM94]:

$$d_f(s_p, s_q) = min\{||L|| \mid L \text{ is a coupling between } P \text{ and } Q\} \tag{5.9}$$

However, this metric is computationally expensive and does not take shear into account. Interpretation by domain experts is also non-intuitive. Li et al. [LHS08] also describe a similarity metric:

$$d_l = 1 - \frac{1}{2}\left(\frac{v'(p) \cdot v(p)}{|v'(p)||v(p)| + 1}\right) \tag{5.10}$$

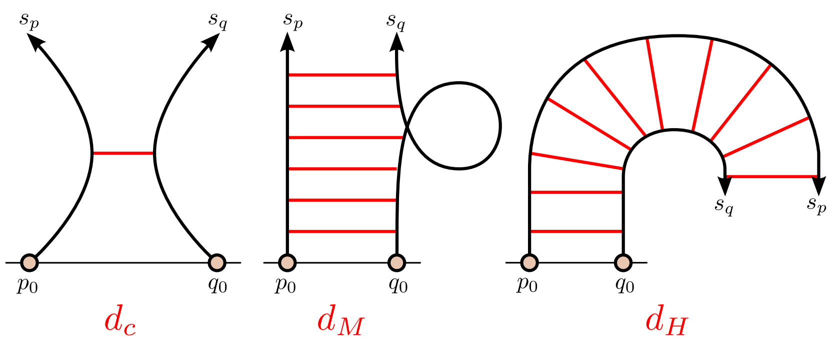


Figure 5.7: *This figure shows some of the cases that inspire our choice of distance metric: d_c does not quantify curvature, d_M may not quantify curvature or shear, d_H does not quantify shear. Our metric measures (1) translational distance, (2) curvature, (3) downstream direction, and (4) shear.*

where v' is an approximate vector at p and $v(p)$ is the original sample vector. This metric compares vectors, not streamlines.

The choice of metric depends on the interests of the user. Figure 5.7 compares some metrics based on two streamlines, s_p and s_q seeded from a rake. It highlights special cases where flow properties are not quantified. We chose the metric described in Section 3.2 because it includes (1) downstream direction, (2) streamline curvature (not only distance), (3) divergence, and (4) shear. However, any streamline or pathline similarity metric can be used in our framework.

5.3 Visualization of Seeding Position and Rake Parameter Sensitivity

The first approach to visualizing the sensitivity field directly is to produce a scalar field. This allows for fast, intuitive exploration of data and provides the user a quick overview of regions with high rates of streamline-based change. With the spatial sensitivity field stored as a scalar field, it is possible to visualize the sensitivity of the seed positions using any volume visualization technique.

Care must be taken by the user when interpreting the results of the similarity field. The initial impression is that the sensitivity field directly maps to features within the flow field. This is not necessarily the case. The sensitivity field is not a feature extraction technique – it highlights regions from where seeded trajectories exhibit more dissimilar behavior. This is an important difference and may provide the user with more information about the flow than just observing features alone. Features, by definition, have a distinctive characteristic and frequently, flow trajectories passing through them exhibit dissimilar behavior. Therefore, these regions may be highlighted by the sensitivity field. In addition, regions where the seed of a trajectory does not originate within a feature, but whose trajectory passes through the feature are also highlighted. Therefore, the similarity field may not only highlight features but regions connected by the flow to these features. This may aid the user in understanding the nature of

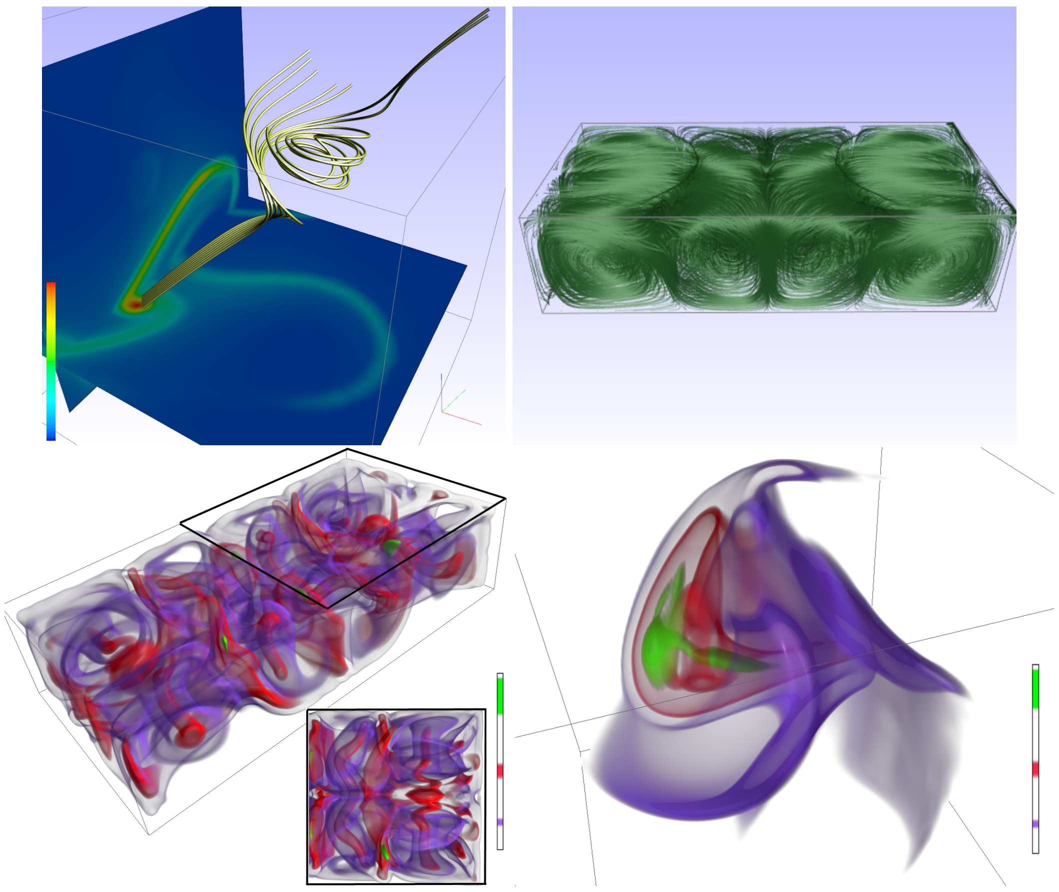


Figure 5.8: *(Top-left) Visualizing the sensitivity field of Arnold-Beltrami-Childress (ABC) [Hal05] flow using two interactive slice probes. Sensitive seeding regions within the field are visualized directly. Streamlines seeded in red regions are very sensitive to seeding position. Using this approach helps the user locate interesting structures more quickly in the domain than a brute force manual search through the spatial domain. In this example, the user quickly navigated the seeding rake to visualize the structure shown using the slice probes for context information. (Top-right) Streamlines on the Bernard flow simulation. (Bottom-left) A direct volume rendering of the sensitivity field, $D(p)$, from Bernard flow. As well as highlighting the regions to which streamline seeding is sensitive to change, the DVR provides an overview of the seed-based features within the simulation. The four main regions of the flow stand out and regions around vortex cores are highlighted by the red tubular forms. (Inset) An aerial view emphasizing the convection cell structures within the field. (Bottom-right) A direct volume rendering of $D(p)$ of the ABC flow and accompanying transfer function. The interesting vortex structure in the flow is highlighted by high sensitivity values mapped to red and green.*

the interesting behavior. Figure 5.8 (top-left) shows this benefit. The seeding rake is placed outside of the vortical behavior of the flow and the streamlines enter this region as they are traced. The rake placement was guided by the sensitivity field.

5.3.1 Visualization of Streamline Seed Position Sensitivity

Using a slice probe provides a fast and simple tool for direct investigation of streamline seeding position sensitivity. The slice probe may be arbitrarily aligned. Figure 5.8 (top-left) depicts two slices of ABC flow. Color is mapped to the similarity distance $D(p)$. This is true for all color-mapping in this chapter unless mentioned otherwise. The color coding conveys positions that are highly sensitive with respect to seeding position, i.e., where a small change in seeding position results in a large change to streamline geometry. The slice probe allows for high levels of interaction as it is computationally inexpensive. They are particularly useful when swept through the domain. As in this case, the slice provides context information which may allow the user to navigate to interesting behavior more easily.

Direct volume rendering (DVR) enables the visualization of the entire 3D domain simultaneously. For DVR we use the Volume Rendering Engine software tool (Voreen) [MSRMH09]. Figure 5.8 (bottom-left) shows the corresponding DVR of the set of streamlines in the top-left image. Rendering $D(p)$ in this way provides an overview of the seeding sensitivity and its characteristics. Figure 5.8 (bottom-right) shows a direct volume rendering of the sensitivity field of a steady Arnold-Beltrami-Childress (ABC) flow which describes a closed-form solution of Euler's equation [Hal05]. The green and red regions in the center of the image highlight an interesting vortex structure within the flow. The curvature of streamlines changes the most when seeded in or around the vortex core. The transparent regions are less sensitive to streamline seeding position. The geometry of streamlines seeded in these regions changes very little with seed position.

5.3.2 Visualization of Seeding Rake Sensitivity

This section introduces techniques used in conjunction with the sensitivity field to provide parameter sensitivity information for a seeding rake. We experimented with a number of different approaches before arriving at the one we found most suitable.

Figure 5.9 (*left*) depicts a naive attempt at an interactive probe constructed with a dense collection of rakes at various orientations. Streamlines are seeded from all rakes within the probe. This probe attempts to demonstrate the effect that varying the orientation of the seeding rake has on the resulting streamlines by showing many rakes simultaneously. However, this probe may produce a considerable amount of occlusion as can be seen by the dense rakes and streamlines in the image. It also provides the user with little intuition as to how to orientate a seeding rake at a given position.

The probe shown in Figure 5.9 (*middle*) uses three axis aligned slices. The seeding rake is interactively rotated about the center point. This approach reduces occlusion and visual complexity from seeding many streamlines. However, the probe itself is prone to self-occlusion and may occlude part of the seeding rake and resulting streamlines. An ideal tool orientates the rake automatically in an appropriate way depending on the local flow properties. A somewhat

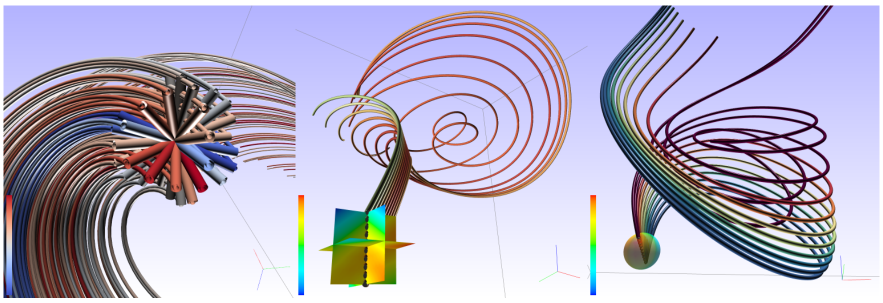


Figure 5.9: *Less successful experiments visualizing seeding rake parameters. (Left) A seeding probe with a dense collection of explicit seeding orientations. This probe produces a visually cluttered result, especially when streamlines are seeded from all possible positions. It also provides the user with little intuition as to how to orient a rake at the given position. (Middle) A seeding probe with axis-aligned slices. This probe provides insight into the sensitivity information in the vicinity of the seeding rake, through color mapped to sensitivity. However, it suffers from self-occlusion. (Right) A seeding probe depicted as a semi-transparent sphere. This probe also provides information about the sensitivity field in a small volume around the seeding rake. However, this approach only shows the sensitivity information on the surface – which corresponds with the rake ends. Therefore sensitivity information across the length of the rake is lost. The orientation of the rake can be observed through the use of a transparent surface.*

better method utilizes a sphere centered at the seeding rake position. See Figure 5.9 (*right*). The sphere provides sensitivity information in all directions around the seeding rake. Occlusion of the seeding rake and the resulting streamlines is reduced through the use of transparency. However, the sphere provides redundant information. Portions of the surface that are aligned with the flow fail to provide further useful information.

The method that we favor is the use of a planar polygon always orthogonal to the flow passing through its center, as in Figure 5.10. We automatically vary the alignment of the probe and the seeding rake to be orthogonal to the local flow field in order to guide the user. There is little benefit from seeding a rake that is tangential to the flow field. In the case of stream surfaces seeded from an interactive rake, a tangential component of the seeding rake to the flow should be avoided [PS09]. Our approach provides the user with guidance on how to orient their seeding rake – as it automatically remains in the plane locally orthogonal to the flow (at its center). Also redundant visualization information is reduced using a local cross-section of the flow. Color is mapped to sensitivity.

The seeding probe is depicted as a planar polygon with regular samples distributed across a central axis. The seeding probe sampling can be set to an arbitrary resolution by the user but the same resolution as the data by default. Figure 5.10 illustrates the effect that a region of high sensitivity can have on streamline geometry with a small change in seeding position. The seeding probe shows a highly sensitive region – the red vertical band on the probe. The streamlines are seeded at equidistant positions along the rake. By comparing neighboring streamlines the effect of a small change in seeding position becomes apparent. Two distinct groups of streamlines can be seen. The sensitive region highlighted by the seeding probe corresponds to the separating region in the flow, where the streamlines flow into either toroid of the Lorenz attractor.

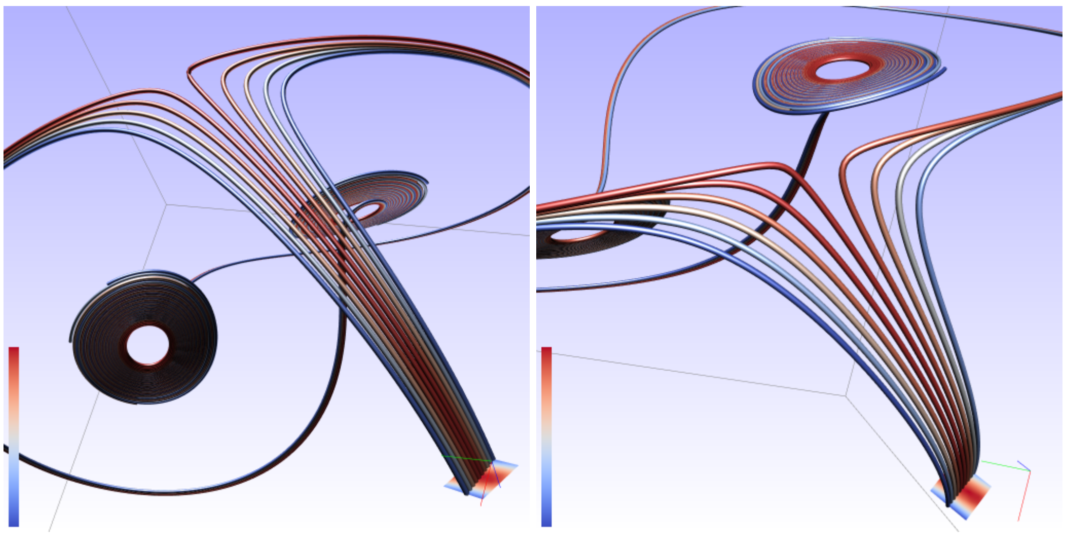


Figure 5.10: *A set of streamlines depicting a Lorenz attractor. Streamlines are seeded with the seeding rake spanning a sensitive region within the domain. The color-mapping shows streamlines from the center of the rake have the highest sensitivity values. By looking at the resulting streamline geometry it can be seen that a large change results from a small change in seeding position within this region. Sensitivity quantifies divergence.*

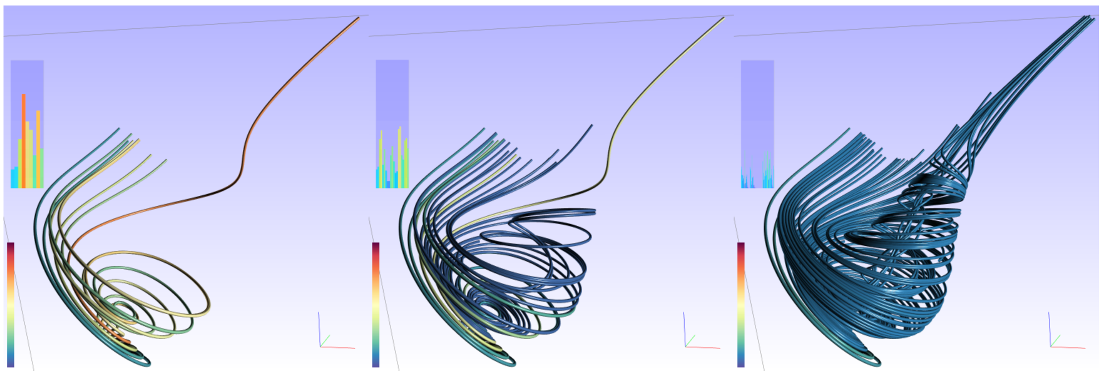


Figure 5.11: *(Left) Streamlines seeded at equidistant intervals along a seeding rake. The bar graphs shows the individual streamline similarity values. The large values in the bar graph represent a dissimilarity between pairs of neighboring streamlines. (Middle) The previous set of streamline seeds enhanced by our method. New streamlines are automatically seeded along the rake between pairs of streamlines with high dissimilarity. These new streamlines aid the user by visualizing the regions with relatively high streamline variation. A threshold of 50% of the maximum value of $D(p)$ is set. (Right) The resulting streamlines with a smaller threshold of 30% of the maximum value of $D(p)$. Again, this produces a more dense set of streamlines automatically to provide a more detailed visualization. Note that streamlines are only added where needed. If the similarity of neighboring streamlines are within the specified threshold, no insertions take place. This simultaneously helps reduce visual clutter in the visualization.*

5.4 Adaptive, Automatic Inter-Seed Spacing

Our spatial similarity metric allows us to introduce a novel idea of placing seed points in a distribution that produces a more illustrative set of streamlines. Choosing the optimal distance between seed points along a seeding rake is a topic that has received relatively little attention. Yet this is a virtually universal parameter for flow visualization using 3D streamlines. In general, the user selects a seeding rake length and the number of streamlines to be seeded from it. Streamline seeds are then distributed at equidistant intervals along the seeding curve. However, this may not be the optimal distribution. Choosing inter-seed distance along a rake is done using trail-and-error in practice. Using our spatial similarity metric it is possible to create an automatic distribution. This is based on the idea of stream surface refinement [Hul92]. In regions of divergence, streamlines are inserted into a stream surface in order to improve its approximation of the underlying velocity field. Similarly, we introduce extra streamlines along the seeding rake in regions of high dissimilarity between neighbors.

For a given rake, we begin by distributing the seeds uniformly along its length. For each pair of neighboring streamlines, s_p and s_q, we compute $d(s_p, s_q)$. We emphasize this similarity information in the form of a bar graph. The streamline pairs are plotted along the x-axis and their corresponding similarity values are plotted along the y-axis (Figure 5.11). The range along the y-axis is $[0..D_{max}]$. This enables the user to see the current streamline sensitivity values in the context of $D(p)$. Next, we allow the user to select a threshold, d_τ, as a percentage of D_{max}. The similarity values of $d(s_p, s_q)$ are compared to d_τ. If $d_i > d_\tau$ a new streamline is introduced between s_p and s_q. Figure 5.11 demonstrates this. The image on the left shows a set of streamlines distributed evenly across the seeding rake. Note the bar chart visualization shows a large dissimilarity between one pair of streamlines. The center and right images show seeding distributions enhanced using our method. These sets of streamlines fill the large gaps that were present using the original seeding pattern.

In region of very large dissimilarity across the rake, such as if the rake intersects a separating surface, care must be taken. It may be possible that floating-point accuracy raises problems. This is due to streamlines being packed so close together that a new seed position cannot be represented accurately enough using floating-point precision. One workaround for this would be to use double-precision floating-point numbers. However, in these extreme cases there is diminishing returns in seeding all the streamlines in these regions as they communicate little new information. An example of where this would occur is shown in Figure 5.10. The region in between the two center streamlines would not benefit from a greater seeding distribution frequency. To this end, we restrict the number of passes of the seed insertion routine. In our experiments we have found 10 passes to be more than adequate to obtain high quality results.

5.5 Visualization of Temporal Seeding Position

An extension of our method may be applied visualizing the seeding parameter sensitivity of unsteady flow fields. In these cases regions in which seeding at different times results in large changes to streamlines or pathlines are highlighted. This has several uses. First, even though a large amount of research addresses the visualization of time-dependent data, important in-

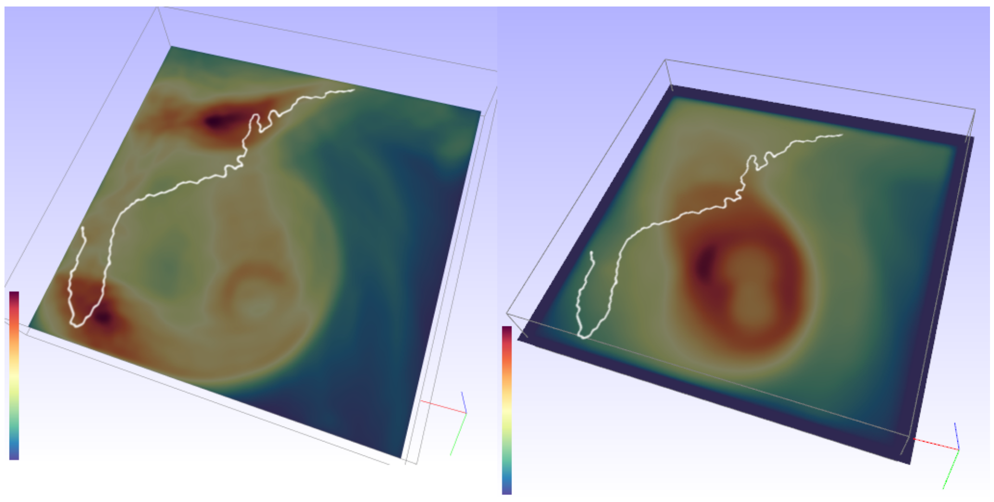


Figure 5.12: *A slice visualization of $D(p,t)$ from a simulation of Hurricane Isabel. This shows the regions with the most variation to streamlines over all time steps $0 \leq t \leq t_{max}$. The path of the hurricane eye and the regions close to it correspond to the most sensitive areas. The left image shows the field constructed using streamlines. The field in the right image was constructed using pathlines seeded at t_0 and traced until $t_{max} = 48$.*

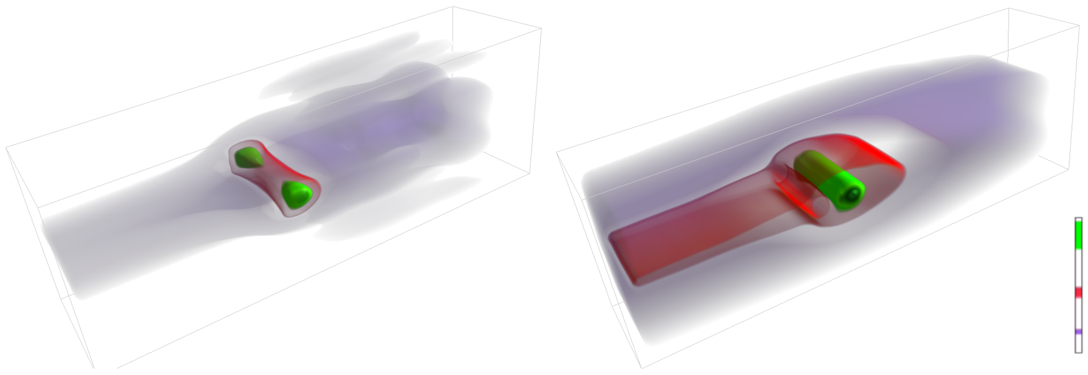


Figure 5.13: *A DVR visualization of the temporal sensitivity (Eq. 5.11) field from flow behind a square cylinder [CSBI05]. This visualization shows the region behind the square cylinder is the most sensitive over time. The left image shows the spatio-temporal sensitivity field constructed using streamlines. The right image shows the entire spatio-temporal field constructed using pathlines seeded at t_0 and traced over until the end of the simulation, $t_{max} = 102$.*

formation is still obtained using steady-state methods, e.g., streamlines, on a single time-step. Second, searching the entire space-time domain for optimal seeding positions can be a daunting task for the user. Therefore, capturing the essential information in a single static image or volume is valuable. Third, this scheme may be used to guide down-sampling a data set temporally. Down sampling is common due to the large output from modern simulations. It is often desirable to use a down-sampled version of the simulation that fits into main memory for the fast performance needed to support interaction. Using our temporal sensitivity field, the user can measure the uncertainty/change between time-steps i.e., $D(p,t_n)$ and $D(p,t_{n+1})$. This can be used as a guide to prevent excessive down-sampling e.g., if the user wanted to down-sample the data if they encountered a case where $||D(p,t_n) - D(p,t_{n+1})|| > D_\gamma$, the user knows that temporal aliasing may reduce visualization result quality. D_γ is a user defined threshold to control the quality of the down-sampling.

In order to handle unsteady flow data, a modification is made to how $D(p)$ is computed. Streamlines or pathlines are again seeded at every sample point, $p_i \in \mathbb{R}^3$. However, they are seeded at every $p_i(t_n)$ for every time-step, t_n. Thus for N time-steps, there are N streamlines or pathlines seeded at p_i. The value of $d(p,t)$ is computed starting with the streamline seeded at time-step t_0 and computing the similarity between $p_i(t_n)$ and $p_i(t_{n+1})$. This is repeated for $t \in T$ with Eq. 5.11

$$D(p,t) = \sum_{n=0}^{N-2} d(S(p,t_n), S(p,t_{n+1})) \tag{5.11}$$

where $S(p_i,t_n)$ is the streamline or pathline seeded at time-step t_n and $S(p_i,t_{n+1})$ is the streamline or pathline seeded at the time-step t_{n+1}.

Figure 5.12 shows visualizations of the temporal sensitivity field, $D(p,t)$, from a simulation of Hurricane Isabel. This visualization shows that the regions that are most sensitive to streamline seeding over the entire 4D space-time domain. These regions are located on the path that the eye of the hurricane takes and in the area around that path. The color-coding is mapped to $D(p,t)$. Figure 5.13 is a direct volume rendering of the temporal sensitivity field from a flow behind a square cylinder simulation [CSBI05]. The region most sensitive throughout the entire spatio-temporal domain is located behind the square cylinder.

5.6 Performance

The computation of the spatial and temporal sensitivity fields is a one-time computation. After $D(p,t)$ has been computed it can be saved to disk for faster loading in the future. If the user requires fast response times it is possible for the user to take a progressive approach to the sensitivity field construction. Shorter streamlines can be used to create a less accurate approximation. The streamlines can be lengthened to improve the accuracy, see Figure 5.14 for an example. In a multi-threaded environment this computation can be performed in the background and the results are progressively refined as they become available. This provides the user with immediate feedback earlier and allows them to quickly start exploring the data. Also, each sensitivity field is customized for a certain streamline length. Using streamlines of a different maximum length would not match the sensitivity field completely. By storing

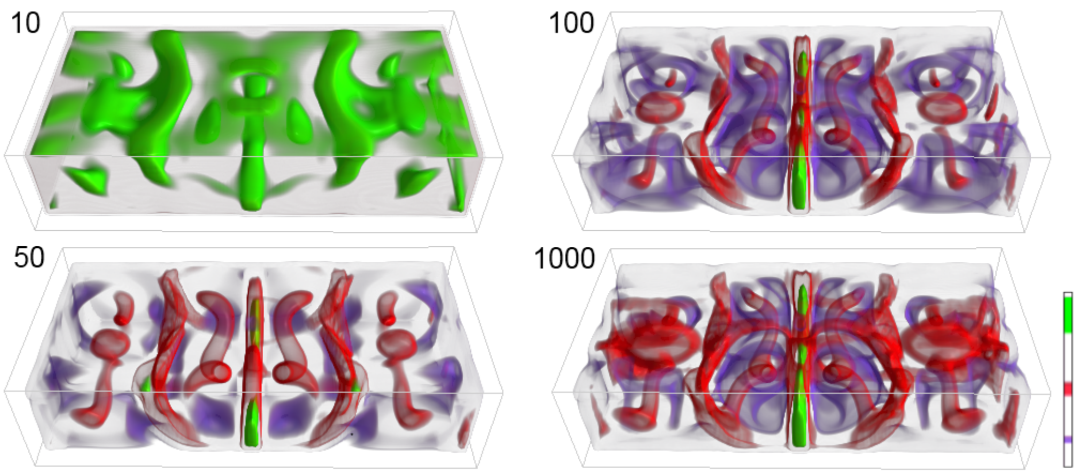


Figure 5.14: *The progressive creation of the $D(p)$ of the simulation of Bernard flow. This field is computed progressively by gradually extending the length of the streamlines. The streamline lengths used to create the field are 10, 50, 100 and 1000 points. The color-coding in normalized to streamline length in each image.*

the set of sensitivity fields as the generated for different streamline lengths, we can use the sensitivity field that matches the streamline length. Saving every single sensitivity for a large range of streamline lengths may require large amounts of memory. Instead, sensitivity fields can be created for streamline length increasing in l, where l is supplied by the user based on their requirements. If a streamline length is selected that lies between one of the computed length, the value may be interpolated using the two appropriate sensitivity fields.

Table 5.1 shows the computations times for a sample D using varying length streamlines. Streamline integration was performed using a 4^{th}-order Rune-Kutta integrator. The results show the computation time grows linearly with streamline length. We note that the computation does not prohibit visualization of 3D, time-dependent data. Also, the computation time is very fast when compared to a user manually adjusting the input parameters, i.e., searching the entire space-time domain. Chapters 5.6.1 and 5.6.1 are a domain expert evaluation provided by Harry Yeh. Harry Yeh is a Professor of Fluid Mechanics at Oregon State University, Corvalis.

Streamline Length	Computation of (128x32x64)
10	6.79s
50	28.53s
100	55.02s
1000	627.08s

Table 5.1: *Performance times for computing $D(p)$ using streamlines of varying lengths. These times were measured on the simulation of Bernard flow with domain resolution 128x32x64. The algorithm runs in approximately $O(n)$.*

5.6.1 Domain Expert Review

The use of sensitivity fields for the visualization of input parameter space in the context of flow visualization is a novel topic. While the initial computation of the sensitivity field may require some seconds or minutes, the benefits of fast navigation through parameter space outweigh the one-time pre-processing cost. Finding expressive seeding positions for streamline rakes can be a laborious and time-consuming process. The sensitivity field facilitates this process which is typically done manually by trial and error. For example, the slice visualization of the sensitivity field in Figure 5.8 (left) aided the navigation of the rake to capture the main vortical structure. This structure was found more quickly than performing a brute-force search through the domain and since automatic feature detection algorithms cannot be implemented for every type of feature, general strategies are very helpful.

The use of the seeding probe helps provide confidence to the visualization results. In highly sensitive regions extra care must be taken to ensure that the rake captures the important flow characteristics. A small change in rake position may make the difference whether a certain behavior is captured or not. This leads to the topic of automatic, adaptive inter-seed spacing which is of particular interest. Having the rake automatically distribute the seeds frees the user from this burden. As mentioned previously a small change in position may result in streamlines missing a feature which was present previously. Using the adaptive seeding strategy presents several benefits: (1) The adaptive seeding provides a higher sampling frequency in the required regions and provides a more detailed visualization result, providing greater insight into the seeding behavior and flow structure. (2) Inserting streamlines only where necessary simplifies the visualization by reducing occlusion and visual clutter, making the results easier to interpret with less visual overload. (3) Adaptive seeding alleviates the problem arising from a feature being lost as the rake is moved slightly. The accompanying video shows an example of an interactive session highlighting this phenomena. This problem could be alleviated by using equidistant seeding and increasing the sampling resolution. However, this places the responsibility of detecting this problem and manually updating the seeding resolution on the user.

The sensitivity field provides additional information about the nature of the flow field which are not communicated using integral curves alone. Section 5.6.2 describes a case study based on the parameter sensitivity data generated from the simulation of Hurricane Isabel.

5.6.2 Case Study: Hurricane Isabel

Densely distributed streamlines shown in Figure 5.15 (top) help observe seeding position sensitivity and the flow pattern of the hurricane. First, the tightly organized flow pattern is shown near the eye of the hurricane. In the area away from the eye, we see the different flow pattern in the upper and lower altitudes: the lower atmosphere tends to spiral into the hurricane, while the flow spreads out in the upper atmosphere. The streamline seeding pattern in the later time $t = t_{24}$ is less coherent than from the previous one t = t_0; this could be a sign of deterioration as the hurricane approaches the landmass. Unlike the streamlines, the pathlines maintain the same pattern as those taken early time (top second) including the preferred influence to/from the upper atmosphere in the northeast side of the hurricane; more pathlines are present there.

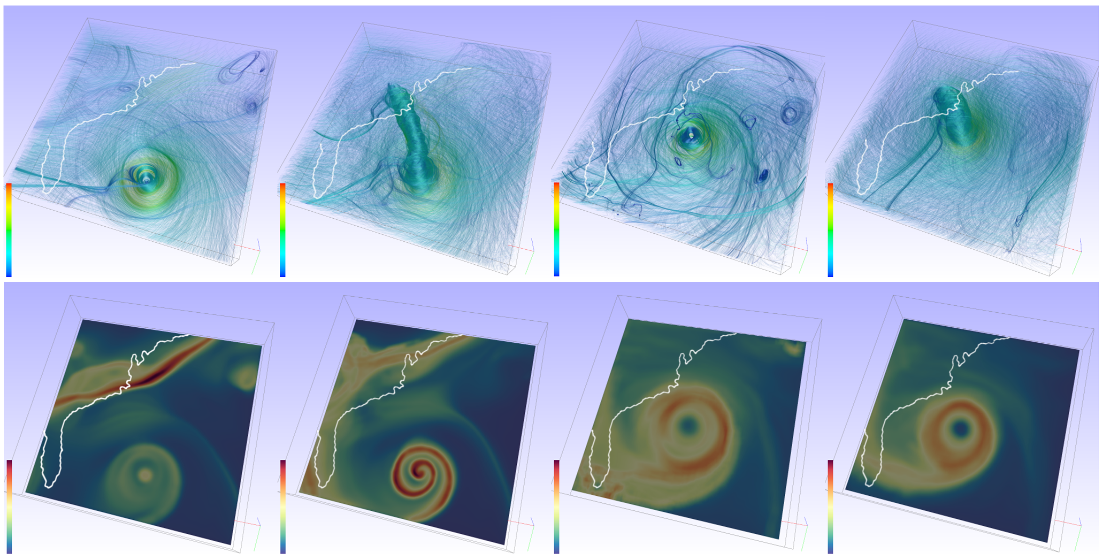


Figure 5.15: *(Top row) Dense sets of streamlines and pathlines seeded equidistantly on the simulation of Hurricane Isabel. The first two images are taken from t_0 out of 48 time-steps of the simulation. The images are of streamlines and pathlines respectively. The next two images show streamlines and pathlines seeded at t_{24} of the simulation. In both pathline cases the pathlines are traced until $t_{max} = 48$. All integral curves are colored according to local velocity magnitude. (Bottom row) Spatial sensitivity visualizations corresponding to the set of integral curves in the image above.*

As demonstrated, the developed technique is to provide guidelines to determine seeding rake and orientation to construct effective streamline and/or pathline field visualization. Figures 5.15 (bottom) and 5.12 however show an additional application. Visualization of the spatio-sensitivity field, $D(p)$, and the spatio-temporal field, $D(p,t)$ in the entire flow domain can be used for the exploratory flow analysis. The high sensitivity means divergence of stream (or path) lines. Figure 5.15 (bottom left) shows the high sensitivity region of the weather front along Eastern Seaboard, which must be strengthened by the approaching hurricane. The eye of the hurricane has high sensitivity but that is not the case 24 hours later (see bottom third). This evolution might reflect the growing hurricane at $t = t_0$, and the subsequent weakening at $t = t_{24}$. Also, see the large arc shaped boundary of the high sensitivity ahead of the hurricane. This represents the boundary of air mass influenced by the hurricane; the mass inside of the boundary flows toward the hurricane, while the air mass outside of the boundary flows away from the boundary just like the example of Bernard convection cell shown in Fig. 9 (middle). Later at time $t = t_{24}$, the appearance of the sensitivity field is different from that of the earlier time. Note that no weather front line can be seen here. It must be pushed further north. It appears that the hurricane has been somewhat diffused and the center of the hurricane (eye) is no longer the location of high sensitivity.

The sensitivity field $D(p)$ in terms of pathlines seeded at $t = t_0$ (Fig 5.15 bottom) also shows the weather front strengthened by the hurricane, but the location is shifted to the north.

This is because the front is pushed toward north by the hurricane as it approaches the coast. Just as the sensitivity field of the streamlines, the arc shaped boundary is observed ahead of the hurricane: the boundary separates the air masses inside and outside the hurricane. The distinct curly shape of the hurricane is intriguing, which implies that the hurricane must form the spiral cell toward the eye. The sensitivity field based on the pathlines at $t = t_{24}$ is grossly different from that of earlier time $t = t_0$. No more spiral pattern is present; instead, a single diffused ring of high sensitivity has emerged. This must be a sign of weakening.

5.7 Conclusion

We present methods to quantify and visualize parameters associated with the seeding of streamlines or pathlines for flow visualization. We define and compare various spatial and temporal similarity metrics which are utilized to create a sensitivity field. This facilitates real-time interaction using our method. Several tools are demonstrated to visualize the sensitivity field in both space and time for seeding rake parameters. We provide a tool to highlight the portions of a seeding rake which would benefit from a denser sampling of seeds, producing a more expressive set of streamlines that would not be achieved by naively increasing the sampling frequency. We present an extension to time-dependent data which highlights the most interesting regions over the lifetime of the simulation. We also report the reaction of a fluid dynamics expert in Sections 5.6.1 and 5.6.2.

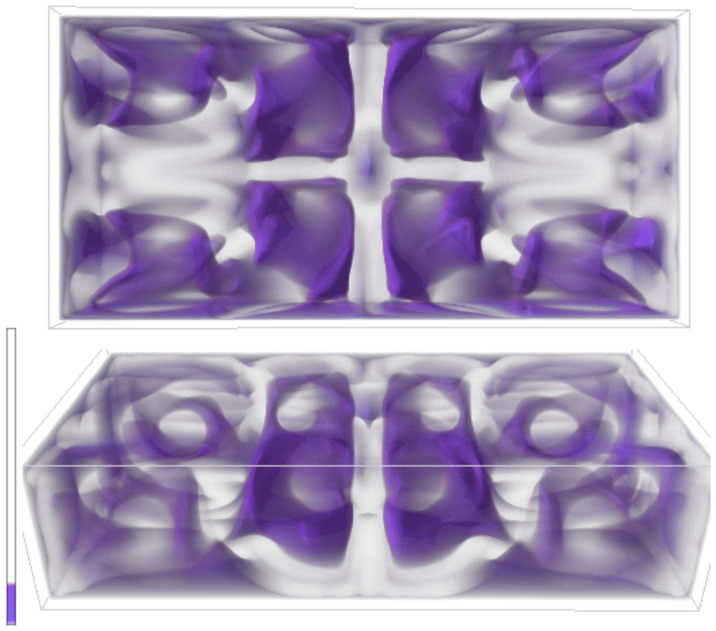


Figure 5.16: *Two views of the sensitivity field on the simulation of Bernard flow. This example demonstrates our spatial similarity highlighting the regions of similar behavior. This is achieved by simply setting the DVR transfer function to represent the minimum sensitivity field values.*

Throughout this chapter we have focused on presenting the regions that are most sensitive to change. However, the user may also want to see the opposite. Regions where flow is advected in a steady fashion are also interesting to domain users. These regions correspond to the regions of low sensitivity and thus steady flow advection. An example is shown in Figure 5.16.

For future research we plan to extend this technique to streaklines. One of the central challenges of analyzing this method in conjunction with streaklines is finding the optimal moment to both seed integral curves and, finding an optimal length.

6

Similarity Measures for Enhancing Interactive Streamline Seeding

"*Aerodynamically, the bumble bee shouldn't be able to fly, but the bumble bee doesn't know it so it goes on flying anyway.*"
-Mary Kay Ash (1918–2001)[1]

Contents

[1]Was an American business woman and founder of Mary Kay Cosmetics

STREAMLINES are curves that are everywhere tangent to a steady-state (time-invariant) vector field. They depict the path a massless fluid element traverses at any given time. The placement of these curves strongly affects the impact of the resultant visualization. Many automatic streamline seeding strategies exist in visualization literature (see Chapter 2). However, in practice these are not commonly used by Computation Fluid Dynamics (CFD) experts. Reasons for this stem from knowledge of the seeding algorithm being required to fully interpret the results. Also some seeding strategies place emphasis on equal coverage using evenly-spaced streamlines [JL97a], however changes in the physical proximity of streamlines may convey important properties of the flow that are lost while using a technique based on producing a fixed resolution output. Also domain engineers may not be interested in the entire spatial domain and their efforts may be focused on investigating a specific sub-region. In this case, a global seeding strategy may add visual clutter to the resulting visualization and impede the investigation by the user. Consequently, CFD engineers rely heavily on manual seeding. In fact, the popular visualization package, TECPLOT [TEC], includes no automatic seeding of streamlines and relies entirely on the user to do so.

However, there has been less focus on research enhancing the user experience while employing manual seeding. Typically, streamlines are seeded at equidistant positions along a curve or plane with little further opportunity for interaction. In many cases, this does not result in a visually optimal set of streamlines for the given seeding object. Whilst working with CFD experts we found that they predominantly use interactive seeding when using streamlines to investigate their data. CFD experts rely heavily on the derived visualizations for disseminating the results of their simulations. The work presented here aims to enhance the domain expert user's experience while employing this frequently used tool. We provide novel interaction with, and control of, the set of streamlines produced from interactive seeding objects. This allows the user to easily customize the resultant visualization enabling them to portray their results with more flexibility. Our method relies on only a small number of parameters which are simple to navigate. We place a high-level of importance on this observation in order to provide an improved user experience. The user is not required to navigate an unintuitive, high-dimensional parameter-space.

The core of our method is a set of similarity measures to compare streamlines. Clustering based on similarity is then performed, which then allows several enhancements such as a focus+context visualization and filtering of streamlines to leave an expressive subset of streamlines. The main contributions of this chapter are:

- A novel approach for computing a signature for an integral curve, and its use for similarity testing using the χ^2 (Chi squared) test.
- The use of old and novel measures for integral curves, and their comparison to existing state-of-the-art techniques. The combination of these measures and signature offers computation two orders of magnitude faster.
- A greedy algorithm for clustering based on the similarity matrix.
- An interactive algorithm for streamline filtering along the seeding primitive.

- A focus+context visualization based on the streamline clusters.
- An algorithm that maintains a high level of interaction with a large number of streamlines per seeding rake.

Many observations motivate this work. Our discussion with domain scientists demonstrates that they primarily use rakes to visualize and explore vector fields. Rakes tend to be the first tool of choice because they offer real-time interactivity with no pre-computation, provide an intuitive visual representation of the data and do not rely on complicated user parameters. The drawbacks of rakes are that resulting visualizations can be cluttered, there is no existing way to highlight streamlines or to customize the visualization to produce high quality rendering for communication and presentation purposes. Controlling the streamlines using streamline seeding or placement algorithms could improve this situation, but this introduces lengthy pre-computation. We investigate this area and report an approach that offers a solution to these problems. This work is related to the well researched topic of seeding to control streamline placement and bundling of DTI fibers. Our approach is compared to existing algorithms in those areas. The comparison demonstrates that our new approach has applications to general integral curve similarity calculations.

The rest of the chapter is organized as follows. Section 6.1 provides a survey of related literature. Section 6.2 provides the overview and detailed description of our method. Enhancements to our algorithm are presented in Section 6.3. Section 6.4 contains a discussion of our algorithm in comparison with other state-of-the-art techniques and provides performance results. Finally, Section 6.5 concludes the chapter with directions of future work.

6.1 Related Work

Here, we discuss related work in the areas of similarity metrics for streamlines and other integral curves, automatic seeding strategies for global placement of streamlines and clustering from a similarity matrix.

6.1.1 Streamline Similarity Metrics

Streamline similarity metrics have been widely used to control the number and proximity of streamlines for streamline placement applications. The goal is to produce uncluttered visualizations of flow fields whilst maintaining the depiction of the major features. The area was introduced by Turk and Banks [TB96] through streamline seeding whilst minimizing an image-space energy function. This was extended using a farthest point seeding by Mebarki et al. [MAD05]. Evenly-spaced streamlines [JL97a] are another solution to the seeding problem. For example, Liu et al. [LM06], incorporate the goals of maximizing streamline length, seeding based upon distance controls and loop detection to place streamlines. Chen et al. [CCK07] observe that (a) streamline placement algorithms tend to use a uniform resolution that either potentially misses salient features or contains redundant streamlines; or (b) rely on feature detection in order to sample streamlines adequately, leading to problems due to incorrect feature identification. They propose a similarity metric over the domain that allows them to adapt streamline resolution in the vicinity of dissimilar streamlines. Their similarity metric is based on computing distances between points along a streamline that leads to slower non-interactive computational times compared to our approach. Li et al. [LHS08] present a ‘less is more’ approach to streamline seeding. The goal is to capture the most important flow features using the fewest number of streamlines. This produces results comparable to hand-drawn diagrams. This similarity metric is also distance based. It is demonstrated in 2D with low numbers of streamlines (relying on a distance transform). Extending to a large 3D volume with the number of streamlines we enable and maintaining interactivity is unrealistic as we demonstrate with our comparison in Section 6.4.1. Other relevant work includes streamline predicates by Salzbrunn and Scheuermann [SS06] which are boolean maps that are used to differentiate streamlines based on input queries from the user. Similar to flow topology, the idea is to partition the domain into regions of coherent flow behavior.

6.1.2 Similarity for DTI Fiber Tracts

Distance metrics have also been applied in the domain of DTI fiber clustering. For an introduction to the area see Moberts et al. [MVvW05] where they review various clustering approaches and distance metrics for DTI fiber clustering. Two widely implemented and state-of-the-art techniques are by Corouge et al. [CGG04] and Zhang et al. [ZCL08]. Corouge et al. [CGG04] introduce a symmetric distance measure based on the mean of all the distances of the closest point on curve B from each point on curve A. Zhang et al. [ZCL08] also introduce a threshold into the distance so that curves that are close for a good portion of their length but then diverge widely at the end are regarded as distant. They also compare their method to Corouge et al. Deiralp and Laidlaw [DL09] introduce a weighting term in order to weight the ends of the curve more in the distance calculation and also introduce a perceptual coloring. Jianu et al. [JDL09] extend that work further ([DL09]) with a coordinated views representation of the DTI model and the clustering. They use average linkage hierarchical agglomerative clustering. See Jain et al. [JMF99] for a classification of clustering. In section 6.1 we compare our method to the measures by Corouge et al. [CGG04], Zhang et al. [ZCL08] and Chen [CCK07].

6.1.3 Streamline Perception in 3D

The goal of good streamline placement is a representation that is free of visual clutter, and contains the salient features. There are many algorithms for 2D streamline placement, but 3D placement remains a more challenging problem. Mattausch et al. [MTHG03] provide several strategies for interacting with evenly-spaced flow data in 3D, also providing a focus+context like visualization by treating the separation distance as a measure of interesting features. More recently Marchesin et al. [MCHM10] present a view-dependent strategy for seeding streamlines in 3D vector fields. Based on the observation that no distribution of streamlines is ideal for all viewpoints, this method produces a set of streamlines tailored to the current viewpoint. The algorithm begins by seeding a random set of initial streamlines. These are then filtered according to an occupancy buffer, which tracks the number of streamlines for a given pixel, and various filtering techniques such as angular entropy.

Visual clutter can be reduced by using differing visualization techniques. For example Mallo et al. [MPSS05], demonstrate an improvement on illuminated lines [MTHG03] that

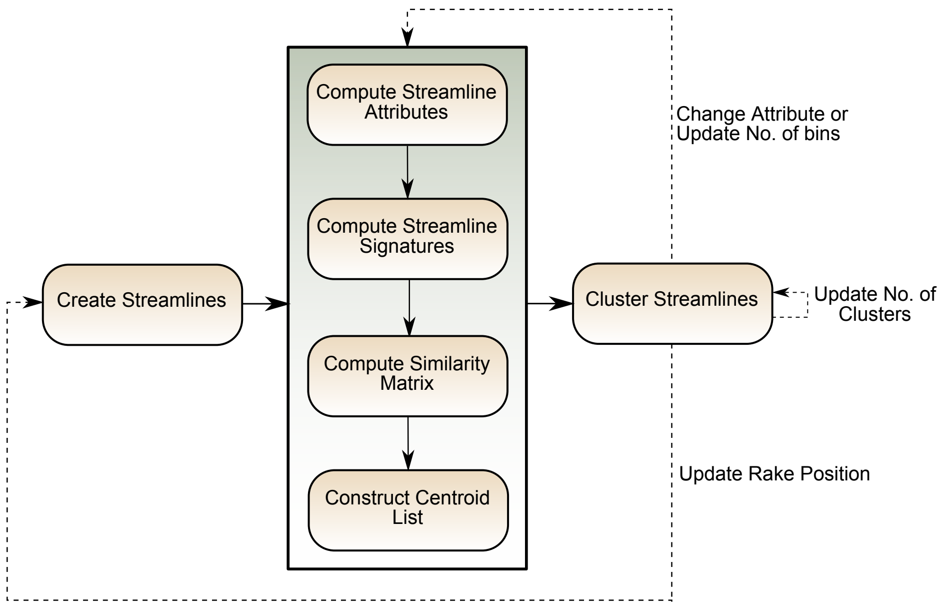

Compute Streamline Attributes
Change Attribute or Update No. of bins
Compute Streamline Signatures
Create Streamlines
Cluster Streamlines
Update No. of Clusters
Compute Similarity Matrix
Construct Centroid List
Update Rake Position

Figure 6.1: *Overview of our algorithm pipeline. First, streamlines are computed. Streamline signatures are then computed based upon streamline attributes. The streamline signatures are used to order the streamlines based on similarity. This ordering is then used to extract the cluster centroids and the streamlines are assigned to the relevant cluster. Dotted lines show user interaction and which parts of the pipeline need to be re-computed due to that interaction.*

exploits the use of diffuse and specular reflection to streamlines to create better perception of spatial structure. The introduction of such a shading technique also helps reduce the visual clutter of large numbers of similarly colored lines. Additional techniques include additive blending and edge bundling [Hol06] techniques for streamlines. Our solution is to take some of the ideas from distance-based similarity metrics, improve upon them for computational speed and apply them to interactive seeding rakes.

6.2 Streamline Similarity

Our algorithm begins with the user seeding a set of streamlines using an interactive seeding object. Once the seed positions have been set the streamline trajectories through the vector field are computed. Next, streamline signatures are computed based on the set of attributes (Sections 6.2.1 and 6.2.3). A similarity matrix is constructed using the χ^2 test (Section 6.2.4). Streamlines are ordered, based on their dissimilarity using a greedy clustering approach. The user selects the desired number of clusters, and then the remaining streamlines are associated with an appropriate cluster. The user can vary the number of clusters interactively to customize the level of detail and their desired visualization (Section 6.2.5).

In order to facilitate user interaction our system only re-computes the necessary parts of the pipeline as the user interacts with the algorithm parameters. For example, once the similarity matrix and streamline ordering has been computed the user can vary the number of clusters

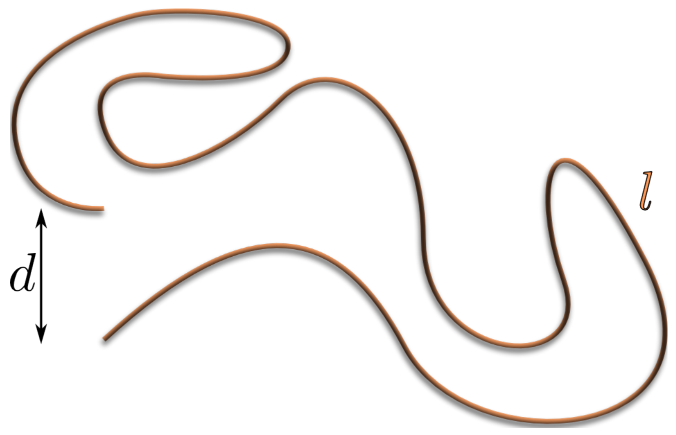


Figure 6.2: *The tortuosity of a streamline, S_T, is the ratio between the length of the streamline, l, and the (shortest) distance, d, between the start and end points, $S_T = \frac{l}{d}$*

without recomputing those stages (only the new clustering needs to be computed). If the user changes the streamline attribute or the number of bins for the streamline signature, then the signatures, similarity matrix and streamline ordering are re-computed (followed by the streamline cluster assignment). If the user moves the seeding rake, the whole algorithm is performed starting with the streamline integration. Figure 6.1 depicts the algorithm overview. The dashed lines show the stages in the pipeline that are affected by the corresponding user-interaction. It is important that no stage of the algorithm introduces a bottleneck into the interaction.

6.2.1 Streamline Attributes

In order to compute the similarity between streamlines we use a number of new and existing attribute measures – curvature, torsion and tortuosity. Curvature measures how much a curve deviates from a straight line. Torsion measures how much a curve bends out of its osculating plane. Tortuosity quantifies how twisted a curve is.

We compute a curvature field for the entire spatial domain. The curvature field is computed with the same sampling as the underlying vector field. To find the curvature for a given position along a streamline we interpolate the value at that position using the curvature field, in a similar way that velocity is interpolated from the velocity field. Curvature, **c**, is computed by [Rot00]:

$$\mathbf{c} = \frac{\mathbf{v} \times \mathbf{a}}{|\mathbf{v}|^3} \tag{6.1}$$

where **v** is the local velocity and **a** is the local acceleration computed by multiplying the local velocity gradient (Jacobian) with the local velocity$(\nabla\mathbf{v})\mathbf{v}$. Note that curvature is a vector quantity. We only require the amount of curvature and thus use the magnitude of the curvature, $|\mathbf{c}|$.

Similarly we pre-compute a torsion field and assign the values to the streamlines. Torsion, τ, is computed by:

$$\tau = \frac{(\mathbf{v} \times \mathbf{a}) \cdot ((\nabla\mathbf{a})\mathbf{a})}{|\mathbf{v} \times \mathbf{a}|^2} \tag{6.2}$$

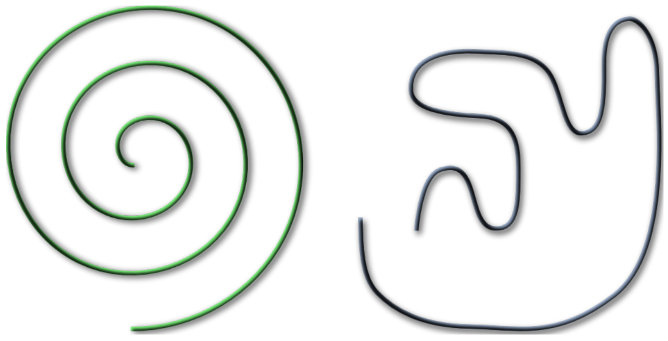


Figure 6.3: *Both of these curves exhibit very similar curvature magnitude values. Therefore, a single global measure fails to distinguish streamline sufficiently. In this case, the tortuosity attribute would fair better but there are cases where this would also fail. A better method creates a distinctive signature for the streamlines using frequency-based attributes.*

The final attribute we use is *tortuosity*. We have found this to produce good results on streamlines whilst having a low cost to compute. It is ratio of the length of curve compared the shortest distance between its start and end points. We apply this to streamlines as a measure of deviation from the shortest path. The tortuosity of a streamline is computed by first summing the distances between all streamline segments. This value is then divided by the distance between the start and end points of the streamline (see also Figure 6.2):

$$S_T = \frac{1}{||f(N) - f(1)||} \sum_{i=1}^{N-1} ||f(i+1) - f(i)|| \tag{6.3}$$

where $f(x)$ is the spatial location of each sample in the vector field and N is the number of points in the streamline. Following this definition, the tortuosity of a straight line is one, and streamlines with higher tortuosity will demonstrate greater deviation from the direct path.

6.2.2 Combining the Metrics

For all streamline points we compute the curvature, torsion and tortuosity values. Each attribute value is then normalized to the range [0,1] over all streamlines. All attributes for a given point are then summed. Normalizing each attribute places equal importance on each attribute and prevents a large value in one attribute from reducing the importance of other attributes. This eliminates the requirement of user-defines weightings for each parameter to counteract this effect, thus, making the computation fully automatic.

6.2.3 Streamline Signatures of Frequency-Based Streamline Attributes

In some cases an overall quantity using the above metrics may produce the same or similar value for a range of streamlines. Thus, dissimilar streamlines may appear similar according to a given measure. For example, using the curvature criterion, a streamline that spirals three times would produce the same result as a more random curve that exhibits the same amount

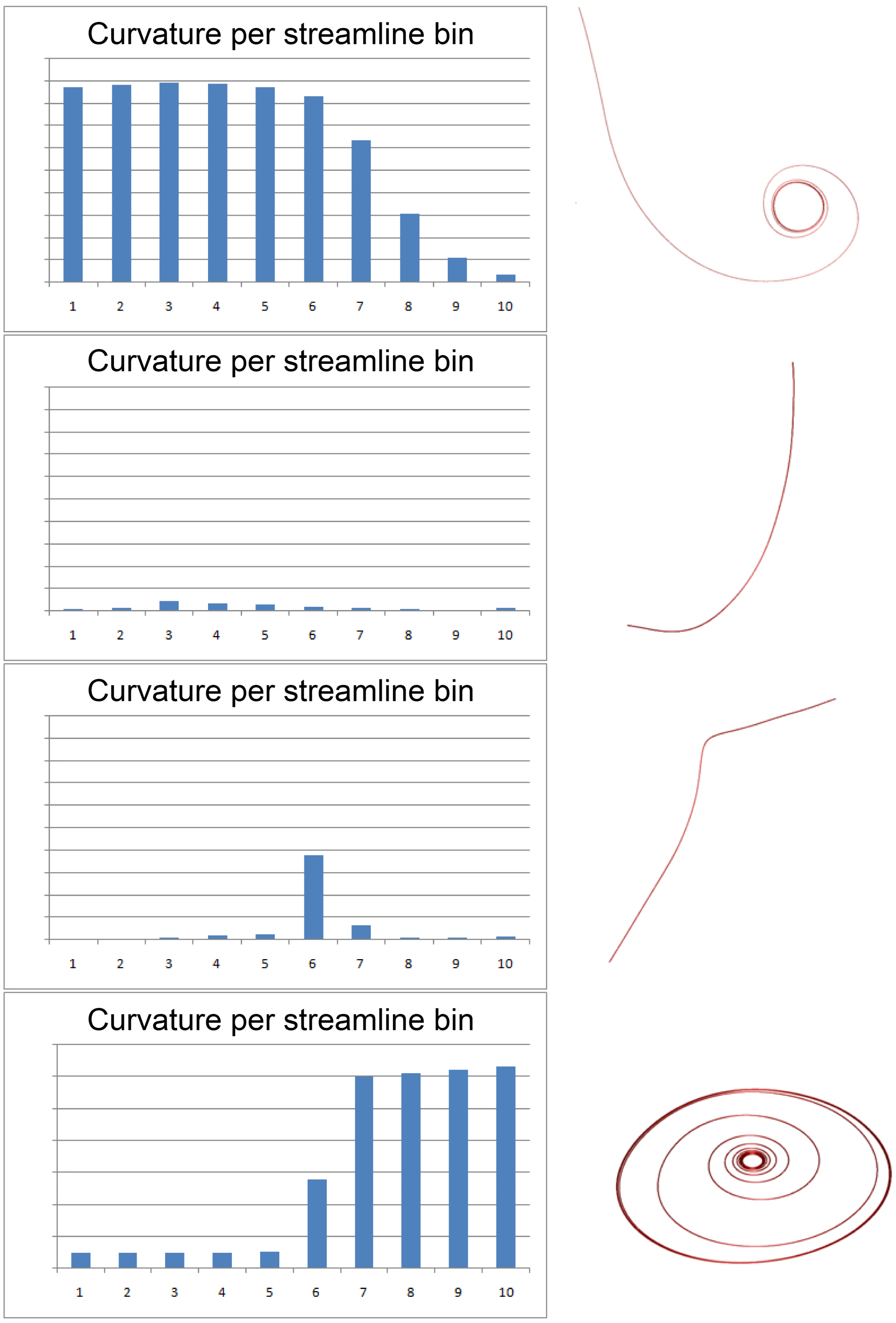


Figure 6.4: Curvature over the streamline intervals. Top: The streamline starts in a vortex, but opens out. Second row: The streamline follows a large arc. Third row: A similar streamline, but this one approaches a saddle point in the middle. Bottom row: A vortex spirals inwards.

of curvature over its length (Figure 6.3). To alleviate this problem, and further differentiate streamlines, we introduce the novel concept of a *streamline signature*. Our motivation for this approach is that this stores a compact description of a streamline and facilitates a matching algorithm (hence the term signature). It should be more descriptive than just the attributes from section 6.2.1. The matching algorithm (section 6.2.4) produces a single dissimilarity rating based on the signature and is shown to be very effective at distinguishing streamlines. The streamline signature is computed by splitting the streamline into several portions/bins consisting of equal numbers of points. The metric is then computed for each bin. These set of values then describe how the attribute changes over the length of the streamline. We now describe this process in more detail.

We set a number of points per bin. We discuss the effects of increasing and decreasing this number later (Section 6.2.4.3). We then iterate over each streamline point and calculate which bin it lies in. The point attributes are then computed (as outlined in the previous sections) and the value is added to the bin. When the entire streamline has been traversed the signature is complete. Note that longer streamlines will have longer signatures. This computation creates a frequency-based pattern for each streamline. Figure 6.4 demonstrates some example frequency-based signatures.

6.2.4 Similarity Measure

We now introduce a novel approach to computing a similarity measure using the streamline signatures. This measure compares streamline signature patterns using the χ^2 test:

$$\chi^2(P_A, P_B) = \sum_{bin \in B} ((P_{bin,A} - P_{bin,B})^2 / (P_{bin,A} + P_{bin,B})) \tag{6.4}$$

where patterns P_A and P_B correspond to the streamline signatures of two streamlines, A and B. The χ^2 test utilizes the streamline signatures to provide a single value that more accurately measures the dissimilarity between streamlines. Identical streamlines result in $\chi^2 = 0$, and $\chi^2 > 0$ for non-identical streamlines. A larger result describes a greater magnitude of dissimilarity. The advantage of using the χ^2 test is that it produces a single value measure of dissimilarity between two streamlines just using their signatures. It operates on the binned data, and is therefore fast to compute (compared to operating on the raw streamline data or for example using the distance metrics [CCK07] [LHS08] where distances between numerous points along both streamlines need to be evaluated).

In the case the number of bins in P_A and P_B are not equal, we iterate only over the number of bins contained in the shorter streamline. This produces partial matching, where only the corresponding portion of the longer streamline is compared to the shorter one. This produces a lower value when the χ^2 test is performed, i.e., the curves are more similar. Another alternative is to give the smaller streamlines the same number of bins as the largest streamline and assign the bins with a value of 0. Thus, when the χ^2 test is performed a greater value is produced, resulting in the streamlines being more dissimilar. However, we favor the first approach. Smaller streamlines, in general, convey less information and many automatic streamline seeding algorithms favor longer streamlines. Taking the latter approach small streamlines rank high in terms of dissimilarity and results in a high probability of them being selected as a cluster centroid in the clustering stage and produce undesirable results.

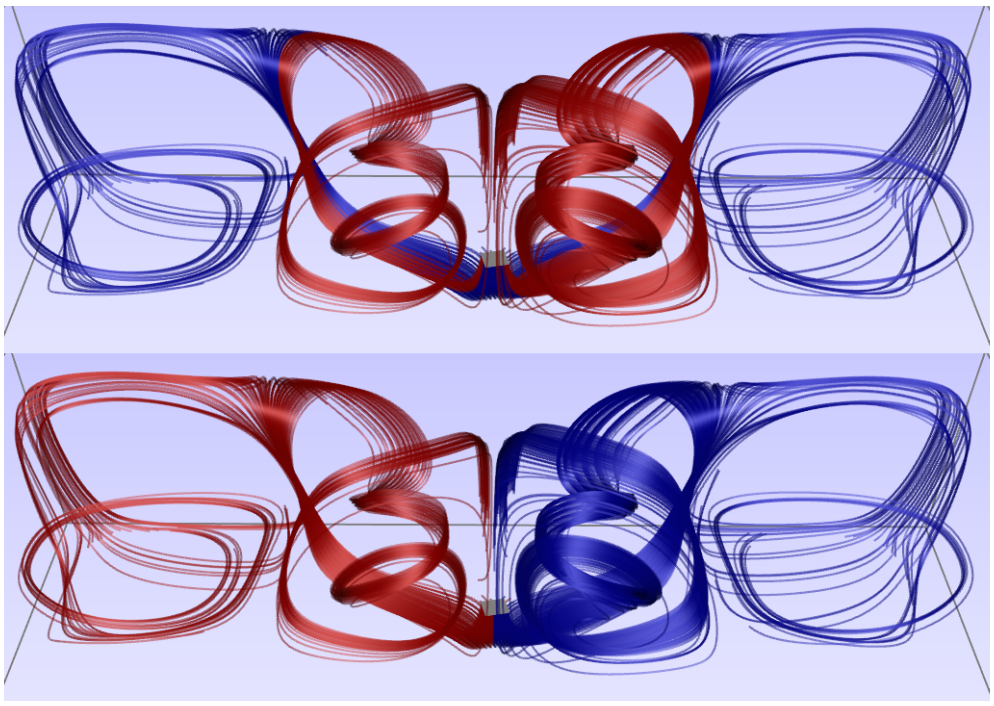


Figure 6.5: *Streamlines seeded from a seeding plane. The top image shows streamline clustering based solely using the streamline signatures ($\alpha = 1$). The bottom image shows clustering using only the Euclidean distance measure ($\alpha = 0$). Which set of clusters is correct is subjective. Our method provides the flexibility to allow the user to quickly navigate to their preferred results.*

6.2.4.1 Similarity Matrix

The χ^2 test is performed for all streamline pairs, from which, a 2D matrix, M^{sim}, of similarity values is constructed. The similarity matrix provides a fast lookup table for the clustering phase of our algorithm. Each column in the matrix corresponds to the set of similarity values for a streamline against all others and the row determines which streamline it is measured against. Entry $M^{sim}_{i,j}$ corresponds to the dissimilarity between streamlines i and j. The similarity matrix is therefore a symmetric matrix, whose main diagonal is composed of zeros, i.e., $M^{sim}_{i,j} = M^{sim}_{j,i}$ and $M^{sim}_{i,i} = 0$.

6.2.4.2 Euclidean Distance Measure

Our method matches streamlines using their signature. There is no weighting attached to proximity. Previous distance metrics attach a high weight to proximity. In those approaches two similarly shaped streamlines far apart are more dissimilar than two dissimilarly shaped streamlines collocated. Our approach compares based on signature which is related to streamline shape. This can lead to streamlines being identified as similar even if they are not collocated. The user may wish for the distances between streamlines to be factored in. Therefore we introduce a weighting based on distance to give the user more control over this aspect. The default for the weighting co-efficient is zero – only the signature is matched. If the user desires close streamlines to have a higher similarity, the weighting can be increased using a slider. This occurs in real-time, so the user can explore this parameter space interactively. We provide this option by adding a lightweight distance measure into our pipeline.

Many distance tests result in the degradation in performance of similarity algorithms, this is demonstrated in Section 6.4.1. We keep the number of distance tests to a minimum as they are only meant to supplement our χ^2 similarity measure. We record the position of the last point in every bin. These points are then used in the computation of the distance tests. The mean of these distances is used to construct a second similarity matrix.

The distance similarity table is then combined with the χ^2 similarity table and a weighting coefficient to produce the final result. The similarity value for a given similarity matrix element, $M^{sim}_{i,j}$, is equivalent to this single measure:

$$M^{sim}_{i,j} = \alpha\chi^2_{i,j} + (1-\alpha) mean\_dist(i,j) \tag{6.5}$$

where α is the weighting coefficient, $\chi^2_{i,j}$ is the χ^2 similarity measure between streamlines i and j (as described in Section 6.2.4), and $mean\_dist(i,j)$ is the mean distance between streamlines i and j, computed using only a subset of their points as outlined above. Providing this extra measure affords the expert user more control over the clustering results. Figure 6.5 shows the effect of this parameter.

6.2.4.3 Choice of Bin Size

Using too few point per bin raises problems with the alignment of the bins in the signature between streamlines. If we have too many bins a finer sampled signature is produced. In some cases this may produce a very localized change in the signature. This can lead to problems with the streamline bins not correctly aligning between a pair of streamlines. For example, if we imagine a pair of neighboring streamlines that both have a point of inflexion, in their signatures there will be a spike due to a large change in curvature. However, if the spike occurs at a slightly different position (arc-length) along each streamline. A finer sampling of the signatures may results in the inflexion point occurring in different bins on the streamlines. This would result in the χ^2 test producing a high dissimilarity for these streamlines. A slightly more coarser sampling for the signatures provides a greater probability that the feature is captured by the same bin(s) and thus gives the desired result. This problem is also greatly reduced when using rakes and seeding planes. We set the seeding object to be orthogonal to the local flow. Thus, any problems with shift invariance are minimized.

Conversely, large numbers of points per bin is equivalent to an under-sampling problem with the possibility that small-scale characteristics are missed. Taking this idea to its extreme is to have a single bin per streamline. This leads to similar problems as outlined in Figure 6.3. Through experimentation we found that a bin size of 5% of the number of points in the longest streamlines provided good results.

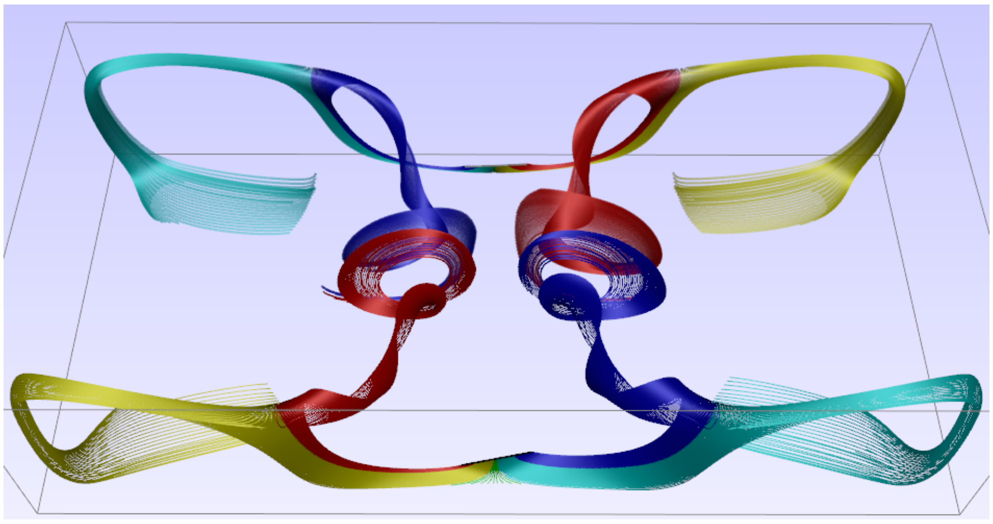


Figure 6.6: *Our similarity measures and clustering algorithm clearly segment the streamlines on this rake into distinct, intuitive clusters. Two rakes were used to generate two sets of streamlines on a simulation of Bernard convection in this image. Color indicates cluster membership. Note colors are re-used for each rake (i.e., the two red clusters are distinct clusters).*

6.2.5 Clustering

We use a greedy clustering technique to compute a list of streamlines such that the first n streamlines can be used as cluster centers. Parameter n can be varied interactively. Like other approaches based on hierarchical clustering we use the similarity matrix M^{sim}. The highest entry of M^{sim} corresponds to the two streamlines that are least similar to each other. These are inserted as the first members of the list of potential cluster centroids C. Subsequent centroids are iteratively added to C by adding the least similar streamline from all current members of C, Section 6.2.5.1 details this further. This process terminates when all streamlines have been added. This results in C being an ordered list of potential cluster centroids. When the user desires n clusters, the first n members from C (i.e., $C_1, \ldots, C_n$) are used as the cluster centroids. The remaining streamlines are then assigned to the relevant cluster based on their position along the rake and their similarity with the relevant cluster centroids. Figure 6.6 shows the results of our clustering algorithm on a simulation of Bérnard convection using two seeding objects. We now discuss this process in more detail.

6.2.5.1 Greedy Clustering

C_1 and C_2 are the two most dissimilar streamlines that are found during the construction of M^{sim}. The subsequent centroids are computed by iteratively applying Algorithm 1 to find the next least similar streamline. The corresponding columns (in M^{sim}) of every streamline stored in the centroid list are used. For each row of these columns the minimum is found. The next centroid is provided by the row in M^{sim} that contains the running maximum of the column minimums for each row. For an example see Table 6.1 and Algorithm 1.

Algorithm 1 Computing Cluster Centroids

```
$C \leftarrow$ initially contains the indices of two least similar streamlines
$n = 2$
while $|M^{sim}| <> numOfStreamlines$ do
  $val = -1$
  for each streamline $i \notin C$ do
    $temp = min(M^{sim}_{C,i})$
    if $temp > val$ then
      $temp = val$
      $index = i$
    end if
  end for
  $C.Append(index)$
  $n = n + 1$
end while
```

M^{sim}						
Streamlines	1	2	3	4	5	6
1	0.0	5.5	5.8	13.9	63.3	10.2
2	5.5	0.0	6.2	14.4	41.6	24.1
3	5.8	6.2	0.0	4.4	57.8	19.6
4	13.9	14.4	4.4	0.0	54.5	30.6
5	63.3	41.6	57.8	54.5	0.0	100.0
6	10.2	24.1	19.6	30.6	100.0	0.0

Table 6.1: *An example of our greedy clustering algorithm. The table represents a real example of M^{sim}. Streamlines 5 and 6 are the most dissimilar so $C = \{5,6\}$. From the two columns 5 and 6, row 4 has the highest minimum (30.6). Therefore, 4 is added to $C = \{5,6,4\}$. Now from the three columns 5, 6 and 4, row 2 has the highest minimum (14.4) and 2 is added to $C = \{5,6,4,2\}$. This is repeated until all streamlines have been added and $C = \{5,6,4,2,1,3\}$. Algorithm note: We find the maximum while building M^{sim}. The minimum for each row to the current C is retained after calculation, so the min() operation only requires 1 comparison for each row per iteration with the latest addition to C.*

6.2.5.2 Cluster Assignment

The centroid list, C, is used to cluster the streamlines. For n clusters, the first n streamlines in C are retrieved. These form the set of cluster centroids to which we add the remaining streamlines. The streamlines of the rake are traversed sequentially from one end to the other. Streamlines that appear before the first centroid along the rake are automatically assigned to the first cluster. Likewise, streamlines that appear after the final centroid are automatically assigned to the last cluster. For the seeding plane, streamlines are assigned to the most similar centroid.

6.2.5.3 Algorithm Analysis

Most integral curve matching algorithms (e.g., [CCK07, CGG04, ZCL08]) use single, average or complete link hierarchical clustering techniques, whereas we target finding candidate centroids first, followed by selection of the number of centroids, followed by cluster association. We take this approach because (1) it is fast, (2) to ensure that the chosen centroids express the least similar regions within the streamlines and (3) to provide consistency as the user adjusts the number of clusters. See Jain et al. [JMF99] for an overview of data clustering. A paper by Gonzalez [Gon85] describes a similar algorithm to ours, differing only by re-clustering at each iteration. Proof of error bound and complexity is also given in that theory-based paper.

6.3 Enhancements and Extensions

This section presents several enhancements and extensions to our algorithm. We also involved a CFD expert to evaluate our approaches.

6.3.1 Focus+context Visualization

Our clustering strategy segregates the streamlines into groups with distinctive behavior. We provide a tool that allows the user to analyze the clusters using a focus+context visualization. This reduces visual clutter and aids in the analysis of the flow. The varying behavior of streamlines along the rake can be quickly and easily explored. The resulting visualization may aid the presentation and communication of results by highlighting a particular flow behavior.

The user selects the cluster they wish to analyze, the streamlines belonging to this cluster are mapped to a high opacity for emphasis. The remaining streamlines are mapped to a lower opacity. This allows the user to focus on the chosen cluster within the context of the entire set of streamlines.

Figure 6.7 shows the focus+context visualization applied to sets of streamlines on the smoke plume simulation, with each row corresponding to a different seeding configuration. The left column of images show streamlines colored according to velocity magnitude. The streamlines exhibit a high-level of visual complexity. Even using transparency it is difficult to distinguish the different flow characteristics and how they interact with each other. The middle and right column images show the results of our clustering strategy with focus+context applied. Streamlines are colored according to cluster membership. In each image a different cluster is highlighted. The CFD expert found that by cycling through the clusters they are able to focus attention on different parts of the flow behavior which aids their understanding. For example, in these sets of streamlines there are several cases where different bundles of streamline separate and converge together. The focus+context tool allows the user to see how these separate bundles interact with each other when they come closer together. Using a constant opacity and coloring according to velocity magnitude, this is much more difficult. They comment that by using the color-coded clusters it is easy to distinguish the portion of the seeding object a particular behavior emanates from. They also indicate throughout the session that the clustering results are, in general, favorable and the ability to fine tune the clustering results is a welcome feature.

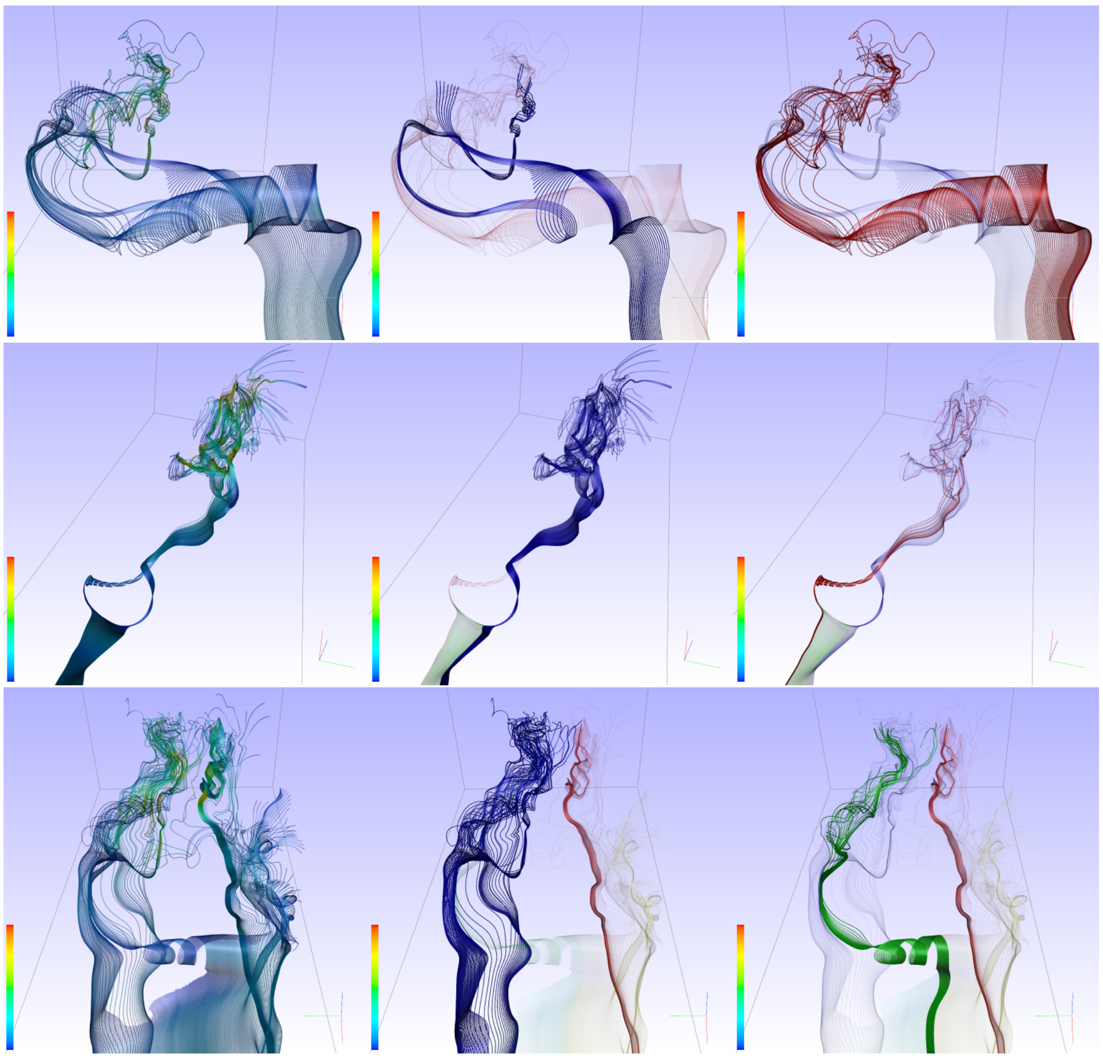


Figure 6.7: *Focus+context visualization of streamline clusters created using our method on the smoke plume simulation. Each row represents a different seeding configuration. Images in the left column show the set of streamlines colored according to velocity magnitude. The middle and right columns show the focus+context views, both setting different clusters as the focus. The selected cluster (the focus) is shown with a high opacity. The remaining clusters (context) are assigned a lower opacity. This allows the user to analyze each cluster more easily while retaining the context of the cluster in regards to the entire rake. It can be seen that this reduces visual complexity and reduces the effects of occlusion. The user can switch between clusters and change the viewpoint interactively. The focus+context view allows the user to analyze the interactions between the streamlines in more detail. Domain expert feedback was that the chosen clusters were good.*

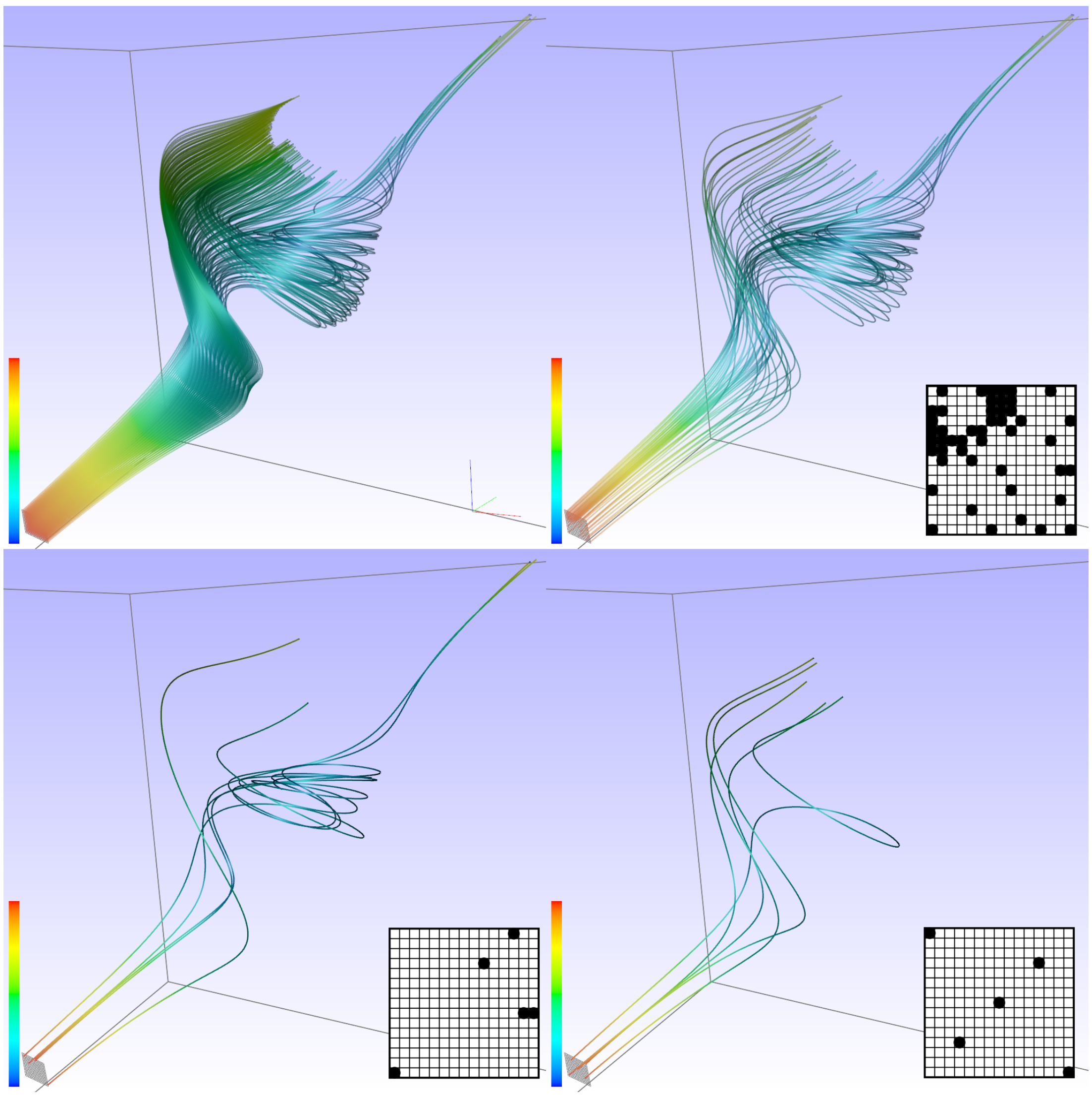


Figure 6.8: *Our filtering technique allows the user to filter out streamlines based on our similarity measures. The streamline that is most similar to the current set is iteratively removed – leaving the most dissimilar (and, hence, the most illustrative) streamlines. The top-left image shows the original dense set of 225 streamlines seeded from a plane. The top-right image shows the our method filtering out approximately 75% of the initial streamlines. The bottom-left image shows filtering using our strategy to leave just 5 streamlines. Using our approach the few streamlines that remain depict the main characteristics of the superset. The bottom-right image shows the 5 remaining streamline using filtering to leave a more uniform distribution of the seeds. A lot of detail is lost when streamline similarity is not taken into account as shown in the bottom-right image. The insets in each image show the seeding distribution for the seeding plane.*

6.3.2 Streamline Filtering

A strong benefit of our approach is that it allows us to introduce streamline filtering to produce variable inter-seed distance along seeding objects. Typically streamlines are seeded at equidistant positions along the seeding curve or at regular points along a seeding plane. This results in the user having to specify a dense set of streamlines if they encounter complex flow behavior. The side-effect of this is that there will be dense bundles of streamlines in regions where the flow is more uniform, which may lead to unnecessary visual clutter and obscure useful information. An ideal solution is to produce an expressive set of streamlines which captures all of the details of flow behavior while reducing the redundancy in the visualization. Burdening the user with the task of manually filtering the streamlines would be a time-consuming process that may have to be repeated many times as they interact with the rake.

Streamline filtering is performed by utilizing the ordered centroid list, C (Section 6.2.5.1). Recall, to construct this list the streamline that is least similar streamline to those in C is added. Looking at this list in reverse the streamlines are now in order of the most similar. This allows us to provide an easy filtering strategy. The user is provided with a simple slider in the graphical user interface. As the user increases the value of the slider streamlines are removed from the visualization. The streamlines are simply not rendered, however, they do stay in memory. Thus no re-computation is necessary if decreases the value of the slider to re-add some of the streamlines that were filtered. The streamlines are filtered based on the reverse of C. If the user filters n streamlines, the last n entries in C are not used.

Figure 6.8 shows our filtering strategy applied to a set of streamlines generated on a simulation of Arnold-Beltrami-Childress (ABC) flow. The top-left image shows the complete set of 225 streamlines. It can be seen that there is a high level of redundancy with the streamlines. The top-right image shows filtering removing approx 75% of the original set of streamlines. In the bottom-left image the number of streamlines is reduced to five using our method. The bottom-right image shows the result of a more uniform filtering strategy leaving only five streamlines. Using our method the remaining streamlines still depict helpful information about the flow characteristics. The uniform filtering loses important information. The insets show the final seed distribution of the seeding plane for each method.

Figure 6.9 shows a comparison between a dense set of streamlines and two sparser sets generated on the simulation of Hurricane Isabel. In this version 4 clusters are selected and streamline filtering is performed on a per-cluster basis. The left image shows the rake at a full resolution of 200 streamlines. The middle image shows the results using our filtering method. Using filtering the expert reduced this number of streamlines down to just 25. Using only 25 streamlines he notes our method still preserves the interesting flow characteristics – in particular the two regions of vortex behavior. The reduced number of streamlines produces a visualization that suffers from occlusion and visual complexity to a much lesser extent. The expert noted that the controls were easy to use and he was able to produce the final result in a matter of seconds. The expert was particularly interested in variable inter-seed distance along the rake, stating it allows the visualization to express more with less, and that is reduces visual information overload. The right image provides a comparison using equidistantly seeded streamlines. The expert noted that the second vortex region was not visualized and that the separation regions were not as clearly defined.

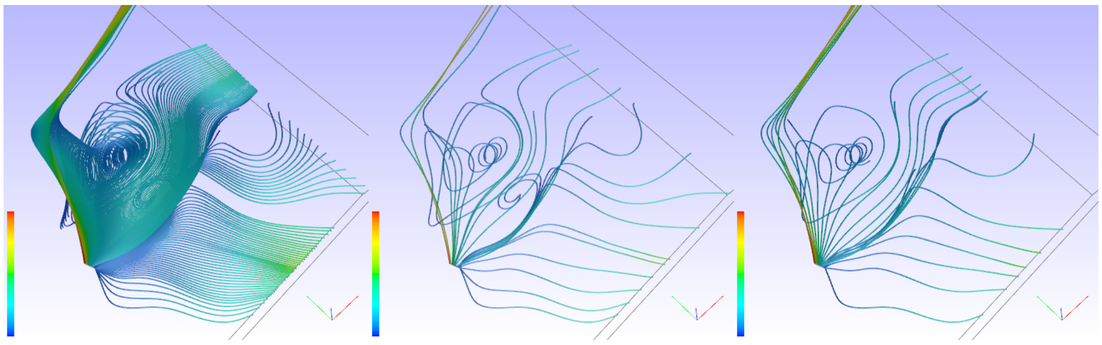


Figure 6.9: *Our filtering strategy applied to the simulation of Hurricane Isabel. 4 clusters were selected and streamlines were filtered on a per-cluster basis. 200 hundred streamlines in the left image are filtered down to 15. The middle image uses filtering based on similarity. The right image shows equidistant seeding of the streamlines. Our method better represents the saddle regions and preserves the second (smaller) vortex structure. The second vortex structure is lost in the right image. Using our method we preserve the main flow characteristics and reduce visual clutter and occlusion.*

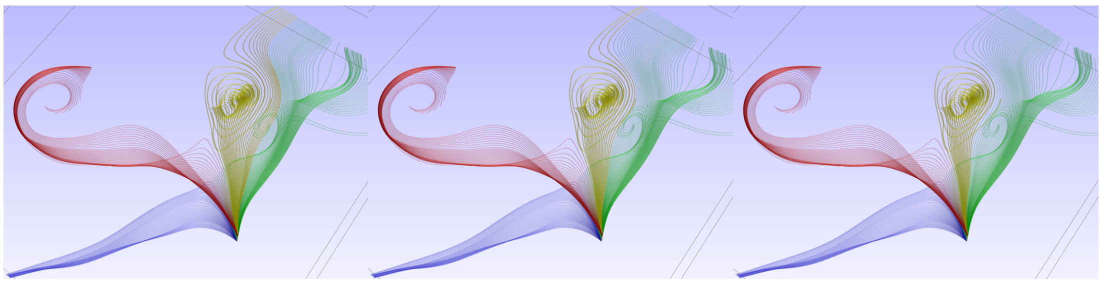


Figure 6.10: *Three sets of four clusters for the same set of streamlines. (Left) The clustering produced using the similarity metric from [CGG04]. This method has produced good quality clustering. However, a user may wish that the gold cluster contains only the streamlines that pass through the vortex region. This method provides no user-parameters for the user to tailor the result to their requirements. (Middle) The set of clusters resulting from our algorithm. Once again, the gold cluster contains streamlines that don't enter the vortex region. However, due to the instantaneous feedback the user can very quickly fine tune the clustering and ensure that the cluster is constrained only to the streamlines that enter the vortex region (Right). Note that the clustering in the left image took just over 24s to produce. Our method produced the streamlines in just over 0.07s. Our weighting parameter space can be fully explored in just a few seconds. Therefore, the user can easily tailor the visualization to their needs before the other method has finished its initial clustering.*

6.3.3 Interaction

In our domain expert interview it was found that the experts desire a high level of interactivity with the visualization application. It was also noted that the parameter space should be kept as small as possible. It should also be simple and intuitive to navigate. The use of a simple GUI widget like the slider provides intuitive interaction. To this end, our algorithm relies on only a few parameters:

- The number of clusters
- The weighting of the χ^2 term
- The number of streamlines to be filtered from a cluster (this is set on a per-cluster basis).

We utilize GUI slider widgets to control these parameters. Sliders facilitate the request from domain experts for a simple, intuitive method of navigating the parameter space. As shown in Section 6.4.1 our algorithm provides interactive rates and changes to these parameters are displayed in real-time to the user. Fast response from parameter updates also aids the user in navigation, allowing them to quickly find a good set of values for the parameters.

All similarity and clustering algorithms have failure cases in which the end result may not match what a user expects with a fully manual clustering scheme. When using a scheme that has no input parameters such as the one by Corouge et al. [CGG04], the user has no control over the final result. If the clustering proves inadequate the user cannot customize the visualization or has to resort to a different algorithm. In contrast, the algorithm of Chen et al. [CCK07] does provide user-modifiable parameters. However, their algorithm is computationally expensive (see Section 6.4.1) and some parameters, such as window size, result in a complete re-computation – meaning the user has to wait for feedback from the application. Also the effect that a change in the user-parameters produces is unintuitive. This means that the user will have to perform a slow search through this parameter space using trial and error. The method of Zhang et al. [ZCL08] also requires a re-computation of all similarity distances when there is a change in the minimum distance threshold.

The domain expert praised our method for allowing the user to quickly and easily customize the final result of the clustering. When navigating through our parameter space they are presented with feedback almost instantaneously – allowing them to explore the flow and customize the final visualization result.

6.3.4 Unsteady Flow

Our method can be extended to unsteady flow. We compute the torsion and curvature fields for every time-step of the simulation. The algorithm then proceeds as outlined in the method overview (Figure 6.1). However, instead of streamlines, we trace pathlines. When the pathline attributes are computed, we use the field that matches the corresponding time of the pathline point. Where a pathline point does not lie exactly on a time-step we interpolate between the two closest fields. For example, if we were computing the torsion at a point along a pathline at $t = 4.7$, we take the values at that position from the torsion field computed for time-steps 4 and 5. The final value is then linearly interpolated. Figure 6.11 shows clustering results using our

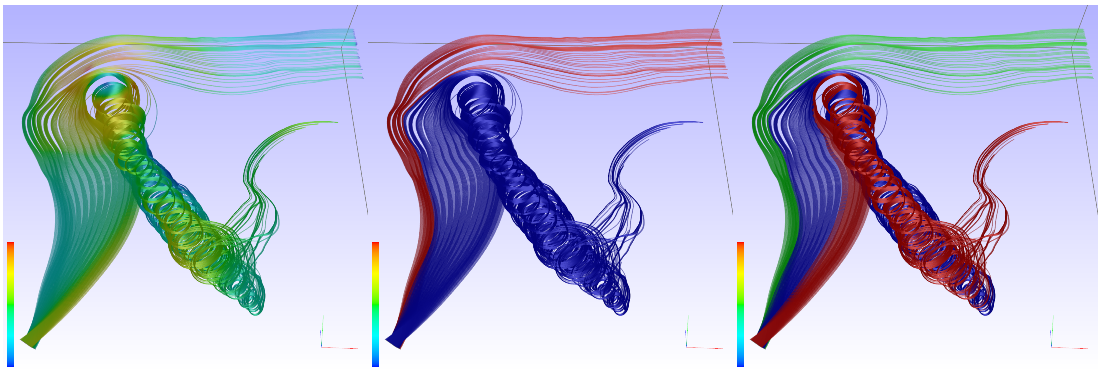


Figure 6.11: *Our method can be applied to unsteady flow. Torsion and curvature fields are generated for every time-step. Pathlines are then traced and the clustering algorithm is performed on them. Interpolation is used to construct field values that do not lie exactly on a time-step. This set of pathlines was generated on the simulation of Hurricane Isabel and seeded from a seeding plane. The left image shows the pathlines with color mapped to velocity magnitude. The middle and right images show the pathlines clusters in to 2 and 3 clusters respectively.*

algorithm. In this figure the pathlines have been seeded using a seeding plane and are traced in the simulation of Hurricane Isabel.

6.4 Comparison

The enhancement of seeding rakes for improved usability is a novel research direction. We have also presented a novel technique of computing streamline signatures and using those for the basis of a novel similarity measure. Our approach is intended to give fast computation, good selection of similarity, leading to interactive and intuitive rake control. Streamline similarity is a well research topic, but as mentioned in the related work section, all the measures involve performing a great number of distance test between streamlines. In this section we compare our approach to state of the art approaches for detecting similar integral curves. The distance measures we compare against are Corouge et al. [CGG04] (equation 2), Zhang et al. [ZCL08] (Section 3.2) and Chen et al. [CCK07] (Section 3).

6.4.1 Performance

Table 6.2 reports the performance times of our algorithm tested on a 2.4Ghz Intel Core 2 Quad CPU with 4GB RAM using a single thread. We compare our running times against algorithms by Chen et al. [CCK07], Corouge et al. [CGG04] and Zhang et al. [ZCL08]. The results in Table 6.2 are generated using 200 streamlines, each consisting of up to 1000 points. We report streamline integration times in order to provide a context which to compare the clustering phase. The last column in the table gives the performance times as a factor of our method. In this scenario our algorithm takes 0.062 seconds to complete – providing interactive results. In contrast, the techniques of Chen et al. [CCK07], Corouge et al. [CGG04] and Zhang et al. [ZCL08] take more than 20 seconds to complete and are thus, prohibitively expensive for

Algorithm	Integration	Similarity & Clustering	Total	Factor
Our Method	0.031s	0.031s	0.062s	1.0x
Chen et al. [CCK07]	0.031s	22.400s	22.431s	361.79x
Corouge et al. [CGG04]	0.031s	20.150s	20.181s	325.50x
Zhang et al. [ZCL08]	0.031s	20.210s	20.241s	326.27x

Table 6.2: *Performance times of our algorithm in comparison with [CCK07], [CGG04] and [ZCL08]. The first column identifies the algorithm used. The second column shows the integration time for the streamlines. The similarity computation and clustering times are combined in the third column and the fourth column shows the total computation time. The final column shows the total computation times as a factor of our algorithm. It can clearly be seen that our method produces interactive results while comparative methods are too expensive for interaction.*

use as an interactive technique. As highlighted in the final column of the table, these algorithm take over 300 times as long as our algorithm to compute. Figure 6.10 demonstrates that our method produces comparable results against the state-of-the-art. However, our method affords the user the flexibility to modify the clustering results.

The vast majority of the computational workload in these algorithms occurs during the large number of distance calculations to compute the similarities. Our algorithm alleviates this by greatly reducing the number of distance tests that need to be performed. The small number of distance tests, coupled with our (less computationally expensive) χ^2 test on the binned streamline signatures, produces good clustering results at a fraction of the expense of pure distance-based similarity metrics. The seeding object type has very little effect on the performance times. The main influence is the number of streamlines used.

6.4.2 Limitations

Our method is shown to be robust when seeding from rakes and planar surfaces. It provides a high degree of interactivity not possible with other methods because of the fewer computations required to create the measure. Streamlines seeded from the rake or surface start orthogonal to the seeding structure and their shape are correlated to this start seed position. If streamlines were placed arbitrarily throughout the domain (e.g., in full domain seeding), then this feature of our algorithm would no longer apply. The results would be that the algorithm, which does not provide shift-invariance in this version, would require more development to extend it to arbitrary full domain coverage.

6.5 Conclusion

We present a tool for enhancing the user experience while interactively seeding streamlines. Our tool enables the user to gain additional insight from a given set of interactively placed seeds. Streamlines can be clustered together and visualized using focus+context methods giving the user the opportunity to reduce visual complexity and target distinct flow behavior that they wish to investigate. The tool also provides a filtering scheme to produce streamlines that are seeded at non-equidistant positions along the seeding object. This technique produces a set of streamlines that preserve the detail of the visualization while greatly reducing the number of streamlines. This is achieved by filtering out the most similar streamlines and leaving the least similar and hence most illustrative set for a given rake.

A large effort is placed on providing interactivity for our tool. From domain expert interviews it was found that experts prefer intuitive tools that they can modify to meet their requirements. We reviewed previous similarity metrics and found that they were too computationally expensive to meet these requirements. Thus, we introduce the novel concept of the streamline signature. The streamline signature is produced from binned data that provides a distinct pattern for each streamline. We also employ the χ^2 test on the streamline signatures as a similarity measure. To the author's knowledge this is the first time the χ^2 test has been used in this context. We also provide a set of attributes that we found useful for the computation of the streamline signature. This is by no means an exhaustive list and further options are available for further research.

We demonstrate the performance of our algorithm compared to other similarity metrics and show that we can provide similar results an order of two magnitudes faster. Finally, our tool allows the user to fine tune the visualization quickly and easily in real-time – reducing the blackbox effect of an automatic algorithm.

In the future we would like to investigate 2D attribute parameter spaces, allowing the user to investigate how one flow attribute changes with another attribute. We would also like to apply this method to DTI fiber bundling and further investigate the possibility of using our method as a fast 3D, full domain streamline seeding method.

Part III

Software Framework and Conclusion

7

Design and Implementation of State-of-the-Art Geometric Flow Visualization Software

"Commit yourself to quality from day one ... it is better to do nothing at all than to do something badly."
-Mark H. McCormack (1930–2003)[1]

Contents

THE demand for flow visualization software stems from the popular (and growing) use of Computational Fluid Dynamics (CFD) and the increasing complexity of simulation data. CFD is popular with manufacturers as it reduces cost and the time of production relative to the expense involved in creating a real physical model. Modifications to a physical model to test new prototypes may be non-trivial and expensive. CFD solvers enable a high degree of software-based testing and refinement before creating a real physical model.

The visualization of CFD data presents many different challenges. There is no single technique that is appropriate for the visualization of all CFD data. Some techniques are only suitable for certain scenarios and sometimes an engineer is only interesting in a sub-set of the data or specific features, such as vortices or separation surfaces. This means that an effective flow visualization application must offer a wide range of techniques to accommodate these requirements. The integration of a wide variety of techniques is non-trivial and care must be taken with the design and implementation of the software.

We describe a flow visualization software framework that offers a rich set of state-of-the-art features. It is the product of over three years of development. This chapter provides more

[1]Was an American lawyer and agent for professional athletes.

details about the design and implementation of the system than are normally provided by typical research papers due to page limit constraints. Our application also serves as a basis for the implementation and evaluation of new algorithms. The application is easily extendable and provides a clean interface for the addition of new modules. More developers can utilize the code base in the future. A group development project greatly varies from an individual effort. To make this viable, strict coding standards [Lar10] and documentation are maintained. This will help to minimize the effort a future developer needs to invest to understand the codebase and expand upon it.

Throughout this chapter we focus on the design and implementation of our system for flow visualization. We address how the systems design is used to address the challenges of visualization of CFD simulation data. We describe several key aspects of our design as well as the contributing factors that lead to these particular design decisions.

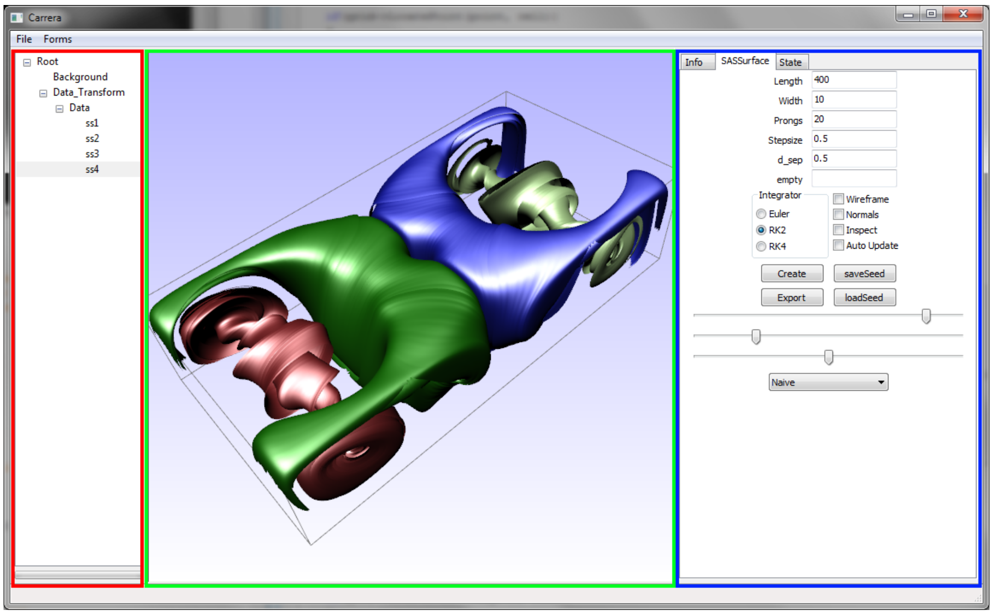
The rest of this chapter is organized as follows: Section 7.1 describes the user requirements and goals for our application. Section 7.2 provides an overview of the application design. A description of the major systems is then provided with the key classes and relationships are discussed. The chapter is concluded in Section 7.3. Throughout the chapter, class hierarchies and collaboration graphs are provided for various important classes of the system.

7.1 System Requirements and Goals

Our application framework is used to implement existing advanced flow visualization techniques as well as being a platform for the development and testing of new algorithms. The framework is designed to be re-used by future developers researching flow visualization algorithms to increase efficiency and research output. Figure 7.1 shows a screenshot of the application in action.

Interactivity Users generally require flexibility over the final visualization and favor feedback as quickly as possible after modifying visualization parameters. Our system is designed to enable a high level of interaction for the user. Providing such a level of interaction allows for easier exploration of the data. The user can also tailor the resulting visualization to their specific needs. This level of interactivity is also of use to the developer. Some algorithms are inherently dependent upon threshold values and parameters. Providing the functionality for these to be modified at run-time allows the programmer to test varying values without having to modify and recompile the code. Once the final value has been found it is then possible to remove the user-option and hard code as a constant if required.

Support for Large, High-dimensional, Time-dependent Simulations The application is used to visualize the results of large simulations comprised of many time-steps. Not every time step has to present in main memory simultaneously. Our application uses a streaming approach to handle large data sets. A separate data management thread continually runs in the background. When a time-step has been used this manager is responsible for unloading the data for a given time-step and loading in the data for the next (offline) time-step. A separate thread in used to minimize the interruption that occurs from the blocking I/O calls. If a single



Length 400
Width 10
Prongs 20
Stepsize 0.5
d_sep 0.5
empty

Integrator: Euler [✓ RK2] RK4

Wireframe ☐ Normals ☐ Inspect ☐ Auto Update ☐

Naive

Figure 7.1: *A screenshot of the application showing the GUI providing controls for stream-surfaces computed on a simulation of Rayleigh-Bénard convection. The application window is split up into three distinct regions. 1. The **Application Tree** (highlighted by the red box) is used to manage the assets in the scene. 2. The **Rendering Window** (highlighted by the green box) displays the visualization results and allows the user to interactively modify the viewing position and orientation. 3. The **Asset Control Pane** (highlighted by the blue box) displays the current set of controls for the selected asset. The GUI is context sensitive for the benefit of the user and the asset control pane only displays the controls for a single tool at any given time. Should a different visualization tool be selected a new set of controls are displayed and the unrequired ones removed. This approach is adopted to provide a simple, uncluttered interface, allowing the user to focus only on the necessary parameters/controls.*

threaded solution was used the system would compute the visualization as far as possible with the in-core data and then have to halt until the new data is loaded. Note that in many cases the visualization computation still out performs the data loading in a multi-threaded solution, however, the delay may be greatly reduced.

Simple API The system is intended for future developers to utilize. In order to achieve this the system must be composed of an intuitive, modular design maintaining a high level of re-usability. Extensive documentation and coding conventions [Lar10] are maintained to allow new users to be able to minimize the overhead required to learn the system. The system is documented using the doxygen documentation system [vH04], the documentation can be found online at http://cs.swan.ac.uk/∼cstony/documentation/.

Support for a Wide-variety of Visualization Methods and Tools Our application is designed as a research platform. A variety of visualization methods have been implemented so that new algorithms can be directly compared with them. Therefore, the system is designed to be easily extensible. Some of the key tools integrated into our application include:

1. Integral Curves (with illumination) [MPSS05]
2. Streamsurfaces and Pathsurfaces [MLZ09]
3. Isosurfaces [LC87]
4. Streaksurfaces [MLZ10]
5. Slice probes
6. Line Integral Convolution (LIC) [CL93]
7. Critical Point Extraction
8. Parameter Sensitivity Visualization [MEL$^+$11a]
9. Clustering of integral curves [MJL11]
10. Vector field resampling
11. Image output to multiple formats
12. Integral curve similarity measures [MJL11]
13. Computation of the Finite Time Lyapunov Exponent (FTLE) [Hal01]

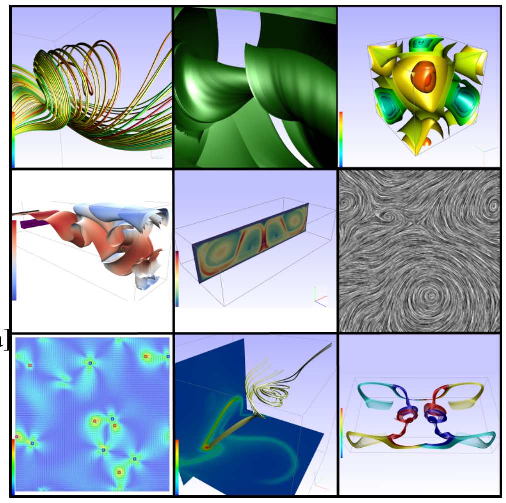


Figure 7.2: *Traversing left-to-right from top-to-bottom, examples of visualization techniques 1-9.*

7.2 System Design and Implementation

Figure 7.3 shows the design of our application. The major subsystems are shown along with the relationships of how they interact with one another.

The *Graphical User Interface* subsystem is responsible for presenting the user with modifiable parameters and firing events in response to the users actions. The user interface is designed to be minimalistic. It is context sensitive and only the relevant controls are displayed to the user at any time. The GUI was created using the wxWidgets library [wGL]. wxWidgets provides a cross-platform API with support for many common graphical widgets – greatly increasing the efficiency of GUI programming. The *3D Viewer* is responsible for all rendering. It supports the rendering of several primitive types such as lines, triangles and quads. The 3D viewer is implemented using OpenGL [ARB00] for its platform independence. The *Simulation Manager* stores the simulation data. It stores vector quantities such as velocity and scalar quantities such as pressure. The simulation manager is also responsible for ensuring the correct time-steps are loaded for the desired time. The *Visualization System* is used to compute the visualization results. This system is comprised of several subsystems. Each major system of the application is now described in more detail.

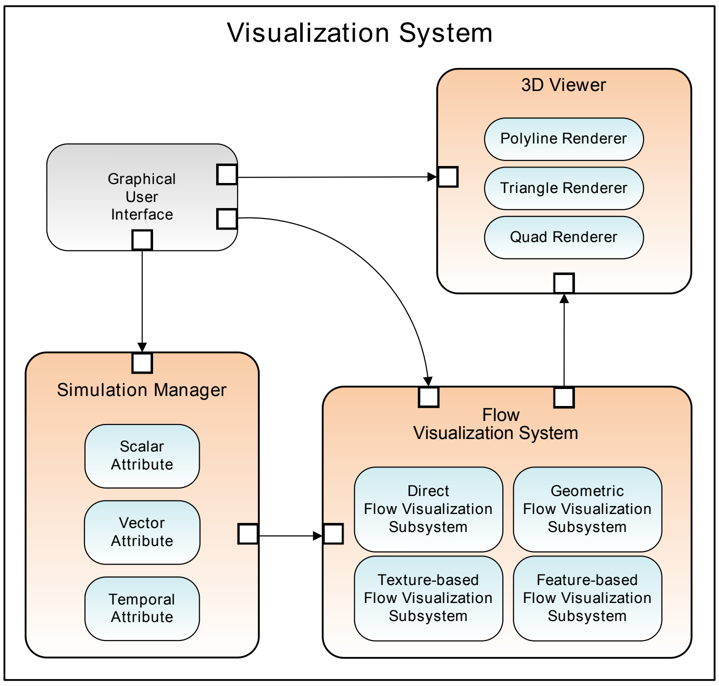

Visualization System
3D Viewer
Polyline Renderer
Triangle Renderer
Quad Renderer
Graphical User Interface
Simulation Manager
Scalar Attribute
Vector Attribute
Temporal Attribute
Flow Visualization System
Direct Flow Visualization Subsystem
Geometric Flow Visualization Subsystem
Texture-based Flow Visualization Subsystem
Feature-based Flow Visualization Subsystem

Figure 7.3: *An overview of our system design. This shows the major subsystems of the framework and which systems interact with one another.*

7.2.1 Visualization System Design

The visualization system is where the visualization algorithms are implemented. The application is designed to separate the visualization algorithm logic, the rendering logic, and the GUI. This allows part of the visualization system to be integrated into other applications – even if they use different rendering and GUI APIs. This system is comprised of four sub-systems.

7.2.1.1 Geometric Flow Visualization Subsystem

Figure 7.4 illustrates the processing pipeline for the geometric flow visualization subsystem. Input and output data is shown using rectangles with rounded corners. Processes are shown in boxes.

The geometric-based visualization subsystem uses the simulation data as its main input. After the user has set a range of integration parameters and specified the seeding conditions the initial seeding positions are created. Numerical integration is then performed to construct the geometry by tracing vertices through the vector field. This is an iterative process with which an optional refinement stage may be undertaken depending on the visualization method. For example, when using streamsurfaces, extra vertices need to be inserted into the mesh to ensure sufficient sampling of the vector field. Afterwards the object geometry is output. The penultimate stage takes the user-defined parameters that direct the rendering result. Most of the implemented algorithms in our application reside within this sub-system.

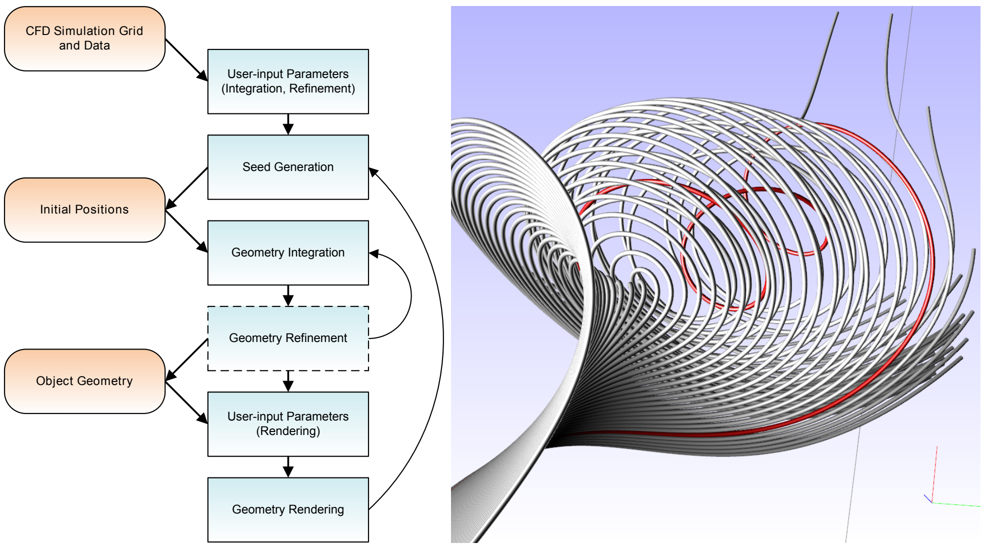

CFD Simulation Grid and Data
User-input Parameters (Integration, Refinement)
Seed Generation
Initial Positions
Geometry Integration
Geometry Refinement
Object Geometry
User-input Parameters (Rendering)
Geometry Rendering

Figure 7.4: *(Left) The processing pipeline for the geometric flow visualization subsystem. (Right) A set of streamlines generated by the geometric flow visualization subsystem. The streamlines are rendered as tube structures to enhance depth perception and provide a more aesthetically appealing result. The visualization depicts interesting vortical behavior in a simulation of Arnold-Beltrami-Childress flow [Hal05].*

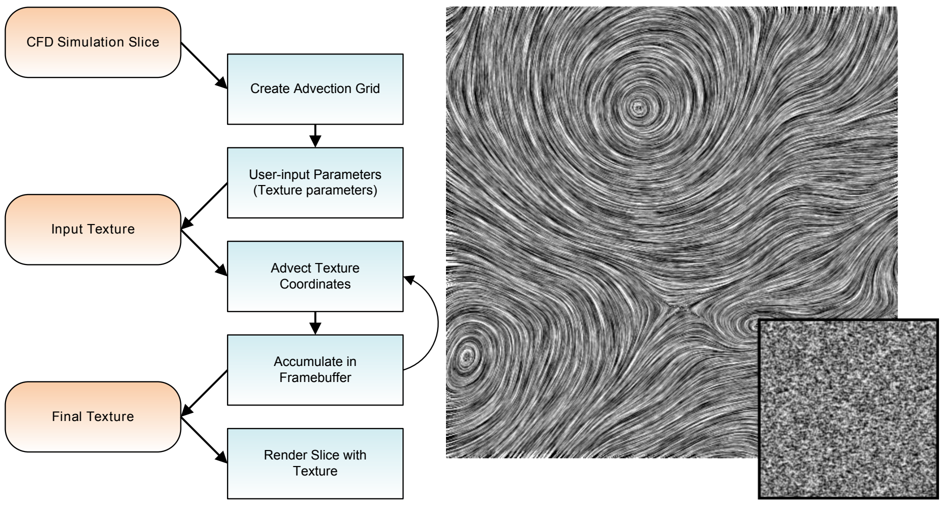

CFD Simulation Slice
Create Advection Grid
User-input Parameters (Texture parameters)
Input Texture
Advect Texture Coordinates
Accumulate in Framebuffer
Final Texture
Render Slice with Texture

Figure 7.5: *(Left) The processing pipeline for the texture-based visualization subsystem. (Right) A Line Integral Convolution (LIC) visualization using the texture-based visualization system. This image was generated by 'smearing' the noise texture (inset) along the direction of the underlying vector field at each pixel. The visualization is of a simulation of Hurricane Isabel. The eye of the hurricane can be seen towards the top of the image.*

7.2.1.2 Texture-based Visualization Subsystem

The texture-based visualization process also takes in the simulation data as input. An advection grid (used to warp the texture) is then set up and user-parameters are specified. An input noise-texture (Figure 7.5 inset) is then ‘smeared’ along the underlying velocity field – depicting the tangent information. The texture advection is performed as an iterative process of integrating the noise texture coordinated through the vector field and accumulating the results after each integration. The resultant texture is then mapped onto a polygon to display the final visualization.

7.2.1.3 Direct Flow Visualization Subsystem

The direct visualization sub-system presents the simplest algorithms. Typical techniques are direct color-coding and glyph plots. The left image of Figure 7.6 shows a basic glyph plot of a simulation of Hurricane Isabel. The right image includes a direct color-mapping of a saliency field showing local regions where a larger change in streamline geometry occurs.

7.2.1.4 Feature-based Flow Visualization Subsystem

Feature-based algorithms may involve a lot of processing to analyze entire the simulation domain. There exists many types of feature that may be extracted (such as vortices), and each feature has a variety of algorithms to detect/extract them. In our application we implemented extraction of critical points (positions at which the velocity diminishes). The right image of Figure 7.6 shows a set of critical points extracted on a synthetic data set. A red highlight indicates a source or sink exists in the cell and a blue highlight indicates that a saddle point is present in the cell.

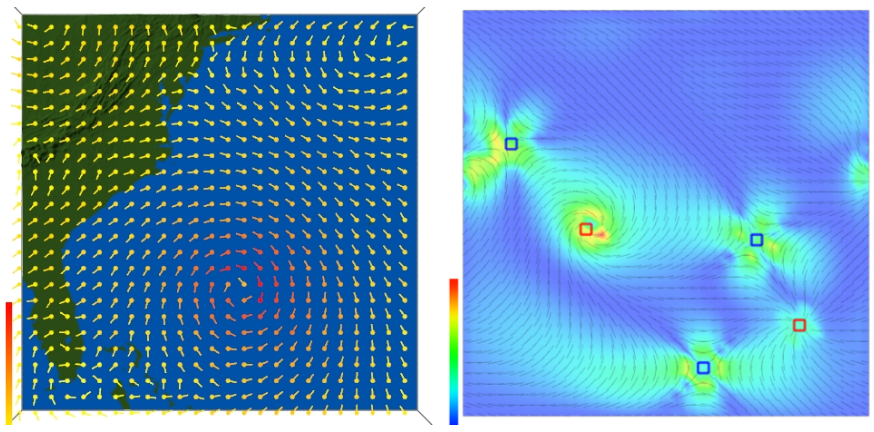


Figure 7.6: *Direct and feature-based visualizations. The left image shows a basic glyph plot of the velocity field of a simulation of Hurricane Isabel. The right image shows the critical points extracted on a synthetic data set. The cells that contain the critical points are highlighted. A red highlight indicates the critical point is a source or a sink and a blue highlight indicates a saddle point. This visualization also contains a direct color-mapping of a saliency field based on local changes in streamline geometry.*

7.2.2 Graphical User Interface and Asset Management

The perfect visualization tool does not (yet) exist. Each piece of research that has been undertaken over the past several decades focuses on a specific problem. Thus, a general solution that is suitable for all visualization problems has not been discovered – and may never be found. To this end, visualization applications must support a variety of techniques in order to be useful. When referring to a visualization tool/technique in terms of our software, we refer to them as *assets*. Our asset management system is designed with the following requirements:

- A common interface for assets, simplifying the process of adding new assets in the future and ensuring the application is extendable.
- A common interface between assets and the application GUI. Again this simplifies expansion in the future and ensures a basic level functionality is guaranteed to be implemented. This also provides a consistent user interface for the user.
- Enforcing the re-use of existing code.
- The same method of adding assets for the visualization at run-time.

Fortunately the object-oriented programming paradigm and the C++ programming language provides us with a powerful set of tools to realize these requirements. The rest of this section discusses aspects of the GUI design and our framework for managing the visualization assets.

7.2.2.1 Application Tree and Scene Graph

In order to provide a flexible system, that allows the user to interactively add and remove assets at run-time, we utilize a scene graph. A scene graph is a tree data structure in which all assets are represented by nodes within the tree. When a frame is rendered, a pre-order, depth-first traversal of the tree is carried out, starting from the root node. As each node is visited, it is sent to the rendering pipeline. Transformations applied to a node are passed onto its children. We provide two node types: *Asset Nodes* and *Camera Nodes*. These are derived from a base node which provides a common interface and are not directly instantiable. The inheritance diagram for the node types is shown in Figure 7.7.

The tree structure used for the scene graph lends itself to be represented using a GUI tree control (see Figure 7.8). The tree control directly depicts all of the nodes in the scene graph and the tree hierarchy. The user manipulates the scene graph through the tree control. Assets can be added to the scene graph by selecting a node to which an asset is attached. Right-clicking upon an asset presents a context menu with an option to add a new node into the scene graph (see Figure 7.9). Following this option another context menu is presented with a variety of assets which the user is able to add. When an asset is selected to be added, it is inserted into the scene graph as a child node of the currently selected node (the node which was right-clicked). Removal of a node is achieved using a similar method – the right-click context menu gives the option of removing a node. When a node is removed from the scene graph all of its children are also removed. This ensures that there are no dangling pointers and acquired resources are freed. The *resource acquisition is initialization* (RAII) [Mey05] programming idiom is obeyed throughout the application to ensure exception safe code and resources are deallocated.

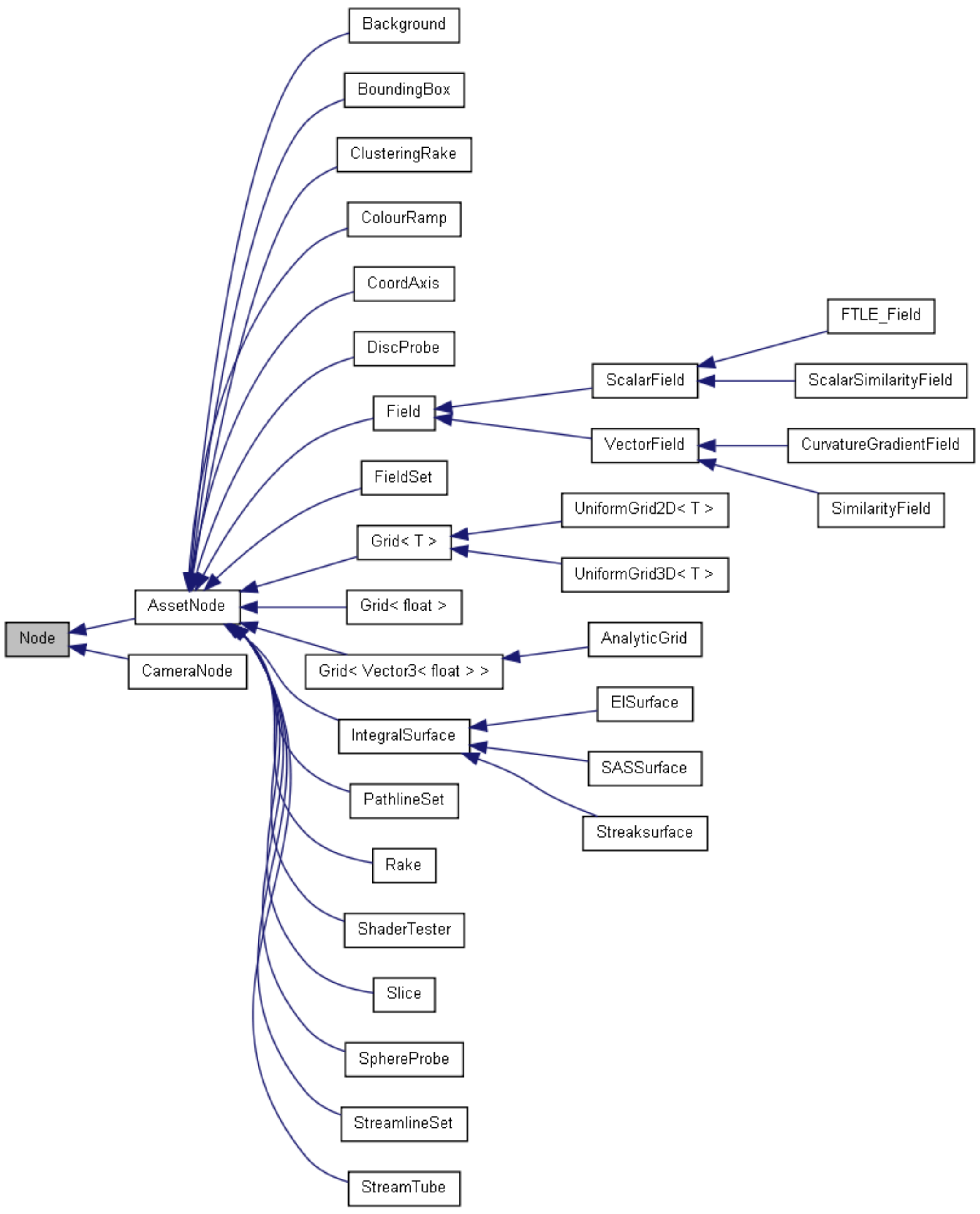


Figure 7.7: *Inheritance diagram for the node classes. The asset node is the interface from which all integrated visualization techniques inherit and implement.*

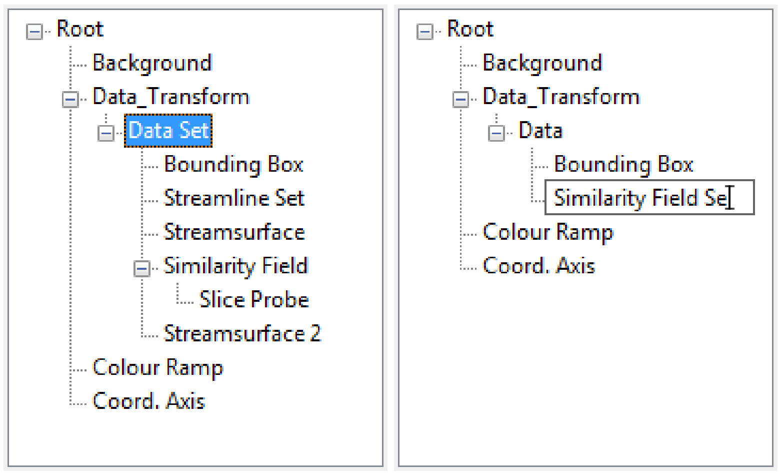


Figure 7.8: *Screenshots of the application tree during the run-time of different sessions. The application tree is a GUI tree control that represents the nodes in the scene graph. (Left) Several visualization assets, such as streamline sets and slice probes, are currently being employed. (Right) The user is editing the label of one of the assets.*

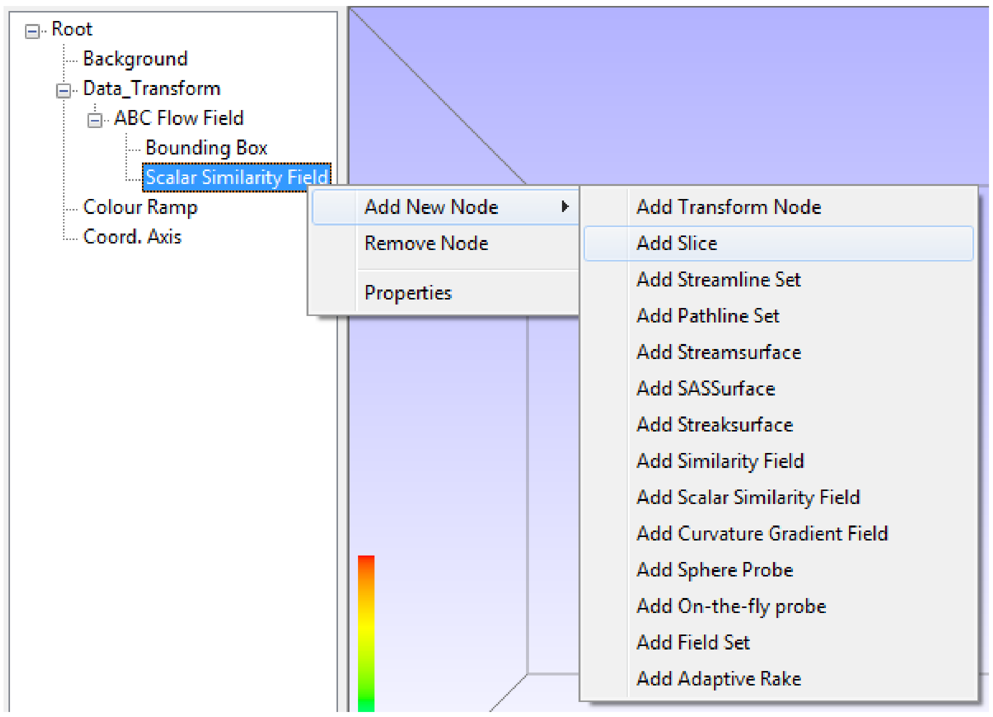


Figure 7.9: *The tree control is used to add new nodes into the scene graph. The user selects which node they want add an asset to. A context menu then presents the user with a list of assets. When an asset is selected it is added to the scene graph as a child node of the currently selected node.*

From a user perspective, this system allows a flexible method with which to interactively add and remove the visualization tools at run-time. The current tool set is always displayed to provide fast and easy access. From a developer perspective, this system provides a consistent interface. The logic for adding and removing a node is maintained in the scene graph, application tree, and node classes. It does not need implementing on a per-asset basis. When a new visualization technique is implemented, all that is required is that the developer inherits from the asset node class and provides the implementation for the pure virtual functions described by the abstract asset node class (described in more detail in Section 7.2.2.3). In addition to the asset node, we provide a class called *camera node* which is responsible for storing and configuring the projection and viewpoint information. We now discuss the camera node and the asset node classes in more detail.

7.2.2.2 Camera Node

3D APIs such as OpenGL and DirectX have no concept of a camera. The viewpoint is always located at the position $(0.0, 0.0, 0.0)$ in eye-space coordinates (for a thorough discussion of coordinate spaces and the OpenGL pipeline we refer the reader to [WNDS07]). However, the concept of a camera navigating through a 3D scene provides an intuitive description. We can give the appearance of a movable camera by moving the scene by the inverse of the desired camera transformation. For example, to simulate the effect that the camera is panning upwards, we simply move the entire scene downwards.

As outlined in Section 7.2.2.1, all child nodes inherit the transformations of their parent. The camera node is set as the root node in the scene graph. The inverse transformation matrix is re-computed when the camera is manipulated. All other nodes are added as a descendant of the camera node and are, therefore, transformed by its transformation matrix. Thus, the camera parameters are the main factor for setting the viewpoint and orientation. This is in line with the camera analogy described at the beginning of this section.

This method can be extended to render to multiple viewports with different view points. This could be realized by maintaining a list of camera nodes, each maintaining their own set of view parameters (like using multiple cameras in real-life). For each view point, the relevant camera node can be inserted into the root node position. The scene graph is then traversed sending each node to the rendering pipeline. This allows the same scene graph to be used, the only change is the initial camera transform.

7.2.2.3 Asset Node

Figure 7.10 shows the collaboration graph for the asset node class. This class is designed to provide a consistent interface for all visualization methods integrated into the application. It is an abstract class and therefore provides an interface that declares common functionality. The class provides three pure virtual function signatures:

- SendToRenderer()
- setDefaultMaterial()
- loadDefaultConfiguration()

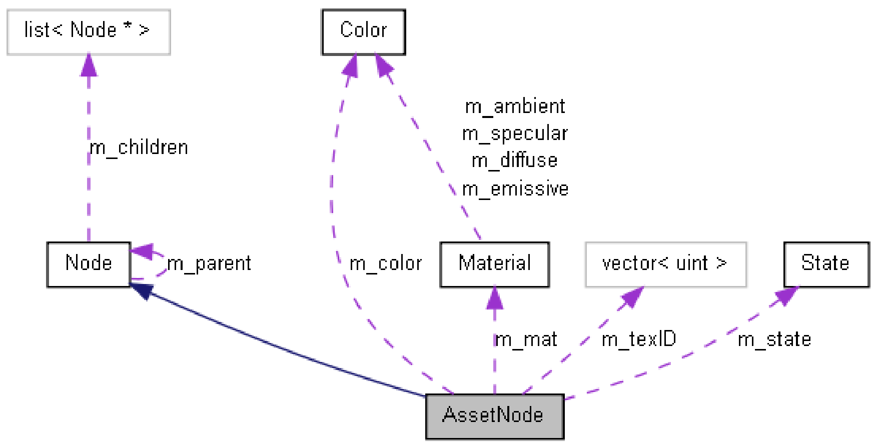


Figure 7.10: *Collaboration diagram for the asset node class. The boxes represent classes in our framework.*

The SendToRenderer() function issues a command to the class to send its geometry to the 3D viewer. The setDefaultMaterial() function is an initialization function that sets the initial/default material properties that are used by the OpenGL API. Finally the loadDefaultConfiguration() function loads the default set of parameters for the visualization method from file. The configuration files follow the INI file format [INI]. This function is provided to ensure that all visualization methods are loaded with sensible default values (where necessary). By providing the configuration information in a file and not hard-coding it into the application brings several benefits. A change to default parameters does not result in any re-compilation bringing speed benefits during development. It also means that the end user can change the default settings without having to have or understand the source code. It also allows users on different machines to each have their own set of default parameters tailored to their requirements. It would be a simple task to allow per-user configuration files on a single machine, however we have not implemented this functionality as it is superfluous to our requirements as a research platform.

The asset node class also provides several member variables that are inherited:

- (unsigned int) m_vboID
- (unsigned int) m_indexID
- (vector⟨unsigned int⟩) m_texID
- (Material) m_mat
- (Color) m_color
- (State) m_state
- (bool) m_inited

OpenGL assigns numeric ID's to all buffers. Asset nodes provide variables to store a vertex buffer (m_vboID), an index buffer (m_indexID) and a list of textures (m_texID). More than one texture can be assigned to an asset node in order to facilitate multi-texturing. Materials are settings that affect how geometry reflects to the light within OpenGL. A material is separated into several components: *ambient, diffuse, specular* and *emissive*. The asset node provides all renderable objects with a material property. It also provides a color property, this is used in a similar fashion to material but it much more lightweight with less flexibility. OpenGL is a state machine, where the current state affects how any primitives passed to it are rendered. Whether lighting and/or texturing is enabled are examples of some of the states used by OpenGL [ARB00]. Every asset node has a state member variable which allows the node to stores various OpenGL state settings plus other non-OpenGL state parameters. The state class is described in more detail in Section 7.2.3.2.

7.2.2.4 Asset User-interaction and the Asset Control Pane

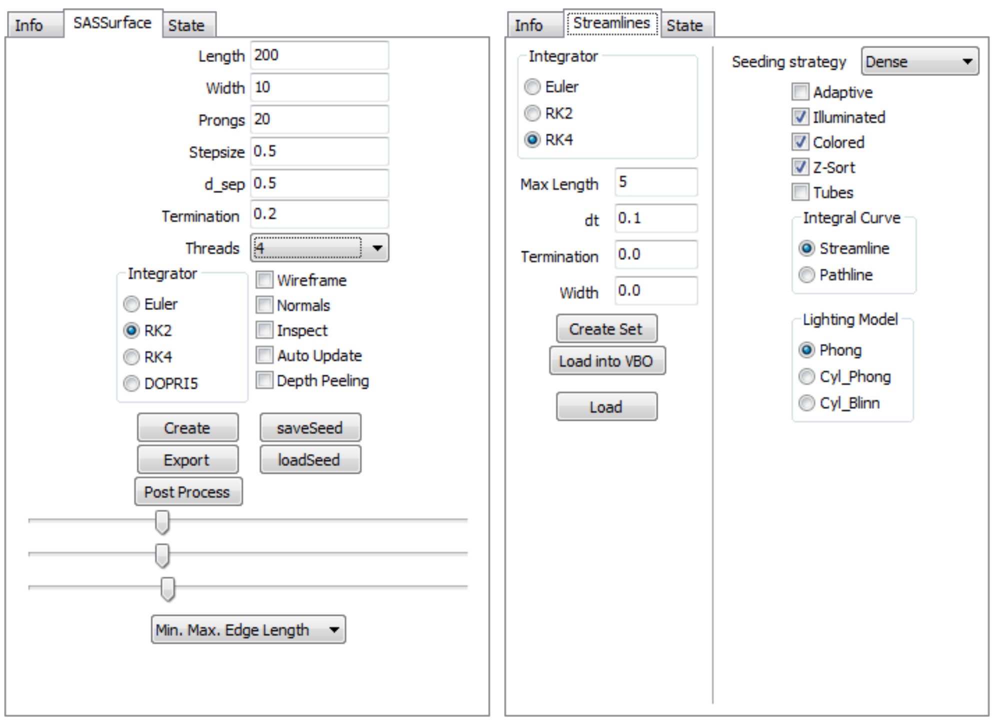
User specified parameters for the various visualization assets are provided through the asset control pane. When an asset is selected in the application tree control (Section 7.2.2.1), the asset control pane is updated. The asset control pane shows only the controls for the currently selected asset. This helps reduce clutter in the GUI and provides an easier experience for the user. The asset control panel also populates the controls with the current values of the asset, therefore the GUI always represents the correct values for the selected asset. The asset panel can be seen in the blue box of Figure 7.1.

The use of C++ pure virtual functions ensures that the GUI panels for all visualization assets must implement functionality to update itself according to the current state of the active asset it is controlling. The GUI panels are now discussed in more detail.

7.2.2.5 Asset Panels

Figure 7.11 shows an examples of the asset panel at runtime. The left panel shows the controls displayed when a streamsurface asset is selected by the user. The right image shows the asset panel when the user has selected a different visualization asset, in this case a streamline set. Note how the streamsurface panel has now been removed and it is replaced with the streamline panel. Other relevant controls for the selected tool (such as state parameters) are neatly set in separate tabs. This has two benefits. It keeps the visualization tool parameters and the OpenGL rendering parameters for the tool separate. We can also re-use the same GUI panel for state controls as the parameters are common across all visualization methods.

Asset panel controls are event-driven. When a control is modified an event is fired which is then handled by the visualization system. The event handler typically obtains an handle to the currently selected visualization asset and calls the correct function. The visualization system is then updated and feedback is presented to the user. Asset panels utilize multiple inheritance. While multiple inheritance has its shortcomings, i.e., the diamond problem, but can provide powerful solutions to problems if used with care. Figure 7.12 shows the inheritance diagram for a typical asset panel (in this case the streamsurface panel). Note that only a single level of inheritance is used. Throughout the design of this system, keeping the inheritance levels as low as possible was set out as a requirement. This ensures shall depth of inheritance trees (DIT) which makes the code easier to extend, test, and maintain. All asset panels inherit from two



Length 200
Width 10
Prongs 20
Stepsize 0.5
d_sep 0.5
Termination 0.2
Threads 4
Integrator: Euler, [✓ RK2], RK4, DOPRI5
☐ Wireframe
☐ Normals
☐ Inspect
☐ Auto Update
☐ Depth Peeling

Integrator: Euler, RK2, [✓ RK4]
Max Length 5
dt 0.1
Termination 0.0
Width 0.0
Seeding strategy Dense
☐ Adaptive
✓ Illuminated
✓ Colored
✓ Z-Sort
☐ Tubes
Integral Curve: [✓ Streamline], Pathline
Lighting Model: [✓ Phong], Cyl_Phong, Cyl_Blinn

Figure 7.11: *Two examples of asset panels taken at runtime. The panel on the left shows the controls for streamsurfaces. When a streamsurface asset is selected in the application tree, this panel is inserted into the asset control pane. The right image shows the result of the user then selecting a streamline set asset. The streamsurface control panel is removed from the asset control pane and the streamline set control pane is inserted in its place. Only the relevant controls to the currently selected asset are displayed to the user. This leads to a less cluttered GUI and the user is not burdened with manually navigating the GUI to find the appropriate controls.*

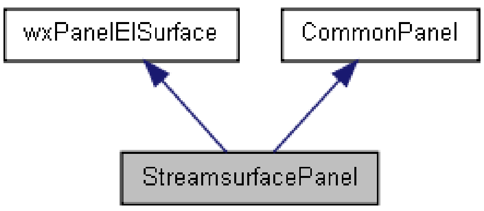


Figure 7.12: *Inheritance diagram for asset panel types. This example shows the streamsurface panel. Asset panels use multiple inheritance, they inherit from* CommonPanel *and another class that is auto-generated using a GUI builder. Using this method provides fast creation of GUI controls (using the form builder and generated class) and allows us to provide a common interface and behavior for all panels (using the common panel class).*

base classes. One of these classes is unique to all derived classes and the other one is common to all derived classes. The class *CommonPanel*, as its name implies, is inherited by all asset panels. It contains information such as the string that is displayed when the panel is shown in the asset control pane and enumeration of the panel type. It also provides the signature for a pure virtual function, $UpdatePanel()$. This function is used to populate the panels controls with the correct values (by querying the currently selected asset). The second class panels inherit from are unique auto-generated classes that are output from using a GUI building tool called wxFormBuilder. The auto-generated classes provide panel layout and controls. They also provide interface for the events that are fired from that panel. The asset panel then provides the implementation for the interface. In our system the auto-generated classes are prefixed with the letters "wx" to differentiate them from user created classes.

The asset panels are designed with both developers and users in mind. The updating panel in the asset control pane ensures that only the relevant controls are displayed. The controls are also located in the same place within the application. Therefore, the user does not have to search around for various options. Similar to the node structures (Section 7.2.2.1), the panels are organized in a manner that facilitates easier implementation that ensures a certain level of functionality. The use of a GUI builder greatly facilitates the developer increasing the productivity when creating GUI components.

7.2.3 3D Viewer

This section details the 3D viewer system of our application. We discuss some key implementation details and outline how our application manages various rendering attributes such as materials and textures.

The 3D viewer system is implemented using the OpenGL API. The OpenGL API was chosen because it provides a high level interface for utilizing graphics hardware and it is platform independent. The 3D viewer is responsible for providing the feedback from the visualization software. Recall that OpenGL defines a state machine whose current rendering state affects how the primitives that are passed through the graphics pipeline are rendered. State-machines can make debugging difficult; unexpected behavior may arise simply from a state being changed that the developer may not have intended. Querying the current state may be difficult at times and almost always relies on dumping text to a console window or file. To try and alleviate this issue our system implements a wrapper around for OpenGL state machine. Our *OGL_Renderer* (OpenGL Renderer) class provides flags for the OpenGL states used within our system. Other states may be added as they are added and utilized by the system. We also provide accessor and mutator functions for retrieving and manipulating state values. Our wrapper provides several benefits:

- Breakpoints may be set to halt the program when a specific state value has been modified.
- Bounds checking may be performed as states as a sanity check, making sure no invalid values are set.
- When using an integrated development environment (IDE), the class can be queried easily and does not rely on the outputting of large volumes of text that the user has to manually search through.

- Some OpenGL code can be simplified making development easier and more efficient.
- Separating the graphics API code allows for other APIs to be used in the future if the requirement arises. This is very difficult if API specific code is embedded throughout the entire codebase.
- It aids the developer by being able to focus more on the visualization algorithms rather than the rendering component. Thus, promoting the system as a research platform.

Our system only requires a single rendering context (if multiple viewports are present, the same rendering context can be used). We utilize the Singleton design pattern [GHJV94] so that instantiation of the OGL_Renderer is restricted to a single instance. We note that a singleton has downsides as it is in essence a global variable. However, the OpenGL state machine is inherently global and the fact we only want a single rendering context makes a singleton suitable for our needs. In our case, a singleton provides a much cleaner solution than continually passing references around is much preferably than every object (that needs to) storing its own reference to the renderer object. Access to the singleton is provided by using the following C++ public static function:

```
static OGL_Renderer& OGL_Renderer::Instance()
{
    static OGL_Renderer instance;
    return instance;
}
```

The first time that this function is called, an instance of the OGL_Renderer is created and the reference to it is returned. Future calls to this function do not create a new instance (due to the static variable) and a reference to the current instance is returned.

7.2.3.1 Rendering

OpenGL rendering code can be ugly and cumbersome if not carefully designed. The API uses C-style syntax which does not necessarily interleave itself well with C++ code in terms of code readability. Many calls are usually made to set the OpenGL state before sending the geometry data along the rendering pipeline. Here is an example of OpenGL code that renders a set of vertices that are already stored in a vertex buffer on the GPU.

```
...
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(Vector3<float>), NULL);

glBindBuffer(GL_ARRAY_BUFFER, normalBufferID);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, sizeof(Vector3<float>), NULL);

glBindBuffer(GL_ARRAY_BUFFER, textureBufferID);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(1, GL_FLOAT, sizeof(float), NULL);
```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexId);

glDrawElements(GL_TRIANGLE_STRIP, numVerts,
               GL_UNSIGNED_INT, NULL);

glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, NULL);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, NULL);
...
```

This renders an indexed set of vertices as a strip of triangles with shading and texturing information. First the buffers and pointers into them are set as well. The vertices are then passed down to the rendering pipeline. The state changes are undone after rendering to put the OpenGL back into its original state. It is clear this is not the simplest code to work with. If the rendering code was merged into the visualization code, all renderable objects would possess similar code chunks. This (1) makes the code harder to read and (2) produces a lot of repetitive code throughout the codebase.

Our system segregates this type of rendering code. We provide classes such as *TriangleRenderer* and *LineRenderer* which contain utility functions that simplify the rendering process. A typical usage of the triangle renderer is shown below.

```
...
TriangleRenderer::RenderTriangle_VBO(m_vboID,
                                     m_indexID,
                                     m_numberOfIndices,
                                     TRIANGLE_STRIP);
...
```

This call to the RenderTriangle_VBO function passes in the required buffers, the number of vertices to be rendered and the rendering mode. This approach allows the developer to take advantage of code re-use and makes the code much more readable.

7.2.3.2 State Objects

We provide a *State* class that encapsulates various OpenGL states that are utilized by our visualization assets. The state class has the following members:

- (bool) m_lighting;
- (bool) m_texturing;
- (bool) m_blend;
- (uint) m_program;
- (int) m_stateBlendSrc;

- (int) m_stateBlendDst;
- (bool) m_render;

The first three bool members are flags indicating whether the matching OpenGL state will be enabled. The m_program member is the ID of the shader program that is used to render the asset. The blend members store the blending states when blending is enabled. The final member, m_render, indicates whether the asset is rendered or ignored. This member has no counterpart in the OpenGL state machine. It is included to allow the user to disable the rendering an asset without removing it from the scene graph. The state class has a member function, *SetState()*, which is called immediately before the asset is rendered.

```
void State::SetState()
{
    OGL_Renderer& renderer = OGL_Renderer::Instance();

    if(m_lighting)   renderer.Enable(LIGHTING);
    if(m_texturing)  renderer.Enable(TEXTURING);
    if(m_blend)
    {
        renderer.Enable(BLEND);
        renderer.SetBlendFunc(m_stateBlendSrc, m_stateBlendDst);
    }
    renderer.UseProgram(m_program);
}
```

After the asset has been rendered the OpenGL state is returned to its original state by calling the *UnsetState()* function of the state.

```
void State::UnsetState()
{
    OGL_Renderer& renderer = OGL_Renderer::Instance();

    if(m_lighting)   renderer.Disable(LIGHTING);
    if(m_texturing)  renderer.Disable(TEXTURING);
    if(m_blend)      renderer.Disable(BLEND);
    renderer.UseProgram(NULL);
}
```

These functions greatly simplify the rendering code and aid the developer in efficiently managing the OpenGL state machine and reduce unexpected behavior arising from incorrectly configures states. The state objects are utilized every time an asset is rendered. The code segment below outlines there usage within our application framework.

```
void AssetType::SendToRenderer()
{
    if(m_state.RenderingEnabled())
    {
        m_state.SetState();
        //Rendering Code
```

```
            ...
            m_state.UnsetState();
        }
    }
```

7.2.3.3 Material Objects and Lights

Within OpenGL (and other rendering APIs), the currently bound material state affects how primitives that are passed to down the rendering pipeline interact with light sources. OpenGL lighting is comprised of various terms that approximate the behavior of light in the real-world. In computer graphics and visualization the aesthetics of the final rendering result are very important for high quality results. A research paper looks more polished and professional with high quality images. We recognize this importance and provide functionality that allows the user to adjust the various lighting and material parameters at run-time.

We allow the user to interactively add and remove light sources. The type of light source and its position can also be controlled by the user. The application also allows the user to set the values of each component of the light source (ambient, diffuse and specular). Likewise with materials we allow the user to adjust each component (ambient, diffuse, specular, emission and specular power). Allowing this level of control at run-time allows the user to receive immediate feedback of the results and prevents any unnecessary recompilation and re-generation of results.

Each asset has its material object which encapsulates the state behavior. Prior to the asset being rendered its material is bound by the OpenGL state machine. Once again, by separating the rendering code from the visualization asset code we are promoting code-reuse and not cluttering up the visualization asset classes with rendering code.

Note we omit a thorough discussion of how OpenGL approximates lighting and materials. Instead we refer the interested reader to [WNDS07].

7.2.3.4 Textures, Texture Management and Texture Editing

As we have previously discussed, our application has served as a research platform for flow visualization techniques. More specifically we have focused on a sub-set of flow visualization techniques that fall into the geometry-based category. These methods compute a geometry that represents some behavior of a flow field. However, by color-mapping this geometry we can depict more information about the flow behavior than the geometry alone. For example, velocity magnitude is often mapped to color.

Color-mapping can be achieved in a variety of ways. A function may be provided that maps the color, although for complex mappings defining a suitable function may be difficult. A large lookup table may be produced, this is a flexible solution but can lead to the developer producing lots of code to produce large look up tables.

Our approach to color-mapping utilizes texture-mapping. Here the texture itself is the lookup table and all we have to do is provide the texture coordinate to retrieve the desired value from the texture. Textures are a very powerful tool in computer graphics and rendering APIs readily provide functionality for various interpolation schemes which we can utilize.

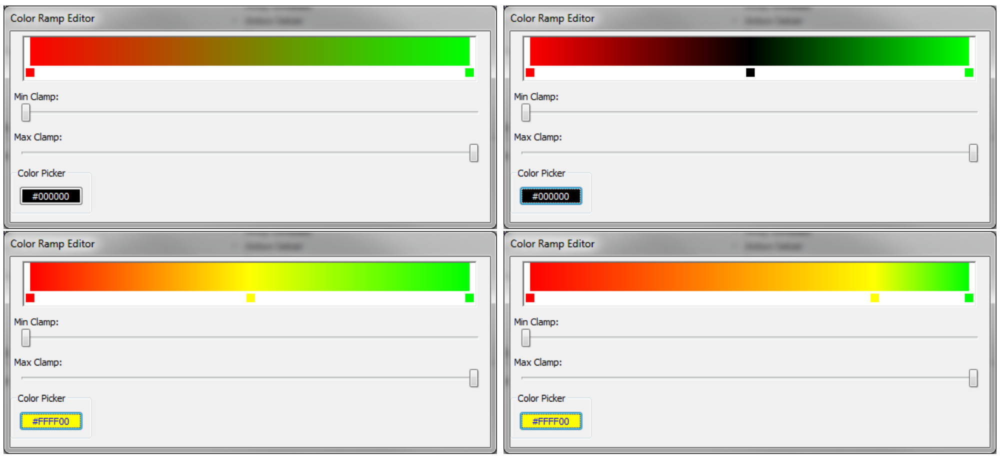


Figure 7.13: *Some images from an interactive session with the color map editor. The top-left image shows the initial state of the editor. The top right image shows the result when the user inserts a new sample (black) in the center of the color map. The bottom-left image shows the result after the user has updated the color of the middle sample to yellow. Finally the bottom-right image shows the effect that dragging the middle sample to right has. The color values between each sample are constructed using interpolation.*

They are also fast as they due to their hardware support. This system is also very flexible, new color-maps (in the form of images) can be dropped into the textures folder of the application and they will automatically be loaded the next time the application is run. Management of the texture is equally simple, the texture manager simply maintains a list of textures, the user can select the texture they wish to use from the GUI and the texture manager binds that texture to the OpenGL state.

We also provide a tool that allows the user to create their own color maps. This allows the user to customize the color-mapping at run-time to ensure that the mapping adequately represents the information they wish to present. Figure 7.13 shows some steps of an interactive session with the editor. The editor allows the user to insert (and remove) samples along the color map. The color of the samples can be altered and the position of the sample can be updated by dragging it around in the editor window. The colors values are interpolated between samples. An up-to-date preview of the color map is always displayed within the editor.

7.2.4 Simulation Manager

The final major system in our application is the simulation manager. The simulation manager is responsible for loading the simulation data and managing the sub-sets of simulation data when it won't fit in core memory. The simulation provides a set of classes for 2D and 3D simulations. The simulation manager handles both discretely sampled data, such as the output from CFD simulations, and analytically defined data by providing the necessary parameters to a function to compute the vector information. Flow simulations are output in a variety of file formats using both ASCII and binary file output. Our application supports a range of formats and provides a simple interface for developers to add support for more formats in future.

The simulation manager is used whenever a vector field evaluation is requested by one of the visualization assets. It is responsible for determining whether a given position lies within the domain (both spatially and temporally). If the position is determined as valid, the simulation manager populates a cell object (of the corresponding grid type) with the appropriate vector values. The cell objects also belong to the simulation manager and are used to construct the final vector value at the desired position using interpolation.

7.2.4.1 Large-Time Dependent Simulation Data

As previously discussed, the output from time-dependent CFD simulations can be of the order of gigabytes or even terabytes. Thus, we have to consider out-of-core methods. Our application handles such large amounts of data by only loading a sub-set of the simulation into memory. In order to perform a single integration step, only two time-steps need to be copied to main memory. For example if each our simulation output data for every second and we need to advect a particle at $t = 3.5s$, only time-steps 3 and 4 four are needed to interpolate the required vector values.

We employ a method similar to Bürger et al. [BSK$^+$07]. We allocate a number of slots equal to the number of time-steps that fit into main memory. These slots are then populated with the data from a single time-step, starting with the first time-step and proceeding consecutively. For example, if we can fit 6 time-steps into memory we allocate 6 slots and populate them with time-steps 0-5. When we have passed through a time-step its data is unloaded and the next time-step is loaded from file in its place. For example, if we are constructing a pathline, when $t \geq 1$, the first slot (which holds the data for time-step 0) is overwritten with the data for time-step 6 – the next unloaded time-step in the simulation. Figure 7.14 illustrates an example.

Time-step	0	1	2	3	4	5
Slot	0	1	2	3	4	5

(a) $0 \leq t < 1$

Time-step	6	1	2	3	4	5
Slot	0	1	2	3	4	5

(b) $1 \leq t < 2$

Time-step	6	7	8	9	10	5
Slot	0	1	2	3	4	5

(c) $5 \leq t < 6$

Time-step	6	7	8	9	10	11
Slot	0	1	2	3	4	5

(d) $6 \leq t < 7$

Figure 7.14: *These four tables show the time-steps that are loaded into the simulation manager slots for given time periods. The gray cells show the time-steps that are used to perform any vector field evaluations for the stated time period. (a) Shows the first time period ($0 \leq t < 1$). (b) shows the next time period, the two slots used in the vector field evaluation have moved over – a sliding window. The previous slot has been updated with the subsequent unloaded time-step in the simulation (slot 0 is loaded with time-step 6). (c) The slots wrap around, when the sliding window reaches the last slot it switched back to the first slot. (d) The process repeats with the new time-steps in the slots.*

Conceptually, a sliding window run over the slots, with the pair of slot covered by the window being used for the current vector field evaluations. When the sliding window has passed a slot, the slot is updated with the next unloaded time-step. When the sliding window reaches the last slot it wraps around to the first slot and the cycle is repeated. The sliding window transition is triggered when a time greater than the current time period covered by the window is requested by the application.

For this method to be effective the simulation manager runs in a separate thread. Disk transfer operations are blocking calls and they halt the rest of the application if a single thread is used. Moving these blocking calls to a separate thread allows the application to proceed computing visualization results while data is loaded in the background. Note, there may be times where the visualization results are computed faster than the simulation manager can load the data. If the required time-steps are not present in memory the application has no option but to halt until they have been loaded. However, even in this case the multi-threaded simulation manager reduces the number and duration of halts compared to a single-threaded solution.

Another consideration that needs to be considered is how the visualization assets are constructed. If we were to generate 10 pathlines by computing the first pathline and then the second one and so on, the simulation manager would have to load all time-steps 10 times (one for each pathline). It is much more efficient to construct all pathlines simultaneously by iterating over them and computing successive points. This ensures that they all require the same sliding window position in the simulation slots and prevents unnecessary paging of data.

7.3 Conclusion

In a typical research paper many implementation details have to be omitted due to space restraints. It is rare to see literature that provides an in-depth discussion concerning the implementation of an entire visualization application. This chapter serves to provide such a discussion. The chapter provides an overview of the high-level application structure and provides details of key systems and classes and the reasoning why these were designed in this way. Many topics are covered ranging from multi-threaded data management for performance gains to GUI design and implementation with considerations both the developer and the user.

We demonstrate that using a good software engineering practices and design methodologies provide an enhanced experience for both software developers and end-users of the software. This serves as proof that research code is not restricted to small ‘one-off’ applications and that implementing proof-of-concept algorithms into a larger framework has many benefits – not least of which is an easier comparison to other techniques.

8 Conclusions

“Try to name a true success story that involves someone giving up before the job was done.”
-Wess Roberts[1]

Contents

THE state-of-the-art in flow visualization is the culmination of over two decades of research. During this time there have been several ‘hot-topics’, FTLE being a current one. Sometimes a particular branch receives little attention for a while and suddenly gains a resurgence of interest. Texture-based methods are one such example; the seminal Line Integral Convolution [CL93] (LIC) algorithm was introduced in 1993. A quiet period in respect to texture-based methods then followed. Through the early to mid 2000s, interest in this branch was re-ignited with a greater number of publications arising in this area [LHD$^+$04].

The use of surfaces for the visualization of flow fields has also recently gained increased interest. The first stream surface algorithm was introduced by Hultquist [Hul92] in 1992. Over the next 16 years only a handful of papers were published regarding this topic. In 2008, a renaissance of surface-based methods arose. Various stream surface construction methods have since been introduced. The use of surfaces has also propagated to include unsteady flow data. In 2008, we have seen path surfaces become more common and the first step towards streak surfaces was undertaken [vFWS$^+$08]. Since then, complete streak surface algorithms have been introduced [MLP$^+$10].

Can we identify what motivates these trends and is it possible to predict future ‘hot topics’? There are several potential driving forces behind these trends, not the least of which is computing hardware. Texture-based techniques may have been driven by the growing popularity of hardware accelerated graphics with increased GPU memory and texture units – allowing for more advanced texture-based methods. Similarly, the popularity of surfaces may be driven by the increased computational power. Streak surfaces are an inherently expensive technique. The sheer number of particle integrations alone amounts to a vast computational workload for even moderate-sized surfaces. However, faster, more powerful processors in addition to the growing

[1]A best-selling American author famed for his business leadership books.

number of CPU cores have put these techniques within the reach of visualization scientists. On older hardware, streak surfaces could have taken of the order of days to compute. This makes their use unfeasible. A small error in input parameters could result in a large waste of computational resources. It also leads to problems with their development and debugging. On modern hardware, streak surfaces can be produced in either hours or mere minutes. Utilizing the highly parallel architecture can reduce this even further, producing consistent, interactive rates in a variety of cases. As to whether we can predict future trends, of course this is impossible. At times, it is clear where a gap in research literature is present, however, that does not necessarily mean that these problems can be addressed now or in the very near future. One thing that is certain, future increases in computing power paired with the enthusiasm and talent of the visualization research community will result in many exciting contributions in the future.

8.1 Future Work

Interactive visualization of unsteady flow is still regarded as a grand challenge of flow visualization [BSK+07]. CFD simulations already output a large quantity of data. However, CFD is also an active field of research. The result of this is multiple runs of massive simulations to test the effect of varying parameters and higher fidelity simulations that simulate more complex behavior. This increases the burden on an already difficult area – coping with the sheer volume of data. This trend is likely to continue for the foreseeable future and it is the responsibility of visualization scientists to tackle this problem. However, the enormity of this challenge means that it may never be solved (or at the very least, in the near future). In fact, some of the very tools that visualization scientists may leverage may even work against them. For example, harnessing the power of GPUs allows the visualization community to deliver more complex visualization algorithms at interactive rates. However, the processing power of GPUs may also be (and is) used in CFD simulations, resulting in data being produced at an ever increasing rate.

Whilst we cannot provide a complete solution to this ongoing problem, we can provide directions in regards to our own work that may serve as sound starting point and direction. First of all, while several algorithms have been created in regards to streak surfaces, we still feel there is room for improvement. Ultimately, a more robust GPU implementation will be introduced. A combination of utilizing the power of the GPU with the flexibility of the CPU is the 'holy grail' in this area. Again, due to advances in hardware, this is becoming a reality. Current generation GPUs have significantly larger memory than a handful of years ago. It is commonplace to see GPUs with over 1GB of memory nowadays. This allows more time-steps to be stored in memory simultaneously. Used in conjunction with techniques such as [BSK+07], handling large data sets on the GPU becomes less of a burden. Another factor is GPU manufacturers unlocking more stages of the graphics pipeline with these stages being supported by graphics APIs. One such stage is the primitive assembly stage. Geometry shaders allow the developer to override primitive assembly. A geometry shader operates on all of the vertices of an incoming primitive (they are not restricted to a single vertex like in a vertex shader). It affords the developer the capability to create and destroy vertices. Outputting zero, one or more primitives and converting primitives to different types. This may allow a more sophisticated handling of divergence, convergence and shear directly on the graphics card.

The second area we would like to investigate is in regards to the visualization of parameter sensitivity. We would like to extend this to produce a similarity vector field which enriches the user-parameter space feedback. This could provide more detail to the user especially in regards to seed orientation. Another interesting direction would be the extension to streaklines. Due to their dynamic geometry over time, streaklines present a variety of challenges and introduce more user-parameters.

The final future direction we would like to highlight is in regards to our similarity measures. Our binning strategy provides large performance gains over the current state-of-the-art. We would like to adapt the binning strategy to handle the streamlines throughout the full domain by addressing the shift-invariance with the current technique. This could be addressed using a hierarchical binning strategy or dynamic bins that aren't fixed to a single portion of the streamlines.

8.2 Summary

During the research, development and discussion of the topics and techniques covered throughout this thesis it is clear that the field of flow visualization is maturing into a developed and well-charted discipline. It is a heavily-researched field, although there is still much left to do before the problem can be considered solved – as highlighted in the discussion above. What is possible, is for us to make strides to solve the presented problems and ultimately allow domain experts to better understand the phenomena their data describes.

The goals of this research were to facilitate user interaction and to proliferate the use of surfaces for 3D flow fields. This resulted in three publications on the construction of various flow surfaces. These results have received praise from several CFD domain experts. The resulting visualizations were novel to the domain experts due to previous algorithms not being present as a standard tool in typical consumer visualization software. Most importantly, they provided information and understanding that the experts had not previously been able to derive.

The work on improving user interaction is also well received and in various stages of publication. This work aids the user in focusing on the most interesting regions throughout the entire spatio-temporal domain and facilitates the user in the presentation of their visualization results by filtering out superfluous information.

This thesis presents research which has forwarded the boundaries of the state-of-the-art in flow visualization. It also provides future directions to build upon this work within. From a personal point of view, the journey of producing this volume of research has been highly enjoyable, rewarding, and challenging. Throughout the journey my confidence has grown and my skills as both an independent researcher and a software engineer have grown substantially.

Part IV

Appendix

A

Data Sets

Several data sets where used to test our algorithms. These are described throughout the thesis were required. We also include them here in this appendix to provide a complete list of the data in a single location.

- **Hurricane Isabel:** The simulation of Hurricane Isabel is sampled at a resolution of $512 \times 512 \times 100$ over 48 timesteps. It is a simulation of a Category 5 hurricane making landfall in North Carolina. This simulation exhibits several examples of interesting behavior such as vortices and saddle points. In the case where we demonstrate our method for unsteady flow using pathlines, we use the entire temporal domain. Hurricane Isabel data produced by the Weather Research and Forecast (WRF) model, courtesy of NCAR and the U.S. National Science Foundation (NSF).

- **Tornado:** A procedural function that simulates behavior similar to a tornado. This is a time dependent simulation. For our purposes we sample the function to a uniform grid at a resolution of $256 \times 256 \times 256$. The function was supplied by Roger Crawfis (Ohio State University, Columbus).

- **Bérnard Convection:** A simulation of Rayleigh-Bénard convection. This simulation is sampled at a resolution of $256 \times 128 \times 64$. A plane is heated at the bottom of the spatial domain creating a pattern of Bérnard convection cells. This simulation was created and provided by Daniel Weiskopf (University of Stuttgart).

- **Flow Behind a Square Cylinder:** This is a direct numerical Navier Stokes simulation by Simone Camarri and Maria-Vittoria Salvetti (University of Pisa), Marcelo Buffoni (Politecnico of Torino), and Angelo Iollo (University of Bordeaux I). It is an incompressible solution with a Reynolds number of 200 and the square cylinder has been positioned symmetrically between two parallel walls. This simulation is initiated from an impulsive start-up and periodic vortex shedding develops with time producing a von karman vortex street. This simulation is in excess of 650GB in its entirety. In our results, we used a re-sampled version supplied by Tino Weinkauf (Max Planck Institute for Informatics, Saarbrcken). The re-sampled version is sampled at a resolution of $192 \times 64 \times 48$ and consists of 102 time-steps.

- **Arnold-Beltrami-Childress (ABC) Flow:** We also use a synthetic dataset of Arnold-Beltrami-Childress (ABC) flow. This describes a closed-form solution of Euler's equation [Hal05]. This type of flow has theoretical importance in fluid dynamics and has been used many times in both fluid dynamics and visualization literature. The vector field is given:

$$v(x,y,z) = \begin{pmatrix} A\ sin(z) + B\ cos(y) \\ B\ sin(x) + C\ cos(z) \\ C\ sin(y) + A\ cos(x) \end{pmatrix},\ x,y,z \in [0, 2\pi] \tag{A.1}$$

where $A = \sqrt{3}$, $B = \sqrt{2}$ and $C = 1$.

- **Smoke Plume:** A simulation of the evolution of a smoke plume. The simulation was supplied by Han Wei Shen (Ohio State University, Columbus) and is sampled at a resolution of $126 \times 126 \times 512$.

- **Lorenz Attractor:** The lorenz attractor was introduced by the mathematician Edward Lorenz. The equations were derived from simplified equations of convection cells rising in the atmosphere. It is famed for its lemniscate (figure-8) shape. It is given by [FH08]:

$$v(x,y,z) = \begin{pmatrix} \sigma(y-x) \\ x(\rho - z) - y \\ xy - \beta z \end{pmatrix},\ x,y \in [-31, 32] \text{ and } z \in [0, 64] \tag{A.2}$$

  in our examples, $\sigma = 10$, $\beta = \frac{8}{3}$ and $\rho = 28$.

- **Ion Front Instability:** A simulation ionization front which is a wall of radiation that resulted during the formation of the first stars in the universe. The data set was provided by D. Whalen and M. L. Norman, "Competition data set and description," in 2008 IEEE Visualization Design Contest, 2008, `http://vis.computer.org/VisWeek2008/vis/contests.html`.

- **Smoke Around a Sphere** A simulation of smoke plume. In this simulation there is a sphere obstacle which the smoke moves around. The fluid solver used to produce this data was provided by Ben Spencer (Swansea University).

B

Streak surface Pseudocode

B.1 Divergence Implementation

In order to simplify the implementation of splitting due to divergence, we test one quad edge at a time and update the topology of the mesh accordingly. For example, Algorithm 2 shows how we update the mesh topology if a quad's west edge length $|E_W| > d_{sep}$. First we test for an existing T-junction on the west edge. If there is one we use it, otherwise we interpolate a new vertex. Likewise for the east edge. Then we construct a new quad for the north half, as in Figure 12 of Chapter 4. Another procedure is called to update the resulting topology (Quad::UpdateNeighborsWest()) shown in Algorithm 3. The methods that handle divergence associated with the other quad edges are similar to Quad::DivideWest().

Figure 12 shows a possible subtle configuration for the Quad::DivideWest() operation. We have to test whether the north-west vertex, V_{NW}, is a T-junction. If it is we change the T-junction's extra neighbor pointer to point to the newly inserted quad. The northern neighbor's southern pointer remains the same. The red lines indicate the correct pointer configuration after the split.

B.2 Convergence Implementation

In order to simplify the implementation of merging a pair of quads, we take a similar approach to that described in Section B.1. We test one quad edge at a time, e.g., north, and test for candidate quads to merge with. Algorithm 4 shows the implementation used to merge with a northern neighbor. The topological cases associated with the merging are illustrated in Figure 7 of Chapter 4. First we test to see which case we fall into by the presence of a T-junction on the east edge of our western neighbor. A similar test is performed on the opposite side. The mesh topology is then updated according to the case the quad is in. The pseudo-code is given in Algorithm 4. Similar functions exist for the other directions e.g., Quad::MergeWest().

B.3 Shear Implementation

The implementation of shear is divided up into five basic functions:

1. Quad::shearEastWithT()

2. Quad::ShearEastNoT()

3. Quad::ShearEastUpdateMesh()

4. Quad::ShearEastDivideWestWithT()

5. Quad::ShearEastDivideWestNoT()

We identify two key cases when we perform the alteration to the mesh topology. Case 1 is the simpler case where there is a T-junction present that we can connect to, this is illustrated in Figure 8 of Chapter 4. The implementation is shown in Algorithm 5. In this case we simply reconnect the north edge to the T-junction. This may result in a new T-junction being added to the northern neighbor (top row).

The second case is slightly more complicated. Here we have no T-junction on the east side. This forms an intermediate triangle in the mesh. See Figure 9 of Chapter 4 and Algorithm 6. We decompose the triangle into three quads by inserting a new point in the middle of the intermediate triangle and on each of its edges. We then sub-divide the triangle into three quads. This method is adapted from Alliez et al. [ACSD$^+$03]. Alliez et al. show how triangular meshes can be converted to quad meshes.

Algorithms 6 - 8 show the implementation associated with Figures 9 and 10 of Chapter 4. The algorithm starts off by testing for the presence of a western T-junction on our eastern neighbor. It then calls Quad::shearEastUpdateMesh() shown in Algorithm 7. Algorithm 7 starts off by creating two new mesh vertices: the vertex on the north edge and a vertex in the center of the intermediate triangle.

It then calls procedures to create the new west and east quads shown in Figure 10 of Chapter 4. The sub-procedure for creating the new west quad is shown in Algorithms 8 and 9.

Algorithm 2 Quad::DivideWest(*void*)

This procedure is called whenever $|E_W| > d_{sep}$.

```
Quad newQuad;
Vertex newEastVertex, newWestVertex;

IF(this->HasT_Junction(WEST)) THEN
   newWestVertex = this->GetT_Junction(WEST).GetVertex();
ELSE
   newWestVertex = Mid(this->GetVertex(NW), this->GetVertex(SW));
   this->AddT_JunctionToWestNeighbor(newWestVertex, newQuad);
ENDIF

IF(this->HasT_Junction(EAST) THEN
   newEastVertex = this->GetT_Junction(EAST).GetVertex();
ELSE
   newEastVertex = Mid(this->GetVertex(NE), this->GetVertex(SE));
   this->AddT_JunctionToEastNeighbor(newEastVertex, newQuad);
ENDIF

//Construct a new quad object on north half.
newQuad.SetNWVertex(this->GetNWVertex());
newQuad.SetNEVertex(this->GetNEVertex());
newQuad.SetSWVertex(newWestVertex);
newQuad.SetSEVertex(newEastVertex);

//Update this quad's vertices
this->SetNWVertex(newWestVertex);
this->SetNEVertex(newEastVertex);

//Update neighbor pointers for new quad
this->UpdateNeighborsWest(newQuad);

RETURN;
```

Algorithm 3 Quad::UpdateNeighboursWest(*Quad newQuad*)

This procedure updates this quad's and neighboring quad's topology after a divide. See also Algorithm 2.

```
/* Update new quad's neighbor pointers */
newQuad.SetNorthNeighbor(this->GetNeighbor(NORTH));
newQuad.SetSouthNeighbor(this);

IF(this->HasT_Junction(WEST)) THEN
   newQuad.SetWestNeighbor(this->GetT_Junction(WEST).GetNeighbor());
   this->GetT_Junction(WEST).GetNeighbor().SetEastNeighbor(newQuad);
   this->deleteT_Junction(WEST);
ELSE
   newQuad.SetWestNeighbor(this->GetNeighbor(WEST));
ENDIF

IF(this->HasT_Junction(EAST)) THEN
   newQuad.SetEastNeighbor(this->GetT_Junction(EAST).GetNeighbor());
   this->GetT_Junction(EAST).GetNeighbor().SetWestNeighbor(newQuad);
   this->deleteT_Junction(EAST);
ELSE
   newQuad.SetEastNeighbor(this->GetNeighbor(EAST));
ENDIF

/**
* Update this quad's northern neighbor's southern
* pointer to new quad. This handles the subtle case
   *highlighted in Figure 12 of the manuscript.
*/
IF(this->GetNeighbor(NORTH).GetNeighbor(SOUTH) == this) THEN
   this->GetNeighbor(NORTH).SetSouthNeighbor(newQuad);
ELSE IF((this->GetNeighbor(NORTH).HasT_Junction(SOUTH))
  this->GetNeighbor(NORTH).GetT_Junction(SOUTH).SetNeighbour(newQuad);
ENDIF

/**
* Update this quad's northern neighbor's pointer to
* new quad.
*/
this->SetNorthNeighbor(newQuad);

/**
* If this quad had a northern T-junction,
* move it to the new quad.
*/
IF(this->HasT_Junction(NORTH)) THEN
   newQuad.SetT_JunctionNorth(this->GetT_Junction(NORTH));
   this->SetT_JunctionNorth(NULL);
ENDIF

RETURN;
```

Algorithm 4 Quad::MergeNorth(*void*)

This procedure is called whenever the edge length of the west AND east sides $< d_{converge}$. One of the following must also be true:
1. The NW vertex is a T-Junction OR
2. The NE vertex is a T-Junction OR
3. None of the corner vertices are T-Junctions.
In these three cases, this quad merges with it's northern neighbor.

```
/**
* If our WESTern neighbor has an EAST T-junction
* THEN delete it. (Cases 2,3,4,5)
* ELSE introduce WESTern T-junction to the new merged
* quad (this). (Cases 1,6)
*/
IF(this->GetNeighbor(WEST).HasT_Junction(EAST)) THEN
   this->GetNeighbor(WEST).GetT_Junction(EAST).DeleteVertex();
   this->GetNeighbor(WEST).DeleteT_Junction(EAST);
ELSE
   this->NewT_Junction(WEST);
   this->GetT_Junction(WEST).SetVertex(this.GetVertex(NW));
   this->GetT_Junction(WEST).SetWesternNeighbor(this.GetNeighbor(NORTH).GetNeighbor(WEST));
ENDIF

/* Perform mirror opposite on the EAST side. */
IF(this->GetNeighbor(EAST).HasT_Junction(WEST)) THEN
   this->GetNeighbor(EAST).GetT_Junction(WEST).DeleteVertex();
   this->GetNeighbor(EAST).DeleteT_Junction(WEST);
ELSE
   this->NewT_Junction(EAST);
   this->GetT_Junction(EAST).SetVertex(this.GetVertex(NE));
   this->GetT_Junction(EAST).SetEasternNeighbor(this.GetNeighbor(NORTH).GetNeighbor(EAST));
ENDIF

/* Update to new northern vertices. */
this->SetNWVertex.(this->GetNeighbor(NORTH).GetVertex(NW));
this->SetNEVertex.(this->GetNeighbor(NORTH).GetVertex(NE));
formerNorthNeighbor = this.GetNeighbor(NORTH);

/* Update new north neighbor's south pointer. */
this->GetNeighbor(NORTH).GetNeighbor(NORTH).SetSouthNeighbor(this);

/* Update new neighbor pointer. */
this.SetNorthNeighbor(this.GetNeighbor(NORTH).GetNeighbor(NORTH));

/* Delete old north neighbor quad object. */
formerNorthNeighbor.Delete();

RETURN;
```

Algorithm 5 Quad::ShearWithEastT($void$)
This method is called whenever a quad is sheared and there’s a T-Junction on the east edge, i.e.
IF
1. $\frac{d_{short}}{d_{long}} < \varepsilon_{shear}$ AND
2. $\theta_{NorthEast} < \theta_{shear}$ AND
3. this-¿HasT_Junction($EAST$) $==$ TRUE
Shear adjustment is not applied to quads at the boundaries of the mesh.

```
Quad NEquad = this->GetNeighbor(NORTH).GetNeighbor(EAST)
IF (NEquad == NULL)
   RETURN

/* IF our NORTHEASTern neighbor has a WESTern
 *    T-junction
 * THEN snap the two T-Junctions together
 * ELSE create a new EAST T-junction for
 *    our NORTHern neighbor.
 * Update our NE vertex AND delete our EASTERN T-Junction
 */
IF (NEquad.HasT_Junction(WEST)) THEN
  NEQuad.SetWestNeighbour(this->GetNeighbor(NORTH));
  NEquad.GetT_Junction(WEST).DeleteVertex();
  NEquad.DeleteT_Junction(WEST);
ELSE
  this->GetNeighbor(NORTH).NewT_Junction(EAST);
  this->GetNeighbor(NORTH).GetT_Junction(EAST).SetVertex(this->GetNEvertex())
  this->GetNeighbor(NORTH).SetEasternNeighbor(this->GetT_Junction(EAST)->GetNeighbor());
END IF
this->SetNEvertex(this->GetT_Junction(EAST)->GetVertex());

/* Set new south east vertex of northern neighbor */
this->GetNeighbor(NORTH).SetSEVertex(this->GetT_Junction(EAST)->GetVertex());

this->DeleteT_Junction(EAST);
```

Algorithm 6

Procedure: Quad::ShearEastNoT(void)

This method is called whenever a quad is sheared and there's no T-Junction on the EAST edge, i.e. IF

1. $\frac{d_{short}}{d_{long}} < \varepsilon_{shear}$ AND
2. $\theta_{NorthEast} < \theta_{shear}$ AND
3. this-¿HasT_Junction($EAST$) $==$ FALSE

Shear update is not applied to quads at the boundaries of the mesh.

```
Quad Nquad = this->GetNeighbor(NORTH)
IF (Nquad == NULL)
  RETURN

/*
* IF our EASTern neighbor has a WESTern T-junction
* THEN delete the T-Junction
* ELSE create a new EAST T-junction for
*      our NORTHern neighbor.
*/
IF (this->GetNeighbor(EAST).HasT_junction(WEST)) THEN
  this->GetNeighbor(EAST).GetT_junction(WEST).DeleteVertex()
  this->GetNeighbor(EAST).DeleteT_junction(WEST)
ELSE
  this->GetNeighbor(NORTH).NewTjunction(EAST)
  this->GetNeighbor(NORTH).GetTjunction(EAST).SetVertex(this->GetNEvertex())
  this->GetNeighbor(NORTH).SetEastNeighbor(this->GetNeighbor(EAST))
END IF

/* Update south-east vertex of out northern neighbor */
this->GetNeighbor(NORTH).SetSEVertex(this->GetSEVertex());

/*
* Create 2 new quads and 3 new T-junctions
* Update our vertices and topology.
*/
this->shearEastUpdateMesh()
RETURN
```

Algorithm 7 Procedure: Quad::shearEastUpdateMesh(void)
This method updates the mesh topology resulting from the shear. See Algorithm 6

```
Vertex vertexNorth, vertexCenter
Quad quadWest, quadEast
/* This quad’s north-east vertex is updated in Algorithm 5 */

vertexNorth  = new Vertex(interpolate(this->GetNEvertex(), this->GetNWvertex())
vertexCenter = new Vertex(interpolate(this->GetNWvertex(), this->GetNWvertex(),
                                      this->GetSEvertex(), this->GetSWvertex())

/* First, create the WEST quad. */
IF (this->HasT_Junction(WEST)
THEN
  quadWest this->shearEastDivideWestWithT(vertexNorth, vertexCenter);
ELSE
  quadWest this->shearEastDivideWestNoT(vertexNorth, vertexCenter);

/* Second, create the EAST quad. */
IF (this->HasT_Junction(EAST)
THEN
  quadEast this->shearEastDivideEastWithT(vertexNorth, vertexCenter);
ELSE
  quadEast this->shearEastDivideEastNoT(vertexNorth, vertexCenter);
/* Update the quadWest and quadEast neighbor topology. */
quadWest.SetEastNeighbor(quadEast);
quadEast.SetEastNeighbor(quadWest);

/* Update this quad’s vertices. */
this->SetNWvertex(quadWest.GetSWvertex());
this->SetNEvertex(vertexCenter);
this->SetSEvertex(quadEast.GetSEvertex());
/* SW vertex stays the same. */

/* Update this quad’s topology. */
this->SetNorthNeighbor(quadWest);
this->SetEastNeighbor(quadEast);
/* West and South neighbors remain the same */

/* Add new quads to central quad list. */
this->GetQuadList()->Add(quadWest, quadEast);
```

Algorithm 8

Procedure: Quad::shearEastDivideWestWithT(
Vertex vertexNorth, Vertex vertexCenter)

This procedure creates a new west quad when a sheared quad is identified. In this case, this quad has a WESTern T-junction.

```
Quad quadWest

/* Update the WEST quad's vertices. */
quadWest.SetNWvertex(this->GetNWvertex());
quadWest.SetNEvertex(vertexNorth);
quadWest.SetSEvertex(vertexCenter);
quadWest.SetSWvertex(this->GetTjunction(WEST).GetVertex());

/* Update the WEST quad's topology
(except for new EAST quad). */
quadWest.SetNorthNeighbor(this->GetNeighbor(NORTH));
quadWest.SetSouthNeighbor(this);
quadWest.SetWestNeighbor(this->GetTjunction(WEST).GetNeighbor());

/* Update our neighbor's topology
(except for new EAST quad). */
this->GetNeighbor(NORTH).SetSouthNeighbor(quadWest);
this->GetTjunction(WEST).GetNeighbor().SetEastNeighbor(quadWest);

/* Delete the WESTern T-junction.*/
this->DeleteTjunction(WEST);

RETURN quadWest;
```

Algorithm 9 Procedure: Quad::shearEastDivideWestWithNoT(Vertex vertexNorth, Vertex vertexCenter)
This procedure creates a new west quad when a sheared quad is identified. In this case, this quad has no WESTern T-junction.

```
Quad quadWest

/* Update the WEST quad's vertices. */
quadWest.SetNWvertex(this->GetNWvertex());
quadWest.SetNEvertex(vertexNorth);
quadWest.SetSEvertex(vertexCenter);
quadWest.SetSWvertex(Interpolate(this->GetNWvertex(), this->GetSWvertex());

/* Update the WEST quad's topology
   (except for new EAST quad).*/
quadWest.SetNorthNeighbor(this->GetNeighbor(NORTH));
quadWest.SetSouthNeighbor(this);
quadWest.SetWestNeighbor(this->GetNeighbor(WEST));

/* Create a new T-junction for our WEST neighbor. */
this->GetNeighbor(WEST).NewTjunction(EAST);
this->GetNeighbor(WEST).GetTjunction(EAST).SetVertex(quadWest.GetSWvertex());
this->GetNeighbor(WEST).GetTjunction(EAST).SetEastNeighbor(quadWest);

/* Update our neighbor's topology
(except for new EAST quad). */
this->GetNeighbor(NORTH).SetSouthNeighbor(quadWest);

RETURN quadWest;
```

Bibliography

[ACSD$^+$03] P. Alliez, D. Cohen-Steiner, O. Devillers, B. Lévy, and M. Desbrun. Anisotropic polygonal remeshing. *SIGGRAPH 03 - ACM Transactions on Graphics*, 22(3):485–493, 2003.

[ARB00] OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Addison Wesley, 2000. D. Schreiner, editor.

[Ari90] R. Aris. *Vectors, Tensors, and the Basic Equations of Fluid Mechanics*. Dover Publications Inc., 1990.

[BFTW09] K. Bürger, F. Ferstl, H. Theisel, and R. Westermann. Interactive Streak Surface Visualization on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1259–1266, November-December 2009.

[BHR$^+$94] M. Brill, H. Hagen, H.-C. Rodrian, W. Djatschin, and S. V. Klimenko. Streamball Techniques for Flow Visualization. In *Proceedings IEEE Visualization '94*, pages 225–231, October 1994.

[BL92] S. Bryson and C. Levit. The Virtual Wind Tunnel. *IEEE Computer Graphics and Applications*, 12(4):25–34, July 1992.

[BLM95] B. G. Becker, D. A. Lane, and N. L. Max. Unsteady Flow Volumes. In *Proceedings IEEE Visualization '95*, pages 329–335, 1995.

[BMP$^+$90] G. V. Bancroft, F. J. Merritt, T. C. Plessel, P. G. Kelaita, R. K. McCabe, and A. Globus. FAST: A Multiprocessed Environment for Visualization of Computational Fluid Dynamics. In *Proceedings of IEEE Visualization*, pages 14–27, 1990.

[BPFG11] W. Berger, H. Piringer, P. Filzmoser, and E. Gröller. Uncertainty-Aware Exploration of Continuous Parameter Spaces Using Multivariate Prediction. *Computer Graphics Forum*, 30(3):911–920, 2011.

[Bro04] R. Brown. Animated visual vibrations as an uncertainty visualisation technique. In *Proceedings GRAPHITE '04*, pages 84–89, New York, NY, USA, 2004. ACM.

[BS87] P. G. Buning and J. L. Steger. Graphics and Flow Visualization in Computational Fluid Dynamics. In *Proc. American Institute of Aeronautics and Astronautics 8th Computational Fluid Dynamics Conf*, pages 814–820, 1987.

[BSK$^+$07] K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, and R. Westermann. Interactive Visual Exploration of Unsteady 3D Flows. In *Proc. EuroVis*, pages 251–258, 2007.

[Bun89] P. G. Buning. Numerical Algorithms for CFD Post-Processing. *van Karman Inst. for Fluid Dynamics*, pages 1–20, 1989.

[BVPBtHR09] R. Brecheisen, A. Vilanova, B Platel, and Bart B. ter Haar Romeny. Parameter Sensitivity Visualization for DTI Fiber Tracking. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1441–1448, 2009.

[BWE05] R. Botchen, D. Weiskopf, and T. Ertl. Texture-based visualization of uncertainty in flow fields. In *In Proceedings of IEEE Visualization 2005*, pages 647–654, 2005.

[BWF$^+$10] S. Born, A. Wiebel, J Friedrich, G Scheuermann, and D Bartz. Illustrative Stream Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16:1329–1338, November 2010.

[CCK07] Y. Chen, J. D. Cohen, and J. Krolik. Similarity-Guided Streamline Placement with Error Evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1448–1455, 2007.

[Cd80] S. D. Conte and C. de Boor. *Elementary Numerical Analysis*. McGraw-Hill, New York, 1980.

[CGG04] I. Corouge, S. Gouttard, and G. Gerig. Towards a shape model of white matter fiber bundles using diffusion tensor MRI. In *International Symposium on Biomedical Imaging (ISBI)*, pages 344–347, 2004.

[CL93] B. Cabral and L. C. Leedom. Imaging Vector Fields Using Line Integral Convolution. In *Poceedings of ACM SIGGRAPH 1993*, Annual Conference Series, pages 263–272, 1993.

[CR00] A. Cedilnik and P. Rheingans. Procedural Annotation of Uncertain Information. In *Proceedings IEEE Visualization 2000*, pages 77–84. IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[CSBI05] S. Camarri, M. V. Salvetti, M. Buffoni, and A. Iollo. Simulation of the Three-Dimensional Flow Around a Square Cylinder Between Parallel Walls at Moderate Reynolds Numbers. In *Congresso di Meccanica ed Applicata*, pages 11–15, sep 2005.

[CSvS86] C. Cuvelier, A. Segal, and A.A. van Steenhoven. *Finite Element Methods and Navier-Stokes Equations*. Springer, 1986.

[DBG^{+}06] S. Dong, P-T. Bremer, M. Garland, V. Pascucci, and J. C. Hart. Spectral surface quadrangulation. *ACM Trans. Graph.*, 25(3):1057–1066, 2006.

[DGKP09] D. Dussel, E. J. Griffith, M. Koutek, and F. H. Post. Interactive Particle Tracing for Visualizing Large, Time-Varying Flow Fields. Technical report, TU Delft Data Visualization Group, 2009.

[DH92] D. Darmofal and R. Haimes. Visualization of 3-D vector fields: Variations on a stream. Paper 92-0074, AIAA, 1992.

[DKG05] S. Dong, S. Kircher, and M. Garland. Harmonic functions for quadrilateral remeshing of arbitrary manifolds. *Comput. Aided Geom. Des.*, 22(5):392–423, 2005.

[DL09] C. Demiralp and D. H. Laidlaw. Similarity coloring of DTI fiber tracts. In *Proceedings of DMFC Workshop at MICCAI*, 2009.

[DSSC08] J. Daniels, C. T. Silva, J. Shepherd, and E. Cohen. Quadrilateral mesh simplification. In *Proc of Siggraph Asia*, page forthcoming, 2008.

[EM94] T. Eiter and H. Mannila. Computing discrete frchet distance. Technical Report CD-TR 94/64, Technische Universitt Wien, 1994.

[FBTW10] F. Ferstl, K. Bürger, H. Theisel, and R. Westermann. Interactive Separating Streak Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16:1569–1577, 2010.

[FG98] A. L. Fuhrmann and M. E. Gröller. Real-Time Techniques for 3D Flow Visualization. In *Proceedings IEEE Visualization '98*, pages 305–312. IEEE, 1998.

[FH08] G. E. Farin and D. Hansford. *Mathematical Principles for Scientific Computing and Visualization*. A. K. Peters, 2008.

[GH90] M. B. Giles and R. Haimes. Advanced Interactive Visualization for CFD. *Computing Systems in Education*, 1(1):51–62, 1990.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GKT^{+}08] C. Garth, H. Krishnan, X. Tricoche, T. Bobach, and K. I. Joy. Generation of Accurate Integral Surfaces in Time-Dependent Vector Fields. *Proceedings of IEEE Visualization 2008*, October 2008.

[Gon85] T. F. Gonzalez. Clustering to Minimize Intercluster Distance. *Theoretical Computer Science*, 38:293–306, 1985.

[GTS$^+$04] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann. Surface Techniques for Vortex Visualization. In *Data Visualization, Proceedings of the 6th Joint IEEE TCVG–EUROGRAPHICS Symposium on Visualization (VisSym 2004)*, pages 155–164, May 2004.

[Hai94] R. Haimes. pV3: A distributed system for large-scale unsteady CFD visualization. Paper 94-0321, AIAA, 1994.

[Hal01] G. Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Phys. D*, 149:248–277, March 2001.

[Hal05] G. Haller. An objective definition of a vortex. *Journal of Fluid Mechanics*, 525:1–26, 2005.

[HE06] A. Helgeland and T. Elboth. High-Quality and Interactive Animations of 3d Time-Varying Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1535–1546, 2006.

[HG91] R. Haimes and M. Giles. VISUAL3 Interactive Unsteady Unstructured 3D Visualization. Technical Report 91-0794, AIAA Paper, 1991.

[HGB$^+$10] M. Hummel, C. Garth, Hamann B., H. Hagen, and K. Joy. IRIS: Illustrative Rendering for Integral Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1319–1328, November 2010.

[Hol06] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[HP93] A. J. S. Hin and F. H. Post. Visualization of turbulent flow with particles. In *Proceedings of IEEE Visualization '93*, pages 46–53, 1993.

[Hul90] J. P. M. Hultquist. Interactive Numerical Flow Visualization Using Stream Surfaces. *Computing Systems in Engineering*, 1(2-4):349–353, 1990.

[Hul92] J. P. M. Hultquist. Constructing Stream Surfaces in Steady 3D Vector Fields. In *Proceedings IEEE Visualization '92*, pages 171–178, 1992.

[INI] INI File. http://en.wikipedia.org/wiki/INI_file.

[JDL09] R. Jianu, C. Demiralp, and D.H. Laidlaw. Exploring 3D DTI fiber tracts with linked 2D representations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1449 –1456, 2009.

[JEH$^+$04] G. Johnson, D. Ebert, C. Hansen, D. Kirk, B. Mark, and H. Pfister. Panel: The Future Visualization Platform. In *Proceedings IEEE Visualization 2004*, pages 569–571, 2004.

[JL97a] B. Jobard and W. Lefer. Creating Evenly–Spaced Streamlines of Arbitrary Density. In *Proceedings of the Eurographics Workshop on Visualization in Scientific Computing '97*, volume 7, pages 45–55, 1997.

[JL97b] B. Jobard and W. Lefer. The Motion Map: Efficient Computation of Steady Flow Animations. In *Proceedings IEEE Visualization '97*, pages 323–328. IEEE Computer Society, October 19–24 1997.

[JL00] B. Jobard and W. Lefer. Unsteady Flow Visualization by Animating Evenly-Spaced Streamlines. In *Computer Graphics Forum (Eurographics 2000)*, volume 19(3), pages 21–31, 2000.

[JL01] B. Jobard and W. Lefer. Multiresolution Flow Visualization. In *WSCG 2001 Conference Proceedings*, pages 33–37, Plzen, Czech Republic, February 2001.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

[KGJ09] H. Krishnan, C. Garth, and K. I. Joy. Time and streak surfaces for flow visualization in large time-varying data sets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1267–1274, October 2009.

[KKKW05] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744 – 756, 2005.

[KL95] D. N. Kenwright and D. A. Lane. Optimization of Time-Dependent Particle Tracing Using Tetrahedral Decomposition. In *Proceedings of IEEE Visualization 1995*, pages 321–328, 1995.

[KL96] D. N. Kenwright and D. A. Lane. Interactive Time-Dependent Particle Tracing Using Tetrahedral Decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):120–129, June 1996.

[KM92] D. N. Kenwright and G. D. Mallinson. A 3-D Streamline Tracking Algorithm Using Dual Stream Functions. In *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pages 62–68, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[KNP07] F. Kälberer, M. Nieser, and K. Polthier. Quadcover - surface parameterization using branched coverings. *Computer Graphics Forum*, 26(3):375–384, 2007.

[KOD$^+$05] B. Krauskopf, H. M. Osinga, E. J. Doedel, M. E. Henderson, J. Guckenheimer, A. Vladimirsky, M. Dellnitz, and O. Junge. A survey of methods for computing (un)stable manifolds of vector fields. *I. J. Bifurcation and Chaos*, 15(3):763–791, 2005.

[Lan93] D. A. Lane. Visualization of Time-dependent Flow Fields. In *Proceedings of Visualization '93*, pages 32–38, 1993.

[Lan94] D. A. Lane. UFAT: A Particle Tracer for Time-dependent Flow Fields. In *Proceedings of Visualization '94*, pages 257–264, 1994.

[Lar02] R. S. Laramee. Interactive 3D Flow Visualization Using a Streamrunner. In *CHI 2002, Conference on Human Factors in Computing Systems, Extended Abstracts*, pages 804–805. ACM SIGCHI, ACM Press, April 20–25 2002.

[Lar04] R. S. Laramee. *Interactive 3D Flow Visualization Using Textures and Geometric Primitives*. PhD thesis, Vienna University of Technology, Institute for Computer Graphics and Algorithms, Vienna, Austria, December 2004.

[Lar10] R. S. Laramee. Bob's Concise Coding Conventions (C^3). *Advances in Computer Science and Engineering (ACSE)*, 4(1):23–36, February 2010.

[LC87] W. E. Lorensen and H. E. Cline. Marching Cubes: a High Resolution 3D Surface Construction Algorithm. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87, Anaheim, CA)*, pages 163–170. ACM, July 27–31 1987.

[LGD$^+$05] R. S. Laramee, C. Garth, H. Doleisch, J. Schneider, H. Hauser, and H. Hagen. Visual Analysis and Exploration of Fluid Flow in a Cooling Jacket. In *Proceedings IEEE Visualization 2005*, pages 623–630, 2005.

[LGSH06] R. S. Laramee, C. Garth, J. Schneider, and H. Hauser. Texture-Advection on Stream Surfaces: A Novel Hybrid Visualization Applied to CFD Results. In *Data Visualization, The Joint Eurographics-IEEE VGTC Symposium on Visualization (EuroVis 2006)*, pages 155–162,368. Eurographics Association, 2006.

[LH05] R. S. Laramee and H. Hauser. Geometric Flow Visualization Techniques for CFD Simulation Data. In *Proceedings of the 21st Spring Conference on Computer Graphics*, pages 213–216, May 2005.

[LHD$^+$04] R. S. Laramee, H. Hauser, H. Doleisch, F. H. Post, B. Vrolijk, and D. Weiskopf. The State of the Art in Flow Visualization: Dense and Texture-Based Techniques. *Computer Graphics Forum*, 23(2):203–221, June 2004.

[LHS08] L. Li, H. H. Hsien, and H. W. Shen. Illustrative streamline placement and visualization. In *IEEE Pacific Visualization Symposium 2008*, pages 79–86, 2008.

[LHZP07] R.S. Laramee, H. Hauser, L. Zhao, and F. H. Post. Topology Based Flow Visualization: The State of the Art. In *Topology-Based Methods in Visualization (Proceedings of Topo-in-Vis 2005)*, Mathematics and Visualization, pages 1–19. Springer, 2007.

[LJL04] W. Lefer, B. Jobard, and C. Leduc. High-quality animation of 2d steady vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):2–14, 2004.

[LK07] R. S. Laramee and R. Kosara. *Human-Centered Visualization Environments*, chapter Future Challenges and Unsolved Problems. Springer Verlag, 2007.

[LM06] Z. P. Liu and R. J. Moorhead, II. An Advanced Evenly-Spaced Streamline Placement Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):965–972, Sep/Oct 2006.

[LMG97] H. Löffelmann, L. Mroz, and E. Gröller. Hierarchical Streamarrows for the Visualization of Dynamical Systems. In *Visualization in Scientific Computing '97*, Eurographics, pages 155–164. Springer-Verlag, 1997.

[LMGP97] H. Löffelmann, L. Mroz, E. Gröller, and W. Purgathofer. Stream Arrows: Enhancing the Use of Streamsurfaces for the Visualization of Dynamical Systems. *The Visual Computer*, 13:359–369, 1997.

[Löf98] H. Löffelmann. *Visualizing Local Properties and Characteristic Structures of Dynamical Systems*. PhD thesis, Technical University of Vienna, 1998.

[LPSW96] S. K. Lodha, A. Pang, R. E. Sheehan, and C. M. Wittenbrink. UFLOW: Visualizing Uncertainty in Fluid Flow. In *Proceedings IEEE Visualization '96*, pages 249–254, October 27–November 1 1996.

[LS07] L. Li and H.-W. Shen. Image-Based Streamline Generation and Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):630–640, 2007.

[LSH04] R. S. Laramee, J. Schneider, and H. Hauser. Texture-Based Flow Visualization on Isosurfaces from Computational Fluid Dynamics. In *Data Visualization, The Joint Eurographics-IEEE TVCG Symposium on Visualization (VisSym '04)*, pages 85–90,342. Eurographics Association, 2004.

[LWS96] S. K. Lodha, C. M. Wilson, and R. E. Sheehan. LISTEN: Sounding Uncertainty Visualization. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings IEEE Visualization '96*, pages 189–196. IEEE, 1996.

[LWSH04] R. S. Laramee, D. Weiskopf, J. Schneider, and H. Hauser. Investigating Swirl and Tumble Flow with a Comparison of Visualization Techniques. In *Proceedings IEEE Visualization 2004*, pages 51–58, 2004.

[MAD05] A. Mebarki, P. Alliez, and O. Devillers. Farthest Point Seeding for Efficient Placement of Streamlines. In *Proceedings IEEE Visualization 2005*, pages 479–486, 2005.

[Mas99] G. T. Mase. *Continuum Mechanics for Engineers*. CRC Press, 1999.

[MBC93] N. Max, B. Becker, and R. Crawfis. Flow Volumes for Interactive Vector Field Visualization. In *Proceedings IEEE Visualization '93*, pages 19–24. IEEE Computer Society, October 1993.

[MBS+04] K. Mahrous, J. C. Bennett, G. Scheuermann, B. Hamann, and K. I. Joy. Topological segmentation in three-dimensional vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):198–205, 2004.

[MCHM10] S. Marchesin, C-K. Chen, C. Ho, and K-L. Ma. View-dependent streamlines for 3D vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1578–1586, 2010.

[MEL+11a] T. McLoughlin, M. Edmunds, R. S. Laramee, G. Chen, N. Max, H. Yeh, and E. Zhang. Visualization of User-Parameter Sensitivity for Streamline Seeding. Technical report, Dept. Computer Science, Swansea University, 2011.

[MEL+11b] T. McLoughlin, Matthew Edmunds, R. S. Laramee, Mark W. Jones, Guoning Chen, and E. Zhang. Using Integral Surfaces to Visualize CFD Simulation Results. In *Proceedings NAFEMS World Congress, The International Association for the Engineering Analysis Community*, pages 100–109, 2011.

[Mey05] S. D. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 2005.

[MHHI98] X. Mao, Y. Hatanaka, H. Higashida, and A. Imamiya. Image-Guided Streamline Placement on Curvilinear Grid Surfaces. In *Proceedings IEEE Visualization '98*, pages 135–142, 1998.

[MJL11] T. McLoughlin, M. W. Jones, and R. S. Laramee. Similarity Measures for Streamline Seeding Rake Enhancement. Technical report, Dept. Computer Science, Swansea University, 2011.

[MK04] M. Marinov and L. Kobbelt. Direct anisotropic quad-dominant remeshing. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, pages 207–216, Washington, DC, USA, 2004. IEEE Computer Society.

[ML12] T. McLoughlin and R. S. Laramee. Design and Implementation of Geometric Flow Visualization software. In *Computer Graphics*. InTech, 2012.

[MLP+09] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over Two Decades of Integration-Based, Geometric Flow Visualization. In *Eurographics 2009: State-of-the-Art Reports*, pages 73–92, 30 March – 3 April 2009.

[MLP+10] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over Two Decades of Integration-Based, Geometric Flow Visualization. *Computer Graphics Forum*, 29(6):1807–1829, 2010.

[MLZ] T. McLoughlin, R. S. Laramee, and E. Zhang. Easy stream surfaces: Supplementary video accompanied with paper submission. http://cs.swan.ac.uk/$\sim$cstony/Video/EIS.mpg.

[MLZ09] T. McLoughlin, R. S. Laramee, and E. Zhang. Easy Integral Surfaces: A Fast, Quad-based Stream and Path Surface Algorithm. In *Proceedings Computer Graphics International 2009*, pages 67–76, 2009.

[MLZ10] T. McLoughlin, R. S. Laramee, and E. Zhang. Constructing Streak Surfaces for 3D Unsteady Vector Fields. In *Proceedings of the Spring Conference on Computer Graphics (SCCG)*, pages 25–32, 2010.

[MPSS05] O. Mallo, R. Peikert, C. Sigg, and F. Sadlo. Illuminated Lines Revisited. In *Proceedings IEEE Visualization 2005*, pages 19–26, 2005.

[MSRMH09] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. H. Hinrichs. Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications (Applications Department)*, 29(6):6–13, Nov./Dec. 2009.

[MTHG03] O. Mattausch, T. Theußl, H. Hauser, and E. Gröller. Strategies for Interactive Exploration of 3D Flow Using Evenly-Spaced Illuminated Streamlines. In *Proceedings of the 19th Spring Conference on Computer Graphics*, pages 213–222, 2003.

[MVvW05] B. Moberts, A. Vilanova, and J.J. van Wijk. Evaluation of fiber clustering methods for diffusion tensor imaging. In *Proceedings IEEE Visualization 2005*, pages 65 – 72, 2005.

[NJ99] G. M. Nielson and I.-H. Jung. Tools for Computing Tangent Curves for Linearly Varying Vector Fields over Tetrahedral Domains. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):360–372, October – December 1999. ISSN 1077-2626.

[NJS$^+$97] G. M. Nielson, I. H. Jung, N. Srinivasan, J. Sung, and J. B. Yoon. *Scientific Visualization, Overviews, Methodologies, and Techniques*, chapter Tools for Computing Tangent Curves and Topological Graphs for Visualizing Piecewise Linearly Varying Vector Fields over Triangulated Domains, pages 527–562. IEEE Computer Society, 1997.

[PKRJ10] K. Potter, J. Kniss, R. Riesenfeld, and C. R. Johhson. Visualizing summary statistics and uncertainty. *Computer Graphics Forum (Proceedings of EuroVis 2010)*, 29(3):823–831, 2010.

[PPF$^+$10] A. Pobitzer, R. Peikert, R. Fuchs, B. Schindler, A. Kuhn, H. Theisel, K. Matkovic, and H. Hauser. On the Way Towards Topology-Based Visualization of Unsteady Flow the State of the Art. In Helwig Hauser and Erik

Reinhard, editors, *State of the Art Reports*, pages 137–154. Eurographics Association, May 2010.

[PRH10] J. Praßni, T. Ropinski, and K. H. Hinrichs. Uncertainty-aware guided volume segmentation. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Vis Conference Issue)*, 16(6):1358–1365, nov, dec 2010.

[PS09] R. Peikert and F. Sadlo. Topologically Relevant Stream Surfaces for Flow Visualization. In H. Hauser, editor, *Proc. Spring Conference on Computer Graphics*, pages 43–50, April 2009.

[PTVF02] W. H. Press, S. A. Teukolsky, W. T. Vettering, and B. P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2 edition, 2002.

[PVH$^+$03] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The State of the Art in Flow Visualization: Feature Extraction and Tracking. *Computer Graphics Forum*, 22(4):775–792, Dec. 2003.

[PWL96] A. T. Pang, C. M. Wittenbrink, and S. K. Lodha. Approaches to Uncertainty Visualization. *The Visual Computer*, 13:370–390, 1996.

[PZ07] J. Palacios and E. Zhang. Rotational symmetry field design on surfaces. *ACM Transactions on Graphics*, 26(3):55, 2007.

[RBM87] S. E. Rogers, P. G. Buning, and F. J. Merrit. Distributed Interactive Graphics Applications in Computational Fluid Dynamics. *International Journal of Supercomputer Applications*, 1(4):96–105, 1987.

[Rey95] O. Reynolds. On the Dynamical Theory of Incompressible Viscous Fluids and the Determination of the Criterion. *Royal Society of London Philosophical Transactions Series A*, 186:123–164, 1895.

[RLBS03] P. J. Rhodes, R. S. Laramee, R. D. Bergeron, and T. M. Sparr. Uncertainty Visualization Methods in Isosurface Rendering. In M. Chover, H. Hagen, and D. Tost, editors, *Eurographics 2003, Short Papers*, pages 83–88. The Eurographics Association, September 1-5 2003.

[RLL$^+$06] N. Ray, W. C. Li, B. Lévy, A. Sheffer, and P. Alliez. Periodic global parameterization. *ACM Trans. Graph.*, 25(4):1460–1485, 2006.

[Rot00] M. Roth. *Automatic Extraction of Vortex Core Lines and Other Line-Type Features for Scientific Visualization*. PhD Dissertation No. 13673, ETH Zurich, 2000. published by Hartung-Gorre Verlag, Konstanz, ISBN 3-89649-582-8.

[RPP$^+$09] O. Rosanwo, C. Petz, S. Prohaska, I. Hotz, and H-C. Hege. Dual streamline seeding. In Peter Eades, Thomas Ertl, and Han-Wei Shen, editors, *Proceedings of the IEEE Pacific Visualization Symposium*, pages 9 – 16, 2009.

[Sab88] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 51–58, New York, NY, USA, 1988. ACM.

[SBH$^+$01] G. Scheuermann, T. Bobach, H. Hagen, K. Mahrous, B. Hamann, K. I. Joy, and W. Kollmann. A Tetrahedral-based Stream Surface Algorithm. In *Proceedings IEEE Visualization 2001*, pages 151–157, October 2001.

[SdBPM98] I. A. Sadarjoen, A. J. de Boer, F. H. Post, and A. E. Mynett. Particle Tracing in σ-Transformed Grids Using Tetrahedral 6-Decomposition. In *Visualization in Scientific Computing '98*, Eurographics, pages 71–80. Springer, 1998.

[SGvR$^+$03] M. Schirski, A. Gerndt, T. van Reimersdahl, T. Kuhlen, P. Adomeit, O. Lang, S. Pischinger, and C. Bischof. ViSTA FlowLib - Framework for Interactive Visualization and Exploration of Unsteady Flows in virtual environments. In *In Proc. of the Eurographics Workshop on Virtual Environments*, pages 77–85, 2003.

[She96] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *WACG: 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering, WACG*, pages 203–222. LNCS, 1996.

[SLCZ09] B. Spencer, R. S. Laramee, G. Chen, and E. Zhang. Evenly-spaced Streamlines for Surfaces. *Computer Graphics Forum*, 28(6):1618–1631, 2009.

[SM04] G. L. Schussman and K. L. Ma. Anisotropic Volume Rendering for Extremely Dense, Thin Line Data. In *Proceedings IEEE Visualization '04*, pages 107–114, 2004.

[SP99] I. A. Sadarjoen and F. H. Post. Geometric Methods for Vortex Extraction. In *Data Visualization '99*, Eurographics, pages 53–62. Springer, May 1999.

[SRBE99] M. Schulz, F. Reck, W. Bartelheimer, and T. Ertl. Interactive Visualization of Fluid Dynamics Simulations in Locally Refined Cartesian Grids. In *Proceedings IEEE Visualization '99*, pages 413–416, 1999.

[SS06] T. Salzbrunn and G. Scheuermann. Streamline predicates. *IEEE Transactions on Visualization and Computer Graphics*, 12:1601–1612, 2006.

[STWE07] T. Schafhitzel, E. Tejada, D. Weiskopf, and T. Ertl. Point-Based Stream Surfaces and Path Surfaces. In *GI '07: Proceedings of Graphics Interface 2007*, pages 289–296, New York, NY, USA, 2007. ACM.

[SVL91] W. Schroeder, C. R. Volpe, and W. E. Lorensen. The Stream Polygon: A Technique for 3D Vector Field Visualization. In *Proceedings IEEE Visualization '91*, pages 126–132, 1991.

[SvWHP94] I. A. Sadarjoen, T. van Walsum, A. J. S. Him, and F. H. Post. Particle Tracing Algorithms for 3D Curvilinear Grids. In *Scientific Visualization*, pages 311–335, 1994.

[SvWHP97] I. Ari Sadarjoen, Theo van Walsum, Andrea J. S. Him, and Frits H. Post. *Scientific Visualization, Overviews, Methodologies, and Techniques*, chapter Practical Tracing Algorithms for 3D Curvilinear Grids, pages 299–323. IEEE Computer Society, 1997.

[SWS09] D. Schneider, A. Wiebel, and G. Scheuermann. Smooth Stream Surfaces of Fourth Order Precision. *Computer Graphics Forum*, 28(3):871–878, 2009.

[SZD$^+$10] J. Sanyal, S. Zhang, J. Dyer, A. Mercer, P. Amburn, and R. Moorhead. Noodles: A tool for visualization of numerical weather model ensemble uncertainty. *IEEE Transactions on Visualization and Computer Graphics*, 16:1421–1430, 2010.

[TACSD06] Y. Tong, P. Alliez, D. Cohen-Steiner, and M. Desbrun. Designing Quadrangulations with Discrete Harmonic Forms. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, pages 201–210, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[TB96] G. Turk and D. Banks. Image-Guided Streamline Placement. In *ACM SIGGRAPH 96 Conference Proceedings*, pages 453–460, August 1996.

[TC05] J.J. Thomas and K.A. Cook. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. IEEE, 2005.

[TE99] C. Teitzel and T. Ertl. New Approaches for Particle Tracing on Sparse Grids. In *Data Visualization '99*, Eurographics, pages 73–84. Springer-Verlag, May 1999.

[TEC] TECPLOT. http://www.tecplot.com/, last accessed 2012-02-21.

[TGE97] C. Teitzel, R. Grosso, and T. Ertl. Efficient and Reliable Integration Methods for Particle Tracing in Unsteady Flows on Discrete Meshes. In *Visualization in Scientific Computing '97*, Eurographics, pages 31–42. Springer-Verlag Wien New York, 1997.

[TGE98] C. Teitzel, R. Grosso, and T. Ertl. Particle Tracing on Sparse Grids. In *Visualization in Scientific Computing '98*, Eurographics, pages 81–90. Springer-Verlag Wien New York, 1998.

[USM96] S. K. Ueng, C. Sikorski, and K. L. Ma. Efficient Streamline, Streamribbon, and Streamtube Constructions on Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):100–110, June 1996.

[vFWS$^+$08] W. von Funck, T. Weinkauf, J. Sahner, H. Theisel, and H.-C. Hege. Smoke Surfaces: An Interactive Flow Visualization Technique Inspired by Real-World Flow Experiments. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2008)*, 14(6):1396–1403, November - December 2008.

[vH04] D. van Heesch. *Doxygen, Manual for version 1.3.9.1.* The Netherlands, 197-2004.

[VKP00] V. Verma, D. Kao, and A. Pang. A Flow-guided Streamline Seeding Strategy. In *Proceedings IEEE Visualization 2000*, pages 163–170, 2000.

[VP04] V. Verma and A. Pang. Comparative flow visualization. *IEEE Trans. Vis. Comput. Graph.*, 10(6):609–624, 2004.

[vW92] J. J. van Wijk. Rendering Surface-particles. In *Proceedings of IEEE Visualization '92*, pages 54–61, 1992.

[vW93a] J. J. van Wijk. Flow Visualization with Surface Particles. *IEEE Computer Graphics and Applications*, 13(4):18–24, July 1993.

[vW93b] J. J. van Wijk. Implicit Stream Surfaces. In *Proceedings of Visualization '93*, pages 245–252, 1993.

[WBPE90] P. P. Walatka, P.G. Buning, L. Pierce, and P.A. Elson. *PLOT3D User's Manual.* NASA, mar 1990. http://www.openchannelfoundation.org/ [September 2007].

[wGL] wxWidgets GUI Library. `http://www.wxwidgets.org/`.

[WHN$^+$03] T. Weinkauf, H.-C. Hege, B.R. Noack, M. Schlegel, and A. Dillmann. Coherent structures in a transitional flow around a backward-facing step. *Physics of Fluids*, 15(9):S3, September 2003.

[WJE00] R. Westermann, C. Johnson, and T. Ertl. A Level-set Method for Flow Visualization. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 147–154, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[WMW86] G. Wyvill, C. McPheeters, and B. Wyvill. Data Structure for *Soft* Objects. *The Visual Computer*, 2(4):227–234, 1986.

[WNDS07] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide, The Official Guide to Learning OpenGL, Version 2.1.* Addison Wesley, 6 edition, 2007.

[WPL96] C. M. Wittenbrink, A. T. Pang, and S. K. Lodha. Glyphs for Visualizing Uncertainty in Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):266–279, September 1996.

[WS05] A. Wiebel and G. Scheuermann. Eyelet Particle Tracing - Steady Visualization of Unsteady Flow. In *Proceedings of IEEE Visualization 2005*, pages 77–84, 2005.

[WT02] T. Weinkauf and H. Theisel. Curvature measures of 3d vector fields and their applications. *Journal of WSCG*, 10(2):507–514, 2002. WSCG 2002, Plzen, Czech Republic, February 4 - 8.

[XZC04] D. Xue, C. Zhang, and R. Crawfis. Rendering Implicit Flow Volumes. In *Proceedings IEEE Visualization 2004*, pages 99–106, 2004.

[YKP05] X. Ye, D. Kao, and A. Pang. Strategy for Seeding 3D Streamlines. In *Proceedings of IEEE Visualization 2005*, pages 471–476, 2005.

[ZCL08] S. Zhang, S. Correia, and D. H. Laidlaw. Identifying white-matter fiber bundles in DTI data using an automated proximity-based fiber-clustering method. *IEEE Transactions on Visualization and Computer Graphics*, 14(5):1044–53, 2008.

[ZSH96] M. Zöckler, D. Stalling, and H-C. Hege. Interactive Visualization of 3d-Vector Fields using Illuminated Stream Lines. In *Proceedings of IEEE Visualization '96*, pages 107–ff., 1996.